Model-driven Development of a Secure eHealth Application

Miguel A. García de Dios[1], Carolina Dania[1], David Basin[2], and Manuel Clavel[1]

[1] IMDEA Software Institute, Madrid, Spain
[miguelangel.garcia,carolina.dania,manuel.clavel]@imdea.org
[2] ETH Zürich, Switzerland
basin@inf.ethz.ch

Abstract. We report on our use of ActionGUI to develop a secure eHealth application based on the NESSoS eHealth case study. ActionGUI is a novel model-driven methodology with an associated tool for developing secure data-management applications with three distinguishing features. First, it enables a model-based separation of concerns, where behavior and security are modeled individually and subsequently combined. Second, it supports model-based quality assurance checks, where the properties proven about the models transfer to the generated applications. Finally, for data-management applications, the ActionGUI tool automatically generates complete, ready-to-deploy, security-aware, web applications. We explain these features in the context of the eHealth application.

1 Introduction

In [3] we proposed a novel methodology, called ActionGUI, for the model-driven development of secure data-management applications. This methodology enables a model-based separation of concerns, where an application's behavior and security are modeled individually and subsequently combined. Moreover, it supports model-based quality assurance checks, where relevant properties may be proven about the combined models. These properties then transfer to the automatically generated data-management applications.

We report here on our use of ActionGUI to develop a secure data-management application. This application is based on a case study proposed within NESSoS, the European Network of Excellence on Engineering Secure Future Internet Software Services and Systems [12]. The eHealth case study consists of a web-based system for electronic health record management (EHRM). Electronic health records (EHR) record information created by, or on behalf of, a health professional in the context of the care of a patient.

Electronic health records are highly sensitive and therefore their access must be controlled. Part of the challenge in this case study was to model the access control policy and build an application that enforces it at runtime. The policy consists of various authorization rules along the lines of: *The access control criteria for an EHR depends, among others, on the type of EHR. For instance,*

a highly sensitive record might be only available to the patient's treating doctor (and perhaps a few others, in rare situations). Such rules necessitate fine-grained access control, where access control decisions 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.

We show how ActionGUI's modeling languages can be used to specify the application's data model (e.g., hospital staff, health records), security policy (e.g., rules like the above) and behavior. Moreover, by illustrative examples, we highlight various features the ActionGUI methodology and associated tool. Overall, the eHealth case study is interesting as an example of developing a secure data-management application and it provides a proof-of-concept for the application of the ActionGUI methodology to an industry-relevant problem.

Organization. In Section 2 we provide background on the ActionGUI methodology and tool. In Section 3 we give an account of our modeling and generation of the EHRM application with ActionGUI. In Section 4 we describe a proof method for checking that the behavior of the modeled data-management application respects the invariants of the application's underlying data model, and we apply it to our EHRM models. Finally, in Section 5, we draw conclusions.

2 ActionGUI

ActionGUI [3] is 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 this 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 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 Figure 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.

The ActionGUI methodology is implemented within a toolkit, also called ActionGUI [1], which performs the aforementioned many-models-to-model transformation. From the resulting security-aware GUI model, ActionGUI generates

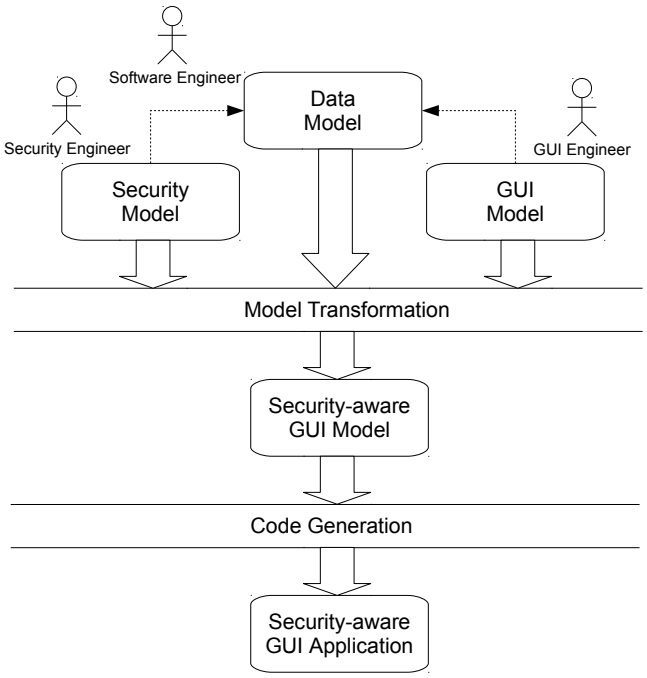

Fig. 1. Model-driven development of security-aware GUIs.

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, i.e., those actions that create, read, update, and delete data, then ActionGUI will generate the complete implementation automatically.

In the remaining part of this section we briefly introduce the languages that are used within the ActionGUI methodology to model the applications' data, security, and GUI models, including their constraints, as well as the tools supporting the ActionGUI methodology.

