Formal verification of QVT transformations for code generation

Kurt Stenzel, Nina Moebius, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Stenzel, Kurt, Nina Moebius, and Wolfgang Reif. 2015. "Formal verification of QVT transformations for code generation." Software & Systems Modeling 14 (2): 981–1002. https://doi.org/10.1007/s10270-013-0351-7.

Formal verification of QVT transformations for code generation

Kurt Stenzel · Nina Moebius · Wolfgang Reif

Abstract We present a formal calculus for operational QVT. The calculus is implemented in the interactive theorem prover KIV and allows to prove properties of QVT transformations for arbitrary meta models. Additionally, we present a framework for provably correct Java code generation. The framework uses a meta model for a Java abstract syntax tree as the target of QVT transformations. This meta model is mapped to a formal Java semantics in KIV. This makes it possible to formally prove (interactively) with the QVT calculus that a transformation always generates a Java model (i.e. a program) that is type correct and has certain semantical properties. The Java model can be used to generate source code by a model-to-text transformation or byte code directly.

Keywords Correctness of model transformations · QVT · Formal verification · Interactive theorem proving

1 Introduction

Model-driven development holds the promise to create better software in shorter time since the modeler can concentrate on the essential properties of the application under development. Technical details will be filled in by model transformations. In specialized areas, the complete source code of an application can be generated from the model.

However, there is the question of how to prove the correctness of the generated code, or—more broadly speaking—of the model transformations. This is a largely unsolved problem. Since transformations are highly versatile, there is no universal notion of correctness (see e.g. [9,31] for surveys). We are mainly concerned with *semantic conformance* [43] and *model level semantic correctness* [31], i.e. that every target model of a transformation has certain semantic properties. Usually a formal semantics of the source and target model is needed, which is definitely not trivial if, for example, UML activity diagrams are transformed into Java code. Second, the transformation must be proved correct. This in turn requires a formal logic and proof support for the transformation language. Third, the transformations itself can be large and complex which makes a formal proof difficult and time-consuming.

In this paper, we present a framework that allows to prove properties of generated Java code for transformations written in operational QVT (QVTO [38]). The source meta model is not fixed even though the example will use UML class diagrams. This means it is possible to prove semantic conformance (the Java code has the same behavior as the source model w.r.t. a semantics for the source). But we are also interested in transformations that *add* behavior to the Java code that is not present in the source—in our example setter and getter methods while the source contains only UML classes and attributes. In this case, we can verify properties that are only expressed in terms of the Java model and do not refer to the source model at all. This work is part of our SecureMDD approach [33,34], a model-driven development method for security-critical applications based on cryptographic protocols. The application, e.g. an electronic purse or a ticketing system, is modeled with UML extended with a profile and

K. Stenzel (✉) · N. Moebius · W. Reif
Institute for Software and Systems Engineering,
Augsburg University, 86135 Augsburg, Germany
e-mail: stenzel@informatik.uni-augsburg.de

N. Moebius
e-mail: moebius@informatik.uni-augsburg.de

W. Reif
e-mail: reif@informatik.uni-augsburg.de

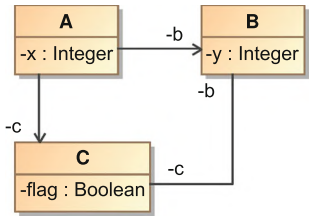

Fig. 1 A simple UML class diagram used for the generation of Java code

```
1  public class A {
2      private B b;
3      private int x;
4      private C c;
5
6      public B getB() { return this.b; }
7      public int getX() { return this.x; }
8      public C getC() { return this.c; }
9
10     public void setB(B b) { this.b = b; }
11     public void setX(int x) { this.x = x; }
12     public void setC(C c) { this.c = c; }
13 }
```

Fig. 2 Java class A generated from the UML class diagram

an abstract programming language MEL. From the model, a formal specification can be generated, and the security of the application can be proved [35,36]. Additionally, the complete Java code for the protocols can be generated by model transformations. In this setting, it is imperative that the code is correct and secure with respect to the formal specification (i.e. a refinement that preserves security [20]).

In the next section, the framework for correct code generation will be presented (Sect. 2) followed by a simple example that generates Java classes from a UML class diagram (Sect. 3). The next two sections describe the formal treatment: Sect. 4 shows how models and meta models can be formalized in a theorem prover, and Sect. 5 presents an overview of the calculus to prove properties of QVTO transformations. Section 6 discusses some experiences, Sect. 7 presents related work, and Sect. 8 concludes. All formal specifications and proofs can be found on our Web page.[1]

2 A framework for correct code generation

We illustrate our approach with a small example. The idea is to generate Java classes with fields, getters, and setters from simple UML class diagrams as shown in Fig. 1. The result are three Java classes A, B, and C. Class A (Fig. 2) has two navigable associations with B and C, and will have two fields of type B and C. The association between B and C is navigable in both directions, so B will have a field of type C, and C a field of type B. The UML primitive type `Boolean` is translated to `boolean`, and `Integer` to `int`. (Mapping unbounded UML integers to bounded Java ints is a deliberate decision by the transformation engineer, because she knows that only small numbers will be used). Other, more complicated methods like object serialization can be generated in the same manner as shown below.

The aim is to formally prove properties about the generated Java code, e.g.,

1. The code is type correct.
2. Correspondence to the UML model: One Java class for every UML class, one Java field for each UML Property, etc.
3. Semantic properties of the generated methods: A getter returns the value of a field, calling a setter, then the corresponding getter returns the same reference, etc.

Formal treatments of Java (either formal semantics of Java, e.g. [51,54,59] or Java calculi for program verification, e.g. [4,23,53]) all work on an annotated abstract syntax tree of a Java program. This makes sense since parsing Java text and annotating an abstract syntax tree are problems of compiler correctness that pose very different challenges, and should be separated from a Java semantics. If we use a model-to-text transformation to generate Java code from the UML class diagram, we essentially concatenate strings. For example, generating the getter methods in XPand [61] looks like

```
«FOREACH this.ownedAttribute AS a»
 public «a.type.toJavaType()» get«a.name.toFirstUpper()»(){
    return this.«a.name»; }
«ENDFOREACH»
```

Formal reasoning about the text requires parsing this mixture of source text and quoted expressions, annotating the resulting syntax tree, and then reasoning about its semantics, thereby mixing the different problems (parsing, annotating, and the dynamic behavior of the program). Our framework introduces a meta model that represents a Java annotated

[1] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/. The direct link is http://swt.informatik.uni-augsburg.de/kiv/projects/secureMDD/uml2jastSimple/index.xml.

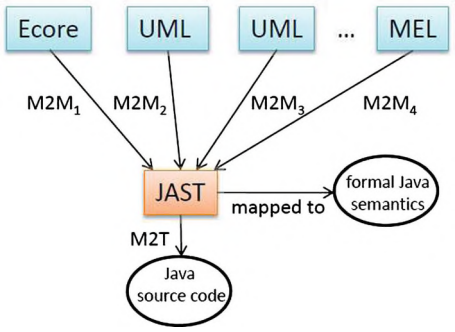

Fig. 3 Generating Java code with an intermediate JAST model

abstract syntax tree (JAST). This allows a separation of concerns (see also Fig. 3):

1. A model-to-model transformation is used to generate an instance of the JAST meta model from an arbitrary source and meta model (MEL is a meta model used in SecureMDD). The transformation is specific for the model-driven application. The JAST model is the basis for formal reasoning.
2. From the JAST model, text can be generated by model-to-text transformation. It is also possible to generate byte code directly. (Not supported in our framework, though). It should be noted that this step depends only on the JAST meta model, and is completely independent of the transformation that created the JAST model.

A small part of the JAST meta model is shown in Fig. 4 as a UML class diagram. The actual model is defined as an Ecore model in Eclipse.

Every Java expression and statement is represented by one EClass in the meta model (37 in our case). Other elements represent method and field declarations and class declarations. Every Java expression has a result type (that would be computed by a compiler when annotating the syntax tree); a binary (infix) expression like `x + y` has a left and right expression and an operation; a local variable access is a separate expression; an instance field access `FieldAccess` has an invoking expression and a reference `FieldCategory` that identifies the accessed field (a Java compiler determines whether `x` is a local variable or an instance field and treats the latter as `this.x`, Sect. 6.5 of *The Java Language Specification* [19]). The full meta model has 64 EClasses. The exact details are a result of the formal Java semantics in KIV, because the JAST meta model is as close as possible to the semantics specification.

3 Transforming UML to JAST

In our example, we use operational QVT to transform the UML class diagram into a JAST model. QVT (Query/View/Transform) [38] is standardized by the OMG and supported by several tools, including Eclipse. QVT distinguishes between a relational and an operational part. Operational QVT (chapter 8 in [38]) is essentially a programming language based on OCL [40] with several features that are tailored to model transformations.

Figure 5 shows the start of the transformation, Fig. 6 the generation of the fields and methods for an attribute, and Figs. 11 and 12 in the "Appendix" contain auxiliary operations. The figures contain the full code for the transformation.

Lines 1 and 2 declare the meta models and UML and J as abbreviations to resolve naming conflicts. Line 4 declares the transformation `simple` that transforms a UML model into a (JAST) model. Running the transformation will call the main method (line 5) that selects an element of the UML type `Model` from the input (line 6) and calls the mapping `createModel` (line 7). There is no explicit assignment to the output variable `res` because all model elements that are created during the transformation will be collected automatically in the output variable. (This is a syntactical convenience when there is only one output model). A mapping (line 9) is one of the central concepts of QVT and creates a correspondence between a source and a target element. For `createModel`, the source element is a UML Model, and the target element is of JAST type JModel. The target element is created automatically, and in lines 14–17 its properties are set. A `JModel` has a `name` and `ownedElements`, a list of (Java) packages. In this example, only one package is created in line 15 using the QVTO construct **object**. It creates a new target element without a correspondence to a source element, and allows to set its properties. This is the second possibility besides mappings to create target elements. In line 11, the classes from the UML model are selected. This requires knowledge of the UML meta model. **`self`** is a UML `model` which is a specialized UML `package`, and the immediately contained elements of a UML package are contained in the `packagedElement` property. The last part `[Class]` selects only elements of type UML `Class`. In line 12, the UML classes are mapped to Java (JAST) classes. The mapping `toJavaClass` creates a Java class by iterating over the attributes of each class and creating the fields, getters, and setters (lines 25–27). Each mapping creates a new correspondence between the same source element (the UML class) and different target elements that can be distinguished by the name of the mapping.

