A Model-Driven Methodology for Developing Secure Data-Management Applications

David Basin, Manuel Clavel, Marina Egea, Miguel A. García de Dios, Carolina Dania

***Abstract*—We present a novel model-driven methodology for developing secure data-management applications. System developers proceed by modeling three different views of the desired application: its data model, security model, and GUI model. These models formalize respectively the application's data domain, authorization policy, and its graphical interface together with the application's behavior. Afterwards a model-transformation function lifts the policy specified by the security model to the GUI model. This allows a separation of concerns where behavior and security are specified separately, and subsequently combined to generate a security-aware GUI model. Finally, a code generator generates a multi-tier application, along with all support for access control, from the security-aware GUI model. We report on applications built using our approach and the associated tool.**

***Index Terms*—Model-driven development, model-driven security, access control, GUI models, model transformation.**

I. Introduction

Data-management applications are focused around so-called CRUD actions that create, read, update, and delete data from persistent storage. These operations are the building blocks for numerous applications, for example dynamic websites where users create accounts, store and update information, and receive customized views based on their stored data. When the data managed is sensitive, then security is a concern and the use of these actions must be controlled.

Access control is the standard approach to restricting users' actions on data. When the access-control policies are sufficiently simple, it may be possible to formalize them declaratively, independent of the application's business logic. For example, multi-tier systems for web-based applications often build support for role-based access control into the application server, which is configured independently of the application's procedural details. In contrast, fine-grained access control policies may depend not only on the user's credentials but also on the satisfaction of constraints on the state of the persistence layer, i.e. on the values of stored data items. In such cases, authorization checks are typically implemented programmatically, by directly encoding them at appropriate places in the application. Unfortunately, these programmatic additions are cumbersome, error prone, and scale poorly. Moreover, they are difficult to audit and maintain as the authorization checks are spread throughout the code and security policy changes require code changes.

In this paper, we propose a methodology for the model-driven development of secure data-management applications. It consists of languages for modeling multi-tier systems, and a toolkit for generating these systems. Within our methodology, a secure data-management application is modeled using three interrelated models:

1) A *data model* defines the application's data domain in terms of its classes, attributes, associations, and (non-CRUD) methods;
2) A *security model* defines the application's security policy in terms of authorized access to the actions on the resources provided by the data model.
3) A graphical user interface, or *GUI model*, defines the application's graphical interface and application logic. Note, in particular, that this model formalizes both UI structure and behavior.

The heart of this methodology, illustrated in Fig. 1, is a model-transformation function that automatically lifts the policy that is specified in the security model to the GUI model. The idea is simple but powerful. The security model specifies under what conditions actions on data are authorized. The control information in the GUI model specifies which actions are executed in response to which events. Lifting essentially consists of prefixing each data action in the GUI model with the authorization check specified in the security model. The resulting GUI model is security aware. It specifies UI structure, information flow with persistent storage, and all authorization checks.

We have implemented this methodology within a toolkit, called ActionGUI [1], that performs this many-models-to-model transformation. From the resulting security-aware GUI model, ActionGUI generates a deployable application, along with all support for access control. In particular, when the security-aware GUI model contains only calls to execute CRUD actions, then ActionGUI will generate the complete implementation automatically.

The methodology and tool that we report on constitute a substantial further development of [2], [3]. In this previous work, we proposed the idea of using model transformations to lift the security policy, formulated in terms of the data model to the GUI model. Here we improve and generalize this previous work and we provide an updated presentation of the modeling languages, the toolkit, and the example applications that we developed.

D. Basin is with ETH Zürich, Switzerland.

M. Clavel is with IMDEA Software, Madrid, Spain, and with Universidad Complutense de Madrid, Spain.

M. Egea is with Atos Research & Innovation, Madrid, Spain.

C. Dania and M. A. García de Dios are with IMDEA Software, Madrid, Spain.

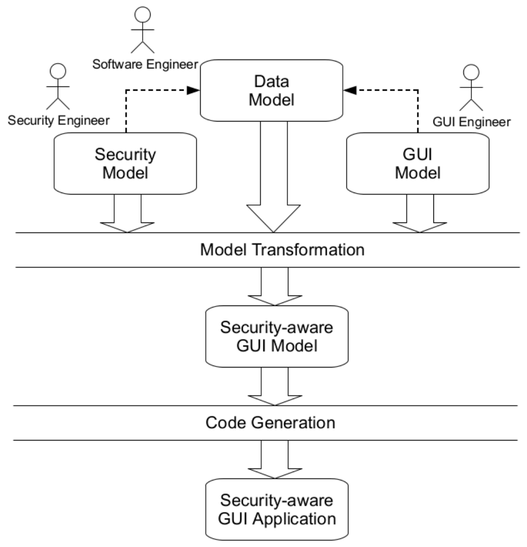


Fig. 1. Model-driven development of security-aware GUIs.

Let us expand on the main generalization with respect to our previous work. Lifting previously consisted of prefixing each event in the GUI model with the authorization check specified in the security model. However since the data actions executed by an event may change the persistence layer's state, checking authorization at the level of events, and therefore before executing any data action, is sufficient only if the underlying security policy does not contain authorization constraints (which was explicitly assumed in [3]), or if they do not depend on values that are changed during the execution of the event's data actions (as was the case in the examples discussed in [2]). To overcome this limitation, we now check authorization before executing each event's data actions and we provide events with a *transaction* semantics: either all of the data actions are executed in the given order, or none of them are executed at all.

Overall, we see our contributions as follows. First, our methodology offers Model Driven Architecture's purported benefits [4], [5] for data-management applications. By working with models, designers can focus on the application's data, behavior, security, and presentation, independent of the different, often complex, technologies that are used to implement them. Second, our use of model transformations leads to modularity and separation of concerns: the GUI model and the security model can be changed independently and by different developers, if desired. This avoids the problems mentioned earlier with fine-grained, hardcoded security policies that are difficult to maintain and audit. Finally, our methodology is quite powerful and compares favorably to alternatives, which are described in detail in Section VI on related work. In particular, it leverages well-known security languages [6] for modeling rich, fine-grained access control policies, which must often be manually encoded in other proposals. Moreover, our new language for GUIs supports modeling realistic, dynamic web interfaces (where the web content varies based on the user's actions or user-provided data), without limiting the interfaces to a fixed set of templates or interaction patterns, as in other methodologies. Of course, the proof of the pudding is in the eating and we report on applications that we developed, which provide evidence of the applicability of this approach.

Organization: The remainder of this paper is organized as follows. In Section II we present background on the existing modeling languages that we use, namely, ComponentUML and SecureUML; in Appendix A we provide additional explanation on the semantics of the latter. In Section III we introduce a new modeling language, called GUIML, for modeling graphical user interfaces together with their behavior. In Section IV we discuss our many-models-to-model transformation and in Section V we report on our tool support and on example applications that we developed using it. Finally, in Section VI we survey related work and we draw conclusions in Section VII. Due to space limitations, we omit the formal account of our methodology, which is given in the technical report [7]. Instead, we provide in Appendix B a high-level account of the correctness of the model-transformation function, which lies at our methodology's core.

II. Background

For modeling an application's data and security policy, we leverage existing modeling languages, namely, ComponentUML and SecureUML [6]. In this section we briefly introduce these languages. Since SecureUML uses the Object Constraint Language (OCL) [8] to model authorization policies, we also summarize its main features.

A. ComponentUML

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. Our methodology employs ComponentUML for data modeling. ComponentUML provides a subset of UML class models where *entities* (classes) can be related by *association* and may have *attributes* and *methods*. In ComponentUML, associations are binary: they always have two *association-ends* connecting two, not necessarily distinct, entities.