2.1 Data models

Data models provide a data-oriented view of a system. They typically specify how data is structured, the format of data items, and their logical organization, i.e., how data items are grouped and related. ActionGUI employs ComponentUML [4] for data modeling. ComponentUML provides a subset of UML class models where *entities* (classes) can be related by *associations* and may have *attributes* and *methods*.

2.2 Constraints

The Object Constraint Language (OCL) [13] is a language for specifying constraints and queries using a textual notation. ActionGUI supports different uses of OCL: it is used in data models to specify *data invariants*, in security models to specify *authorization constraints*, and in GUI models to specify if-then-else *conditions* and action *arguments*.

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. The contextual model for the OCL expressions in ActionGUI models is always the underlying data model.

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; a dot-operator to access the values of the objects' attributes and association-ends in the given scenario; and operators to iterate over collections. Particularly relevant for its use in ActionGUI models, OCL includes two constants, null and invalid, to represent undefinedness. Intuitively, null represents unknown or undefined values, whereas invalid represents error and exceptions. To check if a value is null or invalid, OCL provides, respectively, the Boolean operators oclIsUndefined() and oclIsInvalid().

2.3 Security models

SecureUML [4] extends Role-Based Access Control (RBAC) [9] with *authorization constraints*. These constraints are used to specify policies that depend on properties of the system state. SecureUML supports the modeling of *roles* and their hierarchies, *permissions*, *actions*, *resources*, and *authorization constraints*.

In ActionGUI, we use an extension of SecureUML for specifying security policies over data models. In this extension:

- The protected *resources* are the entities, along with their attributes, methods, and association-ends.
- The controlled *actions* are: to create and delete entities; to read and update attributes; to read, create, and delete association-ends; and to execute methods.
- The authorization constraints are specified using OCL.

The contextual model of the authorization constraints is the underlying data model. Additionally, authorization constraints 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 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 to (or removed from) the (root) resource at an association-end if the permission is granted.

2.4 GUI models

GUI models provide a human-interface oriented view of a system. A GUI consists of widgets, which are visual elements that display information and trigger events that execute actions. In ActionGUI, we use GUIML [3] for modeling both

– the GUI's *structure*, i.e, the elements (*widgets*) that comprise it,
– and the GUI's *behavior*, i.e., how its elements will react (*actions*) in response to user interactions with them (*events*).

Behavioral modeling is a key feature of GUIML and uses OCL to specify both the conditions and the arguments for the different actions. This 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*.

Widgets. A GUI model consists of widgets of different kinds. Examples include windows (pages, when referring to web applications), combo-boxes (selectable lists), tables, date fields, boolean fields (check boxes), buttons, text fields, and labels.

Variables. Widgets may own *variables*, which store values for later use. Each widget declaration may contain variable declarations, listing the variables owned by the widget. There are variables that are, by default, owned by every widget of a given type. In particular, 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. 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 corresponds to this row. Finally, the variable selected is also predefined in every combo box or table where it stores the item(s) selected in the combo box or table.

Events. Widget may trigger events, which 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, a try-catch, or a sequence of statements. The conditions in conditional statements are specified using OCL expressions, whose context is the underlying data model. Additionally, they can refer to the widget variables. Note that each sequence of statements associated to an event is executed as a single *transaction*: either all statements in the sequence successfully execute in the given order, or none of them are executed at all.

Actions. Events trigger actions that can be executed either on objects belonging to the persistence tier or on objects belonging to the presentation tier. The former are called *data actions* and the latter are called *GUI actions*. Data actions are precisely those controlled in the security model, namely: to create and delete entities; to read and update attributes; to read, create, and delete association-ends; and to execute methods. GUI actions include a set action for widget variables, and the actions **open**, **back**, **fail**, and **skip**. The action **fail**, in particular, forces a rollback of the current transaction. Note that some actions may take arguments. The values of these arguments are specified using OCL expressions, whose context is the underlying data model, and they can also refer to the widget variables.

2.5 Security-aware GUI models

The heart of ActionGUI is 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 an if-then-else statement ensures that *act* will only be executed if the constraints associated to the corresponding permissions are satisfied. Moreover, if these constraints are not satisfied, then the action **fail** will be executed, forcing a rollback in the current transaction.

2.6 Tool support

Security-aware GUI models are platform independent and can be mapped to implementations employing different technologies. This includes desktop applications, web applications, and mobile applications. The ActionGUI Toolkit [1], 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. Crucially, the

ActionGUI Toolkit implements our model transformation to generate security-aware GUIML models. Moreover, it includes a code generator that, given a security-aware GUIML model, produces a web application based on the following 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.
3. Persistence tier (also known as data tier or back-end): The generated application manages information stored in a database.

3 The EHRM ActionGUI Application

The NESSoS EHRM application scenario defines different system use cases along with the associated access control policy. The use cases include: register new patients in a hospital and assign them to clinicians, such as nurses or doctors; 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 reassign patients to doctors. Due to space limitations, we will not describe how we model all of these use cases. We focus instead on a representative use case as a running example: reassigning patients to doctors. We will use this example to illustrate ActionGUI's modeling languages as well as the model-based separation of concerns supported by the ActionGUI methodology.

