Self-Monitoring Software: Folding Assertions into Model-Generated Code

Anthony J H Simons

Department of Computer Science, University of Sheffield,
211 Portobello, Sheffield S1 4DP, UK
A.J.Simons@shef.ac.uk

Abstract. For model-driven engineering to achieve acceptance as a trustworthy technology, correctness properties must be preserved at every stage, including verifying that the compiled code in different target languages executes with the same semantics as expressed in the models. This paper reports on a series of experiments that fold Eiffel-style assertion monitoring into Java code generated from abstract models of programs. Assertions may be added incrementally to the models, and the model transformation performs suitable modifications to the target code, reorganising expressions, variables, exceptions and introducing new program structure to handle error recovery. The generated software monitors abstract properties expressed in the models, executing according to the programming-by-contract metaphor. The described strategy extends to code generation in multiple languages from the same models.

Keywords: Model transformation, model semantics, code generation, assertion monitoring, program correctness, Java, Eiffel.

1 Introduction

Model-driven engineering (MDE) is an ambitious and fast-developing strategy in software engineering for synthesizing software systems from high-level models that represent the abstract structure and behaviour of those systems. One of the biggest challenges facing MDE is the need to ensure correctness at every stage of the transformation process from abstract models to executing systems. The initial models must be checked for consistency and completeness, with respect to some declared model syntax and semantics. The transformation process must be proven correct, in the sense that each transformation step should execute in a verified, predictable way and map all the source model elements onto the correct target model elements, every time it is applied. Finally, the resulting compiled software systems should execute predictably, with the same semantics as expressed in the original models.

1.1 Model-to-Model: Focus on Transformation Correctness

Within the MDE community, the greatest attention has been focused on investigating model transformations. The impetus for this is provided by the Object Management Group's drive towards a specific Model-Driven Architecture (MDA) [1], which explicitly integrates other OMG standards such as the Unified Modeling Language (UML) for its notation [2], the Meta-Object Facility (MOF) [3] and Object Constraint Language (OCL) [4] for bootstrapping its syntax and semantics and the rule-based graph-matching strategy known as Query/Value/Transformation (QVT) for its model transformations [5]. Specifically, QVT is intended to express model transformations declaratively, all within the same meta-modelling framework as the rest of UML.

This has tended to set the agenda for model transformation research, highlighting the value of UML, OCL and QVT-style model transformation by rule. Declarative rules were adopted in Atlas Transformation Language, which expresses model transformations as morphisms from the source to the target metamodels [6]. Others prefer to describe ordered transformations imperatively, as in the meta-modelling language Kermeta [7], an object-oriented language enriched with constraints and set comprehensions. Others focus on the need to model the formal properties of QVT-like transformations, as in UML-RSDS [8], where translation rules are expressed as dependent quantifications ($\forall s \in$ source, $\exists t \in$ target), a mathematically elegant approach that offers the prospect of proving the correctness of transformations.

1.2 Model-to-System: Focus on Execution Semantics

One area that has so far received less attention is the need to ensure that generated software systems execute with predictable semantics. This is much harder to achieve than it sounds: take four potential target implementation languages, such as Java, C++, Eiffel and C#, and it is not long before you find ways in which these treat the same program differently (consider combinations of dynamic binding and visibility; or overriding and type redefinition [9]). So, the final code generation step needs as much care as the earlier model transformation steps.

This motivated the background for the work reported in this paper, which was to provide a common abstract model of object-oriented programming capable of being implemented in multiple target languages and nonetheless execute with the identical semantics in each. The model is expressed in a bespoke dialect of XML, called ReMoDeL OOP [10]. This captures a large subset of features common to strongly typed object-oriented languages, including: packages, classes, interfaces, basic types, generic types, fields, methods, overriding, dynamic binding, visibility, expressions, assignment, assertions and exceptions. Section 2 introduces the ReMoDeL transformation and code generation framework in general and describes the specific modelling language ReMoDeL OOP in more detail.