While ComponentUML and SecureUML have a graphical concrete syntax (see [6]), to simplify and clarify the presentation, we shall use textual concrete syntax. In this syntax, entities are declared with the keyword **Entity** followed by the entity's name, and its attributes and association-ends, which are enclosed within brackets. Attributes and association-ends are declared together with their types. Moreover, since associations are binary, each association-end is declared together with its opposite association-end, designated by the keyword **oppositeTo**. Multiplicities other than * and **1** are specified using OCL invariants. Finally, comments are introduced with //.

As the following example illustrates, ComponentUML models specify how the application's data is structured, independently of how it will be visualized or accessed.

Example 1: We use a simple chatroom application as a running example throughout this paper. A demo version of this application can be found at [1]. The application provides an online discussion site where users converse by posting messages. Note that there are two types of users: registered and unregistered users. Registered users have their nicknames and passwords stored in the persistence layer. As usual, some options are only available to registered users, who log into the application by entering a valid nickname and password.

Here we use ComponentUML's textual syntax to model the chatroom's data model. The model, called ChatRoomDTM, consists of three entities representing chatrooms, registered users, and messages. The associations between these entities represent the relations between the registered users and the chatrooms in which they participate, the relations between the registered users and the messages that they have written, and the relations between the messages and the chatrooms where they have been posted. The entities' attributes represent that each chatroom has a topic, each chatroom can be public or not, each registered user has a nickname and a password, and each message has a body.

```
Entity Chatroom {
  String topic
  Boolean public
  //registered users participating in this chatroom
  Set(User) participants oppositeTo chatrooms
  //messages posted in this chatroom
  Set(Message) messages oppositeTo chatroom }

Entity User {
  String nickname
  String password
  //chatrooms in which this registered user participates
  Set(Chatroom) chatrooms oppositeTo participants
  //messages written by this registered user
  Set(Message) messages oppositeTo owner }
Entity Message {
  String body
  //chatroom where this message is posted
  Chatroom chatroom oppositeTo messages
  //registered user that wrote this message
  User owner oppositeTo messages }
```

B. Object Constraint Language (OCL)

The Object Constraint Language (OCL) [8] is a language for specifying 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 expressed using graphical notation, such as class invariants in a UML class diagram. Every OCL expression is written in the context of a model (called the *contextual model*), and is evaluated on an object model (also called the *instance* or *scenario*) of the contextual model. This evaluation returns a value but does not alter the given object model, since OCL's evaluation is side-effect free.

OCL is strongly typed. Expressions either have a primitive type, a class type, a tuple type, or a collection type. OCL provides standard operators on primitive data, tuples, and collections. For example, the operator →includes checks whether an object is part of a collection. OCL also provides a dot-operator to access the values of the objects' attributes and association-ends in the given scenario. For example, suppose that the contextual model includes a class c with an attribute at and an association-end as. Then, if o is an object of the class c in the given scenario, the expression $o.at$ refers to the value of the attribute at for the object o in this scenario, and $o.as$ refers to the objects linked to the object o through the association-end as. Finally, OCL provides operators to iterate over collections, such as →forAll, →exists, →select, →reject, →collect, and →iterate.

C. SecureUML

SecureUML [6] extends Role-Based Access Control (RBAC) [9] with *authorization constraints*. These constraints can be used to specify policies that depend on properties of the system state, for example, that a user can only post a message to a chatroom where the user participates. More specifically, SecureUML allows one to formalize access control decisions that depend on two kinds of information:

1) *static information*, namely the assignments of users and permissions to roles, and the role hierarchy, and
2) *dynamic information*, namely the satisfaction of authorization constraints in the current system state.

SecureUML therefore supports the modeling of *roles* and their hierarchies, *permissions*, *actions*, *resources*, and *authorization constraints*. Moreover, one can also model assignments: which permissions are assigned to a role, which actions are allowed by a permission, which resources are affected by a permission, and which authorization constraint must be satisfied before granting a permission.

In our methodology, we use an extension of SecureUML to specify security policies over ComponentUML models.

TABLE I
SECUREUML+COMPONENTUML: ACTIONS AND RESOURCES.

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

In this extension, the protected *resources* are the entities, along with their attributes, methods, and association-ends, while the *actions* are those shown in Table I.

Note that there are two classes of actions: atomic and composite. *Atomic actions* are intended to map directly onto existing operations on the persistence layer. *Composite actions* hierarchically group lower-level actions. For example, the full access action for an attribute groups together the read and update actions for this attribute. Finally, authorization constraints are specified using OCL, where the context of an OCL expression is the underlying ComponentUML model. Additionally, OCL expressions in SecureUML models may contain the variables self, caller, value, and target, which are interpreted as follows:

- self refers to the root resource upon which the action will be performed if the permission is granted. The root resource of an attribute, a method, or an association-end is the entity to which it belongs.
- caller refers to the user 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 added (or removed) at an association-end if the permission is granted.

The reader familiar with the original presentation of SecureUML [6] may notice that we have introduced two new variables that can be used in authorization constraints: the variables value and target. Furthermore, to avoid potential ambiguities, we have refined the association-end update action into two separate actions: association-end create and association-end delete.

In our concrete syntax, the entity modeling the system's users (or, more specifically, the system's *callers*) is declared with the keyword **User**. The roles that these users can take are declared with the keyword **Role** followed by the role's name, and its permissions, which are enclosed within brackets. The keyword **inherits**, appearing between two roles, declares that the first role is subordinated to the second role in the role hierarchy, and therefore inherits all its permissions.

Permissions are introduced by naming the root resources to which they grant access. Each permission consists of a list of actions through which the corresponding root resource can be accessed. Actions on attributes, methods, or association-ends are declared along with their names. For example, **Read::***attr* denotes the read action on the attribute *attr*. The **if–then** construction is used to declare that the permission to execute an action is constrained by a condition. This condition is the authorization constraint that is associated to the permission.

As the following example illustrates, SecureUML models specify the application's access control policy in a fine-grained way. These models depend, of course, on how the application's data is structured, but not on how it is visualized or accessed through the application's graphical user interface.

Example 2: We use the SecureUML's textual syntax to model a policy for posting and reading chatroom messages. Our model, called ChatRoomSTM, has two roles: the role DefaultR represents everybody, i.e., both registered and unregistered users, and the role UserR represents only registered users. Our policy states that everybody can read any message posted in a public chatroom, but that only registered users can read messages posted in a private chatroom, provided they participate in that chatroom. Moreover, only registered users can post messages in public chatrooms; they can also post to private chatrooms, provided they also participate in that chatroom.

```
User User
Role DefaultR {
  Chatroom {
    //everybody can access the messages posted in a
    //public chatroom
    if self.public then Read::messages }
  Message {
    //everybody can read the body of any message
    //posted in a public chatroom
    if self.chatroom.public then Read::body } }
Role UserR inherits DefaultR {
  Chatroom {
    //every registered user can access the messages that
    //are posted in a chatroom in which she participates
    if self.participants→includes(caller)
      then Read::messages }
  Message {
    //every registered user can read the body of any
    //message that is posted
    //in a chatroom in which she participates
    if self.chatroom.participants→includes(caller)
      then Read::body
    //every registered user can create a new message
    Create
    //every registered user can claim ownership of any
    //unowned message
    if self.owner.oclIsUndefined() and target=caller
      then Create::owner
    //every registered user can change the body of any
    //message she owns
    //provided it is not yet posted anywhere
    if self.owner=caller
        and self.chatroom.oclIsUndefined()
      then Update::body
    //every registered user can post in a public chatroom any
    //message she owns,
    //provided it is not yet posted anywhere
    if self.owner=caller and target.public
        and self.chatroom.oclIsUndefined()
      then Create::chatroom
    //every registered user can post, in a chatroom in which
42      //she participates, any message she owns,
43      //provided it is not yet posted anywhere
44      if self.owner=caller and target.participants
45            →includes(caller)
46         and self.chatroom.oclIsUndefined()
47         then Create::chatroom } }
```

SecureUML provides various constructs for expressing complex access control policies compactly and intuitively, for example, by using action and role hierarchies or by declaring default policies. Nevertheless, as described in Appendix A, every SecureUML model S can be uniquely transformed into a semantically equivalent model $S^\flat$ for which the following holds:

Remark 1: Let S be a SecureUML model. Then, for every atomic action act and every role r in S, there is exactly one permission in $S^\flat$ (possibly constrained by false) for r to execute act.

