Electronic Communications of the EASST
Volume 5 (2006)

Proceedings of the Sixth OCL Workshop OCL for (Meta-)Models in Multiple Application Domains (OCLApps 2006)

An MDA Framework Supporting OCL

Achim D. Brucker, Jürgen Doser, and Burkhart Wolff

18 pages

Guest Editors: Dan Chiorean, Birgit Demuth, Martin Gogolla, Jos Warmer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

An MDA Framework Supporting OCL

Achim D. Brucker, Jürgen Doser, and Burkhart Wolff

Information Security, ETH Zurich, 8092 Zurich, Switzerland
{brucker,doserj,bwolff}@inf.ethz.ch

Abstract: We present a model-driven architecture (MDA) framework that integrates formal analysis techniques into an industrial software development process model. This comprises modeling using UML/OCL, processing models by model transformations, code generation (including runtime-test environments) and formal analysis using the theorem proving environment HOL-OCL. Moreover, our frameworks supports the verification of proof obligations that are generated during model transformations.

We show the extensibility of our approach by providing a SecureUML extension of the framework, which allows for an integrated specification of security properties, their analysis and their conversion to code.

Keywords: MDE, MDA, OCL, model transformation, code-generation, verification

1 Introduction

Model-Driven Engineering refers to the systematic use of models as primary engineering artifacts throughout the development life-cycle of software systems. In the broadest sense, the term "models" is used for descriptions in a machine-supported format, while the term "systematic" refers to machine-supported transformations between models or from models to code. The instance of Model-Driven Engineering based on the UML and defined by the Object Management Group (OMG) is called model-driven architecture (MDA). In UML, various model elements like classes or state machines can be annotated by logical constraints using the Object Constraint Language (OCL); for this reason, UML can be used as a formal specification language with diagrammatic syntax.

For MDE in general and MDA in particular, a *technical framework* ranging over several stages of the software development process—requirements analysis, design, code generation—is vital. This holds to an even larger extent if semantic information like formal specifications are processed; for MDA, an infrastructure is needed that supports model elements annotated by OCL. In such a framework, model transformations can be implemented that represent *data refinements* (for example, forward-simulation [WD96]) or *retrenchments* (for example, replace the integer type by bounded integer [BS05]). Necessary side-conditions of such transformations can be represented as proof-obligations in OCL. In another scenario, OCL constraints can be used for generating code for runtime-testing or, if combined with the HOL-TestGen system [BW04, BW05], for generating test-data for a systematic test of injected, hand-crafted code.

In this paper, we present such a framework, comprising a toolchain that guides the development process from modeling in a CASE tool to code-generation and formal verification. In

particular, our framework consists of a type-checking component allowing to represent OCL in a structured format which can be imported into our model repository (su4sml). This model repository can serve as a basis for model transformations. Moreover, su4sml is the basis for a template-based code generator supporting code-generation for the UML core and state machines, enriched by OCL specifications. This model can be directly transformed into a (formal) model for the theorem proving environment HOL-OCL [BW06a].

As a distinguishing feature, su4sml is developed in the functional programming language SML [Pau96]. For this reason, implementers of model transformations can profit from several techniques that have proven to be of major importance for symbolic computations occurring naturally in compiler construction or theorem proving: pattern matching allows for direct representations of rules to be performed during transformation, higher-order functions allow for the compact description of search- and replacement strategies, and having a strongly typed language helps to detect many errors at compile time.

We also present an implementation of one particular extension of our framework for UML/OCL: namely support for the UML-based language SecureUML [BDL06]. SecureUML is designed to enrich the business model of a system (represented by class diagrams or statecharts) with a concrete access control model. By a model transformation, class systems and operation specifications are transformed such that a combined model is generated, incorporating security and functional aspects. During the transformation, several proof obligations are generated, making explicit under which conditions the functional model of a system is not interfered by its security model. With the help of our framework, the combined model can be transformed to code, while the proof obligations making this transformation "correct" (in the sense of "no bad interference") can be proven by HOL-OCL. Thus, our framework can be seen as a first step towards a uniform framework supporting both semantic and code-generative aspects of UML/OCL specifications.

The Plan of the Paper. After a general overview of the framework, we present its main components: In Section 3, we describe the implementation of our model repository, in Section 4 we present the template-based code generator and in Section 5 we describe the interface to HOL-OCL. Then, in Section 6, we describe the SecureUML instance of the framework and in Section 7 how model transformations, including support for proof obligation generation, can conveniently be expressed in this framework. Finally, we discuss our experiences and observations.

2 Our Framework: An Overview

In this section, we give an overview of our framework and present a toolchain in which it can be used. But first, we introduce the tools and technologies our framework is based on.

