Model-Driven Development of Security-Aware GUIs for Data-Centric Applications

David Basin[1], Manuel Clavel[2,3], Marina Egea[2], Miguel A. García de Dios[2], Carolina Dania[2], Gonzalo Ortiz[2], and Javier Valdazo[2]

[1] ETH Zürich, Switzerland
basin@inf.ethz.ch
[2] IMDEA Software Institute, Madrid, Spain
[manuel.clavel,marina.egea,miguelangel.garcia]@imdea.org
[carolina.dania,gonzalo.ortiz,javier.valdazo]@imdea.org
[3] Universidad Complutense, Madrid, Spain
clavel@sip.ucm.es

Abstract. In this tutorial we survey a very promising instance of model-driven security: the full generation of security-aware graphical user interfaces (GUIs) from models for data-centric applications with access control policies. We describe the modeling concepts and languages employed and how model transformation can be used to automatically lift security policies from data models to GUI models. We work through a case study where we generate a security-aware GUI for a chatroom application. We also present a toolkit that supports the construction of security, data, and GUI models and generates complete, deployable, web applications from these models.

1 Introduction

Model building is at the heart of system design. This is true in many engineering disciplines and is increasingly the case in software engineering. Model-driven engineering (MDE) [7] is a software development methodology that focuses on creating models of different system views from which system artifacts such as code and configuration data are automatically generated. Proponents of model-driven engineering have in the past been guilty of making overambitious claims: positioning it as the Holy Grail of software engineering where modeling completely replaces programming. This vision is, of course, unrealizable in its entirety for simple complexity-theoretic reasons. If the modeling languages are sufficiently expressive then basic problems such as the consistency of the different models or views of a system become undecidable. However, there are specialized domains where MDE can truly deliver its full potential: in our opinion, security-aware GUIs for data-centric applications is one of them.

Data-centric applications are applications that manage information, typically stored in a database. In many cases, users access this information through graphical user interfaces (GUIs). Informally, a GUI consists of widgets (e.g., windows, text-fields, lists, and combo-boxes), which are visual elements that display and

store information and support events (like "clicking-on" or "typing-in"). A GUI defines the layout for the widgets, as well as the actions that the widgets' events trigger either on the application's database (e.g., to create, delete, or update information) or upon other widgets (e.g., to open or close a window).

There is an important, but little explored, link between visualization and security: When the application data is protected by an access control policy, the application GUI should be aware of and respect this policy. For example, the GUI should not display options to users for actions (e.g., to read or update information) that they are not authorized to execute on application data. This, of course, prevents the users from getting (often cryptic) security warnings or error messages directly from the database management system. It also prevents user frustration, for example from filling out a long electronic form only to have the server reject it because the user lacks a permission to execute some associated action on the application data. However, manual encoding the application's security policy within the GUI code is cumbersome and error prone. Moreover, the resulting code is difficult to maintain, since any changes in the security policy will require manual changes to the GUI code.

In this tutorial we spell out our model-driven engineering approach for developing security-aware GUIs for data-centric applications. The backbone of this approach, illustrated in Figure 1, is a model transformation that automatically lifts the access control policy modeled at the level of the data to the level of the GUI [2]. More precisely, given a security model (specifying the access control policy on the application data) and a GUI model (specifying the actions triggered by the events supported by the GUI's widgets), our model transformation generates a GUI model that is security-aware. The key idea underlying this transformation is that the link between visualization and security is ultimately defined in terms of data actions, since data actions are both controlled by the security policy and triggered by the events supported by the GUI. Thus, under our approach, the process of modeling and generating security-aware GUIs has the following parts:

1. Software engineers specify the application-data model.
2. Security engineers specify the security-design model.
3. GUI designers specify the application GUI model.
4. A model transformation automatically generates a security-aware GUI model from the security model and the GUI model.
5. A code generator automatically produces a security-aware GUI from the security-aware model.

The other key components of this approach are the languages that we propose for modeling the data (ComponentUML), the access control policy (SecureUML), and the GUI (ActionGUI). These languages are defined by their corresponding metamodels and support the rigorous modeling of a large class of data models, security models, and GUI models. For data models, the main modeling elements are entities, along with their attributes and associations; for security models, these elements are roles, permissions (possibly constrained at

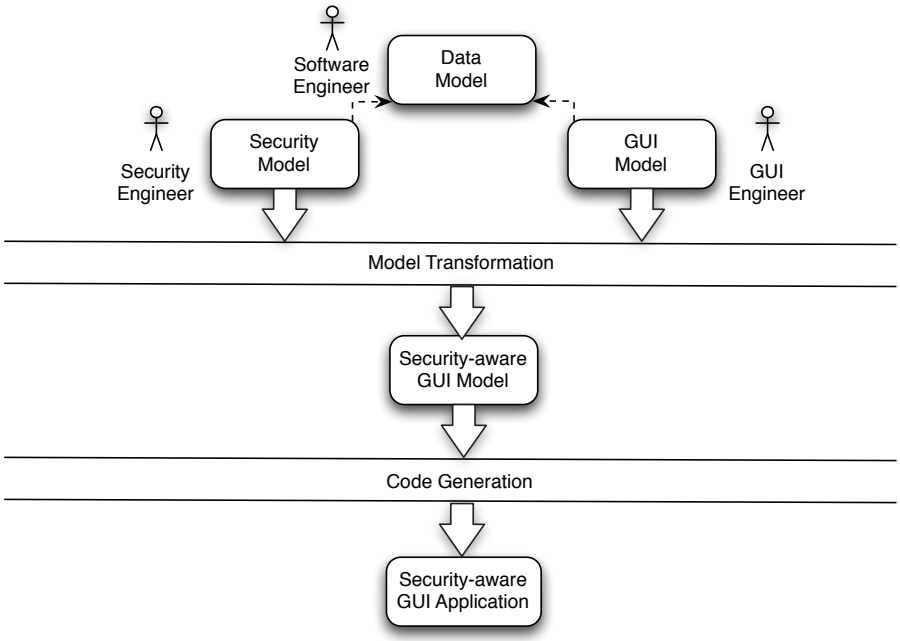

Fig. 1. Model-driven development of security-aware GUIs.

runtime to satisfy given properties), and the actions associated to these permissions. For GUI models, these elements are widgets, the (possibly conditional) events associated to these widgets, and the (possibly conditional) actions associated to these events. The constraint language OCL [10] is used in all of these models. For security models, OCL is used to formalize the constraints on the permissions. For GUI models, it is used to formalize the conditions on the actions, as well as to specify the information to be displayed in widgets, updated in the database, or passed from one widget to another.