Informally, the model $S^\flat$ makes the security policy specified in S completely explicit. Thus, let *Auth* be the function that, for every SecureUML model S, role r, and action act, returns the authorization constraint associated to the unique permission that is defined in $S^\flat$ for r to execute act. We will use this function *Auth* to define the model-transformation that, in our methodology, lifts the security policy from the security model to the GUI model. We conclude this section with some examples that illustrate in which sense *Auth* makes the security policy specified in a security model explicit.

Example 3: Consider the chatroom's security model, **ChatRoomSTM**, in Example 2. Note that UserR is a subrole of DefaultR (in line 11), which means that UserR will inherit all the DefaultR's permissions. Thus, Auth(**ChatRoomSTM**, UserR, **Read::**body) returns:

self.chatroom.public (from ln. 10)
or
self.chatroom.participants→includes(caller) (from ln. 21).

Note also that the association-end messages is opposite to the association-end owner. This means that a create (respectively delete) action on messages will be constrained by the same authorization that constrains a create (respectively delete) action on owner, having simultaneously replaced the variable self by target and the variable target by self. Thus, although no permission is explicitly given for the role UserR to execute a create action on messages, Auth(**ChatRoomSTM**, UserR, **Create::**messages) returns:

target.owner.oclIsUndefined() and self=caller (from ln. 27).

Finally, note that there is no permission explicitly given to the role DefaultR for executing an action update on the attribute body. Since permissions are denied by default (and no other rules can be applied in this case, like the ones for role inheritance or opposite association-ends) Auth(**ChatRoomSTM**, UserR, **Update::**body) returns false.

III. GUI Models

GUI models provide a human-interface oriented view of a system. Together with data models, they constitute platform independent application models, omitting security aspects.

Informally, a GUI consists of widgets, which are visual elements that display information and trigger events that execute actions. In this section we present a key component of our methodology: a novel language for modeling GUIs for data-management applications, called GUIML (GUI Modeling Language). It is important, however, to understand that GUIML is a language for modeling not only the *structure* of a GUI, i.e, the elements (widgets) that comprise it, but also the GUI's *behavior*, i.e., how its elements will react (actions) in response to user interactions with them (events). In fact, the key feature of GUIML is the language it provides for modeling the GUI's behavior, which uses OCL to specify both the conditions and the arguments for the different actions. This feature enables both the security model and the GUI model to "speak" the same language (namely, OCL in the context of the common, underlying data model). This allows us to define rigorously the transformation function that lifts the security policy to the GUI level.

We next briefly describe the main elements of GUIML, namely, *widgets* (with their associated *variables*), *events*, and *actions*. We will also illustrate them later with a simple example: a window for our chatroom application, where users can read and post messages in a chatroom.

Widgets: A GUIML model consists of widgets of different types: windows (pages, when referring to web applications), combo-boxes (selectable lists), tables, date fields, boolean fields (check boxes), buttons, text fields, and labels. Widgets can be displayed in *containers*, which are also widgets. Widgets other than windows must be contained in another widget, and only windows, combo-boxes and tables may contain other widgets. Widgets may own variables, which store values for later use, and trigger events, which execute actions.

In concrete syntax, a widget is declared with a keyword like **Window**, **Button**, and **TextField**, according to its type, followed by the widget's (local) name, and the declaration of the variables it owns, the events it triggers, and the widgets it contains, all enclosed in brackets. The *global name* of each widget must be unique. If a widget is a window, its global name is the name given in its declaration. Otherwise, the global name results from concatenating, using dot, the global name of the widget's container with the name given in its declaration.

Variables: Each widget declaration may contain variable declarations, listing the variables owned by the widget. In concrete syntax, a variable declaration consists of the variable's type followed by its name.

There are also variables that are, by default, owned by every widget of a given type. These variables are implicitly declared in every widget declaration, and their values are handled in special ways. Here we only discuss the predefined variables that we will use in our example. The variables caller and role are predefined in every window. They store, respectively, the application's user and the user's role. The variable text is predefined in every label, button, and text field. This variable stores the string displayed on the screen within the label, button, and text field; also, when a user types in a text field, the value of its variable text is automatically updated. The variable rows is predefined in every combo-box and table. This variable stores the collection of items that can be selected from the combo-box or table. The variable row is also predefined in every combo-box and table where, for each row, it stores the item that can be selected.

Events: Each widget declaration may contain event declarations. Events are triggered when specific actions are executed upon their widgets, and they themselves can execute actions either on data or on other widgets.

The actions executed when an event is triggered are specified using *statements*. A statement is either an action, a conditional statement, an iteration, or a sequence of statements. In GUIML, the conditions in both conditional statements and iterations are specified using OCL expressions, whose context is the underlying ComponentUML model. Additionally, they can refer to the widget variables. In GUIML, when widget variables are used within OCL expressions, they are enclosed in square brackets. Note that each sequence of statements associated to an event is executed as a single *transaction*: either all its statements successfully execute in the given order, or none of them are executed at all.

In concrete syntax, events are declared by indicating their types followed by the sequence of statements that they execute, enclosed in brackets. In our example we will use two types of events: **OnCreate** and **OnClick**. The former are triggered when the widgets are created and the latter are triggered when widgets are clicked upon. In particular, a window is created when an open action that has this window as its target is executed. All the other widgets are created immediately after their corresponding containers are created.

Actions: Every event declaration contains a sequence of statements that specifies the actions executed when the event is triggered. These actions can be executed either on objects belonging to the persistence layer or on objects belonging to the visualization layer. The former are called *data actions*, and the latter are called *GUI actions*. Note that some actions may take arguments whose values are only known at run-time, for example a delete action whose argument is the item selected by the user in a combo-box, or an update action whose argument is the number entered by the user in a text field. In GUIML, these values are specified using OCL. Again, the context of these expressions is the underlying ComponentUML model, but they can also refer to the widget variables.

Next, we briefly describe some of the GUIML data actions and their concrete syntax.

- **Entity create:** It creates a data item in the persistence layer. Its arguments are the *type* of the data item and the *variable* that stores the data item. It is declared by the statement *variable* **:= new** *type*.
- **Entity delete:** It deletes a data item from the persistence layer. Its argument is *object*, which is the data item deleted. It is declared by the statement **delete** *object*.
- **Attribute read:** It reads the value of a data item's attribute in the persistence layer. Its arguments are the data item *object* whose property is read, the *attribute* read, and the *variable* that stores the value read. It is declared by the statement *variable* **:=** *object***.***attribute*.
- **Attribute update:** It modifies the value of a data item's attribute in the persistence layer. Its arguments are the data item *object* whose attribute is modified, the *attribute* modified, and the new *value*. It is declared by the statement *object***.***attribute* **:=** *value*.
- **Association-end read:** It reads the collection of items linked to an item's association-end in the persistence layer. Its arguments are the data item *object* whose property is read, the association-end *assocEnd* read, and the *variable* that stores the collection read. It is declared by the statement *variable* **:=** *object***.***assocEnd*.
- **Association-end create:** It creates a link in the persistence layer between two data items. Its arguments are the source data item *srcObject*, the target data item *tgtObject*, and the association-end *assocEnd* through which the target data item is linked to the source data item. An association-end create action is declared by the statement *srcObject***.***assocEnd* **+=** *tgtObject*.
- **Association-end delete:** It deletes a link in the persistence layer between two data items. Its arguments are the source data item *srcObject*, the target data item *tgtObject*, and the association-end *assocEnd* from which the target data item is removed. It is declared by the statement *srcObject***.***assocEnd* **–=** *tgtObject*.