2.1 Background

2.1.1 HOL-OCL.

HOL-OCL [BW06a] (http://www.brucker.ch/projects/hol-ocl/) is an interactive proof environment for UML/OCL. Its mission is to give the term "object-oriented specification" a formal

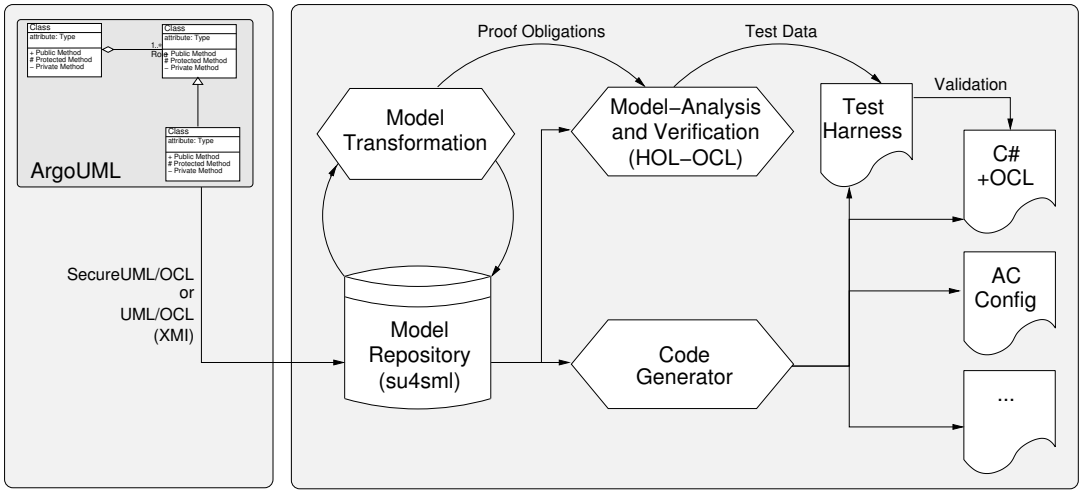

Figure 1: MDA Framework and Toolchain Overview

semantic foundation and to provide effective means to formally reason over object-oriented models. On the theoretical side, this is achieved by representing UML/OCL as a conservative, shallow embedding into the HOL instance of the interactive theorem prover Isabelle [NPW02]. We follow the standard [OMG03a] as closely as possible, in particular, we prove that inheritance can be represented inside the typed λ-calculus with parametric polymorphism. As a consequence of conservativity with respect to HOL, we can guarantee the consistency of the semantic model. On the technical side, this is achieved by automated support for typed, extensible UML data models. Moreover, HOL-OCL provides several derived proof calculi for UML/OCL that allow for formal derivations establishing the validity of UML/OCL formulae. Some automated support for such proofs is also provided, albeit the achieved degree of automation is not yet satisfactory.

2.1.2 SecureUML.

SecureUML [BDL06] is a security modeling language based on RBAC [SCFY96]. In particular, SecureUML supports notions of users, roles and permissions, as well as assignments between them: Users can be assigned to roles, and roles are assigned to specific permission. Users acquire permissions through the roles they are assigned to. Moreover, users are organized into a hierarchy of groups, and roles are organized into a role hierarchy. In addition to this RBAC model, permissions can be restricted by *Authorization Constraints* (expressed in OCL formulae), which have to hold to allow access. SecureUML is generic in the notion of protected actions that can be assigned to permissions. These are specified in a SecureUML *dialect*.

2.2 The Toolchain

Our framework is completed by a toolchain (see Figure 1) that consists of a UML CASE tool with an OCL type-checker for modeling software systems. The framework provides a model repository, model analyzers and various code generators.

We use the UML CASE tool ArgoUML (http://argouml.tigris.org) and combine it with the Dresden OCL2 Toolkit. The Dresden OCL Toolkit uses a specialized metamodel combining the UML 1.5 and the OCL 2.0 metamodel. This results in an upward compatible extension of the UML 1.5 metamodel: every UML 1.5 model is still a model of the combined metamodel. Models expressed in his specialized metamodel can be exported using the XMI export.

At time of writing, our SML-based framework comprises

1. an XMI import supporting the UML 1.4 and 1.5 meta-model (e. g., as used by ArgoUML) and also a metamodel combining UML 1.5 and OCL 2.0 (as used by the Dresden OCL2 Toolkit),
2. a model repository, *su4sml*, which supports the various metamodels we are using, e. g., UML, OCL, SecureUML,
3. a generic, template-based code generator supporting the UML core (e. g., class diagrams), statemachines, OCL and SecureUML, (including the generation of access-control checks for the target languages Java and C#),
4. a code generator setup for checking the OCL specifications at runtime. On the first hand, this serves as a validation that the implementations fulfills the specification, and on the other hand it serves as a basis for further testing activities,
5. a code generator setup for generating test harnesses for unit testing, e. g., for Junit (http://www.junit.org/),
6. model transformations that normalize the models in several normal forms; this comprises the conversion of multiplicities into OCL constraints, etc., and
7. an interface to our theorem prover environment, HOL-OCL, which allows to do (formal) model analysis and verification of UML/OCL models.

The framework is implemented as a set of SML modules that are designed to be easily extensible and also can be used independently. We plan to extend the framework, e. g., by integrating specification based test-generator, i. e., HOL-TESTGEN (http://www.brucker.ch/projects/hol-testgen) into HOL-OCL. The generated test-data can then be used for driving the unit tests and thus provides an automatic test setup for the validation of the (hand-coded) implementation against the specification.

3 The Model Repository: su4sml

The model repository is the component that stores all the (UML) model information that other parts of the framework depend on. It follows the UML/OCL metamodel in representing the model information as closely as this is sensible in a functional programming language. However, some simplifications where made deliberately. For example, we eliminated many indirections that are inherent in the UML metamodel. We also decided to ignore associations between classifiers as such. We only represent their association ends, as part of the participating classifiers.