One of the more interesting challenges in ReMoDeL OOP was how to provide a mechanism for propagating the correctness properties expressed in an abstract design through to the generated code. For this, we were inspired by Eiffel's programming-by-contract metaphor [11], in which semantic assertions are converted into runtime checks placed at suitable points in the executable code and integrated with the

exception handling mechanism. As a means of providing a uniform, modular way of incorporating abstract specifications (c.f. OCL assertions [4]) into concrete designs, we found this attractive. Eiffel's single-minded attitude to guaranteeing correct execution, or failing gracefully, also appealed more than the *ad hoc* Java style of throwing and catching exceptions, which is sometimes abused as a kind of secondary control flow mechanism [12]. The standard monitoring protocol also supported our goal to generate systems with identical semantics in every target language.

1.3 Self-Monitoring Systems: Focus on Code-Folding

Correctness in ReMoDeL OOP is expressed via contracts, each a labelled assertion that guarantees some property of the executing system (c.f. Eiffel [11]). Assertions may be class invariants, method preconditions or method postconditions (c.f. OCL [4]). According to the programming-by-contract metaphor, breaking a postcondition or the invariant indicates a fault in the currently executing method, whereas breaking a precondition is the fault of whatever client code called current method. Exceptions must therefore be raised respectively in the current method, or its calling client. A method may choose to protect itself against failure, with an exception handler. The rescue-code may either clean up the local state and fail gracefully, or optionally reattempt the failed method, if it has a chance of succeeding. A method may therefore only succeed in its original purpose, or fail; it cannot bypass an assertion failure.

Translating from this model into Eiffel is quite straightforward, since the language supports these concepts directly. However, translating this into Java, C++ or C# is a trickier proposition, with regard to raising exceptions, protecting blocks of code and reattempting methods. Likewise, the ability to refer, in postconditions, to variables in their prior states, or to the result-expression of the method, requires a certain degree of code-restructuring and program synthesis.

We approached this as a kind of delayed code-folding problem, similar to aspect-oriented programming [13]. We wanted the code generators to produce as close to the optimal, idiomatic translation as possible in each target language. This meant that the introduction of different specification features would require separate, composable program transformations. Section 3 demonstrates how introducing different invariant, precondition and postcondition assertions into the OOP model affected the generation of executable systems in Java. The same techniques may be applied in other target languages, for which code generators exist in the ReMoDeL framework. More generally, this kind of technique may be used to add a self-monitoring capability to any software system. It may be contrasted with monitoring approaches that add monitors as wrappers, intercepting requests [14]. The approach reported here integrates the monitoring code more closely with the executing system.

2 The ReMoDeL Project

ReMoDeL is a research project based at the University of Sheffield, UK [10]. The acronym stands for *Reusable Model Design Languages*, highlighting the goal to develop related families of models, sharing common concepts, to describe software

systems at every level of abstraction. The name also reflects the intention to support software systems *remodelling*, the continual regeneration of software systems by adaptation at the model-level, in response to rapidly changing user requirements.

2.1 The Transformation and Generation Framework

Within the taxonomy of model transformation approaches [15, 16], ReMoDeL adopts a *forward-transformation* strategy, from high-level models to low-level models and executable code. It adopts the *direct manipulation* strategy, using algorithms encoded directly in Java to manipulate models encoded in XML. We chose this approach in order to prototype different kinds of transformation that were not yet well understood, using the familiar technology of Java and XML. We believed that this would offer a faster route to identifying, prototyping and ordering sets of transformations, than if we had invested effort in developing declarative, pattern-driven rules, c.f. [5, 6].