Finally, we describe some of the GUI actions that are defined in GUIML.

- **Set:** It updates the value of a variable. Its arguments are the name of the *variable* and variable's new *value*. It is declared by the statement *variable* **:=** *value*.
- **Open:** It opens a window. Its argument is *target*, which names the window opened. Additionally, it may take as arguments any number of pairs ($variable_i$, $value_i$), where $variable_i$ is the name of a variable owned by its *target* window, and $value_i$ is the value that is assigned to $variable_i$ when *target* is opened. It is declared by the statement **open** *target* **with** $variable_1$**:=** $value_1$ ... $variable_n$**:=** $value_n$.
- **Back:** It moves back to the previous window. It is declared using the keyword **back**.
- **Fail:** It forces a rollback in the current transaction, whereby the corresponding statement is not successfully executed. It is declared using the keyword **fail**.
- **Skip:** It does nothing. It is declared using the keyword **skip**.

We now provide an example that illustrates the main elements of the GUIML language: a window ReadPostWI for our chatroom application, where users can read and post messages in (previously selected) a chatroom. As this example will show, GUIML models depend on how the application's data is structured — after all, they describe how users interact with this data — but not on the application's access control policy.[1] In our example, this separation of concerns is reflected by the fact that the GUIML model for the window ReadPostWI is completely unaware of the security policy for reading and posting messages in our chatroom application.

Example 4: We use GUIML to model the window of our chatroom application where users can read and post messages in a chatroom. This window is named ReadPostWI. It owns a variable chatroomSel that stores a previously selected chatroom (the action that opens the window ReadPostWI will assign a value to this variable). The window ReadPostWI contains four widgets:

- a table ReadPostsTB for visualizing the messages posted in the selected chatroom;
- a text field WritePostEN for writing a new message;
- a button PostBU for posting in the selected chatroom the message written in the text field WritePostEN; and
- a button BackBU for moving back to the previous window.

The model is as follows:

```
Window ReadPostWI {
  //this variable stores the previously selected chatroom
  Chatroom chatroomSel
  //this table visualizes the messages posted
  //in the selected chatroom
  Table ReadPostsTB {
    OnCreate {
      rows := [ReadPostWI.chatroomSel].messages } }
  //in this text field the user writes its new message
  TextField WritePostEN {
    OnCreate { text := '' } }
  //by clicking on this button, the user posts its new message
  //in the selected chatroom
  Button PostBU {
    OnCreate { text := 'Post' } }
  //by clicking on this button, the user moves back to
  //the previous window
  Button BackBU {
    OnCreate { text := 'Back' } } }
//we continue with this table
Table ReadPostWI.ReadPostsTB {
  //each column of this table shows the body of a message
  //of the selected chatroom
  Label BodyPostLB {
    OnCreate {
      text := [ReadPostWI.ReadPostTB.row].body } } }
//we continue with this button
Button ReadPostWI.PostBU {
  OnClick {
    newPost := new Message
    newPost.owner += [ReadPostWI.caller]
    newPost.body := [ReadPostWI.WritePostEN.text]
    newPost.chatroom += [ReadPostWI.chatroomSel] } }
//we continue with this button
Button ReadPostWI.BackBU {
  OnClick { back } }
```

[1]Of course, in terms of the final application's *usability*, there is a dependency: an application's GUI can end up being unusable precisely because of the application's security policy.

Note that the table ReadPostTB and the buttons PostBU and BackBU are modeled partially inside the window ReadPostWI and partially outside this window. This is supported by our concrete syntax in order to improve the readability of the GUIML models. However, to avoid ambiguities, when a widget is modeled outside its widget container, the widget's global name is used. Note too that the table ReadPostTB is unaware of the security policy for visualizing messages, which in our running example states that only registered users are authorized to read messages posted in private chatrooms. Similarly, the button PostBU is unaware of the security policy for posting messages, which is that only registered users can post messages in public chatrooms and in private chatrooms but, in the latter case, they must also participate in these chatrooms.

IV. Security-aware GUI Models

In this section we describe the heart of our methodology: a model-transformation function *Sec* that, given a GUIML model G and a SecureUML model S, automatically generates a new GUIML model $\mathrm{Sec}(G, S)$. The generated model is identical to G except that it is *security aware* with respect to S. The transformation function *Sec* works by wrapping around every data action act in G an if-then-else statement with the following arguments:

- a condition that reflects the constraints associated to the permissions specified in S, for each of the different roles, to execute the action act;
- a then-branch that contains the action act; and
- an else-branch that contains the action **fail**.

Thus, the semantics of the if-then-else statement ensures that *act* will only be executed if the constraints associated to the corresponding permissions are satisfied. Moreover, this semantics also guarantees that, if these constraints are not satisfied, then the action **fail** will be executed, forcing a rollback in the current transition.

More specifically, to generate the aforementioned if-then-else statement, the function *Sec* makes use of Remark 1. In particular, for each role r in S, it calls the function $Auth(S, r, act)$ to obtain the expression that ultimately (i.e., when the security policy is made completely explicit) constrains the permission given to r for executing act. However, since this expression may contain the variables self, value, target, and caller, the function *Sec* must also replace these variables by the actual arguments of the action act (including its actual user). We denote the resulting OCL expression by $\mathrm{Auth}(S, r, act)[args]$, where $args$ are the arguments specified in the GUI model for the action act. Finally, since different roles may be constrained by different expressions, the condition generated by *Sec* will have the form:

((r_1 = [$Window$.role] and Auth(S, r_1, act)[$args$])
or . . . or
(r_n = [$Window$.role] and Auth(S, r_n, act)[$args$])),

where $r_1, \ldots, r_n$ are all the roles declared in S. (Recall that the actual application's user and its role are always stored in the variables caller and role, which are owned by every window in the GUI model.)

The following examples illustrate the model-transformation function *Sec*. As previously mentioned, the complete, formal account of our methodology, including the model-transformation function *Sec*, is given in [7]. Nevertheless, the interested reader can find in Appendix B a high-level account of the correctness of *Sec*.

Example 5: Consider line 32 in Example 4. It specifies the third action that will be executed when the button ReadPostWI.PostBU is clicked upon, namely,

newPost.body **:=** [ReadPostWI.WritePostEN.text].

Recall that **:=** refers to an update action, in this case to the action **Update::**body. The function *Sec* will replace this by the following if-then-else statement:

```
if ((DefaulR = [ReadPostWI.role] and false)
  or
  (UserR = [ReadPostWI.role] and
    ([newPost].owner = [ReadPostWI.caller]
    and [newPost].chatroom.oclIsUndefined())))
then newPost.body := [ReadPostWI.WritePostEN.text]
else fail.
```

To understand the condition generated by *Sec*, note that Auth(ChatRoomSTM, DefaultR, **Update::**body) is equal to false, but that Auth(ChatRoomSTM, UserR, **Update::**body) is equal to self.owner = caller and self.chatroom.oclIsUndefined(). Thus, the function *Sec* must replace the variable self by the newly created message newPost (since this is the object upon which the action **Update::**body will be executed), and the variable caller by ReadPostWI.caller (since this is the user that will execute the action **Update::**body).

Example 6: Consider line 31 in Example 4. It specifies the second action that will be executed when the button ReadPostWI.PostBU is clicked upon, namely,

newPost.owner **+=** [ReadPostWI.caller].