To support a full model-driven engineering development process, we have built a toolkit, named the ActionGUI Toolkit. This features specialized model editors for data, security, and GUI models, and implements the aforementioned model transformation to automatically generate security-aware GUI models. Moreover, our toolkit includes a code generator that, given a security-aware GUI model, automatically produces a complete web application, ready to be deployed in web containers such as Tomcat or GlassFish. A key component of this code generator is our translator from OCL to an SQL-based query language [5], which handles the OCL expressions appearing in the security-aware GUI models. More information about our ActionGUI Toolkit can be found at the URL `http://www.bm1software.com/actiongui.html`.

Overall, we see the full generation of security-aware GUIs from models for data-centric applications as a very promising application for model-driven engineering. By working with models and using code-generators to produce the final products, GUI designers can focus on the GUI's layout and behavior, instead of wrestling with the different, often complex, technologies that are used to implement them. Moreover, by using model transformations, the problem of establishing the link between visualization and security is successfully addressed.

To appreciate this last point, consider the standard alternative: the default, "ad-hoc" solution of directly hard-coding the security policy within the GUI. This is clearly disadvantageous. First, the GUI designer is often unaware of the application data security policy. Second, even if the designer is aware of it, manual hard-coding the application data security policy within the GUI code is cumbersome and error-prone. Finally, any changes in the security policy will require manual changes to the GUI code that implements this policy, which again is cumbersome and error-prone.

Organization

We explain in this tutorial our approach for developing security-aware GUIs for data-centric applications and present a toolkit, named ActionGUI, supporting this approach. We begin in Sections 2–4 by introducing our modeling languages for data models (ComponentUML), security models (SecureUML), and GUI models (ActionGUI). We also introduce our running example: a basic chatroom application. In Section 5, we discuss the problem of lifting the security requirements from data models to GUI models, and we present our solution: a model transformation that automatically transforms a GUI model into a security-aware GUI model with respect to the security requirements imposed on the underlying data model. We conclude this tutorial with a discussion on current and future work. All of the models we present here (and many more) are available at the ActionGUI home page. The interested reader can also evaluate there the code generated by the ActionGUI Toolkit from these models.

2 Data models: ComponentUML

In this and the next two sections, we introduce the modeling languages that we use for the model-driven development of security-aware GUIs for data centric applications. These languages are: ComponentUML, for modeling data; SecureUML, for modeling the access control policy; and ActionGUI, for modeling the application's GUI. To illustrate the main modeling concepts and relationships provided by these languages, we work through a running example: a simple chat application named ChitChat, which supports multiple chat rooms where users can converse with each other in different chat rooms. We begin then with ComponentUML, which is the language that we use for modeling the application data. ComponentUML also gives us a context to introduce the constraint

language OCL [10], which we intensively use in our approach when modeling both access control policies and the application's GUIs.

Data models provide a data-oriented view of a system. Typically they are used to specify how data is structured, the format of data items, and their logical organization, i.e., how data items are grouped and related. ComponentUML essentially provides a subset of UML class models where *entities* (classes) can be related by *associations* and may have *attributes* and *methods*. Attributes and *association-ends* have *types* (either primitive types or entity types). As expected, the type associated to an attribute is the type of the attribute's values, and the type associated to an association-end is the type of the objects which may be linked at this end of the association. The ComponentUML metamodel is shown in Figure 2.

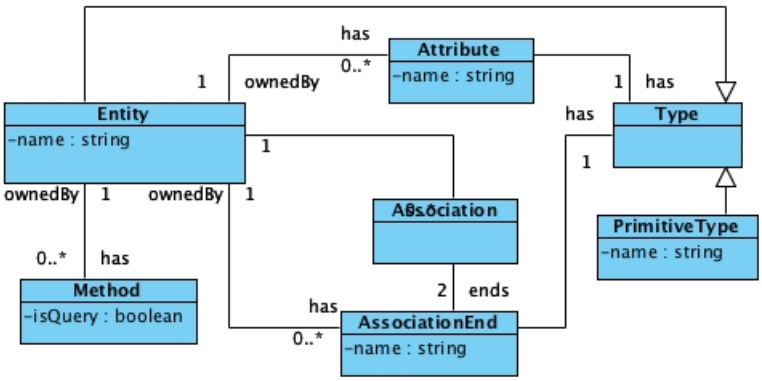

Fig. 2. The ComponentUML metamodel.

The ChitChat data model. In our ChitChat application, each user has a nickname, a password, an e-mail address, a mood message, and a status. The user may participate and be invited to participate in any number of chat rooms. Each chat room is created by a user, has a name, a starting and ending date, and it manages the messages sent by its participants.

The table shown in Figure 3 specifies the data model of the ChitChat application, using the concepts and relationships provided by ComponentUML. Each row in this table corresponds to an entity, and shows its attributes and association-ends, with their corresponding types. Notice that we also indicate, for each association-end, the association-end which corresponds to the other end of the association. The reader can find on the ActionGUI home page the graphical representation of the ChitChat data model using the concrete syntax for ComponentUML that is supported by the ActionGUI Toolkit.

Entity	Attribute	Type	AssocEnd	Type	*Other end*
ChatUser	nickname	String	msgSent	ChatMessage	*from*
	password	String	participates	ChatRoom	*participants*
	email	String	owns	ChatRoom	*ownedBy*
	moodMsg	String	invitedTo	ChatRoom	*invitees*
	status	String			
ChatRoom	name	String	messages	ChatMessage	*chat*
	start	Date	participants	ChatUser	*participates*
	end	Date	ownedBy	ChatUser	*owns*
			invitees	ChatUser	*invitedTo*
ChatMessage	body	String	from	ChatUser	*msgSent*
			chat	ChatRoom	*messages*

Fig. 3. The ChitChat data specification.