As our framework aims at code generation and model analysis and verification, we concentrated on those parts of the UML that have a comparatively rich and well-defined semantics: Currently, the repository supports the model elements featured in UML class and statechart diagrams, including a typed, structured representation of OCL expressions. OCL expressions are, as usual, used as class invariants, operation pre- and postconditions, and transition guards in

```
signature REP_CORE = sig

type operation     = { name          : string,
                       precondition  : (string option * OclTerm) list,
                       postcondition : (string option * OclTerm) list,
                       arguments     : (string * OclType) list,
                       result        : OclType,
                       ...
                     }

type attribute     = { name          : string,
                       attr_type     : OclType,
                       init          : OclTerm option,
                       ...
                     }

type associationend = { ... }

datatype Classifier = Class of { name         : Path,
                                 parent       : Path option,
                                 attributes   : attribute list,
                                 operations   : operation list,
                                 associationends : associationend list,
                                 invariant    : (string option * OclTerm) list,
                                 ...
                               }
                    | Interface   of { ... }   (* similar to Class *)
                    | Enumeration of { ... }
                    | Primitive   of { ... }
end
```

Listing 1: su4sml: Representing the UML Core

statemachines. They can, however, also appear as proof obligations, to be then verified in the HOL-OCL theorem prover. For illustration purposes, Listing 1 and Listing 2 show parts of the repository signatures.

In addition to datatype definitions for the various model elements, the repository structure supports various model transformations. A simple example is the conversion of association ends into attributes with corresponding type, together with the generation of an invariant expressing the corresponding cardinality constraint. A more complex transformation would be the transformation of SecureUML models into pure UML/OCL models (cf., Section 6.3 and [BDW06]), together with the generation of the corresponding security proof obligations.

```
signature REP = sig

include REP_OCLTYPE
include REP_OCLTERM
include REP_CORE
include REP_STATEMACHINES

type Model = Classifier list

end
```

Listing 2: su4sml: The Model Repository

4 A Template-based Code Generator

We developed a *Generic Template-based Code Generator* (GCG) on top of the su4sml repository. Template-based means that for each code artifact to be generated there is a template file which contains a skeleton of what has to be generated intertwined with instructions for the code-generator how to fill out the template. The code generator consists of a generic core and a set of cartridges that can be "plugged" into this core (cf., Figure 2). The core part of GCG is independent both with respect to the input as well as the output language, the cartridges are responsible for interpreting the language-dependent instructions in the template files.

The template language has at the core just three syntactic elements: an @if statement for branching on Boolean predicates, a @foreach statement for iterating over lists, and $variable$ interpolation. In particular, the template language is not Turing-complete. For example, the predicates in @if statements come from a fixed (finite) set that is defined by the cartridges that are plugged into the core. Example predicates are attribute_isPublic or operation_isStatic. Similarly, the lists to iterate over are also defined by the cartridges. Example lists are classifier_list, attribute_list, or operation_list. These lists have an implicit notion of hierarchy. The evaluation of attribute_list, for example, depends on the current classifier that one iterates over in the enclosing @foreach statement. Finally, the variables that can be interpolated are also defined by the cartridges. Typical examples are operation_name or attribute_type, see Listing 3 for an example template file.

While the generic core parses the template file, the actual evaluation of the statements is delegated to the cartridges. For example, when the core executes the statement @if operation_isStatic, it asks the cartridge for the current value of the predicate operation_isStatic. Depending on the answer, the core executes the following statements or not.

We build up cartridge chains supporting increasing functionalities. If one cartridge does not support a requested functionality, it passes the request on to the next cartridge, and the result back to the requester.[1] As a starting point for this cartridge chain, we implemented a base cartridge that implements the most basic functionalities which one would probably need in most languages anyways, for example, variables like attribute_name or lists like operation_list. The design allows for cartridges to override functionalities of "earlier" cartridges by implementing

[1] On the implementation level, the "plugging" of these cartridges is done using SML functors.

```
@// Example template for Java
@foreach classifier_list
        @openfile generated/$classifier_name$.java
        package $classifier_package$;

        public class $classifier_name$
        @if hasParent
                extends $classifier_parent$
        @end
        {
        @foreach attribute_list
                public $attribute_type$ $attribute_name$ ;
        @end
        @foreach operation_list
                public $operation_result_type$ $operation_name$(
                @foreach argument_list
                        $argument_type$ $argument_name$,
                @end )
                {}
        @end
        }
@end
```

Listing 3: A Simplified Template File

them themselves. This is sometimes necessary for language-specific cartridges when the language requires certain syntactic properties.

In this way, different cartridges can independently implement different functionalities: basic code skeletons for different programming languages, runtime assertion monitoring of the OCL invariants, pre- and postconditions, test harnesses, deployment infrastructures for different middleware platforms or web-services architectures, etc.

5 A su4sml-based Datatype Package for HOL-OCL

The encoding of object-oriented data structures in HOL is a tedious and error-prone activity, if done manually. In this section, we give an overview of the su4sml-based datatype package we implemented to automate this process. In the theorem prover community, a *datatype package* [Mel92] is a module that allows one to introduce new datatypes and automatically derive certain properties over them. A (conservative) datatype package has two main tasks:

1. generate all required (conservative) constant definitions, and
2. automatically prove interesting properties over the generated definitions.

Our datatype package is implemented on top of the Isabelle kernel API and exploits its potential to build user-programmed extensions in a logically safe way.

