Model-Based Formal Reasoning about Data-Management Applications*

Carolina Dania and Manuel Clavel

IMDEA Software Institute, Madrid, Spain
{carolina.dania,manuel.clavel}@imdea.org

Abstract. 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. Typically, the application's data is required to satisfy some properties, which we may call the application's data invariants. In this paper, we introduce a tool-supported, model-based methodology for proving that all the actions possibly triggered by a data-management application will indeed preserve the application's data invariants. Moreover, we report on our experience applying this methodology on a non-trivial case study: namely, an application for managing medical records, for which over eighty data invariants need to be proved to be preserved.

1 Introduction

Model-Driven Architecture (MDA) [13] supports the development of complex software systems by generating software from models. Of course, the quality of the generated software 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. Experience shows that even when using powerful, high-level modelling languages, it is easy to make logical errors and omissions. It is critical not only that the modelling language has a well-defined semantics, so one can know what one is doing, but also that there is tool support for analyzing the modelled systems' properties.

ActionGUI [2] is a methodology for the model-driven development of secure data-management applications. It consists of languages for modelling multi-tier systems, and a toolkit for generating these systems. Within this methodology, a secure data-management application is modelled using three interrelated models:

1. A *data model* defines the application's data domain in terms of its classes, attributes, and associations. It also defines the application's *data invariants*, i.e., the properties about the application's data that are required to be satisfied in every state.

* This work is partially supported by the Spanish Ministry of Economy and Competitiveness Project "StrongSoft" (TIN2012-39391-C04-01 and TIN2012-39391-C04-04).

DOI: 10.1007/978-3-662-46675-9_15

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 (GUI) model* defines the application's graphical interface and application logic.

From these models, ActionGUI generates complete, ready-to-deploy, security-aware web applications, along with all support for access control.

In this paper, we enhance ActionGUI with a tool-supported, model-based methodology for proving the *invariant preservation* property, i.e., that all the action possibly triggered by an application will indeed preserve the application's data invariants. In a nutshell, our approach, which was first informally sketched in [6], consists of the following three steps. Suppose that we are interested in checking whether a sequence $\mathcal{A} = \langle act_1, \ldots, act_{n-1}\rangle$ of data actions preserves an invariant ϕ of an application's data model $\mathcal{D}$. We proceed as follows: (Step 1) From the data model $\mathcal{D}$, we automatically generate a new data model $\mathrm{Film}(\mathcal{D}, n)$ for representing all sequences of n states of $\mathcal{D}$. Notice that some of these sequences will correspond to executions of $\mathcal{A}$, but many others will not. (Step 2) We constrain the model $\mathrm{Film}(\mathcal{D}, n)$ in such a way that it will represent exactly the sequences of states corresponding to executions of $\mathcal{A}$. We do so by adding to $\mathrm{Film}(\mathcal{D}, n)$ a set of constraints $\mathrm{Execute}(\mathcal{D}, act_i, i)$ capturing the execution of the action act_i upon the i-th state of a sequence of states, for $i = 1, \ldots, n-1$. (Step 3) We prove that, for every sequence of states represented by the model $\mathrm{Film}(\mathcal{D}, n)$ constrained by $\bigcup_{i=1}^{n-1} \mathrm{Execute}(\mathcal{D}, act_i, i)$, if the invariant ϕ is satisfied in the first state of the sequence then it is also satisfied in the last state of the sequence.

Organization. After describing in more detail the three steps of our methodology, we report on our experience applying it to a non-trivial case study. We conclude with a brief discussion on related and future work.

2 Modelling Sequences of States (Step 1)

A data model provides a data-oriented view of a system, the idea being that each state of a system can be represented by an *instance* of the system's data model. In this section, however, we introduce a special data model: one whose instances do not represent states of a system but instead *sequences of states* of a system. We begin recalling the notions of ActionGUI data models and object models. Notice that, for the sake of readability, we have moved the technical definitions to Appendix A.

2.1 Data Models

ActionGUI employs ComponentUML [1] for data modelling. ComponentUML provides a subset of UML class models where *entities* (classes) can be related by *associations* and may have *attributes*. A formal definition of ComponentUML data models is given in the Appendix A (Definition 2).

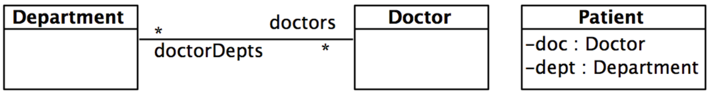

Fig. 1. EHR: A sample data model

Example 1. Consider the ComponentUML model EHR shown in Fig. 1. It consists of three entities: Patient, Department, and Doctor.

Patient. It represents patients. The doctor treating a patient is set in the attribute doc and the department where a patient is treated is set in the attribute dept.

Department. It represents departments. The doctors working in a department are linked to the department through the association-end doctors.

Doctor. It represents doctor's information. Departments where a doctor works are linked to the doctor's information through the association-end doctorDepts.

2.2 Object Models

Object models represent *instances* of data models, consisting of *objects* (instances of entities), with concrete attribute *values*, and *links* (instances of associations). A formal definition of ComponentUML object models is given in the Appendix A (Definition 3).