Recall that **+=** refers to an association-end create action, in this case to the action **Create::**owner. Then, the function *Sec* will replace this line by the following if-then-else statement:

```
if ((DefaulR = [ReadPostWI.role] and false)
   or
   (UserR = [ReadPostWI.role] and
   ([newPost].owner.oclIsUndefined()
     and [ReadPostWI.caller]=[ReadPostWI.caller])))
then newPost.owner += [ReadPostWI.caller]
else fail.
```

To understand the condition generated by *Sec*, note that Auth(ChatRoomSTM, DefaultR, **Create::**owner) is equal to false, but that Auth(ChatRoomSTM, UserR, **Create::**owner) is equal to self.owner.oclIsUndefined() and target=caller. Thus, the function *Sec* must replace the variable self by the newly created message newPost (since this is the object upon which the action **Create::**owner will be executed), the variable caller by ReadPostWI.caller (since this is the user that will execute the action **Create::**owner), and the variable target also by ReadPostWI.caller (since the actual user is precisely the object that will be added by the **Create::**owner as the owner of the newly created message).

Our next example illustrates how our model transformation *Sec* leads to modularity and separation of concerns whereby the GUI model and the security model can be changed independently, if desired.

Example 7: Suppose that we decide to allow anyone (not only registered users, but also unregistered ones) to post messages in public chatrooms. To update the chatroom application's security-aware GUIML model, we just carry out the following steps:

- **Step 1** Change the original chatroom's SecureUML model to reflect our security policy changes. We call the new security model PubChatRoomSTM and show below the new permissions for the role DefaultR (i.e., for everybody using the application) to create a message, update the body of a message, and post a message in a chatroom:

```
Role DefaultR {
  Message {
    //everybody can create a new message
    Create
    //everybody can change the body of
    //any unowned message
    //provided it is not yet posted anywhere
    if self.owner.oclIsUndefined()
      and self.chatroom.oclIsUndefined()
      then Update::body
    //everybody can post in a public chatroom
    //any unowned message
    //provided it is not yet posted anywhere
    if self.owner.oclIsUndefined() and target.public
      and self.chatroom.oclIsUndefined()
      then Create::chatroom } }
```

- **Step 2** Apply our model transformation to the original chatroom GUIML model and the modified chatroom SecureUML model to generate the updated security-aware GUIML model. We show below the result of this transformation for line 32 in Example 4.

```
if ((DefaultR = [ReadPostWI.role] and
    ([newPost].owner.oclIsUndefined()
      and [newPost].chatroom.oclIsUndefined()))
   or
   (UserR = [ReadPostWI.role] and
    (([newPost].owner.oclIsUndefined()
      and [newPost].chatroom.oclIsUndefined())
    or
    ([newPost].owner = [ReadPostWI.caller]
      and [newPost].chatroom.oclIsUndefined())))
then newPost.body := [ReadPostWI.WritePostEN.text]
else fail.
```

It is interesting to compare this result with the one explained in Example 5 for the case of the security model ChatRoomSTM. To understand the differences, note that Auth(ChatRoomSTM, DefaultR, **Update::**body) is equal to false, but that Auth(PubChatRoomSTM, DefaultR, **Update::**body) is equal to self.owner.oclIsUndefined() and self.chatroom.oclIsUndefined(). Also, recall that UserR inherits all permissions from DefaultR and, in particular, its new permission for updating the body of a message, which is constrained by Auth(PubChatRoomSTM, DefaultR, **Update::**body).

Note that in this example, the function *Sec* may generate conditions that can be further simplified. However, for the sake of illustration, here and elsewhere, we show the results of *Sec* without further simplification.

V. ActionGUI Toolkit and Applications

A. ActionGUI Toolkit

Security-aware GUIML models are platform independent and can be mapped to implementations employing different technologies. This includes desktop applications, web applications, and mobile applications. As part of our work, we built the ActionGUI Toolkit [1], which automatically generates web-based data-management applications from security-aware GUIML models.

The ActionGUI Toolkit features model editors for constructing and manipulating ComponentUML, SecureUML, and GUIML models. These editors share our own OCL parser, which takes as additional input the variables introduced by the different models, along with their respective types: in the case of SecureUML models, the variables self, caller, target, and value, and in the case of GUIML models, all the given widget variables. Crucially, the ActionGUI Toolkit implements our model transformation to generate security-aware GUIML models. Finally, it includes a code generator that, given a security-aware GUIML model, produces a web application based on the following, standard three-tier architecture.

1) Presentation tier (also known as front-end): Users access web applications through standard web browsers, which render the content (HTML and JavaScript) dynamically provided by the application server.
2) Application tier: The toolkit generates Java Web Applications, implemented using the Vaadin framework. The applications run in a servlet container (such as Tomcat or GlassFish), process client requests and, generate content, which is sent back to the client for rendering. They may also manipulate data stored in the persistence tier. When processing client requests, the generated application *interprets* its underlying security-aware GUIML model. In particular, it performs the required security checks before modifying any data stored in the persistence tier or sending any data to the presentation tier. This involves, of course, dynamically evaluating the OCL expressions appearing in the security-aware GUIML model.
3) Persistence tier (also known as data tier or back-end): The generated application manages information stored in a database. For each application, the toolkit generates the corresponding database schema from the application's ComponentUML model.

As a model-driven development tool, the ActionGUI Toolkit produces its best results when the data-management application's functionality can be reduced to CRUD actions and its dynamics consists of navigating and passing information through windows, and exchanging information with the underlying database. For applications in this category, ActionGUI automatically generates the complete implementation from the corresponding ComponentUML, SecureUML, and GUIML model. Note that calling CRUD actions is modeled in GUIML using data actions, and navigating and passing information through windows is modeled using GUI actions, namely, **open**, **back**, and **set**.

Of course, some data-management applications will require functionality that goes beyond CRUD actions. For example, they may need to send emails, print tables, or export data in some desired format. As expected, the ActionGUI Toolkit does not generate code for such methods. Instead, it includes their implementation — which must be provided by the application developer — in the generated application. When the application needs to interpret one of these methods, it simply calls the method provided.

B. Applications

We report here on five web applications that we developed using ActionGUI. Our objective is to show that one can use our methodology and the ActionGUI Toolkit to develop *non-toy* secure data-management applications. We begin by briefly describing our applications. In Table II we provide different measurements of the applications' size, defined in terms of their underlying GUIML models.

a) Customer Relationship Management (CRMApp): We have developed a web application for managing customers of a Hospital and Care Center. This application allows marketing and public relations personnel to manage contact information, including filtering contacts based on different criteria and exporting the results in Excel files. As customer data is highly sensitive, data is subject to a restrictive access-control policy. For example, a marketing and PR staff member can only access the contact information of those contacts previously selected as targets of a marketing campaign to which he is assigned. The application also allows a General Manager to create marketing campaigns, select the targeted patients, and assign marketing and PR staff members to campaigns.

b) Volunteer Management (VMApp): We have developed a web application for managing a care center's volunteer program. Using this application, the program's coordinators can take actions such as: introduce new volunteers; create, edit, and modify tasks; and propose these tasks to the volunteers, based on the volunteers' time availability and preferences. The access-control policy stipulates, for example, that volunteers are only authorized to edit their own personal information, such as their preferences and time availability, and to accept or reject their own tasks.

c) Meal Service Management (MSMApp): This is a web application for managing a student residence's meal service. Using this application, a resident can notify the administration whether he will have a meal at the residence's cafeteria, in which of the available time slots, and if he will bring a guest. A resident shall only edit his own meal selection and within a specific time window, which depends on the selected meal. Administrators can create new resident accounts, and list the meals requested for each available time slot.

d) EHealth Record Management (eHRMApp): This is a web application for managing eHealth records. It allows users with the appropriate roles to: register new patients in a hospital and assign to them clinicians (doctor, nurses, etc.); retrieve patient information; register new nurses and doctors in a hospital and assign them to a ward; change nurses or doctors from one ward to another; and move patients to a different practice. The access-control policy regulates, in particular, access to the patients' highly sensitive records. These records shall only be retrieved by their handling doctors, although this policy can be relaxed in an emergency situation.

e) Chatroom (ChatApp): This is an extension of our running example: in addition to posting messages in a selected chatrooms, users can also create and delete chatrooms, under specific conditions.

