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 an MDA framework, developed in the functional programming language SML, that tries to bridge the gap between formal software development and the needs of industrial software development, e.g., code generation. Overall, our toolchain provides support for software modeling using UML/OCL and guides the user from type-checking and model transformations to code generation and formal analysis of the UML/OCL model. We conclude with a report on our experiences in using a functional language for implementing MDA tools.

1 Introduction

Model-Driven Engineering refers to the systematic use of models as primary engineering artifacts throughout the development life-cycle of software systems. 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 Model-Driven Engineering in general and MDA in particular, *technical support* 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. Thus, a technical framework is needed that provides an infrastructure for model elements annotated by OCL.

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 and access control policies specified using SecureUML. Further, this model can be directly transformed into a (formal) model for the theorem proving environment HOL-OCL [4].

As a distinguishing feature, su4sml is developed in the functional programming language SML [11]. For this reason, implementers of model transformations can profit from several techniques that have proven to be of major importance

B. Demuth, D. Chiorean, M. Gogolla, J. Warmer (Eds.): OCL for (Meta-)Models in Multiple Application Domains, TUD-FI06-04-Sept. 2006, University Dresden, Germany, pp. 166–181, 2006.

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 [2]. SecureUML is designed to enrich the business logic of a system (represented by a class diagram or a statechart) with a concrete access control model for objects and operations. By a model transformation [3], 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 business logic 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 a template-based code generator and in section 5 we describe the interface to HOL-OCL. Finally, we describe the SecureUML instance and discuss our experiences and observations.

2 Our Framework: An Overview

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

2.1 Background

SecureUML. SecureUML [2] is a security modeling language based on RBAC [12]. 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*.

The Dresden OCL2 Toolkit. The software platform provided by the Dresden OCL2 Toolkit (`http://dresden-ocl.sf.net/`), written in Java, provides manifold support for OCL. Among other tools, a parser and type-checker for OCL is included. The toolkit is designed for modularity and flexibility. Thus, the Dresden OCL2 Toolkit is a good basis for building new OCL-based tools, either by integrating it into a CASE tool directly or by using it as a standalone tool leveraging the provided XMI import and export facilities. In our setting, we especially benefit from the XMI export, which includes the typed-checked OCL constraints as abstract syntax using an XML-based encoding.

HOL-OCL. HOL-OCL [4] (`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 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 [8] while following the standard [9] 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 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.

2.2 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.

We also developed a Java-based transformation tool, *su2holocl*, on top of the Dresden OCL Toolkit which transforms a SecureUML model into a semantically identical pure UML/OCL model. This model transformation is explained in more detail elsewhere [3].

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),

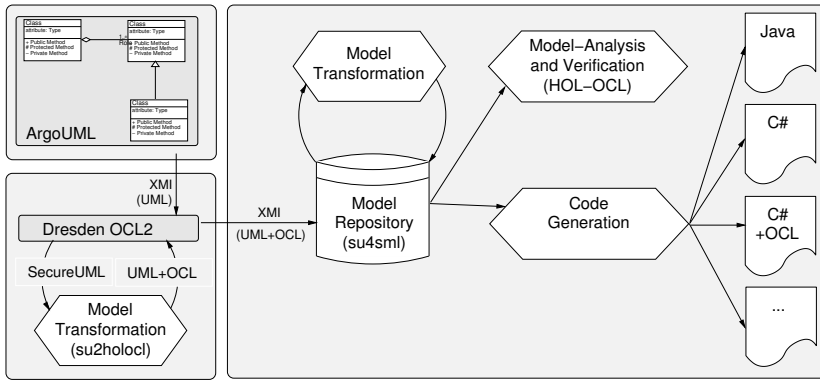

Figure 1. MDA Framework and Toolchain Overview

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 SecureUML (including the generation of access-control checks for the target languages Java and C#), the UML core (e.g., class diagrams), state machines, and OCL,
4. model transformations that normalize the models in several normal forms; this comprises the conversion of multiplicities into OCL constraints, etc., and
5. 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.

3 The Model Repository: su4sml

When implementing an object-oriented model repository in a functional programming language one has to solve several challenges: first one has to decide how to represent the inherently graph-based structure of object-oriented models into a tree-structure that is suitable in a functional programming language. Of course, one can always simulate pointers, but then one loses convenient features of functional programming languages, like safeness and strong typing.

For class models, we decided to employ the inherent tree structure given by the "containment hierarchy." For example, a class contains attributes, operations, or statemachines. We also decided to ignore associations as such. We only represent their association ends, again as part of the participating classifiers.