The mapping `createGetter` (line 10 in Fig. 6) is defined for the type `Property` since an attribute has type `Property` in the UML meta model. It has as an additional input the name of the class, because it is needed in some of the JAST expressions. A JAST method declaration (see Fig. 4) has a list of modifiers. The list is initially empty, and one element is added in line 11 using the $+=$ notation instead of $:=$. `toFirstUpper()` (line 12) is an operation of the pre-

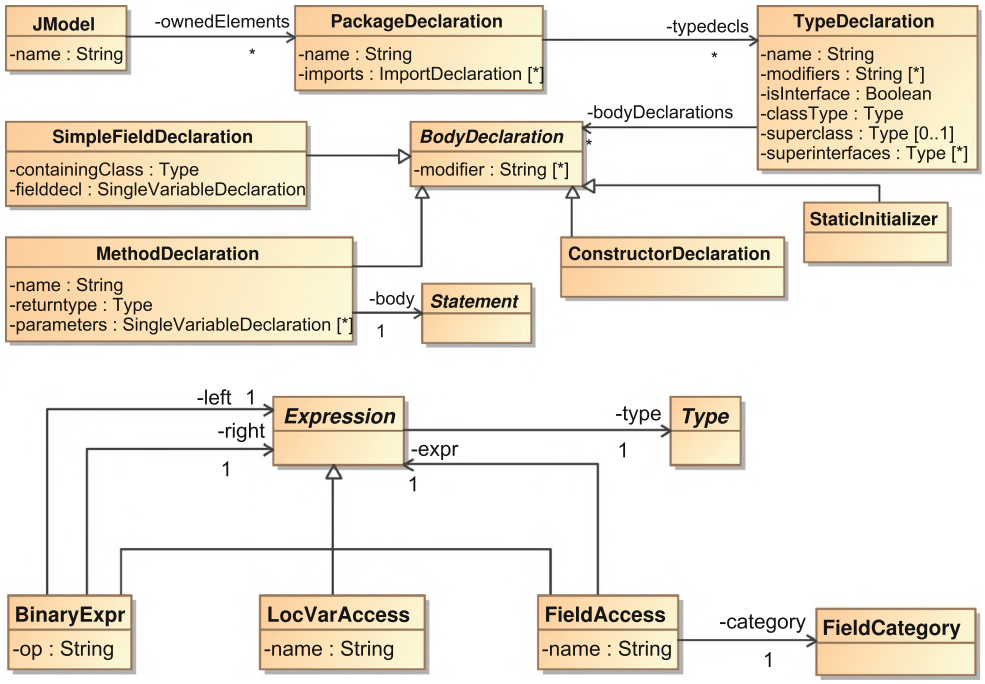

Fig. 4 Two excerpts of the JAST meta model

Fig. 5 Start of the QVTO transformation

```
1  modeltype UML uses 'http://www.eclipse.org/uml2/3.0.0/UML';
2  modeltype J uses 'http://isse.de/JAST';
3
4  transformation simple(in umlmodel : UML, out res : J);
5  main() {
6    var model : Model := umlmodel.rootObjects()![Model];
7    model.map createModel();
8  }
9  mapping Model::createModel() : JModel {
10   init {
11         var css := self.packagedElement[Class];
12         var jcs := css->map toJavaClass();
13   }
14   name := self.name;
15   ownedElements += object PackageDeclaration
16                          { name := packageName();
17                            typedecls := jcs; }
18 }
19 /* create every class */
20 mapping Class::toJavaClass(): TypeDeclaration {
21   init { var atts := self.ownedAttribute; }
22   modifiers += 'public';
23   name := self.name;
24   isInterface := false;
25   classtype := classType(self.name);
26   members += atts->map declareField(self.name);
27   members += atts->map createGetter(self.name);
28   members += atts->map createSetter(self.name);
29 }
```

defined QVT/OCL library, while `toJast()` (line 13) is an operation defined in the transformation itself (see Fig. 12). The body of the method (line 14) is a block containing only a return statement that returns a JAST field access (line 16) (i.e. `this.field`). The statements and expressions are not mapped, but generated as new objects (with the keyword

Fig. 6 Mapping attributes to Java fields and methods with QVTO

```
1  mapping Property::declareField(c:String):
2                                        SimpleFieldDeclaration {
3          modifiers += 'private';
4          containingClass := classType(c);
5          fielddecl := object SingleVariableDeclaration {
6                  name := self.name;
7                  type := self.type.toJast();
8          };
9  }
10 mapping Property::createGetter(c:String):MethodDeclaration{
11   modifiers += 'public';
12   name := 'get' + self.name.firstToUpper();
13   returnType := self.type.toJast();
14   methodbody := object Block {
15         stms += object ReturnStm {
16                  expr := self.fa(thisExpr(c), c);
17   }; };
18 }
19 mapping Property::createSetter(c:String):MethodDeclaration{
20         modifiers += 'public';
21         name := 'set' + self.name.firstToUpper();
22         parameters += self.toParameter();
23         returnType := void();
24         methodbody := object Block {
25                          stms += self.assignThis(c);
26         };
27 }
```

`object`). This is a choice made by the transformation writer that allows to reuse the code in different contexts (even if this does not happen in the example). Actually, mappings are more difficult to handle in the verification than new objects as will be discussed in Sect. 6.5. The rest of the transformation (Figs. 11, 12) contains auxiliary operations to create JAST elements that can be reused in other transformations.

Obviously the QVTO code is much more verbose than the actual Java source code, because the JAST model contains all the information explicitly that is otherwise computed by the Java compiler. Defining suitable reusable auxiliary operations helps to structure the transformations.

A programmer of the transformation needs a good knowledge of the UML and JAST meta models, and some knowledge about QVTO. A QVTO transformation is difficult to understand in detail for non-experts because an arbitrary expression like `.name` could be part of the source meta model, of the target meta model, of OCL/QVTO, or of the predefined library of OCL/QVTO. QVTO is a typed language, but types are mostly omitted in the code, and inferred by the transformation engine, which does not improve readability. Additionally, QVTO is a very flexible language with many features and often there are several possibilities to obtain the same functionality.

Still, the QVTO transformation of the example is quite simple. For example, no recursion or loops are used, and only simple iterations over collections. The programming style shown in this example is typical for operational QVT, and well suited for the creation of abstract syntax trees.

4 Formal reasoning about models and meta models

For the formal part, we use the KIV system [3,21,28] that is developed in our group. KIV is an interactive theorem prover based on algebraic specifications with several logical extensions (e.g. Dynamic Logic for imperative programs and Java and temporal logic for parallel programs and state charts).

The calculus for operational QVT presented in the next section is not limited to UML and JAST, but supports arbitrary input and output (meta) models. This means we need an algebraic specification for meta models and models. Part of the formal specification for meta models is shown in Fig. 7.[2] It specifies a subset of the Ecore meta model from EMF [52].

It is a standard algebraic specification of a freely generated data type. In `ePackage = mkEPackage(eName : string; eClassifiers : eClassifiers)`, `ePackage` is the sort that is specified, and `mkEPackage` the only constructor. It has two arguments of sort `string` and `eClassifiers` which can be selected with `eName` and `eClassifiers`, respectively. In this slightly simplified version, a meta model is an `ePackage` containing a list of `eClassifiers` that define the model

[2] The full specification can be found at http://swt.informatik.uni-augsburg.de/kiv/projects/secureMDD/ecore/project.xml.

```
ePackage = mkEPackage(eName : string;
                      eClassifiers : eClassifiers)
         ;
eClassifier = eClass(eName : string;
                     eIsAbstract : bool;
                     eSuperTypes : stringlist;
                     esfs : eStructuralFeatures;
                    ) with isEClass
            | eDataType(eType : eType) with isEDataType
            | eEnum(eName : string; eLiterals : stringlist;
                   ) with isEEnum
            ;
eStructuralFeature = eAttribute(eName : string;
                                eLowerBound : int;
                                eUpperBound : int;
                                isEOrdered : bool;
                                isEUnique : bool;
                                eBaseType : eType;
                               ) with isEAttribute
                   | eReference(eName : string;
                                eLowerBound : int;
                                eUpperBound : int;
                                isEOrdered : bool;
                                isEUnique : bool;
                                isEContainment : bool;
                                eBaseType : eType;
                               ) with isEReference
                   ;
```

Fig. 7 Part of the formal specification for meta models

elements. An `eClassifier` is either an `eClass`, an `eDataType`, or an `eEnum`. An `eClass` contains a list of `eStructuralFeatures` that are either `eAttributes` for properties with primitive types or `eReferences` for properties containing other model elements (i.e. other EClassifiers). The specification of lists and `eType` is not shown. An `eType` is an `eClassType` containing the name of an `eClass` as a string, a primitive type, an enumeration type, or a collection type. Since an `eClassType` contains only the name of an `eClass`, not a reference to the `eClass` itself, the full structure is a term, i.e. a tree, which is helpful for verification. On the other hand, a meta model must fulfill some well-formedness constraints, e.g. that the `eClassTypes` really exist as `eClasses`, that `eAttributes` really have primitive types and `eReferences` have `eClassTypes`, and that there are no duplicate names. These constraints are specified together with the definition when a model is a correct instance of a meta model (described below). Multi-valued attributes or references have an `eUpperBound` $\neq 1, -1$ for a * multiplicity. The flags `isEOrdered` and `isEUnique` are used to specify one of the standard four collection types (sets, bags, sequences, and ordered sets). We have implemented a plugin for Eclipse that allows to export an Ecore meta model (UML or JAST or others) as a term of this specification.

While the specification of meta models for our purpose is fairly straightforward, the specification of models is not. Models form a graph structure, and there are several possibilities to specify graphs algebraically. Additionally, a more complicated structure than just nodes and edges is needed. We decided to follow the internal representation of models in Eclipse EMF for two reasons. First, it allows to export models from Eclipse to KIV, and second the internal representation is based on a tree structure that is well suited for algebraic specifications and verification. Graphs are difficult because cycles always need a special and careful treatment. Otherwise, operations that are specified on graphs may be inconsistent because recursive definitions may not be well founded. With trees one can always follow the structure from top to bottom. A similar approach was taken in [16] where models are formalized in Isabelle/HOL [37].