CRMApp, VMApp, and MSMApp are commercial applications. They were developed for actual customers, and they are currently being used by their different stakeholders. In contrast, EHRMApp was developed as part of a case study proposed by industrial partners in a European project. The interested reader can find more information about this case study, as well as demo versions of the EHRMApp and ChatApp applications, at [1]. With respect to code-generation, for MSMApp, EHRMApp, and ChatApp, the ActionGUI's code generator automatically generates 100% of their implementation (no non-CRUD actions are ever called). In contrast, CRMApp and VMApp contain custom code for sending mails and for generating Excel files, which we borrowed from existing Java libraries. For all our examples, the ActionGUI's code generator produces the corresponding applications in under a minute.

We conclude this section by summarizing the key contributions of our methodology and toolkit. Our experience developing the reported applications provides evidence of the methodology's potential for developing real-world applications. First, the use of model-transformation and code generation frees the developer from programming fine-grained authorization constraints and inserting them at all the required places throughout the application's code and with the correct arguments. Except for small applications, this is cumbersome and error-prone, since the number of data actions associated to events may be on the order of hundreds; see, for example, the applications CRMApp and VMApp in Table II. Second, our methodology supports modularity and separation of concerns. In particular, the security model can be changed independently of the GUI model, without requiring one to re-program and re-insert all the new fine-grained authorization constraints since this is automatically done by our model-transformation. This substantially aided developing our applications as our clients changed, several times, their security policies for CRMApp, VMApp, and MSMApp.

VI. Related Work

Over the past 15 years, there have been numerous research advances in the model-driven development of data management applications. Among these, UWE [10], [11], [12], [13] and ZOOM [14] are the most closely related to our work.

As a modeling tool, UWE [10], [11], [12], [13] provides the modeler with a higher-level of abstraction than ActionGUI. In particular, the actions executed by the widgets' events are described in UWE using natural language. Thus, unless the models are appropriately refined, as discussed in [13], UWE does not support code-generation. In contrast, UWE provides specific diagrams for modeling GUI *presentations* and *navigations*, which facilitate the task of GUI modeling. In this respect, we define in [15] a mapping that transforms high-level UWE models into more concrete ActionGUI models that, once completed by the modeler, can be directly used to generate the intended applications. Finally, [12] extends UWE to use SecureUML for modeling security policies. However, this work does not use a model-transformation to lift the security policy to the GUI level. Instead the UWE modeler is responsible for adding all the appropriate authorization checks to the GUI model.

Like ActionGUI, ZOOM [14] allows GUI modelers to specify widgets, their events, and their actions. Moreover, using an extension of Z [16], one can specify the conditions of the actions and their arguments, similar to how this is done in ActionGUI using OCL. In contrast to ActionGUI, ZOOM does not provide a language for modeling security and security aspects are not explicitly considered in this approach. Moreover, ZOOM does not support code-generation. It only provides interpreters for model animation.

In contrast to ActionGUI, UWE, and ZOOM, the approaches presented in [17], [18], [19] do not provide a language for modeling GUIs. They instead implement different *rules* for automatically deriving GUIs based on either the application's data model, as in [17], [18], or the application's prototypical scenarios, as in [19]. As expected, the behavior of the resulting GUIs is limited and, based on our experience, insufficient to cope with the logic embedded in real data-management applications. Moreover, security aspects are not addressed in these proposals.

There is other related work that falls between the two extremes of full GUI modeling and full GUI derivation. Both the OO-method [20], [21] and WebML [22], [23]

TABLE II
EXAMPLE APPLICATIONS: SIZE OF THE APPLICATION'S MODELS

	CRMApp	VMApp	MSMApp	eHRMApp	ChatApp
Widgets					
Number of windows	49	102	11	8	3
Number of buttons	182	293	30	18	10
Number of labels	691	697	83	66	7
Number of text fields	159	169	10	19	4
Number of boolean fields	67	9	0	5	1
Number of date fields	14	16	2	1	0
Number of combo boxes	52	33	24	1	0
Number of tables	65	85	7	9	2
Statements					
Number of if-then-else	650	334	150	35	7
Number of iterate	66	13	0	0	1
Data actions					
Number of creates (entity)	50	22	4	11	2
Number of deletes (entity)	14	33	0	0	2
Number of updates	268	180	15	25	4
Number of creates (assoc)	111	66	3	21	4
Number of deletes (assoc)	32	30	0	4	0
GUI actions					
Number of sets	1840	1553	569	120	24
Number of opens	164	234	18	7	7
OCL Expressions					
Number of expressions	3847	3221	925	331	74
Number of non-literal expressions	1478	1105	390	80	16

support building GUIs using UI-*patterns*. These patterns specify the possible interactions with the application's data based on the classes, attributes, and associations that are declared in the underlying data model. These approaches have the advantage of reducing the time required for modeling GUIs. However, the UI-patterns impose restrictions on the type of GUIs that can be modeled, both in terms of their structure and their behavior. Moreover, these approaches only support role-based access control, but not fine-grained access control. Other approaches that fall in this category are [24], [25]. In both cases, the modeler must associate to each widget container the specific data type accessed using the widget. As before, the possible interactions with the underlying data is limited by the default behavior implemented for these widget containers. Security aspects are also not considered.

Finally, there are approaches whose primary focus is to support UI design at different levels of abstraction. Prominent examples are the XML User Interface Language (XUL) [26] and the USer Interface eXtensible Markup Language (UsiXML) [27]. XUL is Mozilla's XML-based language for building user interfaces of applications like Firefox. UsiXML is an XML-compliant markup language that describes the UI for multiple usage contexts, such as Character User Interfaces (CUIs), Graphical User Interfaces (GUIs), Auditory User Interfaces, and Multimodal User Interfaces.

Clearly, ActionGUI is designed for a different purpose than XUL and UsiXML. In particular, ActionGUI is designed for developing *secure* data-management applications. A key design decision for ActionGUI was to ensure that the security model and the GUI models "speak" the same language. To the best of our knowledge, neither XUL nor UsiXML are concerned with security aspects of the UIs. Moreover, ActionGUI is designed for the *model-driven* development of secure data-management applications and this has two clear consequences. First, ActionGUI's modeling languages are designed to be technology-agnostics, in contrast with XUL, which is tightly linked to Mozilla-related technologies. Second, ActionGUI's modeling languages are designed to support the automatic generation of ready to be deployed applications from the models. As a result, ActionGUI models are more concrete than general UsiXML models, which can be defined at any of the four abstraction levels specified in the Cameleon Reference Framework (CRF) [28]. In particular, a GUIML modeler always works at the CRF-Concrete UI level, while the WAR (Web application ARchive) file generated from a GUIML model (along with the associated security and data models) belongs to the CRF-Final UI level. Also, we note that [29] has carried out promising work on extending our methodology to cope with business processes, which are typically defined at the Task & Concepts abstraction level in CRF. Along these lines, it would be interesting to investigate ways of extending our methodology to support UI modeling at the CRF-Abstract UI level, where interaction details are abstracted away.

VII. CONCLUSIONS

The methodology we proposed constitutes a further development of the idea of model-driven security [30]. The two main innovations are an expressive language for modeling an application's graphical user interface and behavior, and a many-models-to-model transformation that lifts a security policy specified on the application's data model to this behavioral model. Our transformation function captures the idea that authorization policies regulating complex transactions can be generated uniformly from much simpler policies on data. Despite our use of expressive modeling languages, we have shown for data-management applications that it is possible to generate automatically complete deployable applications.