Statemachines, however, do not present an obvious way of representation in a tree structure. There we fall back to using pointers, for example from transitions to source and target states, or from states to incoming and outgoing transitions.

In contrast, OCL expressions naturally translate into an abstract datatype, as shown in Listing 1.1 and Listing 1.2. This abstract datatype is modeled closely

```
signature REP_OCL_TYPE = sig

  type Path = string list

  datatype OclType = Integer | Real | String | Boolean      (* Primitive Types *)
                   | OclAny | OclVoid
                   | Set of OclType | Sequence of OclType
                   | OrderedSet of OclType | Bag of OclType
                   | Collection of OclType
                   | Classifier of Path           (* user-defined classifiers *)
                   | DummyT              (* dummy type for untyped expressions *)
end
```

Listing 1.1. su4sml: Representing OCL Types

```
signature REP_OCL_TERM = sig
include REP_OCL_TYPE

datatype OclTerm =
     Literal            of string * OclType              (* Literal with type  *)
   | CollectionLiteral  of CollectionPart list * OclType (* content with type  *)
   | If                 of OclTerm * OclType             (* condition          *)
                         * OclTerm * OclType             (* then               *)
                         * OclTerm * OclType             (* else               *)
                         * OclType                       (* result type        *)
   | AssociationEndCall of OclTerm * OclType             (* source             *)
                         * Path                          (* assoc.-enc         *)
                         * OclType                       (* result type        *)
   | AttributeCall      of OclTerm * OclType             (* source             *)
                         * Path                          (* attribute          *)
                         * OclType                       (* result type        *)
   | OperationCall      of OclTerm * OclType             (* source             *)
                         * Path                          (* operation          *)
                         * (OclTerm * OclType) list      (* parameters         *)
                         * OclType                       (* result tupe        *)
   | OperationWithType  of OclTerm * OclType             (* source             *)
                         * string * OclType              (* type parameter     *)
                         * OclType                       (* result type        *)
   | Variable           of string * OclType              (* name with type     *)
   | Let                of string * OclType              (* variable           *)
                         * OclTerm * OclType             (* rhs                *)
                         * OclTerm * OclType             (* in                 *)
   | Iterate            of (string * OclType) list       (* iterator variables *)
                         * string * OclType * OclTerm    (* result variable    *)
                         * OclTerm * OclType             (* source             *)
                         * OclTerm * OclType             (* iterator body      *)
                         * OclType                       (* result type        *)
   | Iterator           of string                        (* name of iterator   *)
                         * (string * OclType) list       (* iterator variables *)
                         * OclTerm * OclType             (* source             *)
                         * OclTerm * OclType             (* iterator-body      *)
                         * OclType                       (* result type        *)
and CollectionPart = CollectionItem  of OclTerm * OclType (* element with type  *)
                   | CollectionRange of OclTerm           (* first              *)
                                      * OclTerm           (* last               *)
                                      * OclType           (* type of range      *)
end
```

Listing 1.2. su4sml: Representing OCL Expressions

```
signature REP_CORE = sig
type Scope
type Visibility
type operation =      { name          : string,
                        precondition  : (string option * OclTerm) list,
                        postcondition : (string option * OclTerm) list,
                        arguments     : (string * OclType) list,
                        result        : OclType,
                        isQuery       : bool,
                        scope         : Scope,
                        visibility    : Visibility }

type associationend = { name          : string,
                        aend_type     : OclType,
                        multiplicity  : (int * int) list,
                        ordered       : bool,
                        visibility    : Visibility,
                        init          : OclTerm option }

type attribute =      { name          : string,
                        attr_type     : OclType,
                        visibility    : Visibility,
                        scope         : Scope,
                        stereotypes   : string list,
                        init          : OclTerm option }

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

Listing 1.3. su4sml: Representing the UML Core

following the standard OCL 2.0 metamodel. Note however, that OCL expressions include a lot of type information in this model. In essence, the type of each subexpression appears twice: once as the type of the subexpression itself, and once as the type expected (or inferred) as part of the enclosing expression. This constructions allows us, for example, to insert explicit typecasts that are only implicit in the original expression.

In addition to these datatype definitions, the repository structure defines a couple of normalization functions, for example for converting association ends into attributes with corresponding type, together with an invariant expressing the cardinality constraint.

Summarizing, the top-level data structures (see Listing 1.1, Listing 1.2 and Listing 1.3) of su4sml are inspired by the metamodels of OCL [9, Chapter 8] and UML [10] and readers familiar with these metamodels should recognize the similarities.