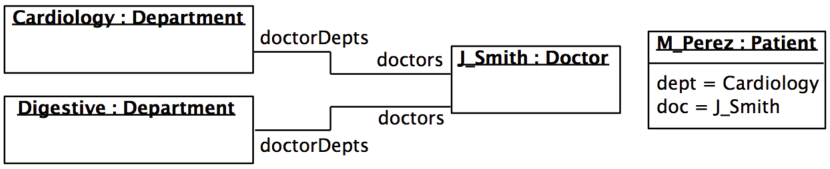

Fig. 2. Inst_EHR: A sample object model

Example 2. Consider the ComponentUML object model Inst_EHR shown in Fig. 2. It represents an instance of the ComponentUML model EHR shown in Fig. 1. In particular, Inst_EHR represents a state of the system in which there are only two departments, namely, Cardiology and Digestive; one doctor, namely, J_Smith, working for both departments; and one patient, M_Perez, treated by doctor J_Smith in the department of Cardiology.

Let $\mathcal{D}$ be a data model. In what follows, we denote by $[\![\mathcal{D}]\!]$ the set of all instances of $\mathcal{D}$.

2.3 Data Invariants

Data invariants are properties that are required to be satisfied in every state of a system.

In ActionGUI we use Object Constraint Language (OCL) [15] to formalize a system's data invariants. OCL is a strongly typed textual language. 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 object model; and operators to iterate over collections. 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 the boolean operator oclIsUndefined().

OCL expressions are written in the context of a data model and can be evaluated on any object model of this data model. The evaluation of an OCL expression returns a value but does not alter the given object model, since OCL evaluation is side-effect free. Let $\mathcal{D}$ be a data model, let $\mathcal{I} \in [\![\mathcal{D}]\!]$ be an object model, and let *expr* be an OCL expression. In what follows, we denote by $[\![expr]\!]^{\mathcal{I}}$ the result of evaluating *expr* in $\mathcal{I}$. Also, let Φ be a set of data invariants over $\mathcal{D}$. Then, we denote by $[\![\mathcal{D}, \Phi]\!] \subseteq [\![\mathcal{D}]\!]$ the set of all the *valid* instances of $\mathcal{D}$ with respect to Φ. More formally,

$$[\![\mathcal{D}, \Phi]\!] = \{\mathcal{I} \in [\![\mathcal{D}]\!] \mid [\![\phi]\!]^{\mathcal{I}} = \mathsf{true},\ \text{for every } \phi \in \Phi\}.$$

Example 3. Suppose that the following data invariants are specified for the data model EHR in Fig. 1:

1. *Each patient is treated by a doctor.*

   Patient.allInstances()→forAll(p|not(p.doc.oclIsUndefined()))

2. *Each patient is treated in a department.*

   Patient.allInstances()→forAll(p|not(p.dept.oclIsUndefined()))

3. *Each patient is treated by a doctor who works in the department where the patient is treated.*

   Patient.allInstances()→forAll(p|p.doc.doctorDepts→includes(p.dept))

Clearly, the object model Inst_EHR in Fig. 2 is a valid instance of EHR with respect to the data invariants (1)–(3), since they evaluate to true in Inst_EHR.

2.4 Filmstrip Models

Next, we introduce the notion of *filmstrips* to model sequences of states of a system. Given a data model $\mathcal{D}$, a *$\mathcal{D}$-filmstrip model* of length n, denoted by $\mathrm{Film}(\mathcal{D}, n)$, is a new data model which contains the same classes as $\mathcal{D}$, but now:

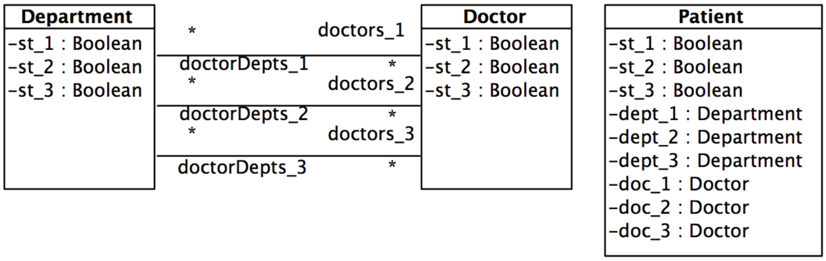

Fig. 3. Film(EHR,3): a filmstrip model of length 3 of EHR

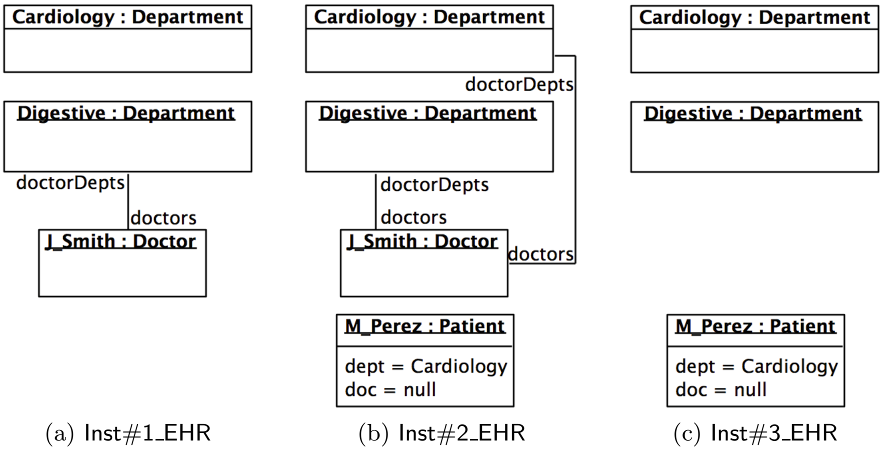

Fig. 4. Three instances of EHR

– To represent that an object may have different attribute values and/or links in each state, each class c contains n different "copies" of each of the attributes and association-ends that c has in $\mathcal{D}$. The idea is that, in each instance of a filmstrip model, the value of the attribute at (respectively, association-end as) for an object o in the i-th state of the sequence of states modelled by this instance is precisely the value of the i-th "copy" of at (respectively, as).
– To represent that an object may exist in some states, but not in others, each class c contains n "copies" of a new boolean attribute `st`. The idea is that, in each instance of a filmstrip model, an object o exists in the i-th state of the sequence of states modelled by this instance if and only if the value of the i-th "copy" of `st` is `true`.

A formal definition of ComponentUML filmstrip models is given in Appendix A (Definition 4).

Example 4. In Fig. 3 we show the filmstrip model Film(EHR, 3). Consider now the three instances of EHR shown in Fig. 4. The first instance (Inst#1_EHR)

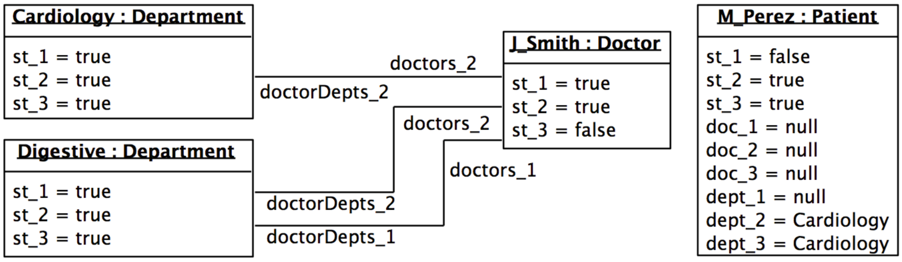

Fig. 5. An instance of Film(EHR, 3)

corresponds to a state where there are two departments, Cardiology and Digestive, and one doctor, J_Smith, working in Digestive. The second instance (Inst#2_EHR) is like the first one, except that now J_Smith also works in Cardiology and, moreover, there is a patient, M_Perez, who is treated in Cardiology, but has no doctor assigned yet. Finally, the third instance (Inst#3_EHR) is like the second one, except that it does not contain any doctor. In Fig. 5 we show how the sequence ⟨Inst#1_EHR, Inst#2_EHR, Inst#3_EHR⟩ can be represented as an instance of Film(EHR, 3).

To conclude this section, we introduce a function Project(), which we will use when reasoning about filmstrip models. Let $\mathcal{D}$ be a data model and let ϕ be an expression. Project($\mathcal{D}, \phi, i$) "projects" the expression ϕ so as to refer to the i-th state in the sequences represented by the instances of Film($\mathcal{D}, n$), for $n \geq i$. A formal definition of Project() is given in Appendix A (Definition 5).

Example 5. Consider the data invariants (1) and (3) presented in the Example 3. Then,

Project(EHR, (1), 1) =
  Patient.allInstances()→select(p|p.st_1)→forAll(p|not(p.doc_1.oclIsUndefined()))
Project(EHR, (3), 1) =
  Patient.allInstances()→select(p|p.st_1)→forAll(p|p.doc_1.doctorDepts_1
    →includes(p.dept_1))

Recall that Patient.allInstances()→select(p|p.st_1) refers to the instances of the entity Patient which exist in the first state of the sequences of states modelled by Film(EHR, 3), while .doc_1 and .doctorDepts_1 refer, respectively, to the value of the attribute doc and the links through the association-end doctorDepts of the instances of the entity Patient also in the first state of the aforementioned sequences of states.

3 Modelling Sequences of Data Actions (Step 2)

As explained before, given a data model $\mathcal{D}$ and a positive number n, the instances of the filmstrip model Film($\mathcal{D}, n$) represent sequences of n states of the system.

Notice, however, that, in the sequence of states represented by an instance of Film($\mathcal{D}, n$), the $(i+1)$-th state does not need to be the result of executing an atomic data action upon the i-th state.

Let $\mathcal{D}$ be a data model and let *act* be a CRUD data action. In this section we introduce a set of boolean OCL expressions, Execute($\mathcal{D}$, *act*, i), which capture the relations that hold between the i-th and $(i+1)$-th states of a sequence, if the latter is the result of executing the action *act* upon the former. For the sake of space limitation, however, we only provide here the expressions that capture the differences between the two states, $(i+1)$-th and i-th, but not the expressions that capture their commonalities.