In the following, we give a brief overview over the functionality of our datatype package ([BW06a, BW06b] describe more details). The datatype package is implemented on top of two

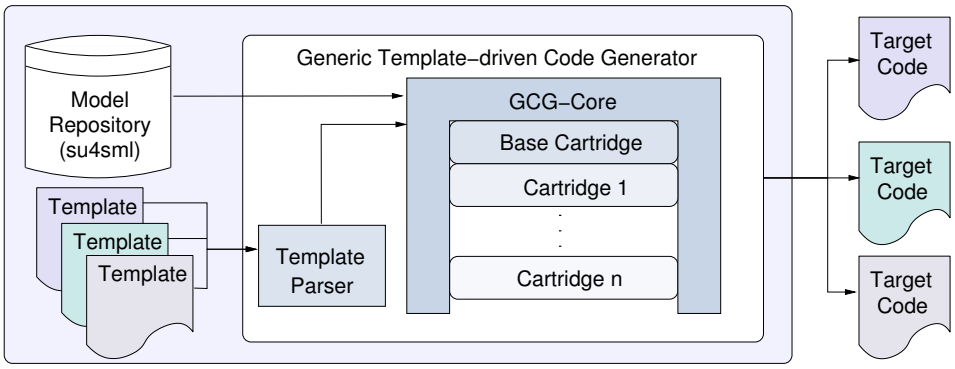

Figure 2: The Cartridge Chain Architecture

```
signature REP_ENCODER = sig
type mdr    = {   theory        : theory,
                  universe      : typ,
                  classifiers   : Classifier list }
  val add_classifiers : Classifier list → mdr → mdr
end
```

Listing 4: The Top-level Interface of the Repository Encoder

components: the su4sml interface and the programming interface of HOL-OCL. Our datatype packages encodes the given UML/OCL model into a HOL-OCL-representation. This is achieved in an extensible way, i. e., classes can be added later on to an existing theory preserving all proven properties ([BW06b] presents more details). The obvious tasks of the datatype package are:

1. declare HOL types for the classifiers of the model,
2. encode the core data model into HOL, and
3. encode the OCL specification and combine it with the core data model.

However, the most important task is probably not that obvious: The package has to generate formal proofs that the generated encoding of object-structures is a faithful representation of object-orientation (in the sense of the UML standard [OMG03b]). To ensure the conservativity (and thus consistency) of our approach we cannot prove these properties "once and for all" on the meta-level. Thus, these theorems have to be proven for each model during its encoding phase. Among many other properties, our package proves that for each pair of classes A and B where B inherits from A the following fact:

$$\frac{\text{self.oclIsType(B)}}{\text{self.oclIsKind(A)}} \tag{1}$$

as well as the more complicated property:

$$\frac{\text{self.oclIsDefined()} \quad \text{self.oclIsType(B)}}{\begin{array}{c}\text{self.oclAsType(A).oclAsType(B).oclIsDefined()}\\ \text{and self.oclAsType(A).oclAsType(B).oclIsType(B)}\end{array}} \tag{2}$$

```
fun cast_class_id class parent thy = let
  val pname = name_of parent
  val cname = name_of class
  val thmname = "cast_"^(cname)^"_id"
  val goal_i = mkGoal_cterm
        (Const(is_class_of class,dummyT)$Free("obj",dummyT))
        (Const("op␣=",dummyT)$(Const(parent2class_of class pname,dummyT)
          $(Const(class2get_parent class pname,dummyT)$Free("obj",dummyT)))
                                $(Free("obj",dummyT)))
  val thm = prove_goalw_cterm thy [] goal_i
          (λ p ⇒ [cut_facts_tac p 1,  (* proof script *)
                  asm_full_simp_tac
                     (HOL_ss addsimps
                            [o_def,
                             get_def thy (parent2class_of class pname),
                             get_def thy (class2get_parent
                             class pname )]) 1,
                  stac (get_thm thy (Name mk_get_parent)) 1,
                  asm_full_simp_tac (HOL_ss addsimps [
                        get_def thy (is_class_of class),
                        get_thm thy (Name ("is_"^pname^"_mk_"^(cname)))]) 1,
                  stac (get_thm thy (Name ("get_mk_"^(cname)^"_id"))) 1,
                  ALLGOALS(simp_tac (HOL_ss))])
in
  (fst(PureThy.add_thms [((thmname,thm),[])] (thy)))
end
```

Listing 5: Proving Cast and Re-Cast (simplified)

Listing 5 presents a simplified version of the SML function cast_class_id that proves the property shown in Equation 2. The expression starting in line 5 generates a type-checked instance of the theorem to prove with respect to the current class (and its parent). Readers familiar with LCF-style theorem provers will recognize the "proof script" in lines 10 to 23. Finally, the function registers the proven theorem in Isabelle's theorem database. Logical rules like those shown in Equation 1 or Equation 2 or co-induction schemes given by class invariants constitute the object-oriented datatype theory of a given class diagram and represent the basic weapon for proofs over them, in particular verifications of UML/OCL specifications.

One could also state these rules by adding corresponding axioms (i. e., unproven facts) during the encoding process, which is definitively easier to implement. Instead, our datatype package generates entirely conservative definitions and formally proves these rules from them; this also includes the definition of recursive class invariants, which are in itself not conservative ([BW06a] describes this construction in detail). This strategy ensures two very important properties:

1. our encoding fulfills the required properties, otherwise the proofs would fail, and
2. our encoding is consistent (provided that HOL is consistent and Isabelle/HOL is a correct implementation).