```
@// 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 1.4. A Simplified Template File

4 A Template-Based Code Generator

We developed a *Generic Template-driven 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. 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. 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 attribute_list, for example, evaluates to the list of attributes of 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 1.4 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

```
signature GCG = sig
  val generate  : Rep.Model → string → unit
end

functor GCG_Core (C: CARTRIDGE): GCG  = struct
    (* misc. auxiliary functions omitted *)

fun generate model template
  = let val env  = C.initEnv  model
        val tree = parse template
        in
            (initOut();
             write env tree;
             closeFile () )
             handle GCG_Error ⇒ (closeFile(); raise GCG_Error)
        end
end
```

Listing 1.5. GCG: the generic code generator

the predicate operation_isStatic. Depending on the answer, the core executes the following statements or not.

On the implementation level, the core is a functor which takes a cartridge as an argument (see Listing 1.5). The functor GCG_Core only takes one cartridge as an argument, whereas we want to be able to plug arbitrarily many cartridges together (see Figure 2). We achieve this by letting each cartridge be a functor itself, which takes another cartridge as an argument. In this way, we can 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. To bootstrap this cartridge chain, we start with a cartridge that is not a functor. This could for example be a trivial cartridge that simply does nothing. For convenience, however, 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 of lists like operation_list. The design allows for cartridges to override these functionalities by implementing them themselves. This is sometimes necessary for language-specific cartridges when the language requires certain syntactic properties. We implemented cartridges for Java and C# in this way.

To get a Java code generator, for example, one has to plug the cartridges together like follows:

```
structure Java_Gcg = GCG_Core (Java_Cartridge(Base_Cartridge));
```

The functor GCG_Core is applied over the cartridge resulting from the application of the functor Java_Cartridge over the base cartridge. The resulting structure Java_Gcg implements the signature GCG and therefore has a function generate which generates Java code from a given UML-model and a given template.

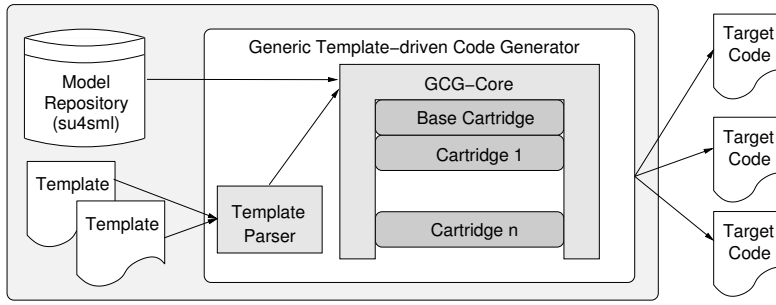

Figure 2. The Cartridge Chain Architecture

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

Listing 1.6. The Top-level Interface of the Repository Encoder

5 A su4sml-based Datatype Package for HOL-OCL

In this section, we present one vital component of HOL-OCL concerned with the encoding of object-oriented data structures in HOL, which is a tedious and error-prone activity to be automated. 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* [7] 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. prove as much (interesting) properties over the generated definitions as possible automatically behind the scenes.

Our datatype package uses the possibility to build SML programs performing symbolic computations over formulae in a logically safe way over derived rules.