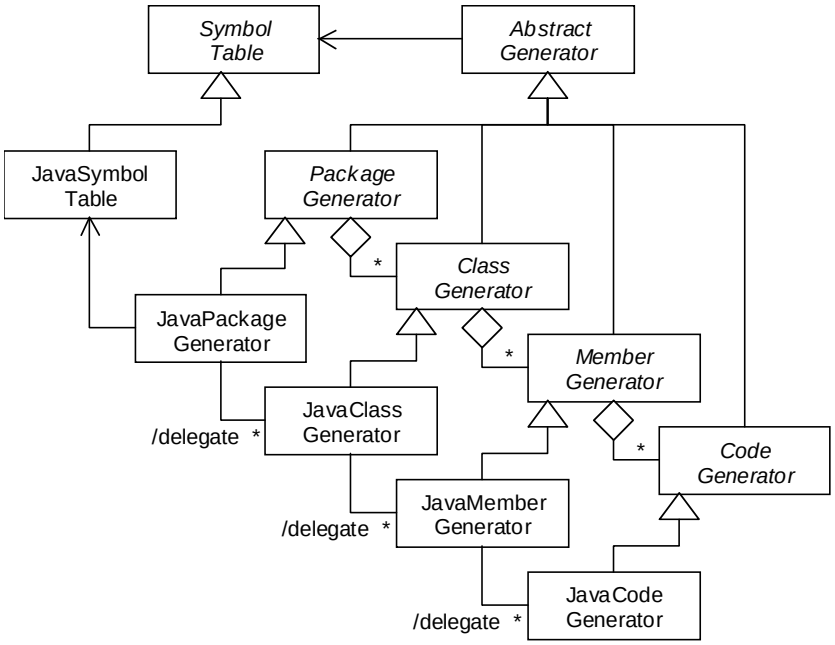

Fig. 1. The OOP generator framework, with specialised components for Java code generation. Generators delegate to subcontractors responsible for handling the next level of detail in the OOP model. Generators for specific languages specialise the abstract classes in the framework, which share common resources, such as symbol tables.

The model transformation architecture consists of three broad kinds of component: *Transformers*, which modify or optimise a model in-place, *Translators*, which map models from one domain to another, and *Generators*, which produce source code in one of the target languages. Figure 1 illustrates part of the *Generator* framework,

based around a compositional hierarchy of *PackageGenerator, ClassGenerator, MemberGenerator* and *CodeGenerator* components. These are responsible for the broad strategy of the code generation algorithm, but rely on more specific components for generating code in particular languages, such as the illustrated subclasses that generate Java code. So far, the framework contains components for generating Java, C++, C# and Eiffel. The *Generator* framework has a four-tier architecture, which follows naturally from the four levels of encapsulation in the OOP model: packages, classes, members and code. Other *Transformer* and *Translator* frameworks used in ReMoDeL follow a similar pattern, according to the models that they process.

2.2 Operation of the Generator Framework

The general model-processing strategy follows a fusion of the *Visitor* and *Composite* Design Patterns [17]. For example, a *JavaPackageGenerator* is the entry point for Java code generation and initiates the translation of an OOP *Package* model into the Java target language. It is responsible for creating the directory within which source files will be placed. It spawns one *JavaClassGenerator* delegate for each class, interface or other type declared in the OOP *Package* model (see 2.3 for an exposition of the model elements used in OOP).

A *JavaClassGenerator* is responsible for creating the file in which the generated source code will be placed, in the directory provided by its parent generator. It dispatches internally to different routines, according to whether the visited model is a *Class, Interface, Symbolic* or *Basic* type; and handles the generation of class-level dependencies on superclass, interface or component types. A *ClassGenerator* then spawns one *JavaMemberGenerator* delegate for each member found in the model.

A *JavaMemberGenerator* is responsible for translating the signature of each member, including its visibility and type. It dispatches internally to different routines, according to whether the visited model is a *Field, Creator* or *Method* member. It spawns *JavaCodeGenerators* as needed, to translate expressions such as field initialisers or code body sequences.

A *JavaCodeGenerator* dispatches internally on the kind of model expression, invoking itself recursively where required, and generates fully idiomatic, executable code in the Java programming language. The generated classes and interfaces may be compiled and executed along with a small, predefined kernel Java library. The same kernel is implemented slightly differently for each target language, and allows all code generated from models to interface with a common set of APIs.

2.3 The Object-Oriented Programming Model

We adopted a bottom-up strategy for the ReMoDeL project, starting with idiomatic system implementations and working backwards to identify what kinds of model and transformation might be used to generate them. That way, we would always have a proof-of-concept, in terms of working, generated systems. While at the higher end of the modelling spectrum we have conceptual data models, control flow graphs,

dataflow models and abstract state machines, it was clear that we first needed a common programming model from which to generate code.