Our methodology is supported by the ActionGUI Toolkit. Applications like those described in Section V-B show the toolkit's potential for developing real-world applications. Nevertheless, there is still much work ahead to turn this toolkit into a full, robust, industrial-strength development platform. In the short term, we plan to develop improved model editors and better support for integrating custom code. In the long term, we would like to support GUIs running on different platforms, like mobile devices. We also plan to add support for handling *privacy policies*: modeling and generating code to enforce that data usage must follow the purpose for which the data was collected and may entail obligations.

Finally, we would like to support model analysis, based on the formal semantics of our models and on the correctness of our model transformation. The following are examples of questions we would like to be able to formally answer. Will every sequence of action executed by every event in the model preserve the data model's invariants? Will authorization checks ever force a transaction roll-back? Do the conditions in the GUI model make redundant the authorization checks generated by the model transformation? Analysis support would allow us to optimize generated code and support assurance activities like system certification.

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), by the Spanish Ministry of Economy and Competitiveness Project "StrongSoft" (TIN2012-39391-C04-04), and by Comunidad de Madrid Program "PROMETIDOS-CM" (S2009TIC-1465).

References

[1] ActionGUI, "The ActionGUI project," 2013, http://www.actiongui.org.

[2] D. A. Basin, M. Clavel, M. Egea, M. A. G. de Dios, C. Dania, G. Ortiz, and J. Valdazo, "Model-driven development of security-aware GUIs for data-centric applications," in *Foundations of Security Analysis and Design VI - FOSAD Tutorial Lectures*, ser. LNCS, A. Aldini and R. Gorrieri, Eds., vol. 6858. Springer, 2011, pp. 101–124.

[3] D. A. Basin, M. Clavel, M. Egea, and M. Schläpfer, "Automatic generation of smart, security-aware GUI models," in *Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings*, ser. LNCS, F. Massacci, D. S. Wallach, and N. Zannone, Eds., vol. 5965. Springer, 2010, pp. 201–217.

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

[5] Object Management Group, "Model driven architecture guide v. 1.0.1," OMG, Tech. Rep., 2003, OMG document available at http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[6] D. Basin, J. Doser, and T. Lodderstedt, "Model driven security: From UML models to access control infrastructures." *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 1, pp. 39–91, 2006.

[7] ActionGUI, "ActionGUI semantics," IMDEA & ETH, Tech. Rep., 2013, available at http://www.actiongui.org.

[8] Object Management Group, "Object constraint language specification version 2.3.1," OMG, Tech. Rep., 2012, http://www.omg.org/spec/OCL/2.3.1.

[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*, vol. 4, no. 3, pp. 224–274, 2001.

[10] H. Baumeister, N. Koch, and L. Mandel, "Towards a UML extension for hypermedia design," in *Proc. of UML'99*, ser. LNCS, R. B. France and B. Rumpe, Eds. Springer, 1999, pp. 614–629.

[11] M. Busch and N. Koch, "MagicUWE - a case tool plugin for modeling web applications," in *Proc. of ICWE'09*, ser. LNCS, M. Gaedke, M. Grossniklaus, and O. Díaz, Eds., vol. 5648. Springer, 2009, pp. 505–508.

[12] M. Busch, "Integration of security aspects in web engineering," Master's thesis, Institut für Informatik, Ludwig-Maximilians-Universität, München, Germany, 2011.

[13] C. Kroiss, N. Koch, and A. Knapp, "UWE4JSF: A model-driven generation approach for web applications," in *Proc. of ICWE'09*, ser. LNCS, M. Gaedke, M. Grossniklaus, and O. Díaz, Eds., vol. 5648. Springer, 2009, pp. 493–496.

[14] X. Jia, A. Steele, L. Qin, H. Liu, and C. Jones, "Executable visual software modeling—the ZOOM approach," *Software Quality Control*, vol. 15, pp. 27–51, March 2007.

[15] M. Busch and M. A. G. de Dios, "ActionUWE: Transformation of UWE to ActionGUI models," 2012, http://uwe.pst.ifi.lmu.de/publications/ActionUWE.pdf.

[16] J. Woodcock and J. Davies, *Using Z: specification, refinement, and proof*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[17] A. M. R. da Cruz, "Automatic generation of user interfaces from rigorous domain and use case models," Ph.D. dissertation, Faculdade de Engenharia da Universidade do Porto, September 2010.

[18] A. M. R. da Cruz and J. P. Faria, "A metamodel-based approach for automatic user interface generation," in *Part I of Proc. of Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010.*, ser. LNCS, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds. Springer, 2010, pp. 256–270.

[19] M. Elkoutbi, I. Khriss, and R. Keller, "Automated prototyping of user interfaces based on UML scenarios," *Automated Software Engineering*, vol. 13, pp. 5–40, 2006.

[20] O. Pastor, J. Gómez, E. Insfrán, and V. Pelechano, "The OO-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming," *Information Systems*, vol. 26, no. 7, pp. 507–534, 2001.

[21] P. J. Molina, S. Meliá, and O. Pastor, "Just-UI : A user interface specification model," in *Proc. of CADUI'02*, C. Kolski and J. Vanderdonckt, Eds., 2002, pp. 63–74.

[22] S. Ceri and P. Fraternali, "The web modeling language - WebML," 2003, http://www.webml.org.

[23] Web Models Company, "Web ratio – you think, you get," 2010, http://www.webratio.com.

[24] A. Schramm, A. Preußner, M. Heinrich, and L. Vogel, "Rapid UI development for enterprise applications: Combining manual and model-driven techniques," in *Part I of Proc. of Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010.*, ser. LNCS, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds. Springer, 2010, pp. 271–285.

[25] V. Kulkarni, S. Reddy, and A. Rajbhoj, "Scaling up model driven engineering - experience and lessons learnt," in *Part II of Proc. of Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010.*, ser. LNCS, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds. Springer, 2010, pp. 331–345.

[26] Mozilla Foundation, "XML user interface language (XUL)," 2013, https://developer.mozilla.org/en-US/docs/XUL.

[27] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero, "UsiXML: A language supporting multi-path development of user interfaces," in *Engineering Human Computer Interaction and Interactive Systems, Joint Working Conferences EHCI-DSVIS 2004, Hamburg, Germany, July 11-13, 2004, Revised Selected Papers*, ser. LNCS, R. Bastide, P. A. Palanque, and J. Roth, Eds., vol. 3425. Springer, 2004, pp. 200–220.

[28] G. Calvary, J. Coutaz, L. Bouillon, M. Florins, Q. Limbourg, L. Marucci, F. Paternò, C. Santoro, N. Souchon, D. Thevenin, and J. Vanderdonckt, "The CAMELEON reference framework," 2002, http://giove.isti.cnr.it/projects/cameleon.html.

[29] J. Valdazo, "Developing secure business applications from secure BPMN models," Master's thesis, Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain, 2012.

[30] D. A. 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 2011)*, vol. 1998443. Innsbruck, Austria: New York, NY, USA, 2011, pp. 1–10.

Appendix

A. Making the Security Policy Explicit

In this appendix we define a transformation that, for every SecureUML model S, produces the SecureUML model $S^\flat$, which makes explicit the security policy declared in S. We define this transformation in four steps. Note that, as stated in Remark 1, the following holds at the end of our transformation: for every atomic action act and every role r in S, there is exactly one permission in $S^\flat$ (possibly constrained by false) for r to execute act.

Step 1: Copy the explicit permissions

- *Atomic actions.* Let act be an atomic action. Suppose that there is a permission in S for a role r to execute act under a constraint $auth$. Then, there is also a permission in $S^\flat$ for r to execute act under the same constraint $auth$.