In the following, we give a brief overview what our package does ([4,5] describes more details). The datatype package is implemented on top of the su4sml interface on one hand and on top of the Isabelle core on the other (see Listing 1.6 for details). During the encoding, our datatype packages extends the given theory by a HOL-OCL-representation of the given UML/OCL model. This is done in an extensible way, i.e., classes can be added later on to an existing theory preserving all proven properties ([5] presents for 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.

```
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 1.7. Proving Cast and Re-Cast (simplified)

In fact, 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 (e.g., in the sense of the UML standard [10], or Java). 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 is a generalizationof A the following fact:

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

as well as the more complicated property:

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

Listing 1.7 presents a simplified version of the SML function cast_class_id that proves the property (2). The expression starting in line 5 generates a type-checked instance of the current 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 (1) or (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. Stating these rules could be achieved by adding axioms (i.e., unproven facts) during the encoding process, which is definitively easier to implement. Instead, our datatype package generates entirely conservative definitions and derives these rules from them; this also includes the definition of recursive class invariants, which are in itself not conservative ([4] describes this construction in detail).

This strategy, i.e., stating entirely conservative definitions and formally proving the datatype properties for them, ensures two very important properties:

1. our encoding fulfills the required properties, otherwise the proofs would fail, and
2. doing all definitions conservatively together with proving all properties ensures the consistency of our model (provided that HOL is consistent and Isabelle/HOL is a correct implementation).

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 *axiom* results in *logical inconsistency* in general. For example:

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

This invariant requires for all instances of type A not to be of type A. Thus, it is in fact possible to state a variant of Russell's paradox which is known to introduce logical inconsistency in naive set theory. Inconsistency means that the OCL logic can derive any fact; this might be exploited by an automated tactic accidentally. Logical inconsistency is different from an *unsatisfiable* class invariant meaning "there is no instance." In particular, in an inconsistent system, each class invariant can be proven both satisfiable and unsatisfiable.

Our conservative construction requires proofs of side-conditions which will fail in paradoxical situations as the one discussed above (c.f. [4] for details) while admitting the "useful" forms of recursion in class invariants. To get an idea for the amount of work needed, the import of the "Company" model (including the OCL specification) presented in the OCL standard [9, Chapter 7] generates 1147 conservative definitions and proven theorems, the larger "Royals and Loyals" model [13] model generates 2472 conservative definitions and proven theorems. The load process usually proceeds in reasonable times.

Using HOL-OCL (see Figure 3) one can formally prove certain properties of UML/OCL specifications. For SecureUML specifications one can generate security-related proof obligations that can be formally analyzed, the details how and which proof obligations are generated is described elsewhere [3]. An example for an important standard property of a class diagram 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) of a model. Another important property is the refinement relation (e.g., forward-simulation [14]) 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

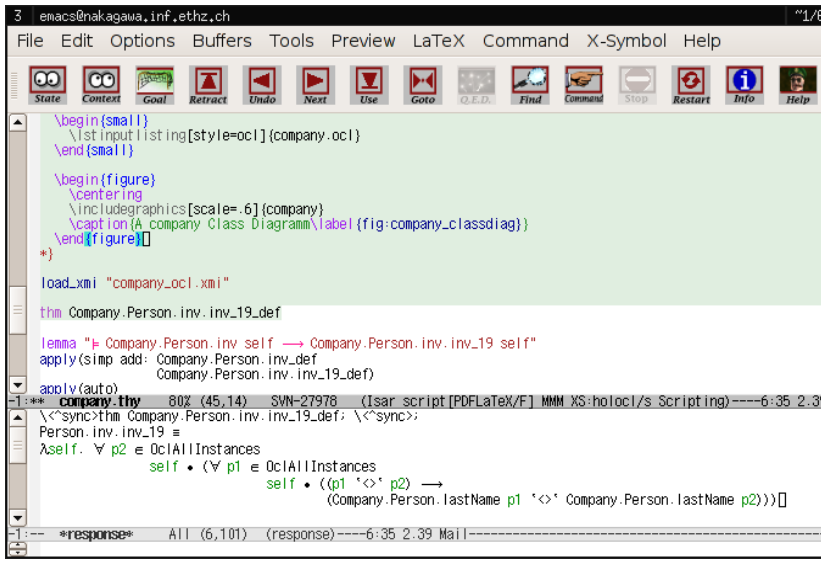

Figure 3. A HOL-OCL session Using the Isar Interface of Isabelle

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 [5].

6 SecureUML Support

As we want to not only support standard UML/OCL models in our framework, but also SecureUML models, we have to extend the framework accordingly. We describe these extensions in the following sections.

6.1 SecureUML Support in the Model Repository

First, we have to extend the model repository to also contain model information coming from a SecureUML dialect.

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

This means, a “secure” model not only contains a list of classifiers (like the unsecured model), but also a security “configuration.” The type of this configuration is parametrized by the concrete security language.

```
signature SECURITY_LANGUAGE = sig
    structure Design : DESIGN_LANGUAGE

    type Configuration
    eqtype Permission

    val getPermissions : Configuration → Permission list

    (* misc. auxiliary functions omitted *)

    val parse: Classifier list → (Classifier list * Configuration)
end
```

We currently have only one implementation of this signature, corresponding to the SecureUML metamodel, i.e., the permissions are given in terms of RBAC with additional authorization constraints. This design allows for other security languages, for example, a hypothetical PrivacyUML language. The function parse 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., by a concrete SecureUML dialect.

```
signature DESIGN_LANGUAGE = sig
    eqtype Resource
    datatype Action = SimpleAction of string * Resource
                    | CompositeAction of string * Resource

    (* The resource hierarchy *)
    val contained_resources : Resource → Resource list