3.1 The EHRM's data model

The full data model for the EHRM application contains 18 entities, 40 attributes, and 48 association-ends. We discuss below just the entities, attributes, and association-ends that are required for our running example.

Figure 2 presents this data model, formalized using ActionGUI's textual syntax. As this example shows, ActionGUI data models specify how the application's data is structured, independently of how it will be visualized or accessed.

Professional. This entity represents the EHRM's users. The role assigned to each user is specified by its role attribute. The roles considered are DIRECTOR, ADMINISTRATOR, DOCTOR, NURSE, and SYSTEM. The medical centers where a user works are linked to the user through the association-end worksIn. If a user is a doctor, then it is linked to the corresponding doctor information through the association-end asDoctor. Similarly, if a user is an administrative staff, then it is linked to staff information through the association-end asAdministrative.

MedicalCenter. This entity represents medical centers. The departments belonging to a medical center are linked to the medical center through the association-end departments. The professionals working for a medical center are linked to the medical center through the association-end employees.

Doctor. This entity represents doctor information. Doctor information is linked to the corresponding professional through the association-end doctorProfessional. The departments where a doctor works are linked to the doctor's information through the association-end doctorDepartments. The patients treated by a doctor are linked to the doctor's information through the association-end doctorPatients.

Administrative. This entity represents administrative staff information. Administrative staff information is linked to the corresponding professional through the association-end administrativeProfessional.

Department. This entity represents departments. The medical center to which a department belongs is linked to the department through the association-end belongsTo. The doctors working in a department are linked to the department through the association-end doctors. The patients treated in a department are linked to the department through the association-end patients.

Patient. This entity represents patients. The doctor treating a patient is linked to the patient through the association-end doctor. The department where a patient is treated is linked to the patient through the association-end department.

3.2 The EHRM data model's invariants

The full EHRM application data model is constrained by 66 data invariants, formalized using OCL. The following three invariants are representative.

1. *Each patient is treated by a doctor.*
   Patient.allInstances()→forAll(p|not(p.doctor.oclIsUndefined()))
2. *Each patient is treated in a department.*
   Patient.allInstances()→forAll(p|not(p.department.oclIsUndefined()))
3. *Each patient is treated by a doctor who works for a set of departments, including the department where the patient is treated.*
   Patient.allInstances()→forAll(p| p.doctor.doctorDepartments→includes(p.department))

These invariants make precise the intended meaning of the associations between the entities Patient, Doctor, and Department. The first two invariants state that the doctor and the department associated to a patient cannot be undefined, i.e., *null*. The third invariant states that a doctor who treats a patient must work in the department where the patient is treated, although the doctor may also work in other departments.

```
entity Professional {
  Role role
  Set(MedicalCenter) worksIn oppositeTo employees
  Doctor asDoctor oppositeTo doctorProfessional
  Administrative asAdministrative oppositeTo administrativeProfessional  }
entity MedicalCenter {
  Set(Department) departments oppositeTo belongsTo
  Set(Professional) employees oppositeTo worksIn  }
entity Doctor {
  Professional doctorProfessional oppositeTo asDoctor
  Set(Department) doctorDepartments oppositeTo doctors
  Set(Patient) doctorPatients oppositeTo doctor  }
entity Administrative {
  Professional administrativeProfessional oppositeTo asAdministrative  }
entity Department {
  MedicalCenter belongsTo oppositeTo departments
  Set(Doctor) doctors oppositeTo doctorDepartments
  Set(Patient) patients oppositeTo department  }
entity Patient {
  Doctor doctor oppositeTo doctorPatients
  Department department oppositeTo patients  }
enum Role { DIRECTOR ADMINISTRATOR DOCTOR NURSE SYSTEM }
```

Fig. 2. The eHRMApp's data model (partial).

3.3 The EHRM's security model

Electronic health records are by their nature highly sensitive and the NESSoS case study informally defines the policy that regulates their access. As expected, the authorization to carry out certain actions is not only role-based, but also context-based. In other words, the EHRM access control policy is *fine grained*.

The full EHRM application's security model contains 5 roles and 573 permissions, where each permission authorizes users in a role to execute an action upon the satisfaction of an authorization constraint formalized in OCL. In Figure 3 we present examples of two permissions, modeled using ActionGUI's textual syntax. The first permission authorizes a user (*caller*) with the role ADMINISTRATOR to reassign a patient to a department (*value*) provided that the user works in a set of medical centers that includes the one to which the department belongs where the patient will be reassigned. The second permission authorizes a user (*caller*) with the role ADMINISTRATOR to reassign a patient (*self*) to a doctor (*value*) provided two conditions are satisfied: (i) among the medical centers where the user works, there is at least one where the doctor to which the patient will be reassigned also works; and (ii) the user works in medical centers that includes the center to which the department belongs where the patient is currently being treated. Note that no other role has permissions associated to the actions of reassigning a patient to a department or to a doctor.