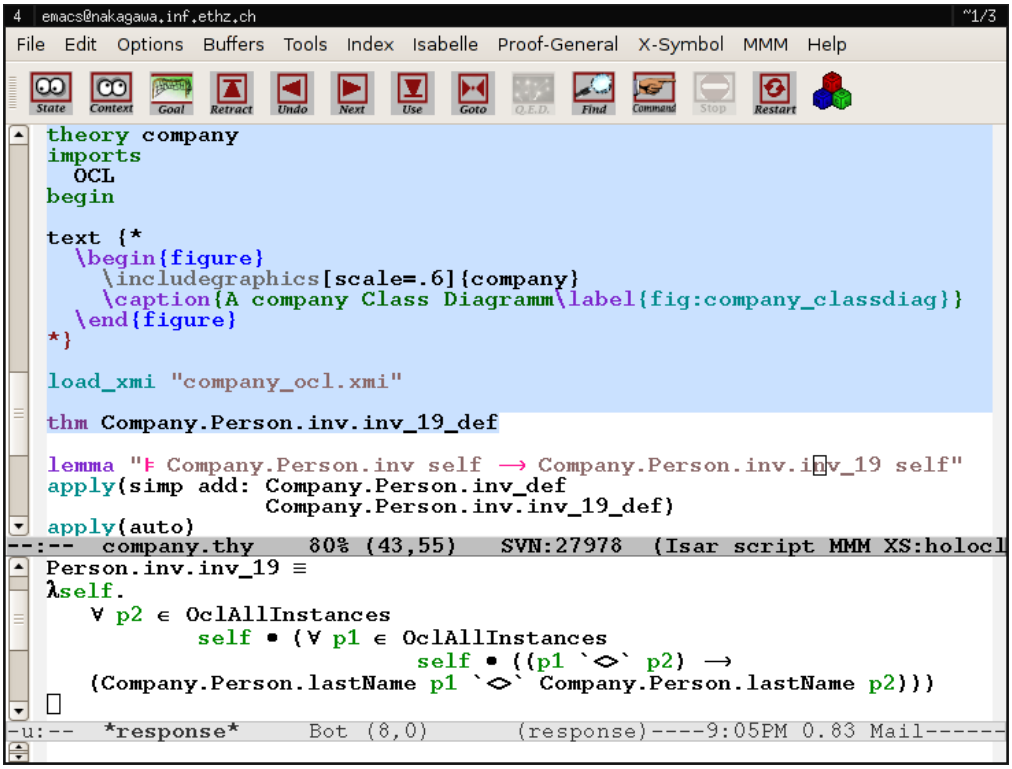

Figure 3: A HOL-OCL session Using the Isar Interface of Isabelle

One might ask what benefit an end-user will get from conservativity after all. Its need becomes apparent when considering recursive object structures or recursive class invariants. Stating recursive predicates as *axioms* can result in *logical inconsistency* in general. For example:

```
context A inv: not self.oclIsKind(A)
```

This invariant requires for all instances of type A not to be of kind A. Thus, it is in fact possible to state a variant of Russell's paradox, which leads to "invalid states" in the sense of the standard.

Our conservative construction requires proofs of side-conditions which will fail in paradoxical situations as the one discussed above (c.f. [BW06a] for details) while admitting the "useful" forms of recursion in class invariants. To get an idea for the amount of work involved, the import of the "Company" model (including the OCL specification) presented in the OCL standard [OMG03a, Chapter 7] generates 1147 conservative definitions and proven theorems, the larger "Royals and Loyals" model [WK03] generates 2472 conservative definitions and proven theorems. The load process usually proceeds in reasonable times.

Using HOL-OCL (see Figure 3) one can now formally prove that a given UML/OCL specification fulfills certain properties. For example, in the case of a SecureUML specification one can verify the generated security-related proof obligations, as described in Section 7. A simple analysis for class diagrams is shown in Figure 3: where the user is about to prove that one invariant of class Person implies another one. A more important property of class diagrams one

can show is consistency, i. e., there is at least one system state fulfilling all invariants, and there exist functions for all operation specifications satisfying the pre- and postconditions for legal states. Another important property is the refinement relation (e. g., forward-simulation [WD96]) between two class diagrams, stating that one model is a refinement of the other. A further interesting formal technique allows for proving that an implementation (i. e., a "method" in UML terminology) is compliant to a specification (i. e., a pair of pre- and postconditions). An in-depth discussion of these issues is out of the scope of this paper; with respect to the compliance problem, the reader might consult [BW06b].

6 SecureUML Support

As we want to not only support standard UML/OCL models, but also SecureUML models, we have to extend the framework accordingly. In this section, we describe these extensions.

6.1 SecureUML Support in the Model Repository

We extend the model repository to also contain model information from a SecureUML dialect.

```
signature REP_SECURE = sig
    structure Security : SECURITY_LANGUAGE
    type Model = Rep.Model * Security.Configuration
    val readXMI: string → Model
end
```

This means, a "secure" model not only contains an "unsecured" model, but also a security "configuration." The type of this configuration is parametrized by the concrete security language.

We currently have one implementation of this signature, corresponding to the SecureUML metamodel, i. e., the permissions are given in terms of RBAC with additional authorization constraints in OCL. The design allows for other security languages, for example, supporting privacy or usage control policies. The implementation of the security language is responsible for extracting the security model information from a UML/OCL model, where it is usually given by a custom UML profile, i. e., stereotypes and tagged values.

The security language is itself parametrized by a design language, i. e., a SecureUML dialect. The dialect specifies the actual resources and which actions are possible on these resources, together with corresponding hierarchies over them. We implemented this signature both for the ComponentUML as well as for the ControllerUML dialect of SecureUML.