Models are specified as a freely generated algebraic data type [60]. The tree structure of a model corresponds exactly to a term denoting an element of the data type. Hence, a model is acyclic by design. The tree structure is induced by the *containment* property of `eReferences` in the meta model. Containment corresponds to UML's composition. If containment is true, a model element is contained exclusively in one container, i.e. it cannot belong to two containers at the same time. Containment shows a strong ownership. For example, the `packagedElements` of a UML `Package` or the `ownedAttributes` of a UML `Class` have containment = true. The graph struc-

```
1  eValue    = eObject (eId : oid;
2                       eClassName : string;
3                       eContents : eContentList)
4            | eList   (eValues  : eValueList)
5            | eOid    (oid      : oid)
6            | eEnum   (eString  : string)
7            | eString (eString  : string)
8            | eInt    (eInt     : int)
9            | eBoolean(eBoolean : bool)
10           | null
11           ;
12 eValueList = []
13    | . + . (. .first : eValue; . .rest : eValueList)
14            ;
15 eContent = ecAttribute    (eName : string;
16                            eValue: eValue)
17           | ecContainedRef(eName : string;
18                            eValue: eValue)
19           | ecExternalRef (eName : string;
20                            eIsList : bool;
21                            eIds : intlist);
22 eContentList = [] with isEmpty
23    | . + . (. .first : eContent; . .rest : eContentList)
24            ;
```

Fig. 8 Main part of the formal specification for models

ture is created by using external references to explicit object identifiers. The main specification for a model is shown in Fig. 8.

Model elements are `eObjects` with an unique object identifier `eId : oid`, the name of the corresponding `eClass`, and a list of `eContents`. An `eContent` has a name and is either an attribute value (`ecAttribute`), a contained reference `ecContainedRef` (for structural features with `isEContainment` = true in Fig. 4), or an external reference `ecExternalRef` (for structural features with `isEContainment` = false in Fig. 4). A model is either a single `eObject` or an `eList`, a list of `eObjects`. Contained references contain again `eObjects` or lists of `eObjects`, while external references simply contain the object identifiers of the referenced objects. Contained references and `eObjects` form a tree because they are specified as a freely generated data type. Therefore, recursion over them is always well founded. The four kinds of collections are represented in the specification as lists (`eValueList`). Their differences are handled in the calculus and in the specification of algebraic functions that operate on collections. In the formal calculus presented in the next section, a QVTO transformation receives an `eObject` (or a list) as input and creates the output model as a list of `eObjects`. This is specified by algebraic functions that operate on models.

One important assumption is that the input model of a transformation is a correct instance of its meta model. This can now be formally specified for an `ePackage` and an `eObject`. The specification is about 190 lines long. [3] The two main properties are as follows:

1. a value in the `eContents` of an `eObject` is of the type mentioned in the meta model for the corresponding `eStructuralFeature`. This is comparable to classical type soundness for programming languages.
   For example, the `ownedAttribute` of a UML Class is defined as a collection of the `eClassType` Property in the UML meta model. This means that an `eContent` named `ownedAttribute` as part of an `eObject` with `eClassName` *UML::Class* must be an `ecContainedRef` with a value that is an `eList` of `eObjects`, and each of these `eObjects` must have an `eClassName` that is a sub-type of *UML::Property* in the meta model.
   Sets and OrderedSets additionally must not contain duplicates. Since collections are represented by lists, they are always ordered.
2. the multiplicity of a value in the `eContents` of an `eObject` is between the lower and upper bound of the corresponding `eStructuralFeature`. In particular, required references with lower = upper bound = 1 must not be missing in the model.
   Eclipse EMF distinguishes between attributes and references. A required, but missing attribute is consid-

[3] http://swt.informatik.uni-augsburg.de/kiv/projects/secureMDD/ecore/specs/modelFitsMetaModel/export/unit.xml.

ered correct. When the attribute is accessed, the default value for its primitive type is returned. However, the Eclipse validation fails if required references are missing.

Other properties are that the object identifiers are unique and that the model contains no additional stuff not mentioned in the meta model. Since QVTO is a typed language, it should be *type sound* in the following sense:

If a successfully type checked transformation is executed with a source model that fulfills the first property mentioned above, then the target model also fulfills this property.

A formal verification of this property requires a formal semantics for QVTO and a formal specification of the type check. Type soundness has been proved for Java [12,59] and other programming languages. The second property cannot be guaranteed by a transformation. It is possible that a target object is created, but a required structural feature is not set. This cannot be detected by a type check, but can be verified for a given QVTO transformation. For example, the JAST meta model requires that every (Java) expression has a result type. This is essential to give a formal semantics to a JAST model.

The QVTO expressions create and modify the target model. In the QVTO calculus, this is done by algebraic operations on models, i.e. `eObjects`. Other operations deal with access to model elements. A rather complex operation is the QVTO assignment to a structural feature with containment = true. Consider the QVTO code fragment in Fig. 9 that creates JAST elements.

The code creates explicitly three objects that will be contained in the target model. As mentioned previously, no explicit assignment to the output variable of the transformation is needed. The `type` of a `LocVarAccess` is a structural feature with containment = true. In line 3, the `inttype` object is assigned as expected. In line 5, the same object is assigned again. However, a contained element cannot belong to two containers at the same time. Therefore, the object is removed from the `x` object and assigned to the `y` object. This behavior is defined in the MOF specification [39], p. 38. As a result, the `x` object has an empty `type` which violates its multiplicity constraint. Experience shows that this behavior is a serious source for errors in transformations and difficult to track down, especially if the target meta model often uses containment. Indeed, using a mapping in the example above would introduce this error because a second call to the mapping operation would return the identical object, not a new one. However, since containment is true in the target model, two different objects are needed.

The corresponding algebraic update operation `update (oid, econtent, model)` gets as argument an object identifier `oid` of an object to update, the new `econtent`, and the model. If the `econtent` is an `ecContainedRef`, then the contained value (or values) is first deleted from the model, and then the actual update takes place. Both deletion and actual update are defined recursively for `eObjects` by following the tree structure induced by containment. Verification of actual QVTO transformations then requires theorems about this `update` operation, e.g. that it does not delete other objects.

There exists a formal Java semantics in KIV [53,54] that was defined several years ago. The formal semantics is based on an algebraic specification of a Java abstract syntax tree. The connection between a JAST model (an `EObject` structure) and the existing abstract syntax tree is obtained by a specification `model2sem` mapping a JAST model to the abstract syntax tree. In this manner, a formal semantics is provided for a JAST model. Invalid JAST models are mapped to a dummy syntax tree. Another approach is to specify the Java semantics in terms of a JAST model, but this requires much effort since a formal Java semantics is huge and complex, especially with a proof of type soundness.

The Java semantics in KIV is a natural big-step operational semantics. A Java expression defines a relation between an initial Java heap and variable binding and a resulting Java heap, variable binding, and result after evaluating the expression: $(v \times h)[\![e]\!]_{tds}(v' \times h' \times result)$ Here, v is the variable binding for local variables and method parameters, h is the Java heap, e is the Java expression, tds is the context consisting of the type (class and interface) declarations, v', h' are the resulting binding and heap, and $result$ is the result of evaluating the expression. If the expression does not terminate (e.g. if it is a call to a recursive, non-terminating method), there is no resulting state, i.e. the relation $[\![.]\!]$ is empty. The semantics faithfully models class initialization (first active use), and exceptions.

The next section presents the QVTO calculus and how to formulate and prove properties of the target JAST model for the transformation of Sect. 3.

Fig. 9 Example for containment

```
1  var inttype := object J::PrimitiveType { name := 'int'; };
2  var x := object J::LocVarAccess { name := 'x';
3                                     type := inttype; };
4  var y := object J::LocVarAccess { name := 'y';
5                                     type := inttype; };
```

5 A calculus for QVTO in dynamic logic

Operational QVT is essentially an object-oriented, imperative programming language that operates on input and output models, and keeps track of mappings with the help of a *trace* that contains source object, target object, and applied mapping operation for every executed mapping call. Our calculus for QVTO is a sequent calculus for dynamic logic (DL, [22]). DL extends predicate logic with two modal operators, box $[\,.\,]$. and diamond $\langle\,.\,\rangle$., written as $[\alpha]\,\varphi$ and $\langle\alpha\,\rangle\varphi$. In standard DL, α is a statement of an abstract programming language, and φ is again a DL formula, i.e. it can contain again boxes or diamonds. The intuitive meaning of $[\alpha]\,\varphi$ is that if α terminates, afterward φ is true. $\langle\alpha\rangle\,\varphi$ is true if α terminates and afterward φ is true. The dualities $[\alpha]\,\varphi \Leftrightarrow \neg\ \langle\alpha\,\rangle\,\neg\ \varphi$ and $\langle\alpha\,\rangle\,\varphi \Leftrightarrow \neg\ [\alpha]\,\neg\ \varphi$ hold.

A sequent $\varphi_1, \ldots, \varphi_m \vdash \psi_1, \ldots, \psi_n$ consists of two lists of formulas $\varphi_1, \ldots, \varphi_m$ and $\psi_1, \ldots, \psi_n$ (often abbreviated by Γ and Δ) divided by $\vdash$ and is equivalent to the formula $\mathrm{Cl}_\forall(\varphi_1 \wedge \ldots \wedge \varphi_m \rightarrow \psi_1 \vee \ldots \vee \psi_n)$. $\mathrm{Cl}_\forall$ is the universal quantification over the free variables of the formula. $\varphi_1, \ldots \varphi_m$ can be thought of as preconditions, while one of $\psi_1, \ldots, \psi_n$ must be proved.

A Hoare triple $\{\varphi_1 \wedge \ldots \wedge \varphi_m\}\alpha\{\psi\}$ can be expressed as $\varphi_1, \ldots, \varphi_m \vdash [\alpha]\psi$ or $\varphi_1, \ldots, \varphi_m \vdash \langle\alpha\rangle\ \psi$ if termination is included.

In our QVTO calculus, the program α is a QVTO expression that is contained in a context `(in, out, trace)`. `in` represents the input model, `out` the output model, and `trace` the trace that is created by the mapping operations. The intuitive idea is that the QVTO expression α is evaluated in the initial context `(in, out, trace)`.

An example is the following sequent:

valid(in, UML), suitable(in), out == [], trace == []
$\vdash \langle$ (in, out, trace) `Simple::main()` $\rangle$
typeCorrect (model2sem(out))

`Simple::main()` is a call to the QVTO transformation's `main()` method, the top-level entry point into the transformation. The `in` model must be a valid UML model `valid(in, UML)` that is additionally suitable for the transformation `suitable(in)`. The output model `out` and the `trace` are initially empty. This is the actual situation when the transformation is started. Then, after running the transformation, the resulting output model `out` converted to the formal Java specification `model2sem(out)` is a type correct Java program `typeCorrect(model2sem (out))`. The value of `out` is changed by evaluating the transformation. The trace is also changed, but not referenced in the postcondition, the input model remains unchanged during the transformation. The predicates used in the pre- and postconditions are specified algebraically. The type correctness proof is described in Sect. 6.5.