As this example illustrates, ActionGUI security models are formulated in terms of the application's data. This formalization is independent of how the data is visualized or accessed through the application's graphical user interface.

```
1  role ADMINISTRATOR {
2    Patient{
3      update (department) constrainedBy [caller.worksIn→includes(value.belongsTo)] }
4    Patient{
5      update (doctor) constrainedBy
6        [caller.worksIn→exists(m | value.doctorProfessional.worksIn→includes(m))
7          and caller.worksIn→includes(self.department.belongsTo)] }
```

Fig. 3. Examples of the EHRM security model's permissions.

3.4 The EHRM's GUI model

The full EHRM application's GUI model contains 8 windows for the following use cases: login to the application; access a medical center's information; register a new patient; review a patient's information; reassign a patient to a doctor and department; access options reserved for the medical center's director; introduce a professional into the system; and reassign a professional to a department.[3]

We discuss below the window relevant for our running example: the window movePatientWI for reassigning a patient to a doctor and a department. Figures 4 and 5 present our model of this window, in ActionGUI's textual syntax. Figure 6 contains a screenshot of the actual window generated from this model.

The window movePatientWI assumes that both a medical center and a patient have previously been selected. This information is stored, respectively, in the variables medicalCenter and patient (lines 2-3). The window movePatientWI contains the following widgets:

- A label patientLa that displays the name and surname of the selected patient (lines 5–7).
- A label departmentLa that displays the name of the department where the selected patient is treated (lines 8–9).
- A label doctorLa that displays the name and surname of the doctor who treats the selected patient (lines 10–13).
- A label departmentsLa that displays a message inviting the user to select a department (lines 14–15).

[3] Here are some other concrete figures about the size of the GUI model: i) Widgets: 19 buttons; 73 labels; 19 text fields; 5 boolean fields; 1 date field; 1 combo box; and 9 tables; ii) Statements: 34 if-then-else statements; iii) Data actions: 11 create actions; 41 update actions; 5 add link actions; and 2 remove link actions; iv) GUI actions: 157 set actions; and 7 open actions; v) OCL expressions: 361 expressions (77 non-literals).

- A label doctorsLa that displays a message inviting the user to select a doctor (lines 16–17).
- A table departmentsTa that displays information about the departments that belong to the selected medical center (line 22); in particular, the name of each of these departments is shown (line 31-34). Also, when the user selects a department from this list, it refreshes the list of doctors displayed in the table doctorsTa (see below) with the doctors who work for the selected department (lines 19–21).
- A table doctorsTA that is initially empty (line 24). As previously explained, upon selection of a department in the table departmentsTa, it displays information about the doctors who work for the selected department (lines 19–21); in particular, the name and surname of each of these doctors are shown (lines 35-41).
- A button moveBu that, when clicked upon, if there is a department selected in the table departmentsTa (line 44), and there is also a doctor selected in the table doctorsTa (line 45), then:
  - it reassigns the selected department to the selected patient (line 46);
  - it reassigns the selected doctor to the selected patient (line 47);
  - it notifies the user that the reassignment succeeded (line 48).

  Otherwise, it notifies the user that either a doctor (line 50) or a department (line 52) must first be selected.
- A button backBU that, when the user clicks on it, it returns to the previous window (line 55).

As this example illustrates, ActionGUI GUI 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 security policy. Of course, in terms of the final application's *usability*, there is a dependency: a GUI can end up being unusable precisely because of the application's security policy.

3.5 The EHRM's security-aware GUI model

As explained in Section 2.5, the heart of ActionGUI is a model-transformation function that, essentially, prefixes each data action in the GUI model with the authorization check specified in the security model. The full EHRM application's GUI model contains 59 data actions, and therefore the automatically generated EHRM application's security-aware GUI model contains the same number of authorization checks.

To illustrate our model-transformation function, we show in Figure 7 the part of the security-aware GUI model for the button moveBu's event **onClick** that is relevant for our running example. The action of reassigning the selected patient to the department selected in the table departmentsTa (line 46 in Figure 5) is now wrapped by an if-then-else statement (lines 46.1-46.5 in Figure 7) whose condition reflects the permission for executing this action given by line 3 in Figure 3. Similarly, the action of reassigning the selected patient to the doctor selected in the table doctorsTa (line 47 in Figure 5) is wrapped by an if-then-else

```
1  Window movePatientWi {
2    MedicalCenter medicalCenter
3    Patient patient
4    String text := ['Move a patient']

5  Label patientLa {
6    String text := ['Patient: '.concat($movePatientWi.patient$.contact.name)
7                    .concat(' ').concat($movePatientWi.patient$.contact.surname)] }
8  Label departmentLa {
9    String text := ['Department: '.concat($movePatientWi.patient$.department.name)] }

10 Label doctorLa {
11   String text := ['Doctor: '.concat($movePatientWi.patient$.doctor.
12                   doctorProfessional.name).concat(' ').
13                   concat($movePatientWi.patient$.doctor.doctorProfessional.surname)] }

14 Label departmentsLa {
15   String text := ['Select the new department:'] }

16 Label doctorsLa {
17   String text := ['Select the new doctor:'] }

18 Table departmentsTa {
19   Department selected {
20     if [not $selected$.oclIsUndefined()] {
21       movePatientWi.doctorsTa.rows := [$selected$.doctors] } }
22   Set(Department) rows := [$movePatientWi.medicalCenter$.departments] }

23 Table doctorsTa {
24   Set(Doctor) rows := [Doctor.allInstances()→select(false)]
25   Doctor selected }

26 Button moveBu {
27   String text := ['Move the patient'] }

28 Button backBu {
29   String text := ['Back']
30 }
```