6.2 SecureUML Support in the Code Generator

The generated code has to ensure that the specified access control policy is actually adhered to. This is in general done by generating access controlled wrapper functions around the sensitive operations. These wrapper functions implement the necessary access control checks by checking the assigned roles of the user and evaluating the associated authorization constraint, throwing an exception in case of no allowed access. In specific cases, for example code generation for certain

middleware platforms where the middleware already supports the configuration of access control policies, the code generator can also be used to generate the corresponding configuration files.

After the template files are adjusted as just described, we need to define a corresponding cartridge that uses the secured model repository. Implementing a cartridge mainly consists of deciding which "features" to support in the template language, i. e., which Boolean predicates, which lists, and which variables. As parts of this strongly depend on the SecureUML dialect, we implemented a SecureUML cartridge that again is parametrized by a SecureUML dialect. The SecureUML cartridge only knows about the global list of permissions, their assigned roles and constraints, which is information that is independent from the used dialect. The dialect specific cartridges then, e. g., deal with the assignment of actions to permissions.

6.3 SecureUML Support for HOL-OCL

At present, our datatype package for HOL-OCL supports SecureUML indirectly using the model transformation described in the following section. For the future, also first-class SecureUML support for HOL-OCL is planned. The development of this support requires:

- the development of a machine-readable, formal semantics for SecureUML, e. g., as an embedding into HOL-OCL. Similar to the already existing theories covering the UML core and OCL, we have to develop a set of theories covering the SecureUML entities and their properties. For example, the development of a generic theory summarizing role-based access control models.
- the extension of the existing datatype package with support for the new SecureUML theories, i. e., the package must be extended to generate definitions for SecureUML entities and, if possible, the generation of proof attempts for security related proof obligations, such as shown below.

7 Model Transformations

In this section, we show how our framework supports the definition of model transformations, and show how HOL-OCL can be used to analyze not only UML models, but also SecureUML models. Namely, we describe a particular model transformation, which converts a given SecureUML model into a semantically equivalent pure UML/OCL model:

```
structure SecureUML2HolOcl:sig
    val transform : Rep_SecureUML_ComponentUML.Model → Rep.Model
end = struct
...
fun transform (design_model,security_conf) =
    let
        val transformed_model = ... (* additional operations with     *)
                                    (* transformed postconditions     *)
        val auth_env          = ... (* classes like Identity and Role *)
    in
         transformed_model @ auth_env
    end
end
```

model element	generated operation with OCL constraints
Class C	**context** C::new():C **post**: result.oclIsNew() and result−>modifiedOnly()
	context C::delete():OclVoid **post**: self.oclIsUndefined() and self@pre−>modifiedOnly()
Attribute att	**context** C::getAtt():D **post**: result=self.att
	context C::setAtt(arg:D):OclVoid **post**: self.att=arg and self.att−>modifiedOnly()
Operation op	**context** C::op_sec(...):... **pre**: pre_{op} **post**: $\text{post}_{op}[\text{f}() \mapsto \text{f\_sec}(), \text{att} \mapsto \text{getAtt}()]$

Table 1: Overview of generated operations

Here Rep_SecureUML_ComponentUML is an implementation of the Rep_Secure signature for the ComponentUML dialect.

The basic idea of this model transformation is to translate every access control requirement in the SecureUML model into a postcondition of the corresponding operation that checks whether this operation is allowed or not. In more detail: If an operation is allowed, the postcondition shall remain unchanged (i. e., the system behaviour is identical to the unsecured one), otherwise the transformed postcondition specifies that calling this operation shall have no effect.

Considering this basic idea, one can deduce certain requirements on the model transformation. For example, the target UML model has to support the specification of OCL constraints that talk about *users* and their *roles*. Also, every access controlled "action" in the SecureUML model has to be represented by an UML operation in the target UML model.

The first requirement is quickly dealt with. Recall the definition:

```
type Rep_SecureUML_ComponentUML.Model = Rep.Model * Security.Configuration
```

For giving the UML model the possibility to talk about *users* and their *roles*, we add, corresponding to the Security.Configuration, appropriate classes like Identity and Role to the Rep.Model contained in the SecureUML model. Role assignments, and role hierarchies are then mapped to invariants of these classes.

The second requirement means that a number of operations have to be added to the classes in the UML model. In the SecureUML dialect we consider here, the access controlled actions are object creation and object destruction, reading or writing on attribute values, and executing operations. Therefore, for each class, attribute and operation in the UML model, the transformation generates operations according to Table 1.

In the transformation for operations, we do not simply copy the existing postcondition, but transform it by substituting operation and attribute calls with the calls to the likewise generated "secured" operations and accessor functions. This is necessary to ensure the security of the generated system: If the postcondition depends on access controlled information, the access control checks for them have to be fulfilled. Otherwise, the system could have a security leak.

Effect of the Security Model Transformation		
pre'_{op}	$\mapsto$	pre'_{op}
$post'_{op}$	$\mapsto$	let auth = $auth_{op}$in if auth then $post'_{op}$ else result.oclIsUndefined() and Set{}−>modifiedOnly() endif

Table 2: Overview of Transformed Constraints

In several of the generated postconditions we not only specify what changes, but also that "nothing else" happens during this operation call. Using standard OCL this is difficult or even impossible for arbitrary methods: one has to specify that the whole system stays unchanged except for this attribute. Therefore, HOL-OCL provides an extension of OCL for specifying frame properties within postconditions: Set(T)::modifiedOnly():Boolean. This allows for specifying explicitly the set of object instances that the system can change during a state transition. For details see [BW06a].