As expected, we define Execute($\mathcal{D}$, *act*, i) by cases. In ActionGUI, we consider the following *atomic data actions*: *create* or *delete* an object of an entity; *read* the value of an attribute of an object; and *add* or *remove* a link between two objects. [1]

Action create. For *act* the action of creating an instance *new* of an entity c, the difference between the states $(i+1)$-th and i-th can be captured by the following expressions in Execute($\mathcal{D}$, *act*, i):

– $new.\mathsf{st\_}i = \mathsf{false}$.
– $new.\mathsf{st\_}(i+1) = \mathsf{true}$.
– $new.at\_(i+1) = \mathsf{null}$, for every attribute at of the entity c.
– $new.as\_(i+1){\rightarrow}\mathsf{isEmpty()}$, for every association-end as of the entity c.

Action delete. For *act* the action of deleting an instance o of an entity c, the difference between the states $(i+1)$-th and i-th can be captured by the following expressions in Execute($\mathcal{D}$, *act*, i):

– $o.\mathsf{st\_}i = \mathsf{true}$.
– $o.\mathsf{st\_}(i+1) = \mathsf{false}$.
– $o.at\_(i+1) = \mathsf{null}$, for every attribute at of the entity c.
– $o.as\_(i+1){\rightarrow}\mathsf{isEmpty()}$, for every association-end as of the entity c.
– $c'.\mathsf{allInstances()}.as'\_(i+1){\rightarrow}\mathsf{excludes}(o)$ for every entity c', and every association-end as' between c' and c.

Action update. For *act* the action of updating an attribute at of an instance o of an entity c with a value v, the difference between the states $(i+1)$-th and i-th can be captured by the following expression in Execute($\mathcal{D}$, *act*, i):

– $o.at\_(i+1) = v$.

[1] ActionGUI supports also *conditional* data actions, where the conditions are boolean OCL expressions. Notice that, when *act* is a conditional data action, we must also include in Execute($\mathcal{D}$, *act*, i), the expression that results from "projecting" its condition, using the function Project(), so as to refer to the i-th state in the sequence.

Action add. For *act* the action of adding an object o' to the objects that are linked with an object o through an association-end *as* (whose opposite association-end is as'), the difference between the states $(i+1)$-th and i-th can be captured by the following expressions in Execute$(\mathcal{D}, act, i)$:

– $o.as\_(i+1) = (o.as\_i) \rightarrow \mathsf{including}(o')$.
– $o'.as'\_(i+1) = (o'.as'\_i) \rightarrow \mathsf{including}(o)$.

Action remove. For *act* the action of removing an object o' to the objects that are linked with an object o through an association-end *as* (whose opposite association-end is as'), the difference between the states $(i+1)$-th and i-th can be captured by the following expressions in Execute$(\mathcal{D}, act, i)$:

– $o.as\_(i+1) = (o.as\_i) \rightarrow \mathsf{excluding}(o')$.
– $o'.as'\_(i+1) = (o'.as'\_i) \rightarrow \mathsf{excluding}(o)$.

To end this section, we list below the expressions in Execute$(\mathcal{D}, act, i)$ that capture the commonalities between the states $(i+1)$-th and i-th, for the case of the action updating an attribute *at* of an instance o of and entity c; the expressions for the other cases are entirely similar.

– $d.\mathsf{allInstances}() \rightarrow \mathsf{select}(\mathsf{x}|\mathsf{x.st\_}(i+1)) = d.\mathsf{allInstances}() \rightarrow \mathsf{select}(\mathsf{x}|\mathsf{x.st\_}i)$, for every entity d.
– $d.\mathsf{allInstances}() \rightarrow \mathsf{select}(\mathsf{x}|\mathsf{x.st\_}i) \rightarrow \mathsf{forAll}(\mathsf{x}|\mathsf{x}.at'\_(i+1) = \mathsf{x}.at'\_i)$, for every entity d and every attribute at' of d, such that $at' \neq at$.
– $c.\mathsf{allInstances}() \rightarrow \mathsf{select}(\mathsf{x}|\mathsf{x.st\_}i) \rightarrow \mathsf{excluding}(o) \rightarrow \mathsf{forAll}(\mathsf{x}|\mathsf{x}.at\_(i+1) = \mathsf{x}.at\_i)$.
– $d.\mathsf{allInstances}() \rightarrow \mathsf{select}(\mathsf{x}|\mathsf{x.st\_}i) \rightarrow \mathsf{forAll}(\mathsf{x}|\mathsf{x}.as\_(i+1) = \mathsf{x}.as\_i)$ for every entity d, and every association-end *as* of d.