Fig. 4. A window for reassigning a selected patient (part I)

```
31 Table movePatientWi.departmentsTa {
32   columns{
33           ['Department'] : Label department {
34                               String text := [$departmentsTa.row$.name] } } }

35 Table movePatientWi.doctorsTa {
36   columns {
37           ['Doctor'] : Label doctor {
38                          String text :=
39                            [$doctorsTa.row$.doctorProfessional.name
40                              .concat(' ')
41                              .concat($doctorsTa.row$.doctorProfessional.surname)] } } }

42 Button movePatientWi.moveBu {
43   event onClick {
44         if [not $departmentsTa.selected$.oclIsUndefined()] {
45           if[not $doctorsTa.selected$.oclIsUndefined()] {
46            [$movePatientWi.patient$.department] := [$departmentsTa.selected$]
47            [$movePatientWi.patient$.doctor] := [$doctorsTa.selected$]
48            notification(['Success'],['The patient has been reassigned.'],[0]) }
49           else {
50            notification(['Error'],['Please, select first a doctor.'],[0]) } }
51         else {
52           notification(['Error'],['Please, select first a department.'],[0]) } } } }

53 Button movePatientWi.backBu {
54   event onClick {
55         back } }
```

Fig. 5. A window for reassigning a selected patient (part II)

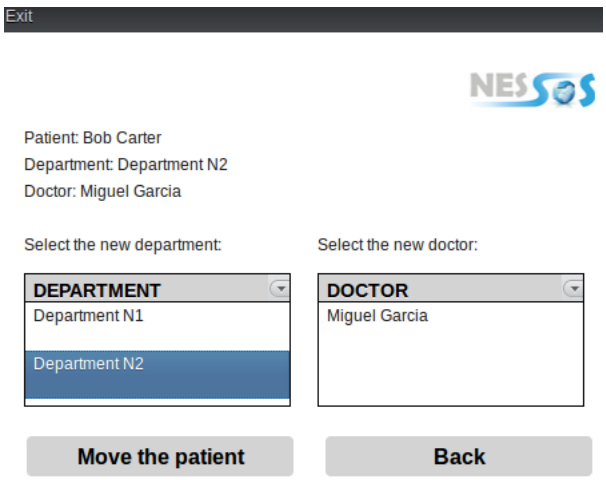

Fig. 6. Screenshot of the window for reassigning a selected patient

statement (lines 47.1-47.7 in Figure 7) whose condition reflects the permission for executing this action given by lines 5–7 in Figure 3.

```
46.1  if [[$movePatientWi.role$ = ADMINISTRATOR
46.2        and $movePatientWi.caller$.worksIn
46.3              →includes($departmentsTa.selected$.belongsTo)] {
46.4      [$movePatientWi.patient$.department] := [$departmentsTa.selected$] }
46.5      else { fail }

47.1  if [[$movePatientWi.role$ = ADMINISTRATOR
47.2        and $movePatientWi.caller$.worksIn→exists( m |
47.3              $doctorsTa.selected$.doctorProfessional.worksIn→includes(m))
47.4            and $movePatientWi.caller$.worksIn
47.5              →includes($movePatientWi.patient$.department.belongsTo))]  {
47.6      [$movePatientWi.patient$.doctor] := [$doctorsTa.selected$] }
47.7      else { fail }
```

Fig. 7. The security-aware actions for reassigning a selected patient

3.6 Generating the EHRM application

The ActionGUI Toolkit automatically generates the complete EHRM application in under 10 seconds. The generated .war file includes the Vaadin library as well as other external libraries. The Vaadin library is responsible of 70% of the size of the generated file and only 10% of this file corresponds to the code that ActionGUI automatically generates to interpret the application's model. The size of the .war file containing the complete application is roughly 15 MB.

4 Analyzing the EHRM ActionGUI Application

Model-Driven Architecture supports the development of complex software systems by generating software from models. Of course, the quality of the generated code depends on the quality of the source models. If the models do not properly specify the system's intended behavior, one should not expect the generated system to do so either. *Quod natura non dat, Salmantica non praestat.*[4] Experience shows that even when using powerful, high-level modeling languages, it is easy to make logical errors and omissions. It is critical not only that the modeling language has a well-defined semantics, so one can know what one is doing, but also that there is tool support for analyzing the modeled systems' properties.

In this section we explain how we can reason about an important property of ActionGUI models, called *data invariant preservation*. We use the EHRM application for illustration.

[4] Less elegantly said, *garbage in, garbage out.*

4.1 Data invariant preservation

We first introduce some terminology. Recall that the actions triggered by an event may be specified using if-then-else statements. At execution time, the exact sequence of actions taken is determined by how the different conditions of each if-then-else statements are evaluated in the system's state at the time of evaluation. Note that this state includes both the state of the persistence layer and the state of the GUI, in particular, its widget variables. Since each action may update the system's state, a sequence of actions gives rise to a sequence of states, which we call an *execution path*.