OCL: constraints and queries

The Object Constraint Language (OCL) [10] is a specification language for expressing constraints and queries using a textual notation. As part of the UML standard, it was originally intended for modeling properties that could not be easily or naturally captured using graphical notation (e.g., class invariants in a UML class diagram). In fact, OCL expressions are always written in the context of a model, and they are evaluated on an instance of this model. This evaluation returns a value but does not change anything; OCL is a side-effect free language.

We summarize here the main elements of the OCL language which are used in this tutorial. OCL is a strongly type language. Expressions either have a primitive type (namely, Boolean, Integer, Real, and String), a class type, or a collection type, whose base type is either a primitive type or a class type. OCL provides the standard operators on primitive types and on collections. For example, the operator includes checks whether an object is part of a collection, and the operator isEmpty checks whether a collection is empty. More interestingly, OCL provides a dot-operator to access the values of the objects' attributes and association-ends. For example, let u be an object of the class ChatUser. Then, the expression u.nickname refers to the value of the attribute nickname for the ChatUser u, and the expression u.participates refers to the objects linked to the ChatUser u through the association-end participates. Furthermore, OCL provides the operator allInstances to access to all the objects of a class. For example, the expression ChatRoom.allInstances() refers to all the objects of the class ChatRoom. Finally, OCL provides operators to iterate on collections. These are forAll, exists, select, reject, and collect. For example, ChatUser.allInstances()−>select(u|u.status='on−line') refers to the collection of objects of the class ChatUsers whose attribute status has the value "on-line".

ChitChat's entity invariants. To illustrate the syntax (and the semantics) of the OCL language, we formalize here some entity (class) invariants for ChitChat's

data model. For example, the following OCL expression formalizes that users' nicknames must be unique:

```
ChatUser.allInstances()->forall(u1,u2| u1 <> u2 implies u1.nickname <> u2.nickname).
```

Similarly, we can formalize that the status of a ChitChat user is either "off-line" or "on-line" using the following OCL expression:

```
ChatUser.allInstances() ->forall(u|u.status='on-line' or u.status='off-line').
```

Finally, we can formalize that each message has exactly one sender:

```
ChatMessage.allInstances()->forAll(m|m.from->size()= 1).
```

3 Security models: SecureUML+ComponentUML

SecureUML [3] is a modeling language for formalizing access control requirements that is based on RBAC [6]. In RBAC, permissions specify which roles are allowed to perform given operations. These roles typically represent job functions within an organization. Users are granted permissions by being assigned to the appropriate roles based on their competencies and responsibilities in the organization. RBAC additionally allows one to organize the roles in a hierarchy where roles can inherit permissions along the hierarchy. In this way, the security policy can be described in terms of the hierarchical structure of an organization. However, it is not possible in RBAC to specify policies that depend on dynamic properties of the system state, for example, to allow an operation only during weekdays. SecureUML extends RBAC with *authorization constraints* to overcome this limitation.

SecureUML provides a language for specifying access control policies for actions on protected resources. However, it leaves open what the protected resources are and which actions they offer to actors. These are specified in a so-called "dialect". Figure 4 shows the SecureUML metamodel. Essentially, it provides a language for modeling *roles* (with their hierarchies), *permissions*, *actions*, *resources*, and *authorization constraints*, along with their assignments, i.e., which permissions are assigned to a role, which actions are allowed by a permission, which resource is affected by the actions allowed by a permission, which constraints need to be satisfied for granting a permission, and, finally, which resource is affected by an action.

In our approach, we use a specific dialect of SecureUML, named SecureUML+ComponentUML, for modeling the access control policy on data models. The SecureUML+ComponentUML metamodel provides the connection between SecureUML and ComponentUML. Essentially, in this dialect of SecureUML, the protected resources are the entities, along with their attributes, methods, and association-ends (but not the associations as such), and the actions that they offer to the actors are those shown in Figure 5. Essentially, there are two classes of actions: atomic and composite. The *atomic* actions are intended to map directly

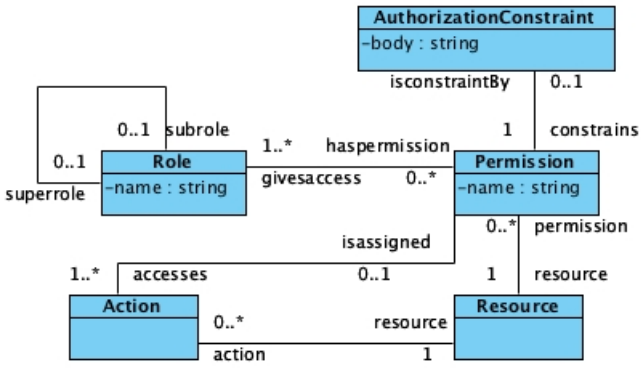

Fig. 4. The SecureUML metamodel.

onto actual operations on the database. These actions are: create and delete for entities; read and update for attributes; create and delete for association-ends; and execute for methods. The underlined actions are the *composite* actions, which hierarchically group lower-level actions. Composite actions allow modelers to conveniently specify permissions for sets of actions. For example, the full access action for an attribute groups together the read and update actions for this attribute.

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

Fig. 5. The SecureUML+ComponentUML actions on protected resources.

Finally, in SecureUML+ComponentUML, authorization constraints are specified using OCL, extended by four keywords, self, caller, value, and target. These keywords have the following meanings:

- self refers to the root resource upon which the action will be performed, if the permission is granted. The root resource of an attribute, an association-end, or a method is the entity to which it belongs.
- caller refers to the actor that will perform the action, if the permission is granted.
- value refers to the value that will be used to update an attribute, if the permission is granted.

– target refers to the object that will be linked at the end of an association, if the permission is granted.

The ChitChat access control policy. For the sake of our running example, consider the following (partial) access control policy for the ChitChat application:

– Only administrators can create or delete users;
– Administrators can read any user's nickname, email, mood message, and status.
– Any user can read and update its own nickname, password, email, mood message, and status.
– Any user can read other users' nicknames, mood messages, and status.
– Users can join a chat room by invitation only, but they can leave at any time.