The main part, however, of the security model transformation is the generation of the security constraints for the operations generated during the design model transformation. The existing constraints on the generated operations are transformed according to Table 2.

Table 2 applies only to operations generated during the design model transformation. Thus, pre'_{op} and $post'_{op}$ refer to constraints generated during the transformation described in Table 1. As noted above, the pre-existing model elements of the design model are preserved. Only the postconditions are changed during this transformation, i. e., the invariants inv_C for classes C of the design model and the preconditions pre_{op} for access-controlled operations stay the same. The transformation wraps the postcondition generated during the design model transformation with an access control check using the authorization expression $auth_{op}$, which evaluates to true if access is granted, and false otherwise. If access is granted, the behavior of this operation will not be changed. Otherwise, the transformed postcondition ensures that no result is returned and the system state does not change.

The expression $auth_{op}$ is built in the following way: Let $perm_1, \ldots, perm_n$ be the permissions for this operation call, and let $roles_i$ be the set of roles, $constr_i$ be the authorization constraint associated with permission $perm_i$, and

$$\overline{constr_i} = (constr_i[\text{caller}\mapsto\text{ctxt.principal.identity.name}]) \\ [\text{f}() \mapsto \text{f@pre}(), \text{att} \mapsto \text{att@pre}, \text{aend} \mapsto \text{aend@pre}]$$

be the OCL expression where every occurrence of the non-standard keyword caller in $constr_i$ is substituted by the expression ctxt.principal.identity.name, which evaluates the name of the current caller using the authorization environment auth_env. Note that operation, attribute, and association end calls are substituted by their post-state equivalents. This is necessary because the authorization constraints are evaluated in a postcondition, i. e., after the operation has been called, whereas they have to hold before the operation is called. $auth_{op}$ is then defined as the following OCL expression:

```
auth_op := let perm1:Boolean = Set{<list of role names r ∈ roles_1>}
                               ->exists(s|ctxt@pre.principal@pre.isInRole@pre(s))
                               and constr_1
           -- analogous for perm2 to permn
           perm:Boolean = perm1 or perm2 or ... permn
         in perm.oclIsDefined() and perm
```

This definition explicitly checks the authorization expression for undefinedness, mapping it to false if it is undefined. This is necessary because undefinedness can be caused by user-specified authorization constraints, which form a part of auth_{op}.

An interesting question to ask now is whether this transformation may turn consistent design models into inconsistent ones, or not. In [BDW06] we answered this question: if the original design model (without security policy) is consistent, and the following security proof obligation spo_{op} holds for all generated operations, then the resulting model is also consistent:

$$\text{spo}_{op} := \text{auth}_{op} \text{ implies post}_{op} \triangleq \text{post}_{op}[\text{f}() \mapsto \text{f\_sec}(), \text{att} \mapsto \text{getAtt}()]$$

The transformation additionally generates these security proof obligations, which can then be proven in HOL-OCL. These proof obligations require that whenever someone has access for an operation (auth_{op} holds), then the postcondition for this operation does not depend on whether the original attributes and operation calls are used, or the access controlled variants. This essentially requires that the caller also has sufficient access rights to "evaluate" the postcondition. If the postcondition only contains operation calls to other operations, the proof boils down to the check that the caller has sufficient permissions for these operations. However, when the postcondition contains recursive calls to itself, the proof requires an inductive argument.

8 Conclusion

We have presented a framework for MDA comprising OCL support in model transformations, code generation and verification, together with one application of such a combined framework, namely SecureUML. In our framework, model transformations can be implemented that produce logical proof-obligations which can be tested or formally verified. In a way, our work can be seen as an approach to extend MDE by model-driven formal reasoning.

The code generator is a template-based generator which can be easily configured to produce code for various parts of models, target languages and application scenarios that range from runtime-test and injection of security mechanisms, from the generation of web-based services to the infra-structure for distributed component platforms. The techniques in itself are by no means new, but having them integrated into a framework and having access to OCL will, in our view, pave the way for new and up to now unexpected applications.

8.1 Related Work

Since code generation is at the heart of model-driven engineering, there is a wealth of similar approaches, e. g., AndroMDA (http://www.andromda.org/), which itself is based on Veloc-

ity (http://jakarta.apache.org/velocity/). While Velocity provides a rich template language with (among others) support for arithmetical, relational and logical operators over user-definable variables, our template language is intentionally simple and restricted, but provides an @eval construct allowing for the execution of arbitrary SML code. Another difference lies in our concept of cartridges: Velocity supports cartridges only in form of an unstructured merging of the "context" object(s). In contrast, our cartridge concept is hierarchically structured, which is type-checked on the SML module level.

There are only few proof environments for OCL; most notably the KeY Tool [ABB+05]. It offers a concrete verification method for a Java-like language (which HOL-OCL does not at present) at the dispense of compliance to the semantic foundations of OCL—the underlying semantics is a two-valued dynamic logic with an axiomatic representation of the data-models resulting from class diagrams.

With UMLsec [Jür04] we share the conviction that security models should be integrated into the software engineering development process by using UML. However, although UMLsec provides a formal semantics, it does only provide rudimentary tool support, both for code generation and for (formal) model analysis.