ActionGUI's data model may include *data invariants*. We have given several examples of these in Section 3.2. These are properties that are *required* to be satisfied in every (reachable) system state. Invariance of a property must be *proven* and the standard way to do this is to show that the property is inductive, that is, it is satisfied in the system's initial state and, whenever it satisfied in a state, it is satisfied in all possible successor states. Below we shall focus on the inductive step: proving invariant preservation.

Formally, let Φ be a collection of data invariants. An event *preserves a data invariant* $\phi \in \Phi$ if and only if for every execution path triggered by the event, if every data invariant $\psi \in \Phi$ is satisfied at the initial state of the execution path, then ϕ is also satisfied at the final state. Here we leverage ActionGUI's transaction semantics and that transactions are implemented in a way that ensures their atomicity: The intermediate states of an execution path may be considered to be internal and may therefore (temporarily) violate ψ. An event is *Φ-data invariant preserving* when it preserves all data invariants in Φ.

Our proof procedure, illustrated below, is based on the fact that each event defines a *action tree*. The nodes in this tree are the actions triggered by the event and branching corresponds to the if-then-else conditions governing the execution of these actions. As expected, every successful transaction corresponds to executing a sequence of actions given by one of the branches of the action tree, from the root to a leaf. Note that, to simplify our exposition we omit both iteration statements and event-triggering actions; including these would lead to action graphs rather than trees.

Reassigning doctors and departments to patients We show in Figure 9 the action tree defined by the the button moveBu's event **onClick**. For ease of later reference, we assign labels for the actions and the if-then-else conditions. Note that:

- Branch 1 corresponds to the case when a department and a doctor are both selected when the button moveBU is clicked-on. In this situation, the patient will be first assigned to the selected department, and then to the selected doctor; finally, a message confirming these actions will be displayed.
- Branch 2 corresponds to the case when a department is not selected when the button moveBU is clicked-on. In this situation, a message stating that a department must be first selected will be displayed.

1	*Each patient is treated by a doctor.* Patient.allInstances()→forAll(p|not(p.doctor.oclIsUndefined()))
2	*Each patient is treated in a department.* Patient.allInstances()→forAll(p|not(p.department.oclIsUndefined()))
3	*Each patient is treated by a doctor who works for a set of departments that includes the department where the patient is treated.* Patient.allInstances()→forAll(p| p.doctor.doctorDepartments→includes(p.department))

Fig. 8. Examples of the EHRM data model's invariants.

– Branch 3 corresponds to the case when a department is selected, but a doctor is not, when the button moveBU is clicked-on. In this situation, a message stating that a doctor must first be selected will be displayed.

Next, we use this action tree to reason about whether the button moveBu's event **onClick** preserves the data invariants 1–3.

Branch 1: Data invariants 1 and 2. Recall that these data invariants state that every patient is assigned to exactly one doctor and one department. Observe that the initial state in every successful transaction in this branch will satisfy the conditions *a_dept_is_selected* and *a_doctor_is_selected*. Therefore the arguments of the actions *assign_dept* and *assign_doctor* will necessarily not be null when these actions are called. Thus, the conditions *a_dept_is_selected* and *a_doctor_is_selected*, together with the postconditions of the actions *assign_dept* and *assign_doctor*, guarantee that every successful transaction in this branch preserves the data invariants 1 and 2.

Branch 1: Data invariant 3. Recall that this data invariant states that every patient is assigned to a department where its doctor works. Interestingly, there is no guarantee that every successful transaction in this branch preserves the data invariant 3. This is because the doctors shown in the table doctorsTa are those belonging to the selected department at the time of this selection (line 19–21 in Figure 4); however, there is no guarantee that, by the time the user clicks on the button moveBu, this relationship still holds for the selected doctor.

To guarantee that data invariant 3 is preserved by every successful transactions in this branch, we can simply enclose the sequence of actions *assing_dept*, *assig_dept*, and *notifiy_reassignment* (lines 46-54 in Figure 5) within an (additional) if-then-else with the following condition:

$departmentsTa.selected$.doctors→includes($doctorsTa.selected$).

Branch 2 and 3. Since these branches do not contain any data actions, every successful transaction in these branches will trivially preserve all the data model's invariants.

Actions

assign_dept	= [$movePatientWi.patient$.department][$departmentsTa.selected$]
assign_doctor	= [$movePatientWi.patient$.doctor] := [$doctorsTa.selected$]
notify_reassign	= notification(['Success'],['The patient is reassigned.'],[0])
error_select_doctor	= notification(['Error'],['Select first a doctor.'],[0])
error_select_dept	= notification(['Error'],['Select first a department.'],[0])

If-then-else conditions

a_dept_is_selected	= not $departmentsTa.selected$.oclIsUndefined()
a_doctor_is_selected	= not $doctorsTA.selected$.oclIsUndefined()

Branch 1

a_dept_is_selected = true ∧ *a_doctor_is_selected* = true	
nodes	actions
1	*assign_dept*
2	*assign_doctor*
3	*notify_reassignment*

Branch 2

a_dept_is_selected = false	
nodes	actions
1	*error_select_dept*

Branch 3

a_dept_is_selected = true ∧ *a_doctor_is_selected* = false	
nodes	actions
1	*error_select_doctor*

Fig. 9. Action tree for the button moveBU's **onClick**.