The table shown in Figure 6 specifies this (partial) access control policy, using the concepts and relationships provided by SecureUML+ComponentUML. Each row in this table corresponds to a role, and shows its permissions. Moreover, for each permission it shows the actions allowed by the permission, the resource affected by each of these actions, and the constraint that must be satisfied for the permission to be granted. For example, the permission DisjointChat authorizes any user to leave a chat room at any anytime. More precisely, it allows any user caller to delete a participates-link between a user self and a chat room target (meaning that the user self leaves the chat room target), but only if the user caller is indeed the user self (that is, the user caller is the one leaving the chat room target), and also the chat room target indeed belongs to the collection of chat rooms linked to the user caller through the association-end participates (that is, the caller is actually participating in the chat room target). The reader can find in the ActionGUI home page the graphical representation of ChitChat's access control policy using the concrete syntax for SecureUML+ComponentUML that is supported by the ActionGUI Toolkit.

4 GUI models: ActionGUI

ActionGUI is a modeling language for formalizing the GUIs of a rich class of data-centric applications. The ActionGUI metamodel is shown in Figure 7. In a nutshell, ActionGUI provides a language to model *widgets* (e.g., windows, text-fields, buttons, lists, and tables), *events* (e.g., clicking-on, typing-in), and *actions*, which can be on data (e.g., to update a property of an element in the database) or on other widgets, (e.g., to open a window), as well as the associations that link the widgets with the events that they support and the events with the actions that they trigger. In addition, ActionGUI provides support to formally model the following features:

– Widgets can be displayed in *containers*, which are also widgets (e.g., a window can contain other widgets).

Role	Permission	Action	Resource	Authorization Constraint
Admin	AnyUser	create, delete	ChatUser	true
		read	nickname	
		read	email	
		read	moodMsg	
		read	status	
User	SelfUser	read, update	nickname	caller=self
		read, update	password	
		read, update	email	
		read, update	moodMsg	
		read, update	status	
	OtherUser	read	nickname	true
		read	moodMsg	
		read	status	
	JointChat	create	participates	self=caller and caller.invitedTo−>includes(target)
	DisjointChat	delete	participates	self=caller and caller.participates−>includes(target)

Fig. 6. The ChitChat access control policy (partial).

- Widgets may own *variables*, which are in charge of storing information for later use.
- Events may be only supported upon the satisfaction of specific *conditions*, whose truth value can depend on the information stored in the widgets' variables or in the database.
- Actions may be only triggered upon the satisfaction of specific *conditions*, whose truth value can depend on the information stored in the widgets' variables or in the database.
- Actions may take their *arguments* (values that instantiate their parameters) from the information stored in the widgets' variables or in the database.

The ActionGUI metamodel's invariants specify: (i) for each type of widget, the "default" variables that widgets of this type always own; (ii) for each type of widget, the type of events that widgets of this type may support; and (iii) for each type of action, the arguments that actions of this type require, as well as the arguments (if any) that these actions may additionally take. In particular, the invariants of ActionGUI's metamodel formalize, among others, the following constraints about the different types of widgets:

- *Windows.* They can contain any type of widget, except windows. Windows are not contained in any widget.
- *Text-field.* They can be typed-in. By default, each text-field owns a variable text of type string, which stores the last string typed-in by the user. The value of the variable text is permanently displayed in the text-field.
- *Button.* They can be clicked-on.
- *List.* They contain exactly one text-field. By default, each list owns a variable rows of type collection. A list displays as many rows as elements are in

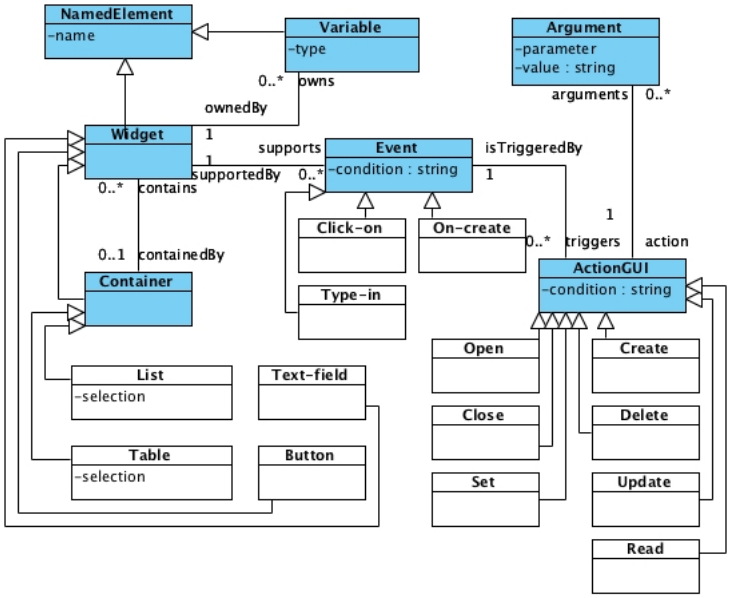

Fig. 7. The ActionGUI metamodel.

the collection stored by the variable rows, each row containing exactly one instance of the text-field contained by the list. By default, each instance of this text-field owns a variable row whose value is the element associated to this row from those stored by the variable rows. Finally, by default, each list owns a variable selected that holds the element associated to the last row selected by the user.

- *Combo-box.* They are similar to lists, except that rows are displayed in a drop-down box.
- *Table.* They are similar to lists, except that they can contain any number of text-fields, buttons, lists, or even tables.

Also, the invariants of ActionGUI's metamodel formalize, among others, the following invariants about the different types of actions:

- *Create.* It creates a data item in the database. It takes two arguments: the type of the new data item (type) and the variable that will store this element for later reference (variable).
- *Delete (entities).* It deletes a data item in the database. It takes as argument the element to be deleted (object).
- *Read.* It reads the value of a data item's attribute in the database. It takes three arguments: the data item whose property is to be read (object); the property to be read (attribute); and the variable that will store, for later reference, the value read (variable).

– *Update.* It modifies the value of a data item's attribute in the database. It takes three arguments: the data item whose attribute is to be modified (object); the attribute to be modified (attribute); and the new value (value).
– *Create (association-ends).* It creates a new link in the database between two data items. It takes three arguments: the source data item (sourceObject); the target data item (targetObject); and the association-end (associationEnd) through which the target data item will be linked to the source data item.
– *Delete (association-ends).* It deletes a link in the database between two data items. It takes three arguments: the source data item (sourceObject); the target data item (targetObject); and the association-end (associationEnd) from where the target data item will be removed.
– *Open.* It opens a window. It takes as argument the window to be opened (target); additionally, for any of this window's variables, it can take as argument a value to be assigned to this variable when opening the window.
– *Back.* It goes back to the window from which a window was open.
– *Set.* It assigns a new value to a widget's variable. It takes two arguments: the variable (target) and the value to be assigned to this variable (value).