It is also possible to prove properties for parts of the transformation. In our example, we have one mapping that creates a Java method declaration for a setter, and one mapping for a getter. We can formulate a property that after calling the setter, the getter returns the setter's argument. In Java, this would look like `a.setB(b); b == a.getB();`. Formally, this looks like

1. valid(in, UML), suitable(in), isProperty(a), a $\in$ in, unmapped (a, trace)
2. $\vdash \langle$ (in, out, trace) `f := a.map declareField(c)` $\rangle$
3. $\langle$ (in, out, trace) `g := a.map createGetter(c)` $\rangle$
4. $\langle$ (in, out, trace) `s := a.map createSetter(c)` $\rangle$
5. $(\forall v, this, b, h_1, h_2, h_3, val, tds.$
6. tds=class c { model2sem(f), model2sem(s), model2sem(g)}
7. $\wedge$ validJava$(v, h_1, this, b, tds)$
8. $\wedge(v \times h_1)[\![this.setB(b)]\!]_{tds}(v \times h_2 \times \bot)$
9. $\wedge(v \times h_2)[\![this.getB()]\!]_{tds}(v \times h_3 \times val)$
10. $\rightarrow val == b)$

In lines 2, 3, and 4, the mappings from the transformation are called for a UML attribute `a` and each result is assigned to a variable. This assignment is not part of the original transformation, but that is no problem. Line 1 contains the preconditions. Since the mappings are defined for a UML `Property`, the invoking variable `a` must be a property (`isProperty(a)`) of the input model (`a` $\in$`in`). Additionally, we need to know that `a` has not yet been mapped by any mapping operation. This is expressed by `(unmapped(a, trace)`. Otherwise, the mapping operation would not be executed, but the target object would be retrieved from the trace. In this case, we would need the precondition that it is the correct target object. However, in the actual transformation, we are sure that the attributes are unmapped.

The three diamonds are executed in order, and the target model `out` and trace `trace` are modified by side effect. This means that `out` and `trace` in line 3 refer to the *resulting* target model and trace of the mapping call in line 2 even though the variable names are the same—this is an intrinsic feature of Dynamic Logic. The postcondition of the three consecutive diamonds begins in line 5. In line 6, the JAST elements `f`, `g`, `s` are converted to their Java semantics counterpart (a Java field declaration and two method declarations), and a class containing them is constructed. In line 8, the semantics of a setter call is used. $setB$ is the name of the setter that is constructed from the name of the UML attribute. $this.setB(b)$ is evaluated with an initial Java heap h_1 and local variable mapping v. $this$ and b must refer to valid objects or values in the h_1 and v. This is expressed by validJava$(v, h_1, this, b, tds)$. The semantics of the setter will result in a modified heap h_2 which is then used in line 9 as the initial heap for the getter call. The semantics of this call will produce a result val which is equal to the setter's argument b (line 10), our desired property.

The sequent is quite long because so many conditions are needed. In particular, the Java semantics is well defined only for type correct Java programs with a compatible heap. Hence, the class must contain the method declarations and a declaration for the field, and the invoker *this* and argument *b* must reference valid objects or values. The proof itself requires only a couple of minutes (real time, i.e. including user interactions) since the generated methods are so simple. In comparison, the type correctness proof—discussed in Sect. 6.5—required about 2 weeks to complete.

5.1 Example rules of the calculus

The calculus essentially has one rule for every operational QVT expression, plus some generic rules. It works by symbolic execution of the QVTO program from its beginning to its end (i.e. computation of strongest postcondition). This means it follows the natural execution of the program. Mapping and operation calls are replaced by their implementation (inlined), then nested expressions are flattened to a sequence of simple expressions that can be executed directly. OCL and QVTO do not distinguish between statements and expression, there are only expressions that return a result. To make this result accessible, the calculus introduces assignments if necessary. The calculus avoids OCL's `invalid` which is discussed in detail in Sect. 6.1. Applying proof rules to a goal creates a proof tree from bottom to top. Starting with the transformation's `main` method the initial proof looks like this:

$$\frac{\vdots}{5.\ \Gamma' \vdash \langle \texttt{model := elems![Model]; ...} \rangle\, \varphi}$$
$$\frac{}{4.\ \Gamma \vdash \langle \texttt{elems := umlmodel.rootObjects(); model := elems![Model]; ...} \rangle\, \varphi}$$
$$\frac{}{3.\ \Gamma \vdash \langle \texttt{model := umlmodel.rootObjects()![Model]; ...} \rangle\, \varphi}$$
$$\frac{}{2.\ \Gamma \vdash \langle \texttt{varmodel := umlmodel.rootObjects()![Model]} \rangle \langle \texttt{...} \rangle\, \varphi}$$
$$\frac{}{1.\ \Gamma \vdash \langle \texttt{main()} \rangle\, \varphi}$$

Goal 2 expands the `main()` call. A sequence of expressions can be divided into individual diamonds, i.e. ⟨ (in, out, trace) $\alpha; \beta\rangle\ \varphi \equiv$ ⟨ (in, out, trace) $\alpha\rangle\langle$ (in, out, trace) $\beta\rangle\ \varphi$. Goal 3 replaces the local variable declaration with a simple assignment. Goal 4 flattens the right-hand side of the assignment. A new variable `elems` is introduced to capture the intermediate result. In the next step, a proof rule for `umlmodel.rootObjects()` can be applied. `rootObjects` is a predefined library operation that selects the top-level elements of the input model. This eliminates the expression by adding new preconditions to Γ. In the next step, a proof rule for `elems![Model]` can be applied and so on.

We present some proof rules to give an impression of the calculus.

5.1.1 Conditional

The rule for a conditional works like this:

$$\frac{1.\ e = true, \Gamma \vdash \langle (in, out, trace)\ x := \alpha \rangle\, \varphi \qquad 2.\ e = false, \Gamma \vdash \langle (in, out, trace)\ x := \beta \rangle\, \varphi}{\Gamma \vdash \langle (in, out, trace)\ x := \textbf{if}\ e\ \textbf{then}\ \alpha\ \textbf{else}\ \beta\ \textbf{endif}; \rangle \varphi}$$

Γ is an arbitrary list of formulas containing preconditions. e, α, and β are QVTO expressions, and φ is again a Dynamic logic formula. The rule has two premises, one for the case that the test is true, one for the case that it is false. The conclusion contains the **if** expression that is assigned to x. The assignment may have been introduced by another rule of the calculus. In case the test is true the, **then** part is assigned to x, otherwise the **else** part. The result of the test cannot be undefined (see Sect. 6.1). This captures the standard meaning of an **if** expression. However, the rule is only applicable if e is a simple expression, either a literal (like $true$ or $false$) or a variable. In these cases, it is possible to write $e = true$ because a literal (or a variable) can be used in predicate logic formulas in our calculus. If the test is a complex expression like `seq->exists(...)`, the flattening rule must be applied first that will introduce a new variable:

```
flag := seq->exists(...);
x := if flag then ...
```

After the expression has been evaluated, the conditional rule is applicable.

5.1.2 Iterators

Collections and iteration over collections are important features of OCL and often used in QVTO transformations. The tricky part about QVTO is that iterators usually compute a result collection, but modify the target model by side effect. This makes reasoning about them difficult. Currently, the proof strategy uses induction which requires an unwinding proof rule that depends on the iterator. The generic proof rule looks like this (the context is omitted):

$$\frac{\begin{array}{l}1.\ \Gamma \vdash seq \neq null \\ 2.\ \Gamma, seq = [] \vdash \langle x := empty\_result; \rangle\, \varphi \\ 3.\ \Gamma, seq \neq [] \vdash \langle e := x\text{->}first(); a := body(e); \\ \qquad\qquad y := seq\text{->}rest()\text{->}op(e|body(e)); \\ \qquad\qquad x := combine(a, y) \rangle\, \varphi\end{array}}{\Gamma \vdash \langle x := seq\text{->}op(e|body(e)); \rangle\, \varphi}$$

The rule is formulated for a sequence `seq`, bags and sets must be converted first. An iterator `op` is applied with a body `body` that uses `e` to bind the successive individual elements. The result is assigned to `x`. The first premise requires the proof that `seq` is not null. This means we have a real collection, and the operations `first()` and `rest()` are well

defined. The second premise covers the case that the sequence is empty ($seq = []$). In this case, `x` is assigned the correct result *empty_result* that depends on the iterator, i.e. `false` for `exists`, `true` for `forAll`. The third premise unwinds the collection and performs one step for the first element of the collection. A new variable e is assigned to the first element, and the body of the iterator is executed. The result is stored in a new variable a. Then, the iterator is applied to the rest of the sequence. Finally, the results are combined (with `or` for `exists`, `and` for `forAll` etc.) and assigned to x. This captures exactly the behavior of the iterators. Operations that can abort the iteration require a modified third premise that includes a conditional. This proof rule is only useful in inductive proofs since the length of a collection is usually not fixed so the proof rule can be applied forever.

5.1.3 Mapping call

One of the most interesting and the most complicated proof rules is the one for a mapping call, i.e. `x := a.map m(args);`. Mapping operations and mapping calls are described in Sect. 8.2.1.5 and 8.2.1.21 in the QVT specification [38]. We describe only a simplified version of the rule. It has three premises (see Fig. 10):

1. The invoking expression `a` may not be null: $a \neq null$
2. As mentioned in Sect. 3, a mapping creates a correspondence between a source and a target element, and the transformation keeps track of this correspondence in a *trace*. So if the source element has been mapped before (i.e. is contained in the trace), the mapping is not executed, but the result is looked up in the trace:

   $$a \in trace \vdash \langle (in, out, trace)\ x := lookup(a, trace); \rangle \varphi$$

   lookup is not a QVTO expression, but a logical function with an algebraic specification that can be used as a QVTO expression. The introduction of "new" expressions is a useful technique in a calculus to modularize complex behavior. QVTO has a `resolve` expression that can be used to search the trace for elements (Sect. 6.5 contains an example why this is useful). In the proof rule, `resolve` is also reduced to *lookup*.
3. Otherwise, the mapping call is replaced by the (modified) body of the mapping operation. The body of a mapping operation consists of three (optional) sections: an initialization section (denoted by `init {...}`), a population section where attributes of the result are computed, and a termination section (denoted by `end {...}`). The mapping `toJavaClass` (Fig. 5) contains only an init section and a population section without any keyword, and the mappings in Fig. 6 contain only the population section. The QVT) specification states that between the init and population section, the following happens: If the predefined `result` variable is still null, a target element of the correct result type is created, assigned to the `result` variable, and the source and target elements are added to the trace.