We conclude this section by summarizing in Figure 10(a) our analysis of data invariant preservation for the button moveBu's event **onClick**. For the sake of illustration, we also consider in Figures 10(b) and 10(c) data invariant preservation for two modified versions of the button moveBu's event **onClick**. In the first case, we have removed the innermost if-then-else, i.e., the one whose condition checks that a doctor has been selected. In the second case, we have removed the outermost if-then-else, i.e., the one whose condition checks that a department has been selected. As expected, if we remove the innermost if-then-else, there is no guarantee that data invariant 1, i.e., that every patient is assigned to exactly one doctor, will be preserved. Similarly, if we remove the outermost if-then-else, there is no guarantee that data invariant 2, i.e., that every patient is assigned to exactly one department, will be preserved.

	Schemas		
Invs.	1	2	3
1	✓	✓	✓
2	✓	✓	✓
3	✗	✓	✓

(a) Original

	Schemas	
Invs.	1	2
1	✗	✓
2	✓	✓
3	✗	✓

(b) Without *a_dept_is_selected*

	Schemas	
Invs.	1	2
1	✓	✓
2	✗	✓
3	✗	✓

(c) Without *a_doctor_is_selected*

Fig. 10. Checking data invariants preservation for different versions of the button moveBu's **onClick**.

4.2 Checking data invariant preservation

We now describe how we check whether modeled events preserve data invariants.

Fix a data model D and a GUI model G. Let Φ be D's declared invariants. Let ev be an event in G and let B be a branch of ev's action tree containing n actions. To check that every instance of B preserves the invariants in Φ, we proceed as follows:

1. We define a ComponentUML data model D_n that represents all sequences of n states. Recall that a state is any instance of the data model D along with any assignment to the widget variables in G.
2. For $1 \leq i < n$, we formalize an OCL expression, in the context of D_n, that the i-th action's postconditions is satisfied in the $(i+1)$-th state. We denote by $\mathit{Posts}(B)$ the resulting set of OCL expressions.
3. For $1 \leq i \leq n$, we formalize an OCL expression, in the context of D_n, that the *guard* of the i-th action is satisfied in the i-th state. We denote by $\mathit{Guards}(B)$ the resulting set of OCL expressions.
4. For each invariant $\phi \in \Phi$, we formalize an OCL expression, in the context of D_n, that ϕ is satisfied in the first state (initial state). We denote by $\Phi(1)$ the resulting set of OCL expressions.

5. For each invariant $\phi \in \Phi$, we formalize an OCL expression $\psi(n)$, in the context of D_n, stating that ψ is satisfied in the n-th (final) state.
6. We prove that there is no instance of D_n that satisfies

$$\Phi(1) \cup \mathit{Posts}(B) \cup \mathit{Guards}(B) \ \cup \{\neg\psi(n)\}\,.$$

This formula expresses that there is no sequence of n states where the first state satisfies all the invariants, each state satisfies the postcondition of the action leading to it, each state satisfies the condition that guards the action leading to the next state, and the final state does not satisfy ψ.

We have built a tool that implements the above steps. For every data model D with invariants Φ, GUI model G, and event ev in G, our tool automatically generates the set of branches Π corresponding to ev. Then, for each branch $B \in \Pi$ and invariant $\psi \in \Phi$, it generates the data model D_n and the sets of OCL expressions $\Phi(1)$, $\mathit{Posts}(B)$, $\mathit{Guards}(B)$, and $\{\neg\psi(n)\}$, where n is B's length. Finally, our tool uses the mapping OCL2FOL$^+$ [8] to generate the first-order proof-score corresponding to step 6 above, both in SMT-LIB syntax [2] and DFG syntax [14].

4.3 Analyzing the EHRM application

We report here on preliminary experiments where we used our tool to check data invariant preservation for the EHRM application. The application's full GUI model only contains 8 events whose associated statements includes data actions, and therefore must be checked. Moreover, the action trees defined by these events contain 49 branches in total, but only 8 of these branches include data actions. Therefore, since the full EHRM application's data model contains 66 invariants, we must perform a total of 528 checks (8 branches $\times$ 66 invariants) to prove data invariant preservation for this application.