4 Proving Invariants Preservation (Step 3)

Invariants are properties that are *required* to be satisfied in every system state. Recall that, in the case of data-management applications, the system states are the states of the applications' persistence layer, which can only be changed by executing the sequences of data actions associated to the applications' GUI events. Also recall that, within ActionGUI, (i) data invariants are specified along with the application's data model, and also that (ii) the sequences of actions triggered by the GUI events are specified in the application's GUI model. We can now formally define the invariant-preservation property as follows:

Definition 1 (Invariant preservation). *Let $\mathcal{D}$ be a data model, with invariants Φ. Let $\mathcal{A} = \langle act_1, \ldots, act_{n-1}\rangle$ be a sequence of data actions. We say that $\mathcal{A}$* preserves *an invariant $\phi \in \Phi$ if and only if*

$$\forall \mathcal{F} \in [\![\mathrm{Film}(\mathcal{D}, n), \bigcup_{i=1}^{n-1} \mathrm{Execute}(\mathcal{D}, act_i, i)]\!] . \tag{1}$$

$$[\![\mathrm{Project}(\mathcal{D}, \bigwedge_{\psi\in\Phi}(\psi), 1) \;\mathsf{implies}\; \mathrm{Project}(\mathcal{D}, \phi, n)]\!]^{\mathcal{F}} = \mathrm{true},$$

*i.e., if and only if, for every $\mathcal{A}$-*valid *instance $\mathcal{F}$ of* Film$(\mathcal{D}, n)$ *the following holds: if all the invariants in Φ evaluate to* true *when "projected" over the first state of the sequence of states represented by $\mathcal{F}$, then the invariant ϕ evaluates to* true *as well when "projected" over the last state of the aforementioned sequence.*

Using SMT Solvers for Checking Invariant-Preservation

In [4,5] we proposed a mapping from OCL to first-order logic, which consists of two, inter-related components: (i) a map from ComponentUML models to first-order formulas, called ocl2fol$_{\text{def}}$; and (ii) a map from boolean OCL expressions to first-order formulas, called ocl2fol. The following remark formalizes the main property of our mapping from OCL to first-order logic.

Remark 1. Let $\mathcal{D}$ be a data model, with data invariants Φ. Let ϕ be a boolean expression. Then,

$$\forall \mathcal{I} \in [\![\mathcal{D}, \Phi]\!] . ([\![\phi]\!]^{\mathcal{I}} = \text{true}) \Longleftrightarrow$$
$$\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\text{ocl2fol}(\gamma) \mid \gamma \in \Phi\} \cup \text{ocl2fol}(\mathsf{not}(\phi)) \text{ is unsatisfiable.}$$

By the previous remark, we can reformulate Definition 1 as follows: Let $\mathcal{D}$ be a data model, with invariants Φ. Let $\mathcal{A} = \langle act_1, \ldots, act_{n-1} \rangle$ be a sequence of data actions. We say that $\mathcal{A}$ *preserves* an invariant $\phi \in \Phi$ if and only if the following set is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\text{Film}(\mathcal{D}, n)) \cup \{\text{ocl2fol}(\gamma) \mid \gamma \in \bigcup_{i=1}^{n-1} \text{Execute}(\mathcal{D}, act_i, i)\} \tag{2}$$
$$\cup\ \text{ocl2fol}(\mathsf{not}(\text{Project}(\mathcal{D}, \bigwedge_{\psi \in \Phi} (\psi), 1) \ \mathsf{implies}\ \text{Project}(\mathcal{D}, \phi, n))).$$

In other words, using our mapping from OCL to first-order logic, we can transform an invariant-preservation problem (1) into a first-order satisfiability problem (2). And by doing so, we open up the possibility of using SMT solvers to automatically (and effectively) check the invariant-preservation property of non-trivial data-management applications, as we will report in the next section.

5 Case Study

In this section we report on a case study about using SMT solvers —in particular, Z3 [7]— for proving the invariant-preservation property. Satisfiability Modulo Theories (SMT) generalizes boolean satisfiability (SAT) by incorporating equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other first-order theories. Of course, when dealing with quantifiers, SMT solvers cannot be complete, and may return "unknown" after a while, meaning that neither they can prove the quantified formula to be unsatisfiable, nor they can find an interpretation that makes it satisfiable.

The data-management application for this case study is the eHealth Record Management System (EHRM) developed, using ActionGUI, within the European Network of Excellence on Engineering Secure Future Internet Software Services and Systems (NESSoS) [14]. The EHRM application consists of a web-based system for electronic health record management. The data model contains 18 entities, 40 attributes, and 48 association-ends. It also contains 86 data invariants. For the sake of illustration, we can group the EHRM's data invariants in the following categories:

G1. Properties about the population of certain entities. E.g., *There must be at least a medical center.*

MedicalCenter.allInstances()→notEmpty().

G2. Properties about the definedness of certain attributes. E.g., *The name of a professional cannot be left undefined.*

Professional.allInstances()→forAll(p|not(p.name.oclIsUndefined())).

G3. Properties about the uniqueness of certain data. E.g.: *There cannot be two different doctors with the same licence number.*

Doctor.allInstances()→forAll(d1,d2|d1<>d2 implies d1.licence<>d2.licence).