Finally, actions' conditions and arguments are specified in ActionGUI models using OCL, extended with the widget's variables (always enclosed in square brackets). As expected, when evaluating an OCL expression that contains a widget's variable, the value of the corresponding subexpression is the value currently stored in the variable. In case of ambiguity, a widget's variable is denoted by its name, prefixed by the name of its widget (followed by a dot). Also, in case of ambiguity, the name of a widget is prefixed by the name of its container (followed by a dot). Notice that, within the same containers, widgets have unique names. Moreover, a widget's variable can only be used within the window that contains its widget, either directly or indirectly.

The ChitChat login window. To continue with our running example, consider the following interface for allowing a registered user to login into the ChitChat application: a window (loginWi) containing:

– a writable text-field (nicknameEn), for the user to type its nickname in;
– a writable text-field (passwordEn), for the user to type its password in; and
– a clickable button (loginBu), for the user to login, using as its nickname and password the strings that it typed in the text-fields nicknameEn and passwordEn, respectively. Upon successful authentication, the user will be directed to the application's main menu window (menuWi) as the logged-in user (caller).

The table shown in Figure 8 specifies this login window, using the concepts and relationships provided by ActionGUI. Each row in this table correspond to a widget, where the containment relationship is denoted by displaying the widgets using tree-like notation. For each widget, we show the variables owned by the widget and the events that it supports. Moreover, for each event, we show the actions triggered by this event, as well as its arguments, indicating the values

for each of the actions' parameters. However, we neither show in this table the "default" widget's variables nor the events supported by the widgets when they do not trigger any action.

Widgets	**Variables**	**Events**	**Actions**	**Arguments**
loginWi				
|nicknameEn				
|passwordEn				
|loginBu | |		click-on	open$^{(\star)}$	window = menuWi menuWi.caller = *authenticated user*

$^{(\star)}$ *Upon successful authentication.*

Fig. 8. The ChitChat login window.

Notice also that there are two elements that we have intentionally left undefined in this table: the value to be assigned to the menuWi's variable caller when opening the window menuWi, and the condition for this action. We now describe how both elements can be defined using ActionGUI's extension of OCL.

- The *authenticated user* should be the registered user in the database whose nickname and password coincide with the values of the text-variables owned, respectively, by text-fields nicknameEn and passwordEn. Using our extended OCL, we can define the *authenticated user* as follows:

```
ChatUser.allInstances()->any(u|
    u.nickname= [nicknameEn.text] and u.password=[passwordEn.text])
```

  Notice that, as one of the invariants of the ChitChat data model, we specified that nicknames shall be unique. Thus, although the any-iterator will return *any* registered user satisfying the body of the any-iterator, there will be at most one such registered users.
- The condition for opening the window menuWi should be the existence in the database of a registered user whose nickname and password coincide with the values of the text-variables owned, respectively, by text-fields nicknameEn and passwordEn. We can define this condition as follows:

```
ChatUser.allInstances()->exists(u|
    u.nickname= [nicknameEn.text] and u.password=[passwordEn.text])
```

The reader can find in the ActionGUI home page the graphical representation of ChitChat's login window using the concrete syntax for ActionGUI that is supported by the ActionGUI Toolkit.

The ChitChat menu window. Consider now the following interface for allowing a logged-in user to choose an option from ChitChat's main menu: a window (menuWi), owning a variable caller which stores the logged-in user, and containing:

- a selectable list (usersLi) with as many rows as registered users are online, each of these rows containing an unwritable text-field (nicknameLa) showing the nickname of the registered user associated to this row;
- a clickable button (editProfileBu) for the caller to access the interface for editing the profile (i.e., name, password, email, mood message, and status) of the user selected in the list usersLi;
- a clickable button (createChatBu) for the caller to access the interface for creating a new chat room; and
- a clickable button (closeChatBu) for the caller to close the window.

The table shown in Figure 9 specifies this menu window, using the concepts and relationships provided by ActionGUI. Notice that the collection of data items to be displayed in the list `usersLi`, namely, the *online users*, is not formally defined in this table. Using ActionGUI's extension of OCL, we can define this collection as follows:

```
ChatUser.allInstances()->select(u|u.status= 'on-line')
```

The reader can find in the ActionGUI home page the graphical representation of ChitChat's menu window using the concrete syntax for ActionGUI that is supported by the ActionGUI Toolkit.

Widgets	**Variables**	**Events**	**Actions**	**Arguments**
menuWi	caller			
| usersLi |		on-create	set	target = rows value = *on-line users*
| | nicknameLa | | | |		on-create	read	object = [UsersLi.row] attribute = nickname variable = text
| editProfileBu | |		click-on	open	window = editProfileWi editProfileWi.selectedUser = [UsersLi.selected]
| createChatBu		click-on	open	window = createChatWi
| closeChatBu		click-on	close	

Fig. 9. The ChitChat menu window.

5 Security-aware ActionGUI models

In this section, we propose our solution to what we believe is the key challenge when modeling security-aware GUIs for data-centric applications: Which method

should the GUI designer use for establishing the link between visualization and security? In other words, how should the GUI designer model their GUIs so as to make them *aware of* and *respect* the access control policy that protects the application data? As mentioned before, establishing this link is crucial when modeling security-aware GUIs for data-centric applications. Basically, a GUI should not display options to users for actions (e.g., to read or update information) that they are not authorized to execute on application data. This prevents the users from getting security warnings or cryptic error messages directly from the database management system. It also prevents user frustration from filling out a long electronic form only to have the server reject it because the user lacks a permission to execute some associated action on the application data.