    (* the action hierarchy *)
    val subordinated_actions:   Action → Action list

    (* misc. auxiliary functions omitted *)

    val parse_action: Classifier → attribute → Action
end
```

The dialect specifies the actual resources and actions that are possible on these resources, together with the corresponding hierarchies over them. We implemented this signature both for the ComponentUML as well as for the ControllerUML dialect of SecureUML. Note the function parse_action, which is responsible for parsing the attributes of permission classes.

6.2 SecureUML Support in the Code Generator

After the repository has been extended, for code generation purposes we only need to define a corresponding cartridge. 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 only indirectly using an external model transformation, su2holocl [3]. This model transformation converts a given SecureUML model into a semantically equivalent pure UML/OCL model. For the future, 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 security related proof obligations, together with proof attempts.

7 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 a way, our work can be seen as an approach to extend MDA with 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 target runtime-environments. The technique in itself is by no means new, but having it integrated into our framework and having access to structured OCL will, in our view, pave the way for new and up to now unexpected applications.

7.1 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 Velocity (`http://jakarta.apache.org/velocity/`). Besides the fact that we apply functional programming techniques, there are two main differences: first, Velocity provides a rich template language with (among others) support for arithmetical, relational and logical operators over user-definable variables. Instead, our template language is intentionally very simple and restricted, but provides an `@eval` construct allowing for the execution of arbitrary SML code. The second difference lies in our concept of cartridges. Since a fixed, static template language is not flexible enough for generic code-generation, a template engine has to provide some support for extensibility. Velocity supports this by customizing and unstructured merging of the "context" object(s). In contrast, our concept of cartridges supports a notion of hierarchy and dependency between cartridges, which is type-checked on the SML module level. Our

cartridges also do not entail the complexity and overhead of AndroMDA cartridges, which include not only the template vocabulary, but also model-facades for the UML profile, and the template files themselves. Keeping these separated both simplifies the development of new cartridges and proves to be more flexible.

There are also some proof environments for OCL; since we focus on tool aspects and integration into MDA in this paper, we only mention the KeY Tool [1]. 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 [6] 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.

7.2 Lessons Learned

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, i.e., mapping classes (with attributes) to constructors over records (with corresponding fields), leads to a quite substantial duplication of code for the inherited attributes and possibly in the pattern matching based functions processing these data structures. This representation of data models can be generated automatically from class diagrams via code generators such as our own (thus overcoming typical errors due to duplication). Nevertheless, pattern matching over constructors has to be designed and prepared with care to be extensible. For example, selector functions of inherited attributes like:

```
fun get_name (Class{name, ... })       = name
  |get_name (Interface{name, ...})     = name
  |get_name (Enumeration{name, ...}) = name
  |get_name (Primitive{name, ...})     = name
```

are sometimes preferable to pattern matching constructs since they are more stable under extensions; on the other hand, representing patterns only as selector and test functions, is feasible, but tedious and in itself very lengthy and error-prone. Thus, finding a suitable balance of re-usability and conciseness in each situation is the key for success.

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. For example, the SML-structure containing the code generator for C# with SecureUML is constructed by the functor application:

```
1 structure CSharpSecure_Gcg
            = GCG_Core (SecureUML_Cartridge(CSharp_Cartridge(Base_Cartridge)),
                                            ComponentUML(Base_Cartridge));
```

which just represents it as a combination of the various compilation aspects.

Building a Toolchain. Our toolchain depends on a common XMI format for exchanging UML/OCL models. 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. Full exchangeability of XMI files between different tools (and versions thereof) is still more a dream than reality. On the other hand, 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. Such adaptions had been necessary several times during the lifetime of our project and could be realized usually in one day of programming work and turned out to be easier in practice than, developing and maintaining appropriate XSLT-transformations.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
2. 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), 2006.
3. A. D. Brucker, J. Doser, and B. Wolff. A model transformation semantics and analysis methodology for SecureUML. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, eds., *MoDELS 2006*, no. 4199 in LNCS, pp. 306–320. Springer, 2006.
4. A. D. Brucker and B. Wolff. The HOL-OCL book. Tech. Rep. 525, ETH Zürich, 2006.
5. A. D. Brucker and B. Wolff. A package for extensible object-oriented data models with an application to IMP++ . In A. Roychoudhury and Z. Yang, eds., *SVV 2006*, Computing Research Repository (CoRR). 2006.
6. J. Jürjens. *Secure Systems Development with* UML . Springer, 2004.
7. T. F. Melham. A package for inductive relation definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, eds., *The* HOL *Theorem Proving System and its Applications*, pp. 350–357. IEEE Computer Society Press, 1992.
8. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/*HOL*—A Proof Assistant for Higher-Order Logic*, LNCS , vol. 2283. Springer, 2002.
9. UML 2.0 OCL specification. 2003. `ptc/2003-10-14`.
10. OMG Unified Modeling Language Specification. 2003. `formal/03-03-01`.
11. L. C. Paulson. ML *for the Working Programmer*. Cambridge Press, 1996.
12. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.
13. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for* MDA . Addison-Wesley, 2nd ed., 2003.
14. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.