G4. Properties about the population of certain association-ends. E.g., *Every medical center should have at least one employee.*

MedicalCenter.allInstances()→forAll(m|m.employees→notEmpty()).

G5. Other properties: E.g., *A patient should be treated in a department where its assigned doctor works.*

```
Patient.allInstances()
            →forAll(p|p.doctor.doctorDepartments→includes(p.department)).
```

In our case study, we have checked the invariant-preservation property for seven non-trivial sequences of data actions: namely, those that create a new admin staff, a new nurse, or a new doctor; those that reassign a doctor or a nurse to another department; and those that register a new patient, and move a patient to a different ward. The result of our case study is shown in Fig. 6. In particular, for each of the aforementioned sequences of actions, we indicate:

– The number of data actions (and conditions) in the sequence.
– The number of data invariants (potentially) affected by the actions in the sequence, indicating how many of them we have proved to be preserved by the sequence and how many to be violated.[2]

[2] Interestingly, when an invariant is violated, Z3 returns also an instance of the given filmstrip model responsibly for this violation. This *counterexample* can then be used to fix accordingly the given sequence of actions.

Sequences	Acts.	Conds.	Invariants			Time		
			affected	preserved	violated	min.	max.	avge.
Create an administrative	8	9	18	18	0	0.03s	0.20s	0.05s
Create a nurse	10	11	22	22	0	0.03s	0.22s	0.06s
Create a doctor	11	12	25	24	1	0.03s	27.00s	0.07s
Reassign a doctor	2	6	2	2	0	6.88s	11.10s	8.94s
Reassign a nurse	2	6	2	1	1	0.10s	17.01s	8.55s
Register patient	30	6	28	26	2	0.03s	0.20s	0.05s
Move a patient	2	3	3	3	0	0.03s	0.03s	0.03s
		Total	100	96	4			

Fig. 6. EHRM case study: summary

– The minimum, maximum, and average time taken for proving that the sequence preserves (or violates) each of the (potentially) affected invariants.

All the proofs have been ran on a machine with an Intel Core2 processor running at 2.83 GHz with 8GB of RAM, using Z3 versions 4.3.1 and 4.3.2. All the Z3 input files have been automatically generated with an ActionGUI plugin which implements our mapping from OCL to first-order logic. These files are available at `http://software.imdea.org/~dania/tools/ehrm.html`, where we also indicate which files are to be ran with which version.

Lessons learned. There are two main lessons that we can learn from this case study. The first lesson is that, when modelling non-trivial data-management applications, it is indeed not difficult to make errors, or at least omissions, even when using a high-level language like ActionGUI. In fact, the four *violated* invariants showed in Fig. 6 arise because the EHRM's modeler inadvertently omitted some conditions for the execution of the corresponding sequence of actions. As an example, for the case of creating a doctor, the invariant that is *violated* is "Every doctor has a unique licence number", and it is so because the modeler omitted a condition for checking that the licence number of the doctor to be created must be different from the licence numbers of the doctors already in the system. As another example, for the case of reassigning a nurse, the invariant that is *violated* is "There should be at least one nurse assigned to each department", and this is produced because the modeler omitted a condition for checking that the department where the nurse to be reassigned currently works must have at least two nurses working in it.

The second lesson that we have learned is that, using our methodology, and, in particular, using Z3 as the back-end prover, the invariant-preservation property can indeed be effectively checked for non-trivial data-management applications. As reported in Fig. 6, we are able to automatically prove that, for each of the sequences of actions under consideration, all the *affected* invariants are either *preserved* or *violated*. This means that Z3 does not return "unknown" for any of the 100 checks that we have to perform (corresponding to the total number of *affected* invariants), despite the fact that in all these checks there are (many) quantifiers involved. Moreover, regarding performance, Fig. 6 shows that, in most of the cases we are able to prove the invariant-preservation property in less than 100ms (worst case: 27s). This great performance is achieved even though, for each case, Z3 needs to check the satisfiability of a first-order theory containing on average 190 declarations (of function, predicate and constant symbols), 20 definitions (of predicates), and 550 assertions. Overall, these results improve very significantly those obtained in a preliminary, more simple case study reported in [6], where some checks failed to terminate after several days, and some others took minutes before returning an answer. However, we should take these new results with a grain of salt. Indeed, we are very much aware (even painfully so) that our current results depend on the (hard-won) interaction between (i) the way we formalize sequences of n states, OCL invariants, actions' conditions, and actions' executions, and (ii) the heuristics implemented in the verification back-end we use, namely Z3. This state-of-affairs is very well illustrated by the fact that, as indicated before, we have had to use two different versions of Z3 (4.3.1 and 4.3.2) to complete our case study for the following reason: there are some checks for which one of the versions returns "unknown", while the other version returns either "sat" or "unsat"; but there are some other checks for which precisely the opposite occurs.