As we will motivate in this section, *manually* modeling (and even worse, manually encoding) the application's security policy within the GUI model is cumbersome, error prone, and scales poorly to large applications. Moreover, the resulting models are difficult to maintain, since any changes in the security policy will require manual changes to the GUI models. Our solution to this problem uses a standard technique in model-driven development: *model transformation*. In particular, we introduce a model transformation that *automatically* transforms a GUI model into a security-aware GUI model with respect to the access control policy imposed on the underlying data model. One additional and substantial advantage of our solution is that, by keeping the security models and the GUI model apart, the security engineers and the GUI designers can independently model what they know best and maintain their models independently.

The ChitChat edit-profile window. To motivate the problem faced by a GUI designer when modeling a security-aware GUI, let us continue with our running example. Consider the interface for allowing a logged-in user (caller) to edit the profile of a previously chosen user (selectedUser), which may of course be the caller itself. More specifically, this interface shall consist of a window such that:

1. The current values of the selectedUser's profile are displayed when opening the window.
2. The caller can type in the new values (if any) for the selectedUser's profile.
3. The selectedUser's profile is updated with the new values typed in by the caller when he or she clicks on a designated button.

Recall that a registered user's profile is composed of the following attributes: nickname, password, mood message, email, and status. Recall also that the access control policy for reading and updating users' profiles, as specified in Figure 6, is the following:

4. A user is always allowed to read and update its own nickname, password, mood message, email, and status.
5. A user is allowed to read another user's nickname, mood message, and status, but not the user's password or email.
6. An administrator is always allowed to read a user's nickname, mood message, status, and email, but not the user's password.

Now, if the GUI designer only takes into consideration the functional requirements (1–3), the ChitChat edit-profile window can be modeled as shown in Figure 10. Namely, a window editProfileWi, which owns the variable selectedUser and caller, and contains:

- A writable text-field nicknameEn, for the caller to type in the new value (if any) with which to update the selectedUser's nickname. Notice that when the text-field nicknameEn is created, its "default" variable text will be assigned the current value of the selectedUser's nickname, and therefore this value will be the string initially displayed in the text-field nicknameEn, as requested.
- Analogous writable text-fields for each of the other elements in a registered user's profile: password, mood message, email, and status.
- A clickable button updateBu for the caller to trigger the sequence of actions that will update, as requested, the selectedUser's nickname, password, mood message, and status, with the new values (if any) typed by the caller in the corresponding text-fields.

Obviously, the edit-profile window modeled in Figure 10 does not satisfy the security requirements (4–6). Any caller can read and update any value contained in the profile of any selectedUser! So how can the GUI designer model the edit-profile window to make it aware of and respect the security requirements (4–6)?

There are essentially two solutions available. Unfortunately both of them, if applied *manually*, are cumbersome and error prone and therefore impractical for large applications with complex security policies. To understand the challenge that faces a GUI designer when modeling security-aware GUIs, let us first highlight the knowledge that he or she must acquire to accomplish this task. We decompose this into two steps.

Step 1 For each action triggered by an event, and for each role considered by the access control policy, the GUI designer must determine (i) *the conditions under which the given action can be securely executed by a user with the given role*. This depends, of course, on the underlying access control policy, and, more specifically, on the authorization constraints assigned to the permissions which grant access to execute the given action for the given role. Also, recall that permissions are inherited along the role hierarchy.
Notice however that, to obtain (i), the GUI designer can not simply compose, using disjunctions, the authorization constraints assigned to the aforementioned permissions. In particular, the variable caller, if it appears in any of these authorization constraints, must be replaced by the (widget) variable that stores the current application's user. Furthermore, the variables self, value, and target, if they appear in any of the aforementioned authorization constraints, must also be replaced by the appropriate expressions, based on the arguments taken by the given action. For example, in the case of a read-action, the variable self should be replaced by the value of the parameter object for this action.

Step 2 For each event supported by a widget, and for each role considered by the access control policy, the GUI designer must determine (ii) *the conditions*

Widgets	Variables	Events	Actions	Arguments
editProfileWi	caller, selectedUser			
| nicknameEn		on-create	read	object = [selectedUser] attribute = nickname variable = text
| passwordEn		on-create	read	object = [selectedUser] attribute = password variable = text
| moodMsgEn		on-create	read	object = [selectedUser] attribute = moodMsg variable = text
| emailEn		on-create	read	object = [selectedUser] attribute = email variable = text
| statusEn		on-create	read	object = [selectedUser] attribute = status variable = text
| updateBu		click-on	update	object = [selectedUser] attribute = nickname value = [nicknameEn.text]
			update	object = [selectedUser] attribute = password value = [passwordEn.text]
			update	object = [selectedUser] attribute = moodMsg value = [moodMsgEn.text]
			update	object = [selectedUser] attribute = email value = [emailEn.text]
			update	object = [selectedUser] attribute = status value = [statusEn.text]

Fig. 10. The ChitChat edit-profile window (although security-unaware).

under which all the actions triggered by the given event can be securely executed by a user with the given role. In this case, for each given role, the GUI designer can simply compose, using conjunctions, the results to determine (i) for every action triggered by the given event.

Let us now illustrate, using our running example, the two solutions that are currently available to the GUI modeler for turning a security-unaware GUI model into a security-aware one. To simplify the discussion, we assume from now on that the user currently logged-in is always stored in a (widget) variable caller, that this variable is owned by every window, and that this variable's value is automatically passed from one window to another window when opening the latter from the former. Similarly, we assume the role of the user currently logged-in (if he or she has several roles, then the "active" role) is always stored in a (widget) variable role, that this variable is owned by every window, and that this variable's value is automatically passed from one window to another window when opening the latter from the former.

Solution A. The first solution for the GUI designer consists in modeling as many different edit-profile windows as possible security scenarios. In our case, the GUI designer must model three different edit-profile windows (editMyProfileWi, editOthersProfile, and editUsersProfile), one for each of the following scenarios:

1. When the caller has the role 'User' and coincides with the selectedUser.
2. When the caller has the role 'User' but does not coincide with the selectedUser.
3. When the caller has the role 'Admin'.

In particular, the window editMyProfileWi associated to the security scenario (1) will contain exactly the same widgets as the window editProfileWi in Figure 10. In contrast, the window editOthersProfileWi associated to the security scenario (2), will only contain the text-fields nicknameEn, moodMsgEn, and statusEn (and not the button updateBu), since a user with the role 'User' can only read (but not update) other user's nickname, mood message, and status. Similarly, the window editProfilesWi will only contain the text-fields nicknameEn, moodMsgEn, emailEn, and statusEn (and not the button updateBu), since a user with the role 'Admin' can read (but not update) any user's profile, except its password.