In the proof rule, a mapping call `x := a.map m();` with declaration

```
mapping m() {init; population; end;}
```

is replaced by

```
{init;addToTrace(result); population; end;
x := result;}
```

First, the init section is executed. Then, the logical extension *addToTrace(result)* is executed. This extension has its own proof rule that either does nothing (if result has been assigned in the init section) or creates the result object, assigns it to `result`, and adds it to the trace. Then, population and end sections are executed, and finally, the result variable is assigned to `x`.

This finished the description. Figure 10 shows the rule. Since mappings can be overridden for more specific invokers, there can be more premises similar to premise 3 for every possible mapping body. (The QVTO specification states that "This follows usual object-oriented virtual call semantics.") Merging and disjuncting mappings is not supported. Additionally, the formal parameters are bound to the actual arguments by equations, and the predefined QVT variable `self` is bound to the invoker `a`. This may require renaming of variables to avoid conflicts.

Fig. 10 The proof rule for mapping calls

$$\frac{\begin{array}{l}1.\ \Gamma \vdash a \neq null \\ 2.\ \Gamma, a \neq null, a \in trace \vdash \langle (in, out, trace)\ x := lookup(a, trace) \rangle\ \varphi \\ 3.\ \Gamma, a \neq null, \neg\, a \in trace, self = a, params = args \\ \quad \vdash \langle (in, out, trace)\ init; addToTrace(result); population; end; x := result; \rangle\ \varphi\end{array}}{\Gamma \vdash \langle (in, out, trace)\ x := a.map\ m(args) \rangle\ \varphi}$$

5.1.4 Dealing with the QVTO trace

The mapping rule contains one premise (the second) for the case that a source object has been mapped before. The trace is specified as a list, and it is simple to specify operations that add to the trace, or search for an object in the trace. The elements in the trace itself contain all the information needed to faithfully handle traces: (1) a reference to the source object (if there is one, it is also possible to have mappings without a source object), (2) the signature of the mapping operation, (3) additional actual arguments to the mapping operation (if any), and (4) a reference to the target object. Adding to the trace is done by a new QVTO expression `addToTrace` that exists only in the calculus, and is introduced by the proof rule for mapping calls (in the conclusion in Fig. 10). As with all other expressions, one proof rule exists to evaluate and discharge the expression. The relevant description in the QVT specification is Sect. 8.2.1.15 (MappingOperation). The `addToTrace` captures the behavior of the "implicit instantiation section":

$$\frac{\begin{array}{ll}1. & \Gamma, target = null, newTarget = eObject(...), \\ & newOut = add(out, newTarget), \\ & newTrace = add(trace, [source, sig, args, newTarget]) \\ & \vdash \varphi\,[target/newTarget][out/newOut][trace, newTrace] \\ 2. & \Gamma, target \neq null, newTrace = add(trace, [source, sig, args, target]) \\ & \vdash \varphi\,[trace/newTrace]\end{array}}{\Gamma \vdash \langle (in, out, trace)\ addToTrace(source, sig, args, target)\rangle\ \varphi}$$

- **Premise 1** deals with the case that the result of the mapping operation `target` is still null. In this case, a new object is created ($newTarget = eObject(...)$), added to the target model `out` ($newOut = add(out, newTarget)$), and a new entry is added to the trace ($newTrace = add(trace, [source, sig, args, newTarget])$). $newTarget, newOut, newTrace$ are new variables that are introduced to refer to the new elements. The `addToTrace` expression is discarded, and in the postcondition φ, the old variables $target, out, trace$ are replaced by the new ones ($\varphi\,[target/newTarget][out/newOut][trace, newTrace]$). This is a standard technique in Dynamic Logic. The effect is that other QVTO expressions in φ use the new values (cf. the theorem for the getter–setter identity at the beginning of the section).
- **Premise 2:** If the `target` is not null (this means it has been instantiated explicitly in the init section of the mapping), then it is simply added to the trace.

This finishes the presentation of the calculus. The next section discusses design decision, experiences, and issues with QVTO.

6 Experiences and discussion

6.1 OCL invalid

QVTO is based on OCL. Normal (essential) OCL does not support error handling, but has a generic error element `invalid` that is used as the result of an operation that would raise an exception in Java. For example, calling an operation on a null source, selecting the first element of an empty sequence, using bad indices in string operations, or casting to the wrong type with `oclAsType` return `invalid`. The reason is that OCL was designed as a specification language that is close to formal logic, not as a programming language ([40], Sect. 7.2 *Why OCL?*). Now, however, OCL is used as a programming language. Most OCL operations are strict, i.e. return `invalid` when passed `invalid` as an argument. This has the effect that an inadvertent occurrence of `invalid` as the result of a programming error propagates through the code, then disappears somewhere. The result is that the transformation will create an unexpected, and usually bogus target model. Programming experience with several thousand lines of QVTO transformations shows that locating the error can be very time-consuming. Therefore, we feel that deliberately programming with `invalid` values should be avoided. Instead of testing whether the result of `x->first()` is `invalid` with `oclIsInvalid`, one can test beforehand if the sequence is empty.

The problem with `invalid` is aggravated by the fact that Eclipse sometimes shows an unexpected behavior. The OCL semantics specifies that neither the `then` nor the `else` branch of a conditional is executed if the test returns `invalid` and that the result of the conditional is also `invalid` (Def. A.30 in Sect. A.5.1.2 in [40]). Eclipse, however, evaluates the `else` branch and returns its result. Other diverging behavior concerns some of the iterators.[4]

6.2 Some open questions in the QVTO specification

OCL is used in QVTO as a programming language. However, the OCL specification does not care about efficiency. For example, the semantics states that `exists` and `forAll` must iterate over the complete collection even if the test for the first element returns `true` for `exists` (or `false` for `forAll`). A normal programming language would abort in this case, and Eclipse does this. Technically, this is wrong since the body of the iterator may contain side effects, e.g. create objects in the target model.

[4] As far back as 2003, several issues were opened to clarify the behavior of the conditional http://www.omg.org/issues/ocl2-rtf.open.html#Issue6554 and forAll/exists http://www.omg.org/issues/ocl2-rtf.open.html#Issue6539 that are still unresolved.

On the other hand, it is obviously extremely bad programming style to used side effects for tests like `exists` and `forAll`. QVTO has a large section on imperative extensions to OCL, but does not redefine the original OCL expressions.

Some iterator expressions are specified precisely by defining their behavior with the help of other QVTO expressions. However, most of the time the semantics is described in natural language only. This is more or less clear for the good cases, but exceptional situations are often not mentioned. Another area which requires clarification is the treatment of class hierarchies in the meta models. Some examples of open issues:

– **Assignments:** (QVT 8.2.2.11)

"Null values are also skipped." For example, in
`modifiers := Sequence{ null, "public" };`
the result will be a list with one element, `"public"`, and in
`modifiers += null;`
the `null` value will not be added to the list. This is a quite nice feature for programming. However, the specification should also mention explicitly that assigning `invalid` also has no effect.

– **Resolve expression:** (QVT 8.2.1.22)

"A resolve expression is an expression that inspects trace objects of the transformation in order to retrieve target objects ..."

One version of resolve returns the first target object that matches a given test. The specification does not mention what happens if there is no such element. One might expect `invalid`, since `first()` is `invalid` for an empty sequence (in the specification and in Eclipse), but Eclipse returns `null`.

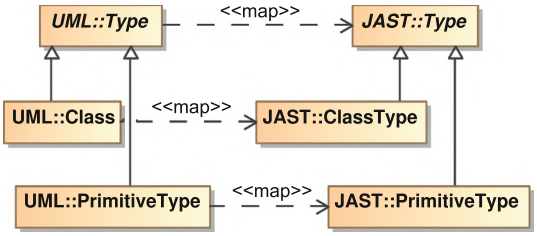

Fig. 11 Mapping UML types to JAST types

– **Switch and alt(ernative) expression:** (QVT 8.2.2.8/8.2.2.9)

"...It returns the executed expression if the condition is satisfied, otherwise it returns null. It is semantically equivalent to an if expression with a null in the else clause."
A conditional returns `invalid` if the test is `invalid`. The specification seems to be contradictory.

Actually, the complete QVT specification uses the word *invalid* only once and in a different context. QVTO is based on OCL, but the specification seems to assume that `invalid` never occurs.

More serious omissions concern class hierarchies. An example is shown in Fig. 11. Suppose we want to map UML types to JAST types. The meta class hierarchy is simplified and shows only the relevant meta classes. Both the UML meta class and the JAST meta class `Type` are abstract. UML classes should be mapped to JAST class types, and primitive types to primitive types. Other sub-classes of `Type` are not considered. As a transformation writer, I would like to use the following code:

```
1  var p : UML::Property := ...
2  var ty := p.map type2type();
3
4  abstract mapping UML::Type::type2type() : JAST::Type { }
5
6  mapping UML::Class::type2type() : JAST::ClassType {
7    name := self.name;
8  }
9  mapping UML::PrimitiveType::type2type() :
10   JAST::PrimitiveType {
11   name := self.name;
12 }
```

The idea is that the concrete mappings `UML::Class::type2type()` and `UML::PrimitiveType::type2type()` override and implement the abstract mapping `UML::Type::type2type()` that should never be called. This code is not accepted by Eclipse QVTO, but the specification is largely silent about this. Mapping operations and map-

Fig. 12 Correct overriding of mapping operations

```
1  mapping UML::Type::type2type() : JAST::Type { init { } }
2
3  mapping UML::Class::type2type() : JAST::Type {
4   init {
5    result := object JAST::ClassType { name := self.name; };
6        }
7  }
8  mapping UML::PrimitiveType::type2type() : JAST::Type {
9   init {
10   result := object JAST::PrimitiveType { name :=self.name; };
11       }
12 }
```

ping calls are covered in the QVT specification in 8.2.1.15 and 8.2.1.21.

- It is not specified whether a mapping operation can be declared as `abstract` (as in line 4).
- The result element of the mapping operation is created automatically after the `init` section unless it was explicitly created there. However, the result type `JAST::Type` is abstract in line 4, and cannot be instantiated. The specification does not mention this case. Eclipse QVTO requires that an init section exists syntactically. However, if the result object is not created, there the transformation fails at run time with a Java exception.
- When does one mapping operation M_{sub} overrides another M? The specification says in 8.2.1.10 that *"…the signature must be compliant …"* without defining what compliant is. It is clear that the source type of M_{sub} must be a sub-type of M's source type (e.g. UML::Class and UML::Type). It is not clear what must hold for the arguments and the result types. One possibility is to require the signatures of the operations to be equal. Another possibility is to allow covariance for parameters, and contravariance for result types. The fictional code above requires contravariance for the result type, because then the result objects of the concrete mappings can be created automatically. Eclipse QVTO requires equality.