6 Related Work

In the past decade, there has been a plethora of proposals for model-based reasoning about the different aspects of a software system. For the case of the static or structural aspects of a system, the challenge lies in mapping the system's data model, along with its data invariants, into a formalism for which reasoning tools may be readily available (see [10] and references). By choosing a particular formalism each proposal commits to a different trade-off between expressiveness and termination, automation, or completeness (see [16,3] and references). On the other hand, for the case of model-based reasoning about a system's dynamic aspects, which is our concern here, the main challenge lies in finding a suitable formalism in which to map the models specifying how the system can change over time. To this extent, it is worthwhile noticing the different attempts made so far to extend OCL with temporal features (see [12] and references). In our case, however, we follow a different line of work, one that is centered around the notion of *filmstrips* [8,18]. A filmstrip is, ultimately, a way of encoding a sequence of snapshots of a system. Interestingly, when this encoding uses the same language employed for modelling the static aspects of a system, then the tools available for reasoning about the latter can be used for reasoning about the former. This is precisely our approach, as well as the one underlying the proposals presented in [11] and [9]. However, the difference between our approach and those are equally important. It has its roots in our different way of mapping data models and data invariants (OCL) into first-order logic [4,5], which allows us to effectively use SMT solvers for reasoning about them, while [9] and [11] resort to SAT solvers. As a consequence, when successful, we are able to prove that all possible executions of a given sequence of data actions preserve a given data invariant. On the contrary, [9] can only validate that a given execution preserves a given invariant, while [11] can prove that all possible executions of a given sequence of data action preserve a given invariant, but only if these executions do not involve more than a given number of objects and links. Finally, [17] proposes also the use of filmstrip models and SMT solvers for model-based reasoning about the dynamic aspects of a system. This proposal, however, at least in its current form, lacks too many details (including non-trivial examples) for us to be able to provide a fair comparison with our approach.

7 Conclusions and Future Work

Data-management applications are focused around the so-called CRUD actions, namely, to 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. In [2] we proposed a model-driven development environment, called ActionGUI, for developing data-management application. With ActionGUI, complete, ready-to-deploy, security-aware web applications can be automatically generated from the applications' data, security, and GUI models. In this paper, we present our work to enhance ActionGUI with a methodology for automatically proving that the application's data invariants, i.e., the properties that are required to hold in every state of the system, are indeed preserved after the execution of the sequences of data actions supported by the application's GUI. We have also reported on a non-trivial case study, in which we have successfully applied our methodology over an eHealth application whose data model contains 80 data invariants, and whose GUI model includes events possibly triggering more than 20 data actions in a sequence.

Finally, we are currently extending our methodology to deal with complex, non-atomic data action. The idea, of course, is to model the execution of these complex actions using OCL, as we have done for the case of CRUD actions. A more challenging goal, however, is to extend our methodology to deal with *iterations*. In ActionGUI, each iteration specifies that a sequence of data actions must be iterated over a collection of data elements. The idea here is to integrate in our methodology the notion of *iteration invariant*, taking advantage of the fact that the collection over which the sequence of data actions must be iterated is also specified using OCL. Finally, we are analyzing in depth the interaction between (i) the way we formalize data-invariants and data-action executions and (ii) the heuristics implemented in the verification back-end we use, namely Z3, to better understand its scope and limitations.

References

1. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. ACM Transactions on Software Engineering and Methodology 15(1), 39–91 (2006)

2. Basin, D.A., Clavel, M., Egea, M., García de Dios, M.A., Dania, C.: A model-driven methodology for developing secure data-management applications. IEEE Trans. Software Eng. 40(4), 324–337 (2014)
3. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83(2), 283–302 (2010)
4. Clavel, M., Egea, M., García de Dios, M.A.: Checking unsatisfiability for OCL constraints. Electronic Communications of the EASST 24, 1–13 (2009)
5. Dania, C., Clavel, M.: OCL2FOL+: Coping with Undefinedness. In: Cabot, J., Gogolla, M., Ráth, I., Willink, E. (eds.) CEUR Workshop Proceedings OCL@MoDELS, vol. 1092, pp. 53–62. CEUR-WS.org (2013)
6. García de Dios, M.A., Dania, C., Basin, D., Clavel, M.: Model-driven development of a secure eHealth application. In: Heisel, M., Joosen, W., Lopez, J., Martinelli, F. (eds.) Engineering Secure Future Internet Services and Systems. LNCS, vol. 8431, pp. 97–118. Springer, Heidelberg (2014)
7. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. D'Souza, D., Wills, A.: Catalysis. Practical Rigor and Refinement: Extending OMT, Fusion, and Objectory. Technical report (1995), `http://catalysis.org`
9. Gogolla, M., Hamann, L., Hilken, F., Kuhlmann, M., France, R.B.: From application models to filmstrip models: An approach to automatic validation of model dynamics. In: Fill, H., Karagiannis, D., Reimer, U. (eds.) *Modellierung*. LNI, vol. 225, pp. 273–288. GI (2014)
10. González, C.A., Cabot, J.: Formal verification of static software models in MDE: A systematic review. Information & Software Technology 56(8), 821–838 (2014)
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
12. Kanso, B., Taha, S.: Temporal constraint support for OCL. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 83–103. Springer, Heidelberg (2013)
13. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
14. NESSoS. The European Network of Excellence on Engineering Secure Future internet Software Services and Systems (2010), `http://www.nessos-project.eu`
15. Object Management Group. Object constraint language specification version 2.4. Technical report, OMG (2014), `http://www.omg.org/spec/OCL/2.4`
16. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. Data & Knowledge Engineering 73, 1–22 (2012)
17. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: DATE, pp. 1341–1344. IEEE (2010)
18. WieringaA, R.: survey of structured and object-oriented software specification methods and techniques. ACM Comput. Surv. 30(4), 459–527 (1998)