We ran these checks on a laptop computer, with a 2.66GHz Intel Core 2 Duo processor and 4Gb 1067MHz. memory, using SPASS [15] as the back-end theorem-prover. Here we summarize the results. First, for branches containing up to 3 data actions (50% of the non-trivial checks fall into this category, including our running example) checking takes less than a second to return the expected answer: "proof found" when the invariant is preserved and "completion found" when (for the sake of experiment) we remove some of conditions from the branch, thereby violating the invariant. Second, when checking branches containing 8-10 actions and 8-10 conditions (45% of the non-trivial checks), we also obtain the expected answer in less than a second, except for several checks that take up to 20 seconds. Third, there is one branch containing 30 actions and 6 conditions for which our check does not terminate (we timed out after four days) after we removed some of the conditions required for proving data invariant preservation. Finally, note that all these results depend on the (so far successful) interaction between (i) the way we formalize sequences of n states, OCL invariants, actions' guards, and actions' post-conditions, and (ii) the heuristics implemented in the verification back-end we use, here SPASS. We are currently analyzing this interaction in depth to better understand the scope and limitations of our tool.

5 Conclusions

This chapter complements the article [3], where we present the ActionGUI methodology and tool in detail. [3] also contains an extensive comparison with related work such as [6, 7, 11, 10] and provides summary statistics from five different developments. The eHealth application was one of the smaller examples considered there and other examples are roughly an order of magnitude larger, e.g., with hundreds of windows, buttons, labels, and if-then-else-statements and thousands of OCL statements. In contrast, in this paper, we present one case study in detail. We also describe model-based property checking, which was not addressed in [3].

In the following we draw some conclusions based on our experience with the eHealth application and developing other applications with ActionGUI. First, ActionGUI's security modelling language is well suited for modeling access control policies that combine both *declarative* and *programmatic* aspects. Declarative access control policies depend on static information, namely the assignments of users and permissions to roles. Programmatic access control depends on dynamic information, namely the satisfaction of authorization constraints in the current system state. Programmatic access control is formalized using authorization constraints and, as Section 3.3 illustrates, this allows us to model directly the kinds of authorization rules considered in the eHealth case study.

Second, ActionGUI's graphical user interface modelling language is well suited for modeling *dynamic web pages*. These are pages, displayed at the client, that are generated at the time of access by a user or that change as a result of user interaction. As Section 3.4 illustrates, an important aspect of our methodology is that developers can model this behavior independent of the access control policy. The policy is later lifted from the security model to this behavioral model, as described in Section 3.5.

Third, as explained in Section 3.6, the ActionGUI code generator can automatically generate ready-to-deploy, security-aware, data-management web applications. By data-management, we mean that most of the behavior described in the GUI model is built from CRUD actions (which create, read, update and delete data). When all behavior can be described this way, then the entire application can be generated from the models, including a complete, configured security infrastructure and back-end database support.

Finally, our case study illustrates how users can specify properties of ActionGUI models, such as invariant preservation. Moreover, as described in Section 4, our approach to checking these properties based on translation to first-order logic is practical, see also [5]. This is a form of model-checking and, as in other domains, it has an important role to play in building and certifying security-critical systems. Designers and system certifiers can reason about systems at the model level using automated tool support. Moreover, with our approach, they can afterwards generate model-conform, and therefore property conform, systems simply by pressing a button. Our experience with ActionGUI shows that this is not merely a vision for the future, but it is realizable today, at least for small and medium-scale data-management applications.

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) and by the Spanish Ministry of Economy and Competitiveness Project "StrongSoft" (TIN2012-39391-C04-04).

References

1. ActionGUI. The ActionGUI project, 2013. `http://www.actiongui.org`.
2. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.
3. D. Basin, M. Clavel, M. Egea, M. A. G. de Dios, and C. Dania. A model-driven methodology for developing secure data-management applications. *IEEE Transactions on Software Engineering*, 2014. To appear.
4. 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.
5. 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)*, volume 1998443, pages 1–10, Innsbruck, Austria, 2011. New York, NY, USA.
6. H. Baumeister, N. Koch, and L. Mandel. Towards a UML extension for hypermedia design. In R. B. France and B. Rumpe, editors, *Proc. of UML'99*, LNCS, pages 614–629. Springer, 1999.
7. M. Busch and N. Koch. MagicUWE - a case tool plugin for modeling web applications. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Proc. of ICWE'09*, volume 5648 of *LNCS*, pages 505–508. Springer, 2009.
8. C. Dania and M. Clavel. OCL2FOL+: Coping with Undefinedness. In J. Cabot, M. Gogolla, I. Ráth, and E. D. Willink, editors, *OCL@MoDELS*, volume 1092 of *CEUR Workshop Proceedings*, pages 53–62. CEUR-WS.org, 2013.
9. D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.
10. X. Jia, A. Steele, L. Qin, H. Liu, and C. Jones. Executable visual software modeling—the ZOOM approach. *Software Quality Control*, 15:27–51, March 2007.
11. C. Kroiss, N. Koch, and A. Knapp. UWE4JSF: A model-driven generation approach for web applications. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Proc. of ICWE'09*, volume 5648 of *LNCS*, pages 493–496. Springer, 2009.
12. NESSoS. The European Network of Excellence on Engineering Secure Future internet Software Services and Systems, 2010. `http://www.nessos-project.eu`.
13. Object Management Group. Object constraint language specification version 2.3.1. Technical report, OMG, 2012. `http://www.omg.org/spec/OCL/2.3.1`.
14. C. Weidenbach. SPASS input syntax version 1.5, 1999.
15. C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.