Furthermore, for this solution to work, every event intended to give access to the interface for editing users' profiles must also be aware of the security scenarios (1)–(3), in order to open the appropriate edit-profile window. In particular, the GUI designer must associate the sequence of conditional actions shown in Figure 11 to each of the aforementioned events. Notice that for each open-action, the imposed condition formalizes *the conditions under which all the actions triggered by all the events supported by the widgets which are contained in the window being opened can be securely executed by a user.* That is, the GUI designer must gather all the knowledge corresponding to (ii), in Step 2 above, for every event supported by any widget contained in the window which the action under consideration is about to open.

Actions		Arguments
if		[role] = 'User' and [caller] = [selected_user]
then	open	window = editMyProfileWi editMyProfileWi.selectedUser = [selected_user]
if		[role] = 'User' and [caller] <> [selected_user]
then	open	window = editOthersProfileWi editOthersProfileWi.selectedUser = [selectedUser]
if		[role] = 'Admin'
then	open	window = editUsersProfileWi editUsersProfileWi.selectedUser = [selectedUser]

Fig. 11. Solution A: Conditions for opening the edit-profile windows.

Solution B. Another solution for the GUI designer consists of specifying, for each of event supported by a widget, *the conditions under which all the actions triggered by the given event can be securely executed by a user*. In our case, the GUI design must convert the edit-profile window model shown in Figure 10 into the one shown in Figure 12.

This solution is certainly simpler and less intrusive than Solution A with respect to the original, security-unaware GUI model (since, for example, no new windows need to be added to the original design). However, in order to implement this solution, the GUI designer must gather all the knowledge corresponding to (ii), in Step 2 above, for every event supported by every widget in the model.

A model-transformation approach. Clearly, manually modeling the application's security policy within the GUI model is problematic. It is cumbersome, error prone, and scales poorly to large applications. Moreover, it requires the GUI designer to have complete knowledge of the access control policy on the application data. Finally, the resulting GUI models are difficult to maintain.

To address the problem of establishing the link between visualization and security, we employ a standard technique from model-driven development: model transformation. A model transformation takes as input a model (or several models) conforming to given metamodel (respectively, several metamodels) and produces as output a model conforming to a given metamodel. More specifically, the ActionGUI Toolkit implements a model transformation that takes as input an ActionGUI model and a SecureUML+ComponentUML model, and *automatically* produces as output an ActionGUI model. This output model is identical to the input ActionGUI model except that it is now security-aware with respect to the access control policy specified in the input SecureUML+ComponentUML model. In particular, our model transformation follows the ideas behind Solution B: it specifies for each event supported by a widget, the conditions under which all the actions triggered by this event can be securely executed by a user. As expected, at the core of this model transformation, we have implemented a function (on the input models) that *automatically* gathers all the knowledge cor-

Widgets	Events		Actions	Arguments
nicknameEn	on-create	if true		
		then	read	object = [selectedUser] attribute = nickname variable = text
passwordEn	on-create	if [role] = 'User' and [caller] = [selectedUser]		
		then	read	object = [selectedUser] attribute = password variable = text
moodMsgEn	on-create	if true		
		then	read	object = [selectedUser] attribute = moodMsg variable = text
emailEn	on-create	if (([role] = 'User' and [caller] = [selectedUser]) or [role] = 'Admin')		
		then	read	object = [selectedUser] attribute = email variable = text
statusEn	on-create	if true		
		then	read	object = [selectedUser] attribute = status variable = text
updateBu	click-on	if [role] = 'User' and [caller] = [selectedUser]		
		then	update	object = [selectedUser] attribute = nickname value = [nicknameEn.text]
			update	object = [selectedUser] attribute = password value = [passwordEn.text]
			update	object = [selectedUser] attribute = moodMsg value = [moodMsgEn.text]
			update	object = [selectedUser] attribute = email value = [emailEn.text]
			update	object = [selectedUser] attribute = status value = [statusEn.text]

Fig. 12. Solution B: The ChitChat edit-profile window (now security-aware).

responding to (i) and (ii), in Step 1 and Step 2 above, for each action triggered by an event and for each event supported by a widget, respectively.

Continuing with our running example, let us consider how models can be made security-aware and maintained over time. First, how can the GUI designer model the edit-profile window to make it aware of and respect the security requirements (4–6)? Using our ActionGUI Toolkit, the designer simply calls the aforementioned model transformation on ChitChat's access control policy shown in Figure 6 and the (security-unaware) ChitChat edit-profile window model shown in Figure 10. Our model transformation then automatically generates (in practically no time) the security-aware ChitChat edit-profile window model shown in Figure 12. Finally, what must the GUI designer do, with respect to the edit-profile window model, if ChitChat's security policy happens to change? For example, suppose that any user is allowed to read the email of other users when the former participates in a chat room where the latter is also participating. The designer must simply call again the model transformation, this time taking as inputs the modified model of ChitChat's access control policy and, as before, ChitChat's (security-unaware) edit-profile window model. shown in Figure 10.

6 Related Work and Conclusions

The ever-growing development and use of information and communication technology is a constant source of security and reliability problems. Clearly we need better ways of developing software systems and approaching software engineering as a well-founded discipline. In many engineering disciplines, model building is at the heart of system design. This is increasingly the case in software engineering since model-driven software engineering was first proposed over a decade ago. In our opinion, the late adoption of this methodology is due to the difficulty in defining effective domain-specific modeling languages and also the effort required for developers to learn modeling languages and the art of model building.

Defining a good domain-specific modeling language requires finding the right abstractions and degree of precision to capture relevant aspects of the structure and the logic of a software system. In addition, for a modeling language to be really usable and useful for software developers, appropriate tools must be provided to build models, analyze them, and keep them synchronized with end products. We wish to emphasize in this regard the need to focus on concrete domains like access controls, GUIs, data models, and the like. In our experience, only by limiting the domain is it possible to build sufficiently precise modeling languages that support the automatic generation of fully functional applications.