A Technical Definitions

Definition 2 (Data models). *A* data model *is a tuple* $\langle C, AT, AS, ASO\rangle$ *such that:*

– *C is a set of class identifiers.*
– *AT is a set of triples $\langle at, c, t\rangle$, also represented as $at_{(c,t)}$, where at is an attribute identifier, $c \in C$, $t \in C \cup \{\textsf{Integer}, \textsf{Real}, \textsf{String}, \textsf{Boolean}\}$, and c and t are, respectively, the class and the type of the attribute at.*
– *AS is a set of tuples $\langle as, c, c'\rangle$, also denoted by $as_{(c,c')}$, where as is an association-end identifier, $c, c' \in C$, c and c' are, respectively, the source and the target classes of as.*
– *ASO is a symmetric relation, $ASO \subseteq AS \times AS$, where $(as_{(c,c')}, as'_{(c',c)}) \in ASO$ represents that as' is the association-end opposite to as, and vice versa, and $c, c' \in C$.*

Definition 3 (Object models). *Let $\mathcal{D}$ be a data model $\langle C, AT, AS, ASO\rangle$. Then, a $\mathcal{D}$-*object model *is a tuple $\langle O, VA, LK\rangle$, such that:*

– *O is a set of pairs $\langle o, c\rangle$, where o is an object identifier and $c \in C$. Each pair $\langle o, c\rangle$, also represented as o_c, denotes that the object o is of the class c.*
– *VA is a set of triples $\langle o_c, at_{(c,t)}, va\rangle$, where $at_{(c,t)} \in AT$, $o_c \in O$, $t \in C \cup \{\textsf{Integer}, \textsf{Real}, \textsf{String}, \textsf{Boolean}\}$, and va is a value of type t. Each triple $\langle o_c, at_{(c,t)}, va\rangle$ denotes that va is the value of the attribute at of the object o.*
– *LK is a set of triples $\langle o_c, as_{(c,c')}, o'_{c'}\rangle$, where $as_{(c,c')} \in AS$, and $o_c, o'_{c'} \in O$. Each tuple $\langle o_c, as_{(c,c')}, o'_{c'}\rangle$ denotes that the object o' is among the objects that are linked to the object o through the association-end as.*

Definition 4 (Filmstrip models). *Let $\mathcal{D}$ be a data model, $\mathcal{D} = \langle C, AT, AS, ASO\rangle$. Let n be a positive number. We denote by* $\mathrm{Film}(\mathcal{D}, n)$ *the model of the sequences of length n of $\mathcal{D}$-object models.* $\mathrm{Film}(\mathcal{D}, n)$ *is defined as follows:*

$$\mathrm{Film}(\mathcal{D}, n) = \langle C, (n\times\{\mathrm{st}\}) \cup (n\times AT), (n\times AS), (n\times ASO)\rangle$$

where

– $(n\times\{\mathrm{st}\}) = \{(\mathrm{st\_}i)_{(c,\textsf{Boolean})} \mid c \in C \wedge 1 \le i \le n\}$.
– $(n\times AT) = \{(at\_i)_{(c,t)} \mid at_{(c,t)} \in AT \wedge 1\le i\le n\}$.
– $(n\times AS) = \{(as\_i)_{(c,c')} \mid as_{(c,c')} \in AS \wedge 1\le i\le n\}$.
– $(n\times ASO) = \{((as\_i)_{(c,c')}, (as'\_i)_{(c',c)}) \mid (as_{(c,c')}, as'_{(c',c)}) \in ASO \wedge 1\le i\le n\}$.

Definition 5 (Project). *Let $\mathcal{D} = \langle C, AT, AS, ASO\rangle$ be a data model. Let n be positive number. Let ϕ be a $\mathcal{D}$-expression. For $1 \le i \le n$,* $\mathrm{Proj}(\mathcal{D}, \phi, i)$ *is the* $\mathrm{Film}(\mathcal{D}, n)$*-expression that "projects" the expression ϕ so as to refer to the i-th state in the sequences represented by the instances of* $\mathrm{Film}(\mathcal{D}, n)$. $\mathrm{Proj}(\mathcal{D}, \phi, i)$ *is obtained from ϕ by executing the following:*

– *For every class $c \in C$, replace every occurrence of* c.allInstances() *by* c.allInstances()→select(o|o.st_(i)).
– *For every attribute $at_{(c,t)} \in AT$, replace every occurrence of $.at$ by $.at\_(i)$.*
– *For every link $as_{(c,c')} \in AS$, replace every occurrence of $.as$ by $.as\_(i)$.*