8.2 Lessons Learned

8.2.1 Using Functional Programming Languages.

Using a functional programming language for an object-oriented data model (e. g., the UML meta model) has advantages and disadvantages: on the one hand, a direct compilation into SML datatypes, leads to a quite substantial duplication of code for the inherited attributes and possibly in the pattern matching based functions processing these data structures. Nevertheless, the advantages of type-safe pattern matching over constructors outweigh by far the disadvantages, in particular in comparison to untyped script-like transformation languages such as XSLT.

We have been very pleased by the degree of abstraction and re-usability that has been achieved in the code generator by using the SML functor concept. To our knowledge, this is the first time that it had been applied to the concept of cartridges, which allows for a type-safe and aspect-oriented way to describe the compilation process.

8.2.2 Building a Toolchain.

Our toolchain for UML/OCL depends on the common XMI format for tool exchange. This has been the key for re-using work of other research groups in the field. However, in practice, each tool uses slightly different variants of the underlying meta-model, and thus different XMI variants. By having an infrastructure based on a general XML parser and pattern matching-based conversions between an imported XMI and the internal su4sml model repository, it turned out to be a fairly easy routine task to adapt to various XMI dialects.

8.3 Future Work

One obvious limitation of our current implementation is the lack of an XMI export from su4sml allowing re-import into ArgoUML: This would pave the way for a re-interpretation and a re-

visualization of the results of model-transformations done on su4sml. Usually, the transformed (and expanded) models are less abstract and lengthy, and modifications of the re-engineered models by hand are not really compatible with the MDE methodology which emphasizes a tool-based generation process.

It remains the question how to deal with injected, hand-written code which can be part of a model in MDE. There are essentially two options: a code-verification approach or a systematic test approach.

With respect to the code-verification, a Hoare Calculus has already been derived for OCL for a simple imperative language [BW06b]. To allow code-verification for programs of at least modest size effectively, however, a verification condition generator has to be added and the degree of automation of tailored proof-tactics must be increased.

With respect to the testing approach, it is tempting to adapt a test-case generation system to OCL. Two of the authors developed the test-data-generator HOL-TESTGEN based on specifications in HOL, which has been used for substantial case-studies in black-box unit test [BW04], gray-box unit test [BW05] and sequence test [BW07]. HOL-TESTGEN and HOL-OCL are both built on the same technical and logical platform, namely Isabelle/HOL. However, the technical challenge of an adaption of HOL-TESTGEN consists in basing its partitioning component (computing symbolically a normal form called TNF; see [BW04] for details) on a three-valued logic such as OCL.

References

[ABB+05] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, P. H. Schmitt. The KeY Tool. *Software and System Modeling* 4:32–54, 2005.
doi:10.1007/s10270-004-0058-x

[BDL06] D. Basin, J. Doser, T. Lodderstedt. Model Driven Security: from UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology* 15(1), January 2006.
doi:10.1145/1125808.1125810

[BDW06] A. D. Brucker, J. Doser, B. Wolff. A Model Transformation Semantics and Analysis Methodology for SecureUML. In Nierstrasz et al. (eds.), *MoDELS 2006*. LNCS 4199, pp. 306–320. Springer, 2006.
doi:10.1007/11880240_22

[BS05] B. Beckert, S. Schlager. Refinement and Retrenchment for Programming Language Data Types. *Formal Aspects of Computing* 17(4):423–442, 2005.
doi:10.1007/s00165-005-0073-x

[BW04] A. D. Brucker, B. Wolff. Symbolic Test Case Generation for Primitive Recursive Functions. In Grabowski and Nielsen (eds.), *Formal Approaches to Testing of Software*. LNCS 3395, pp. 16–32. Springer-Verlag, 2004.
doi:10.1007/b106767

[BW05] A. D. Brucker, B. Wolff. Interactive Testing using HOL-TESTGEN. In Grieskamp and Weise (eds.), *Formal Approaches to Testing of Software*. LNCS 3997, pp. 87–102. Springer-Verlag, Edinburgh, 2005.
doi:10.1007/11759744_7

[BW06a] A. D. Brucker, B. Wolff. The HOL-OCL Book. Technical report 525, ETH Zürich, 2006.

[BW06b] A. D. Brucker, B. Wolff. A Package for Extensible Object-Oriented Data Models with an Application to IMP++ . In Roychoudhury and Yang (eds.), *International Workshop on Software Verification and Validation (SVV 2006)*. Aug. 2006.

[BW07] A. D. Brucker, B. Wolff. Test-Sequence Generation with HOL-TESTGEN – With an Application to Firewall Testing. In *International Conference on Tests and Proofs*. 2007. to appear.

[Jür04] J. Jürjens. *Secure Systems Development with UML*. Springer, 2004.
doi:10.1007/b137706

[Mel92] T. F. Melham. A Package for Inductive Relation Definitions in HOL. In Archer et al. (eds.), *The HOL Theorem Proving System and its Applications*. Pp. 350–357. IEEE Computer Society Press, 1992.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[OMG03a] UML 2.0 OCL Specification. Oct. 2003. Available as OMG document ptc/2003-10-14.

[OMG03b] OMG Unified Modeling Language Specification, v1.5. Mar. 2003. Available as OMG document formal/03-03-01.

[Pau96] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman. Role-Based Access Control Models. *IEEE Computer* 29(2):38–47, Feb. 1996.
doi:10.1109/2.485845

[WD96] J. Woodcock, J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.

[WK03] J. Warmer, A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2nd edition, Aug. 2003.