Automatic code generation brings with it important advantages. Once a code generator is implemented for a platform (a one-off cost), modelers can use it like a compiler for a very high-level language, dramatically increasing their productivity. But even when the model-driven development process is not completely automatic, there are experience reports (see, for instance, [9, 11]), in which the productivity of a developer in industry is said to increase by a factor of 2 or 3.

There is an additional argument for using model-driven development to develop security-critical systems, i.e., for model-driven security [1]. Namely, security is often built redundantly into systems. For example, in a web-application, access control may be enforced at all tiers: at the web server, in the back-end databases, and even in the GUI. There are good reasons for this. Redundant security controls is an example of defense in depth and is also necessary to prevent data access in unanticipated ways, for example, directly from the database thereby circumventing the web application server. Note that access control on the client is also important, but more from the usability rather than the security perspective. Namely, although client-side access control may be easy to circumvent, it enhances usability by presenting honest users an appropriate view of their options: unauthorized options can be suppressed and users can be prevented from entering states where they are unauthorized to perform any action, e.g., where their actions will result in security exceptions thrown by the application server or database.

The above raises the following question: must one specify security policies separately for each of these tiers? The answer is "no" for many applications. Security can often be understood in terms of the criticality of data and an access control policy on data can be specified at the level of component (class) models, as discussed in Section 3. Afterwards, an access control policy modeled at the level of components may be lifted to other tiers. When the tiers are also modeled, this lifting can be accomplished using model transformation techniques and in a precise and meaningful way as we have illustrated in Section 5. In general, model transformations support problem decomposition during development where design aspects can be separated into different models which are later composed. As a methodology for designing security-aware GUIs, this approach supports the consistent propagation of a security policy from component models to GUI models and, via code generation, to GUI implementations. This decomposition also means that security engineers and GUI designers can independently model what they know best and maintain their models independently.

Related Work

There are also other tools like WebRatio [12], Olivanova [4], and Lightswitch [8] that support development methodologies for building data-centric applications that are similar to the model-driven methodology presented in this work. In these tools, application development starts by building a data model that reflects the data structure required for the database. The development process continues by applying different UI generation patterns to the data model. These patterns enable data retrieval, data editing, data creation, and database search. A detailed comparison of the expressiveness provided by the languages supported by these tools with the languages supported by the ActionGUI Toolkit is interesting; however it falls out of the scope of this paper. Nevertheless, we note that, in contrast to what these tools can provide, the ActionGUI Toolkit offers developers the full flexibility to create designs without burdening them with the restrictions imposed by the obligatory use of a fixed number of given patterns. The above

tools also impose a major restriction at the level of data management, i.e., at the level of data access and visualization: The information that can be referenced and therefore that can be accessed and visualized within one screen can only come from one table of the database or, at most, from two tables that are reachable from each other within one navigation step.

The three tools, WebRatio, Olivanova, and Lightswitch, support the definition and generation of RBAC policies at different granularity levels. Lightswitch supports granting or denying permissions (whose actual behavior must be manually programmed) to execute create, read, update, or delete actions on entities for users in different roles. WebRatio and Olivanova also support granting or denying permissions to execute a similar set of actions on entity's properties, individually, for users in different roles. In WebRatio and Olivanova, the role of the authorization constraints could be played by preconditions restraining the invocation of actions through a concrete UI. Note, however, that none of these tools implement an algorithm capable of lifting to the UIs the security policy that governs the access to data.

Future Work

The toolkit we presented supports the construction of security, data, and GUI models and generates complete, deployable, security-aware web applications from these models. However, there is still much work ahead to turn this toolkit into a full, robust commercial application. In particular, we plan to add to the language (and to its code-generator) other actions which do not act upon the database or the GUI elements. Examples are sending an email to a contact selected in a list or printing a table.

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

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

Ultimately we see model-driven security playing an important role in the construction and certification of critical systems. For example, certification un-

der the Common Criteria requires models for the higher Evaluation Assurance Levels. Model-driven security provides many of the ingredients needed: models with a well-defined semantics, which can be rigorously analyzed and have a clear link to code. As the acceptance of model-driven development techniques spread, and as they become better integrated with well-established formal methods that support a detailed behavioral analysis, such applications should become a reality.

Acknowledgements

This work is partially supported by the EU FP7-ICT Project "NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems" (256980) by the Spanish Ministry of Science and Innovation Project "DESAFIOS-10" (TIN2009-14599-C03-01), and by Comunidad de Madrid Program "PROMETIDOS-CM" (S2009TIC-1465).

References

1. D. Basin, M. Clavel, and M. Egea. A decade of model driven security. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies (SACMAT '11)*. ACM Press, June 2011. Invited Paper. In Press.
2. D. Basin, M. Clavel, M. Egea, and M. Schläpfer. Automatic generation of smart, security-aware GUI models. In F. Massacci, D. S. Wallach, and N. Zannone, editors, *Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS'10)*, volume 5965 of *LNCS*, pages 201–217, Pisa, Italy, 2010. Springer.
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. Care Technologies. Olivanova – the programming machine, 2011. `http://www.care-t.com`.
5. M. Egea, C. Dania, and M. Clavel. MySQL4OCL: A stored procedure-based MySQL code generator for OCL. *Electronic Communications of the EASST*, 36, 2010.
6. D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.
7. A. Kleppe, W. Bast, J. B. Warmer, and A. Watson. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley, 2003.
8. Microsoft. Visual studio lightswitch, 2010. `http://www.microsoft.com/visualstudio/en-us/lightswitch`.
9. R. Mohan and V. Kulkarni. Model driven development of graphical user interfaces for enterprise business applications - experience, lessons learnt and a way forward. In Andy Schürr and Bran Selic, editors, *Proc. of MODELS 2009*, LNCS, pages 307–321. Springer, 2009.
10. Object Management Group. *Object Constraint Language specification Version 2.2*, February 2010. OMG document available at `http://www.omg.org/spec/OCL/2.2`.

11. A. Schramm, A. Preußner, M. Heinrich, and L. Vogel. Rapid UI development for enterprise applications: Combining manual and model-driven techniques. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proc. of MODELS 2010*, LNCS, pages 271–285. Springer, 2010.
12. Web Models Company. Web ratio – you think, you get, 2010. `http://www.webratio.com`.