Hence, only the code in Fig. 12 works.

6.3 Development of the calculus

OCL has 12 expressions, imperative OCL 21, and QVT 5 additional expressions, most of them with many features (e.g. the ImperativeIterateExp (QVT 8.2.2.7) defines 6 different iterators). The OCL and QVT standard library have about 175 predefined operations. This means that implementing a calculus for full QVT requires a *huge* effort. Therefore, we follow a pragmatic approach: Currently the calculus has rules only for a subset of all expressions (18) and operations (30), namely those that occur in the transformations we verified so far. More proof rules will be added in the future.

The calculus and used algebraic data types are designed with full QVTO in mind. In particular, exceptions, try-catch, and breaks can be included without any modifications to the existing calculus. Support for relational QVT requires extensions to deal with patterns.

The OCL and QVT specifications are together 500 pages long. However, the description is often imprecise and leaves many issues open, even for rather central language constructs. This raises the question how a correct proof rule should look like. Here, we follow the following strategy:

1. If an issue is not clear from the specification, but is irrelevant for "normal " QVT transformations, the proof rule will have a precondition that excludes the issue.
2. Otherwise we check what the QVTO implementation in Eclipse [42] and SmartQVT [49] do. If their behavior seems reasonable, the proof rule is designed correspondingly. This makes sense because verifying and actually running a transformation should produce the same results (i.e. correct target models).
3. Otherwise we program our transformations around the issue and do not support it. We also make sure that our transformations work in both Eclipse QVTO and SmartQVT.[5] Since Eclipse QVTO and OCL are actively developed, their behavior may change for some language constructs in a new version.
4. For serious differences between the OCL specification and Eclipse like the behavior of `exists` and `forAll`, the proof rule ensures the Eclipse behavior. If no side effects occur, there is no difference (except maybe in efficiency). Our calculus allows to express that an expression has no side effects, and this could be included in the proof rule. However, this would require additional proof effort, and seems pointless because it never happens in reality. Probably the best solution is to implement a syntactical check that the body of `exists` and `forAll` has no side effect.

[5] Development on SmartQVT seems to have stopped. Unfortunately, the available open source version does not work with newer Eclipse releases.

An important design decision was the omission of `invalid` from the calculus. One reason was the lack of precision in the specifications and implementations. A second reason was that `invalid` should not be used in transformations as discussed in Sect. 6.1. The third reason was that it makes proof rules more complicated without any benefit.

This does not mean that the calculus is incorrect or assumes that `invalid` does not occur. On the contrary, the calculus *guarantees* that `invalid` does not occur. In situations where the result of an operation could be `invalid` it must be proved that this situation does not occur. For example, for `x->first()`, it must be proved that `x` is a non-null, non-empty sequence. This simplifies the calculus considerably because a special treatment of `invalid` and its propagation can be avoided. Otherwise every proof rule would have at least one additional premise. For `x->first()`, we would need one premise that `x` is already invalid, a second premise that `x` is null or empty, and a third premise for the good case where the first element is selected. There is no theoretical obstacle against supporting `invalid`, but since a well-designed transformation does not use it, there is no reason either.

6.4 Assumption hunting

Often it is not clear what assumptions are made about the input model of a transformation. A standard assumption is that the model is a correct instance of its meta model. We must include this property explicitly in our proofs for the UML model (valid(in, UML) in the preconditions), but do not assume that the additional constraints mentioned in the UML specification hold. But usually there are more specific assumptions. First, we have syntactical assumptions. The names of classes and attributes must be legal Java names, and not contain duplicates. Next, in our example, it is necessary that all attributes have types, although the type is optional in UML. Additionally, the type must be either a UML class, the primitive type `Boolean`, or the primitive type `Integer`. Other cases are not covered by the transformation. This makes sense since the transformation is used for a special purpose, and is not intended to transform arbitrary UML class diagrams or models to code.

A slightly more subtle assumption is about the structure of the source model. An attribute could reference a class that is contained in a sub-package of the top-level package (a UML model is a package). However, the transformation selects and transforms only the `packagedElements` of the model and would "miss " this class. The result would be a JAST model that is not type correct.

In general, formal verification is very good at finding (hunting for) implicit assumptions. All assumptions must be incorporated explicitly in the preconditions, or the verification will fail. This is comparable to model validation (every source model that passes validation is guaranteed to produce correct target models), but slightly different to defensive programming (i.e. checking assumptions in the transformation itself, e.g. with assertions). With defensive programming, a source model either will produce a correct target model or fail. However, we want to prove that the transformation does *not* fail, but always produces a correct target model—given the "right" source model. This means that defensive programming does not really help verification, but does not complicate it either (we prove that the assertion will never fail anyway).

6.5 Proof experiences

Interactive theorem proving usually requires substantial effort, for example, the verification of an l4 micro-kernel with 8,700 lines of C code by Klein et al. required 11 person years of effort and another 9 person years of research [29]. However, the example also shows that interactive verification can handle large case studies where fully automatic techniques must fail. In this sense, interactive verification scales up.

It requires quite a lot of experience to use interactive theorem provers successfully, because the user must know the logic, the input language, how things are formalized, and how to utilize the strength of the tool.

QVTO transformations often use iterators over sequences or sets, e.g. `css->map toJavaClass()` This avoids loops or recursion, but is similarly difficult for verification. The properties that hold during the iteration must be formulated as invariants. Informally, this looks like:

> *The original list* `css` *can be divided into two parts. The classes of the first part have already been mapped, and the output model already contains corresponding Java type declarations. The classes of the second part and their attributes are unmapped.*

Unfortunately, it is not possible to use an invariant rule that is similar to the well-known Hoare rule for `while` loops because the result list that collects references to the newly created target objects does *not* remain invariant. Structural induction that works well for the verification of algebraic lists or for functional programming languages does not work either because the target model and the trace are modified by side effect. It may be possible to devise a specialized proof rule, but we have not done so yet.

Proving type correctness of the generated Java abstract syntax tree is much more difficult than proving properties about the behavior of the generated setter and getter methods. The getter-/setter property shown in Sect. 5 requires 28 user interactions, and some minutes to finish. The type correctness proof, on the other hand, requires about 70 lemmas, more than 800 user interactions in total (i.e. including lemmas), and about 2 weeks to complete.

One reason for the increased complexity is that the full transformation must be considered, while the setters and getters are created by a much smaller part of the transformation. A second reason is that several iterations are used, and the lemmas about the iterations must fit together. The third reason is, however, that intermediate results are not type correct, only the final result is, for example, class A is generated first and has a field of class type B. However, B is created later so A in itself is not type correct. The proof uses two different ideas:

- It is necessary to establish a structural correspondence between the source and the target model: all Java classes correspond to UML classes and vice versa.
- A kind of look ahead is needed that the missing parts are eventually generated:
  All types that occur in a class (i.e. the types of fields and methods) correspond to UML classes in the source model.
  Assume that this class is checked in a context where all these types already exist as Java classes. Then, it is type correct.

Eventually, for every UML class a Java class is generated. Then every class is type correct because the assumption is fulfilled. The most important lemma that handles both iteration and the look ahead deals with `css->map toJavaClass()` (see "Appendix B"). It took a while to establish this approach, but now it can be reused in other transformations.

Quite a lot of work also goes into reasoning about the structure of the models since the formal specification is quite complex, and UML itself is very complex. As with every new calculus, verification is quite tedious at first, and more experience is needed to see how to improve automation.

7 Related work

Related work can be divided into three areas: formal treatment of OCL and QVT, formal treatment and correctness of model transformations in general, and correctness of Java code generation.

The OCL specification [40] contains a formal semantics on paper (i.e. not tool supported which makes a big difference). Brucker and Wolff formalize OCL in Isabelle/HOL on 500 pages [7]. Their goal is to *"provide a semantic representation compliant with the OCL standard semantics definition"*. They define OCL with respect to UML class diagrams (in contrast to Ecore/MOF used in QVT), and faithfully model undefined and invalid values. The semantics is specified using a shallow embedding. This means for example that there does not exist an abstract syntax tree of a OCL program. They also proved a calculus based on a three-valued logic. Experience shows that a three-valued logic creates a considerable technical overhead for concrete proofs. As discussed in the previous section, our calculus works without invalid values. The shallow embedding means that an OCL expression must be translated into a HOL expression. Then, normal HOL proof rules are used. Our calculus operates on an abstract syntax tree for QVTO, so that there is an explicit proof rule for an OCL/QVTO `if` statement. Proof automation *"…is still at the beginning."*. The whole work is not aimed at proving properties about actual OCL programs (only one example is mentioned, a stack), and it is difficult to judge whether QVTO could be added without major modifications. Not surprisingly, they identify many problems in the specification (see [6] for an update).

There seems to be no other work that supports the verification of arbitrary OCL programs. However, there are a number of automatic approaches that deal with OCL constraints on UML class diagrams. The USE tool by Gogolla et al. [17] supports different techniques for checking OCL constraints for concrete UML models, e.g. with Alloy [18]. Bordbar et al. [1] and Krieger and Knapp [30] also translate OCL expressions into the input language of Alloy [24], and check that a concrete model fulfills the constraints. Soeken et al. [50] encode OCL with bit vectors and use a SMT solver to prove consistency and reachability for OCL constraints for a fixed number of objects. Queralt et al. [41] deal with OCL constraints for class diagrams in general, and write:

> *It is well-known that the problem of automatically reasoning with integrity constraints in their full generality is undecidable. This means that it is impossible to build a reasoning procedure that deals with the full expressiveness of OCL, and that always terminates and answers whether the schema satisfies a certain property or not.*

They go on to discuss two automatic approaches (restricted constraints and incompleteness/non-termination), and combine both in their own tool AuRUS for better results. Interactive verification does not have these limitations, but requires the invention of suitable invariants and generalizations by the user.

There is some work on the semantics of different parts of QVT. Büttner and Kuhlmann [8] identify problems with the design of imperative OCL which is part of QVTO and extends OCL. For example, the evaluation context is not properly defined since imperative OCL has side effects but OCL not. They argue that it should not be possible to use imperative OCL everywhere, e.g. in invariants.