Step 2: Unfold the security model

- *Action hierarchies.* Let CA be a composite action. Suppose that there is a permission in S for a role r to execute CA under a constraint $auth$. Then for every atomic action act contained in CA, there is a new permission in $S^\flat$ for r to execute act under the same constraint $auth$.
- *Role hierarchies.* Let act be an atomic action and let r and r' be two roles. Suppose that r is a subrole of r' in S, and that there is also a permission in S for r' to execute act under the constraint $auth$. There is then a new permission in $S^\flat$ for the role r to execute act under the same constraint $auth$.
- *Delete actions.* Let $entity$ be an entity. Suppose that there is a permission in S for a role r to delete $entity$ under a constraint $auth$. Then for every association-end $assoc$ owned by $entity$, there is a new permission in $S^\flat$ for r to execute the action $\textbf{Delete::}assoc$ under the same constraint $auth$.
- *Opposite association-ends.* Let $assoc$ and $assoc'$ be two opposite association-ends. Let act be the action $\textbf{Create::}assoc$. Suppose that there is a permission in S for a role r to execute act under the constraint $auth$. There is then a new permission in $S^\flat$ for the role r to execute $\textbf{Create::}assoc'$ under the constraint that results from simultaneously replacing in $auth$ the variable self by target and the variable target by self. Unfolding is similar when act is the action $\textbf{Delete::}assoc$.

Step 3. Add default permissions to the security model

- *Denying by default.* Let r be a role and let act be an atomic action. Suppose that there is no permission in $S^\flat$ for the role r to execute act. There is then a new permission in $S^\flat$ for the role r to execute act under the constraint false. That is, the role r will be denied access to execute act in all circumstances.

Step 4. Simplify the resulting security model

- *Disjunction of constraints.* Let r be a role and let act be an action. Suppose that there are n permissions in $S^\flat$ for the role r to execute act. These n permissions are then simplified to a single permission whose authorization constraint results from disjoining together all the authorization constraints of the n individual permissions.

B. Correctness of our Model Transformation

In this paper we have focused on our methodology and tool support for designing and generating secure data-management applications. Our approach also has a formal basis and we sketch here the correctness of our model transformation Sec, which is defined relative to the semantics of GUI models. Full details are provided in [7].

We define the semantics of GUI models by first giving a set of inference rules that defines a transition relation $\longrightarrow$ between triples of the form $\langle stm, I, \theta\rangle$, where stm is a statement, I is a scenario (i.e., an instance of the underlying data model), and θ represents a state of the widget variables. We provide inference rules for each possible statement: namely, for every type of data action and GUI action (base cases), and for arbitrary sequences of statements, conditional statements, and iterator-statements (inductive cases). In particular, for data actions, the inference rules have form

$$\frac{}{\langle act(args), I, \theta\rangle \longrightarrow \langle \textbf{skip}, \mathrm{res}(I), \mathrm{res}(\theta)\rangle}\,,$$

where:

- $args$ are the arguments of the data action act,
- $\mathrm{res}(I)$ specifies the scenario that results from executing $act(args)$ in the scenario I and widget variable state θ, and
- $\mathrm{res}(\theta)$ specifies the widget variables' new state after executing $act(args)$ in the scenario I and widget variable state θ.

Crucially, no inference rule leading to **skip** is defined for the GUI action **fail**.

We then define the *operational semantics* of an event ev that executes the actions specified by an statement stm as the set of all the transitions

$$\langle stm, I, \theta\rangle \longrightarrow^* \langle \textsf{\textbf{skip}}, I', \theta'\rangle \,,$$

where $\longrightarrow^*$ is the reflexive-transitive closure of $\longrightarrow$.

By definition, this operational semantics for events is *security-unaware*: it does not respect the authorization constraints that, according to the given security model, should constrain the execution of data actions. To provide a *security-aware* operational semantics for events, we define the *security-aware versions* of the inference rules. In particular, given a security model S, for each role r, and for each type of data action act, the security-aware version of the inference rule for act and r has the form

$$\frac{[\![(\mathrm{Auth}(S, r, act)[args])\theta]\!]^I = \mathsf{true}}{\langle act(args), I, \theta\rangle \longrightarrow \langle \textsf{\textbf{skip}}, \mathrm{res}(I), \mathrm{res}(\theta)\rangle}\,,$$

where:

- $[\![expr]\!]^I$ denotes the value of the expression $expr$ in the scenario I; and, therefore,
- $[\![(\mathrm{Auth}(S, r, act)[args])\theta]\!]^I$ denotes the evaluation in the scenario I of the authorization $\mathrm{Auth}(S, r, act)$ that constrains the only permission that, according to Remark 1, ultimately allows users with the role r to execute the action act, given that arg are the arguments of act and θ is the state of the widget variables.

These security-aware inference rules define the transition relation $\longrightarrow_S$. Finally, given a security model S, we define the *security-aware operational semantics* of an event ev that executes the actions specified by an statement stm as the set of all the transitions

$$\langle stm, I, \theta\rangle \longrightarrow_S^* \langle \textsf{\textbf{skip}}, I', \theta'\rangle \,.$$

The theorem below formalizes the *correctness* of our model-transformation function Sec. It states that evaluating a statement transformed by Sec following the security-unaware operational semantics returns the same result as evaluating the original statement using the *security-aware* semantics. Hence the transformed statement respects the authorization constraints formalized in the underlying security model.

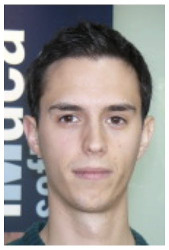
Theorem Let S be a security model and let stm be a statement. Then, for every scenario I, and every widget variable state θ,

$$\langle \mathrm{Sec}(stm, S), I, \theta\rangle \longrightarrow^* \langle \textsf{\textbf{skip}}, I', \theta'\rangle \Longleftrightarrow \\ \langle stm, I, \theta\rangle \longrightarrow_S^* \langle \textsf{\textbf{skip}}, I', \theta'\rangle.$$

David Basin is a full professor and has the chair for Information Security at the Department of Computer Science, ETH Zurich since 2003. From 2003–2011 he was founding director of the ZISC, the Zurich Information Security Center. He received his Ph.D. from Cornell University in 1989, and his Habilitation from the University of Saarbrücken in 1996. His research focuses on information security, in particular methods and tools for modeling, building, and validating secure and reliable systems.

Manuel Clavel received his bachelor's degree in Philosophy from the Universidad de Navarra in 1992, and his Ph.D from the same university in 1998. Currently, he is Associate Research Professor at the IMDEA Software Institute and Associate Professor at the Universidad Complutense de Madrid. His research focuses on rigorous, tool-supported model-driven software development, including: modeling languages, model transformation, model quality assurance, and code-generation.

Marina Egea is a Senior Security Consultant at Atos Research & Innovation, based in Madrid, since September 2011. She received her PhD in Computer Science at Universidad Complutense de Madrid in 2008. From 2008–2011 she was a post doctoral researcher, first at the Information Security Group at ETH Zurich, and later at IMDEA Software Institute. Her current research interests include secure systems development and assurance of security and privacy properties of Cloud services.



Miguel A. García de Dios is a Ph.D student at IMDEA Software in the Modeling Group. He received his bachelor and master degrees from the Universidad Complutense de Madrid in 2007. His research focuses on rigorous, tool-supported model-driven software development, including: modeling languages, model transformation, model quality assurance, and code-generation.

Carolina Dania is a Ph.D student at IMDEA Software in the Modeling Group. She received her bachelor's degree from the Universidad Nacional de Córdoba, Argentina, and her master's degree from the Universidad Complutense de Madrid, Spain. Her research interests include software engineering, formal methods and security. In particular, she is working in tools and techniques for modeling, building and validating secure and reliable software systems.