Favre [15] formalizes the QVT Core language and meta models in CASL. The QVT specification reduces relational QVT to the Core language and defines the semantics of QVTR in terms of the Core semantics. Favre's formalization

uses algebraic specifications to define models and meta models. Here, references to other model elements are algebraic data types on their own. This avoids cyclical algebraic operations. In our specification, we achieve the same by using the containment relation. The Core language is formalized with a shallow embedding (similar to OCL in HOL). This makes it in principle possible to reason about relational QVT transformations by translating them to the Core language and then apply the algebraic definitions. It seems, however, that this has not been done. Romeikat et al. [45] show (informally) how relational QVT can be translated into operational QVT, and Jouault and Kurtev [25] translate QVT to ATL [26]. So in a sense, these languages are more or less exchangeable. However, for interactive proofs, it is essential that the original language is supported. Otherwise one gets lost in the technical overhead and looses track of what to prove.

An (incomplete) overview over model transformation approaches by Czarnecki and Helsen in 2006 [11] concluded that they are *"often ad hoc, that is, without proper theoretical foundation."*. Correctness is mentioned only once as a vague possibility. However, Varró and Pataricza [58] wrote already in 2003

> *In theory it would be advisable to prove that a model transformation preserves certain predefined semantic properties for any well-formed model instance, but this typically requires the use of sophisticated theorem proving techniques and tools with a huge verification cost.*

Then, they go on to argue that the effort is not justified. In contrast, we believe that security- or safety-critical applications justify this effort and that a suitable methodology and techniques can make the task manageable.

Giese et al. [16] transform state automata into PLC (programmable logic controllers) code. The transformations are based on triple graph grammars [48]. The (meta) models and transformations are translated manually to Isabelle/HOL [37], and the semantic equivalence of source and target model is proven. However, this direction of work seems to have been abandoned.

There is a large body of work on using (triple) graph grammars for model transformations. While they have a well-defined mathematical semantics and background theory—like confluence, termination, or modularity—(see e.g. the books by Rozenberg [46] or Ehrig et al. [13]), a verification tool to prove specific properties of individual transformations is missing. This means that a transformation is usually translated into another formalism.

Varró [56] (and Pataricza [58]) develop a framework based on graph transformations that allows to model check properties of the source and target model of a transformation. The properties are also defined by graph transformations. Transformations and properties are translated into a transition system (a Kripke structure), and then can be checked with the SAL tool suite [47]. However, this does not prove that a transformation preserves the properties for arbitrary well-formed source models (cf. the quote above). It is not clear whether this framework is also applicable to the extended graph transformation language VIATRA2 by Varró and Balogh [57], since this language supports imperative constructs and recursion.

Another possible formalism is rewrite logic that can also be used to specify transformations directly. Boronat et al. [5] support relational "QVT-like" model transformations in the Maude tool [10]. It is possible to check automatically for violations of invariants that are given by the user. This problem is undecidable for an infinite state space, i.e. failure to find a violation does not mean that there is none. Therefore, a full verification of a transformation is not possible. Another possibility is to model check propositional Linear Temporal Logic formulas. However, propositional LTL is too restricted to express complex properties of generated Java programs. Lucas and Toval [32] also formalize a subset of relational QVT in Maude, and check that two models are equivalent w.r.t. the transformation.

ATL [26] is a transformation language that is also included in the Eclipse modeling framework. It supports pattern-based rules and imperative constructs, and started with only an informal semantics. Troya and Vallecillo [55] formalize ATL with rewrite logic in Maude. Their main goal is to give a formal semantics to ATL. As a benefit, Maude's tool support for analysis can be used. The tool suite Henshin [2] by Taentzer et al. is based on graph transformations. It contains tools to detect deadlocks for concrete source models (i.e. that the transformation will get stuck without applying all rules), to verify propositional LTL formulas, and to check invariants as in Maude. A full verification of transformations is not possible. GROOVE [44] by Rensink and Kastenberg is a tool for modeling object-oriented systems with the help of graph transformations. Given a finite-state space, e.g. by restricting the length of lists to three, it is possible to model check propositional CTL formulas [27].

There is very little work concerning the correctness of generated Java code which may be due to the complexity of Java. Engels et al. [14] transform activity diagrams to TAAL, a Java-like programming language, with graph transformations, and present a correctness notion based on trace equivalence. They can check the correctness for concrete input and output models, but their *"ultimate aim is a general proof of correctness for the transformation"*. Rahim and Whittle [43] transform state machines to Java code, and add annotations for a model checker to the Java code. The model checker can then be used to prove that the Java code behaves as the state

machine for every concrete input and output model. They argue that *"Verifying the generated output in this way is more efficient than formally verifying the transformation's definition."* This means the verification can fail any time because the output is not correct. In contrast, we aim at a full verification of the transformation itself.

8 Conclusion

We have presented a framework that allows the formal verification of Java code generation in a model-driven setting. It is based on a meta model for an annotated abstract Java syntax tree (JAST), a formal Java semantics, and operational QVT transformations from arbitrary models into a JAST model. The major ingredient is a calculus for operational QVT that allows formal reasoning about transformations. It is implemented in our theorem prover KIV.

Our target are transformations for security-critical systems, primarily in the context of our SecureMDD project. Here, a UML model extended with our MEL programming language of a security protocol is the basis for a formal specification where the security of the system can be proved, and for generating a Java implementation of the protocols. It is essential that the generated code is correct, i.e. a correct refinement of the formal specification. Since we generate Java Card code that runs on resource restricted smart cards, the generated code is not trivial. Serialization and deserialization must work correctly, and detect malicious input. Objects must be reused, values copied, the code must never throw run time exceptions, etc.

The transformations are several thousand lines long. In principle, it is now possible to prove these properties though the effort will be considerable. The work presented in this paper is just a starting point. Future work includes adding heuristics to the prover to increase automation, supporting more QVTO operations, incorporating input validation, and proving the correctness of our SecureMDD transformations.

Acknowledgments This work is partly sponsored by the Deutsche Forschungsgemeinschaft DFG with grant RE 828/8-1 (SecureMDD).

Appendix A: the rest of the example transformation

See Figs. 13 and 14.

Fig. 13 Operations creating JAST expressions

A The rest of the example transformation

```
1  query Property::assignThis(c : String) : Statement {
2    var r : Statement := null;
3    r := object ExprStm {
4    expr := object Assign {
5            expr1 := self.fa(thisExpr(c), c);
6            operator := '=';
7            expr2 := object LocVarAccess {
8              name := self.name; type := self.type.toJast();
9            };
10           type := self.type.toJast();
11   } };
12   return r;
13 }
14 query Property::fa(e:J::Expression,c:String):J::Expression{
15   return object FieldAccess {
16            expr := e;
17            category := object SimpleFieldCategory {
18                                  name := self.name;
19                                  classtype := classType(c);
20                                  isStatic := false;
21                                  type := self.type.toJast();
22                        };
23            type := self.type.toJast();
24          };
25 }
26 query thisExpr(c:String) : J::Expression {
27   return object LocVarAccess {
28            name := 'this'; type := classType(c); };
29 }
```

Fig. 14 Auxiliary operations

```
1  query Property::toParameter() : SingleVariableDeclaration {
2    return object SingleVariableDeclaration {
3             name := self.name;
4             type := self.type.toJast();
5            };
6  }
7  query packageName() : String {
8    return getModel().name;
9  }
10 query boolean() : J::PrimitiveType {
11   return object J::PrimitiveType { name := 'boolean';};
12 }
13 query int() : J::PrimitiveType {
14   return object J::PrimitiveType { name := 'int'; };
15 }
16 query void() : J::VoidType {
17   return object J::VoidType { };
18 }
19 query classType(c : String) : J::ClassType {
20   return object ClassType {
21      name := c;
22      qualifiedName := packageName() + '.' + c; };
23 }
24 query Type::toJast() : J::Type {
25   var resty : J::Type := null;
26   var n := self.name;
27   if (n = 'Boolean') then { resty := boolean(); }
28   else if (n = 'Integer') then { resty := int(); }
29   else { resty := classType(self.name); }
30   endif endif;
31   return resty;
32 }
```

Appendix B: main lemma for the type correctness proof

Below is a simplified version of the man inductive lemma for the mapping `css->map toJavaClass()` (line 12 in Fig. 5, line 10 below). [6] It is generalized to allow induction on then number of classes to map, and it contains the look ahead argument that eventually all classes will be type correct. The formula is explained below.

1. valid(in, UML), suitable(in), classrefs = select('packaged Element', in),
2. css ⊒ classrefs, unmapped(css, trace),
3. valid(out, JAST), out = out' + jcs, isJavaClasses(jcs), # jcs + # css = # classrefs,
4. ∀ tds. tds = model2sem(jcs)
5. → tds.names = firstN(# tds, classrefs.names)
6. ∧ parttc(tds) ∧ types∃ (tds.alltypes, classrefs)
7. ∧ ∀ td, cntxt. td ∈ tds ∧ td ∈ cntxt
8. ∧ types∃ (td.alltypes, cntxt)
9. → fulltc(td, cntxt)
10. ⊢ ⟨(in, out, trace) `res := css->map toJavaClass()`⟩
11. ∃jcs'. out = out' + jcs + jcs' ∧ res = pointsTo(jcs')
12. ∧ ∀ tds. tds = model2sem(jcs + jcs')
13. → tds.names = classrefs.names
14. ∧ parttc(tds) ∧ types∃ (tds.alltypes, classrefs)
15. ∧ ∀ td, cntxt. td ∈ tds ∧ td ∈ cntxt
16. ∧ types∃ (td.alltypes, cntxt)
17. → fulltc(td, cntxt)

– **Generalized invariant:** The precondition describes a situation in the middle of the iteration. The original list is `classrefs`, the UML classes of the input model (line 1).
The classes in `css` have not yet been mapped (unmapped (css, trace), line 2), hence `css` is a postfix of `classrefs` (css ⊒ classrefs, line 2).
The output model contains some Java classes `jcs` (isJavaClasses(jcs) and out = out' + jcs, line 3). The length of `css` plus the length of `jcs` equals the length of the original `classrefs` (# jcs + # css = # classrefs, line 3). The idea is that `jcs` holds the result of the already mapped UML classes.

[6] The original theorem is `toJavaClass-induction-fulltc` in specification `UMLCDSimple` at http://swt.informatik.uni-augsburg.de/kiv/projects/secureMDD/uml2jastSimple/project.xml.

Now let `tds` be the Java classes `jcs` converted to the formal semantics (line 4). Then their class names (tds.names, line 5) equal the first n names of the original UML classes (line 5).

- **Type correctness:** Theses Java classes `tds` are partially type correct (parttc(tds), line 6) and they contain only class types that are UML classes (types∃ (tds.alltypes, classrefs), line 6).
  Every Java class `td` that occurs in `tds` and in an arbitrary context `cntxt` (line 7) and where all types occurring anywhere in `td` also exist as Java classes in `cntxt` (types∃ (td.alltypes, cntxt), line 8) is (fully) type correct (fulltc(td, cntxt), line 9).
- **Postcondition:** Then the output model contains additional Java classes `jcs'` (line 11) that are referenced by the result variable `res`, and the Java classes from the precondition `jcs` plus `jcs'` together (tds = model2sem(jcs + jcs'), line 12) have *all* names of the original UML classes (tds.names = classrefs.names, line 13), i.e. the iteration has finished.

The rest of the postcondition (lines 14–17) is identical to the precondition (lines 6–9).

The postcondition implies that all Java classes are fully type correct.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: a challenging model transformation. In: MODELS 2007—10TH International Conference on Model Driven Engineering Languages and Systems. Springer LNCS 4735 (2007)
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: MODELS, Model Driven Engineering Languages and Systems, 13th International Conference. Springer LNCS 6394 (2010)
3. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Fundamental Approaches to Software Engineering. Springer LNCS 1783 (2000)
4. Beckert, B., Hähnle, R., Schmitt, P. (eds.): Verification of Object-Oriented Software: The KeY Approach. Springer LNAI 4334 (2007)
5. Boronat, A., Heckel, R., Meseguer, J.: Rewriting logic semantics and verification of model transformations. In: FASE 2009. Springer LNCS 5503 (2009)
6. Brucker, A., Krieger, M., Wolff, B.: Extending OCL with null-references: towards a formal semantics for OCL 2.1. In: MODELS 2009 Workshops. Springer LNCS 6002 (2010)
7. Brucker, A.D., Wolff, B.: The hol-ocl book. Technical Report 525, ETH Zürich (2006)
8. Büttner, F., Kuhlmann, M.: Shortcomings of the embedding of OCL into QVT ImperativeOCL. In: Workshops and Symposia at MODELS'08. Springer LNCS 5421 (2009)
9. Calegari, D., Szasz, N.: Verification of model transformations: a survey of the state-of-the-art. Electron. Notes Theor. Comput. Sci. **292**, 5–25 (2013)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude: A High-Performance Logical Framework. Springer LNCS 4350 (2007)
11. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
12. Drossopoulou, S., Eisenbach, S.: Describing the semantics of Java and proving type soundness. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java. Springer LNCS 1523 (1999)
13. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
14. Engels, G., Kleppe, A., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: From UML activities to TAAL: towards behaviour-preserving model transformations. In: ECMDA-FA 2008. Springer LNCS 5095 (2008)
15. Favre, L.: A formal foundation for metamodeling. In: Ada-Europe 2009, Proceedings. Springer LNCS 5570 (2009)
16. Giese, H., Glesner, S., Leitner, J., Schäfer, W., Wagner, R.: Towards verified model transformations. In: Proceedings of the MoDeVa Workshop at MoDELS'06 (2006)
17. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. Sci. Comput. Program. **69** (2007)
18. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, independence and consequences in UML and OCL Models. In: Dubois, C. (ed.) Proceedings of 3rd International Conference Test and Proof (TAP'2009), pp. 90–104. Springer, Berlin, LNCS 5668 (2009)
19. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java (tm) Language Specification, 3rd edn. Addison-Wesley, Boston (2005)
20. Grandy, H., Stenzel, K., Reif, W.: A refinement method for Java programs. In: Formal Methods for Open Object-Based Distributed Systems (FMOODS). Springer LNCS 4468 (2007)
21. Haneberg, D., Bäumler, S., Balser, M., Grandy, H., Ortmeier, F., Reif, W., Schellhorn, G., Schmitt, J., Stenzel, K.: The user interface of the KIV verification system: a system description. Electron. Notes Theor. Comput. Sci. UITP special issue (2006)
22. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
23. Huisman, M., Jacobs, B.: Java program verification via a Hoare logic with abrupt termination. In: Fundamental Approaches to Software Engineering (FASE'00). Springer LNCS 1783 (2000)
24. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
25. Jouault, F., Kurtev, I.: On the architectural alignment of ATL and QVT. In: 21st ACM Symposium on Applied Computing. ACM Press, New York (2006)
26. Jouault, Frédéric, Allilaire, Freddy, Bézivin, Jean, Kurtev, Ivan, Valduriez, Patrick: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
27. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Model Checking Software (SPIN). Springer LNCS 3925 (2006)
28. KIV homepage. http://www.informatik.uni-augsburg.de/swt/kiv
29. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, pp. 207–220. ACM, Big Sky, MT, USA, October (2009)
30. Krieger, M., Knapp, A.: Executing underspecified OCL operation contracts with a SAT solver. In: Proceedings of the 8th International Workshop on OCL Concepts and Tools (OCL 2008) at MoDELS 2008. Electronic Communications of the EASST, vol. 15 (2008)
31. Lano, K., Kolahdouz-Rahimi, S., Poernomo, I.: Comparative evaluation of model transformation specification approaches. Int. J. Softw. Inform. **6**(2), 233–269 (2012)

32. Lucas, F.J., Toval, A.: Model transformations powered by rewriting logic. In: Proceedings of the Forum at the CAiSE'08 Conference. CEUR Workshop Proceedings, 2008. http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-344/
33. Moebius, N., Stenzel, K., Grandy, H., Reif, W.: SecureMDD: a model-driven development method for secure smart card applications. In: Workshop on Secure Software Engineering, SecSE, at ARES 2009. IEEE Press (2009)
34. Moebius, N., Stenzel, K., Reif, W.: Modeling security-critical applications with UML in the secureMDD approach. Int. J. Adv. Softw. **1**(1), 59–79 (2008)
35. Moebius, N., Stenzel, K., Reif, W.: Generating formal specifications for security-critical applications: a model-driven approach. In: ICSE 2009 Workshop: International Workshop on Software Engineering for Secure Systems (SESS'09). IEEE/ACM Digital Library (2009)
36. Moebius, N., Stenzel, K., Reif, W.: Formal verification of application-specific security properties in a model-driven approach. In: Proceedings of ESSoS 2010: International Symposium on Engineering Secure Software and Systems. Springer LNCS 5965 (2010)
37. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS. Springer, Berlin (2002)
38. Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1 (2011)
39. Object Management Group (OMG). Meta Object Facility (MOF) Core, Version 2.4.1 (2011)
40. Object Management Group (OMG). Object Constraint Language, Version 2.3.1 (2012)
41. Queralt, A., Rull, G., Teniente, E., Farré, C., Urpí, T.: AuRUS: automated reasoning on UML/OCL schemas. In: ER 2010, 29th International Conference on Conceptual Modeling. Springer LNCS 6412 (2010)
42. QVT Operational (Eclipse Project). http://www.eclipse.org/projects/project_summary.php?projectid=modeling.m2m.qvt-oml
43. Rahim, L., Whittle, J.: Verifying semantic conformance of state machine-to-java code generators. In: MODELS 2010. Springer LNCS 6394 (2010)
44. Rensink, A.: The GROOVE simulator: a tool for state space generation. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE). Springer LNCS 3062 (2004)
45. Romeikat, R., Roser, S., Müllender, P., Bauer, B.: Translation of QVT relations into QVT operational mappings. In: ICMT 2008. Springer LNCS 5063 (2008)
46. Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, vol. 3. World Scientific Publishing Co., Singapore (1997)
47. The SAL symbolic analysis laboratory. http://sal.csl.sri.com/
48. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 94: Graph-Theoretic Concepts in Computer Science. Springer LNCS 903 (1994)
49. SmartQVT. http://sourceforge.net/projects/smartqvt/
50. Soeken, M., Wille, R., Dreschsler, R.: Encoding OCL data types for SAT-based verification of UML/OCL models. In: Proceedings of the Conference on Tests and Proofs (TAP) 2011. Springer LNCS 6706 (2011)
51. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, Berlin (2001)
52. Steinberg, D., Budensky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2009)
53. Stenzel, K.: A formally verified calculus for full Java Card. In: Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings. Springer LNCS 3116 (2004)
54. Stenzel, K.: Verification of Java Card Programs. PhD thesis, Faculty of Informatics, Augsburg University, Germany (2005)
55. Troya, J., Vallecillo, A.: Towards a rewriting logic semantics for ATL. In: ICMT 2010. Springer LNCS 6142 (2010)
56. Varró, Dániel: Automated formal verification of visual modeling languages by model checking. Softw. Syst. Model. **3**, 85–113 (2004)
57. Varró, Dániel, Balogh, András: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**, 214–234 (2007)
58. Varró, D., Pataricza, A.: Automated formal verification of model transformations. In: Jürjens, J., Rumpe, B., France, R., Fernandez, E.B. (eds.) CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop, number TUM-I0323 in Technical Report. Technische Universität München (2003)
59. von Oheimb, D., Nipkow, T.: Machine-checking the Java specification: proving type-safety. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java. Springer LNCS 1523 (1999)
60. Wirsing, M.: Algebraic specification. In: Handbook of Theoretical Computer Science, vol B, chapter 13, pp. 675–788. Elsevier, Oxford (1990)
61. XP and http://wiki.eclipse.org/Xpand

Author Biographies

Kurt Stenzel studied Computer Sciences in Karlsruhe, Germany, and began working as a research assistant there. After a period at the University of Ulm, Germany, he moved to Augsburg University where he received his Ph.D. in computer sciences in 2005. Since then he is working in Augsburg on formal Java verification, interactive theorem proving, model-driven development of security-critical systems, and information flow control.

Nina Moebius studied Computer Science at the University of Lübeck. Since 2006 she is working as a research assistant at the department of software engineering in Augsburg. Her doctoral thesis is about the model-driven development of secure Smart Card applications. Her research interests are security engineering, verification, model-driven development, software engineering in general, and information flow control.

Wolfgang Reif is full professor for software engineering at Augsburg University, Germany. He is vice president of the University of Augsburg, and director of the Institute for Software & Systems Engineering. He received his doctoral degree in computer science from the University of Karlsruhe, and held a professorship in software engineering at the University of Ulm from 1995 to 2000. His current research interests are software and systems engineering, safety, reliability and security, organic computing, as well as software-driven mechatronics & robotics. Wolfgang Reif is author of a large number of scientific publications, and is consultant to leading technology companies.