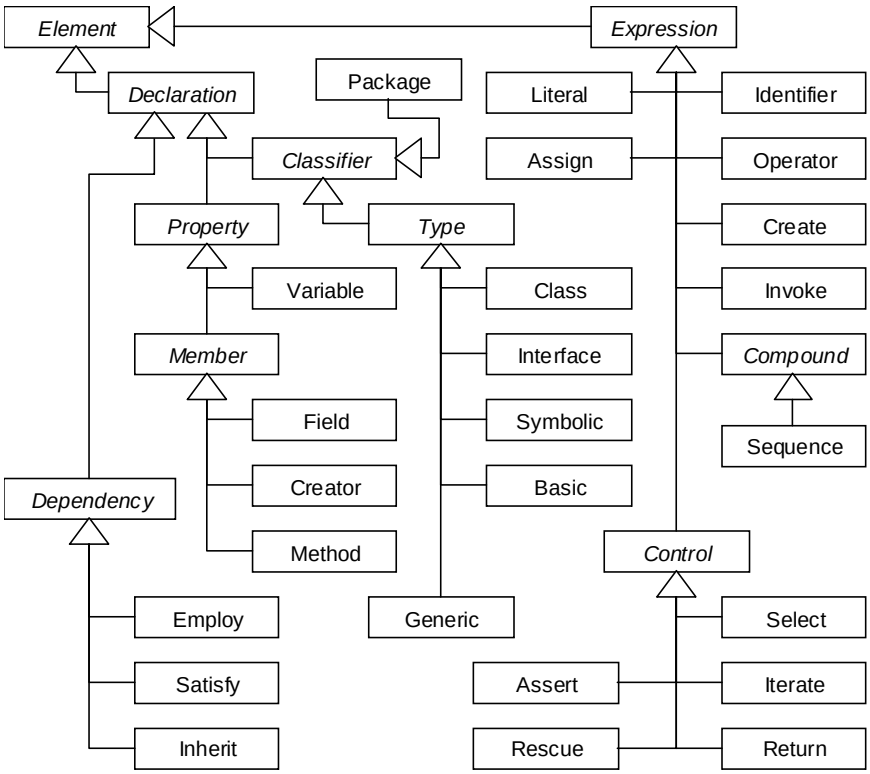

Fig. 2. Derivation of OOP programming concepts in the *ReMoDeL* meta-model. Terminal nodes correspond to XML elements used in OOP models of object-oriented programs.

The Object-Oriented Programming model, ReMoDeL OOP, was developed out of a simpler language of functions and expressions, known as ReMoDeL FUN [10]. This earlier work allowed us to prototype the kinds of expression nodes needed to represent executable code, such as literals, identifiers, selection and iteration. The major OOP concepts are illustrated in figure 2, which depicts the metamodel tree involved in the derivation of OOP. We used a common metamodel as a means of unifying similar concepts across different languages. For example, while both FUN and OOP have variables and identifiers, FUN has functions and application, whereas OOP has methods and invocation. The additional concepts in OOP include imperative programming, classification and inheritance, and assertions with exception handling. As few new nodes as possible were introduced in OOP, following a principle of minimalist design. Nodes represent program concepts at a higher level of abstraction than typical program code, for example the *Select* node represents both

single- and multi-branching selection, while the *Iterate* node represents all kinds of conditional, deterministic and quantified iteration.

2.4 Semantic Properties of the OOP Model

Here, we only have space to sketch the semantic properties of ReMoDeL OOP, which was designed to conform to a Common Semantic Model (CSM) for object-oriented programming, an operational model capable of being translated into multiple target languages. The CSM is in some ways the dual of Microsoft's Common Language Infrastructure (CLI) for the .NET platform [18], in that while the CLI defines a least upper bound, the CSM defines the greatest lower bound of all supported target language features. The OOP model was designed to support:

- Dynamic objects with reference semantics: most target languages allocate objects on the heap and recycle memory automatically – for the C++ translation, a special implementation pattern using smart pointers was adopted;
- Types, subtyping and generic types: objects are strongly typed, types conform in a subtype hierarchy and overriding is restricted to methods with the same signatures; parametric generic types are supported, with upper bounds [11];
- Single inheritance with multiple interfaces: multiple inheritance is supported only by some languages, C++ and Eiffel – but these may mimic the single-derivation, multiple conformance strategy of Java and C# [11];
- No name overloading: no within-class name overloading is permitted, yielding a one name per feature style – to avoid unintended method hiding [9];
- Redefinition and dynamic binding: any redefined method may expect dynamic binding, if declared *public* or *protected – private* methods may not be so redefined, since target languages may unpredictably hide, or bind dynamically [9];
- Method invocation: invocation expects a target object or expression, which may be implicit, in which case the target is *self*, the currently executing object;
- Object construction: creation expressions return objects, whereas creator methods are void procedures – to finesse functional and imperative initialisation styles;
- Namespaces and encapsulation: class members may have *private*, *protected* and *public* visibility; and packaged classes may have *private* or *public* export status;
- Assertions and exception handling: assertion monitoring uses the programming-by-contract metaphor [11] to raise and handle exceptions – handlers may clean up and fail, or reattempt the failed method and succeed.

Examples of the concrete XML syntax used to express OOP are given in section 3 below (see also fig. 2).

3 Folding Assertions into Code

Below, we present a working example of the translation of ReMoDeL OOP into Java. The example is taken from the *Finance* case study, one of several used to motivate the OOP model and translation algorithms [10]. The example is developed by first giving

the plain translation of a class model into Java, then showing progressively how the translation is modified, after adding assertions, and later, an exception handler.

3.1 The *SavingsAccount* Class from the *Finance* Package

Altogether, the *Finance* package defines an enumerated type, *Status*, describing the status of an account; an interface type *Asset* representing something with financial value; an abstract class *Account* that satisfies *Asset*, defines the notion of a *balance* and the abstract methods *deposit* and *withraw*; and a concrete *SavingsAccount* that inherits from *Account*. Listing 1 shows a fragment of this model, depicting the *SavingsAccount* class, prior to adding any assertions.

Listing 1. An OOP class model, defining a *SavingsAccount* class. The class depends on three other types and defines one creator and three methods.

```
<Class name="SavingsAccount" visible="public">
  <Inherit refer="Account" kind="Class" from="Finance"/>
  <Employ refer="Person" kind="Class" from="People"/>
  <Creator name="makeWith" type="Void" visible="public">
    <Variable name="holder" type="Person"/>
    <Variable name="amount" type="Integer"/>
    <Sequence type="Void">
      <Invoke method="openWith" implicit="true" type="Void">
        <Identifier name="holder" type="Person"/>
        <Identifier name="amount" type="Integer"/>
      </Invoke>
    </Sequence>
  </Creator>
  <Method name="deposit" type="Void" visible="public">
    <Variable name="amount" type="Integer"/>
    <Sequence type="Void">
      <Assign symbol="assign" type="Integer">
        <Identifier name="balance" type="Integer"
            scope="object"/>
        <Operator symbol="plus" type="Integer">
          <Identifier name="balance" type="Integer"
              scope="object"/>
          <Identifier name="amount" type="Integer"/>
        </Operator>
      </Assign>
    </Sequence>
  </Method>
  <Method name="withdraw" type="Integer" visible="public">
    <Variable name="amount" type="Integer"/>
    <Sequence type="Integer">
      <Assign symbol="assign" type="Integer">
        <Identifier name="balance" type="Integer"
            scope="object"/>
        <Operator symbol="minus" type="Integer">
          <Identifier name="balance" type="Integer"
```

```
            scope="object"/>
          <Identifier name="amount" type="Integer"/>
        </Operator>
      </Assign>
      <Return type="Integer">
        <Identifier name="amount" type="Integer"/>
      </Return>
    </Sequence>
  </Method>
</Class>
```

This shows the style of XML syntax used in the language-agnostic OOP model. The *SavingsAccount* class depends on the inherited superclass *Account* and the class *Person*, employed from another package. It defines a creator *makeWith*, to initialise *SavingsAccount* objects, and methods *deposit* and *withdraw*, which override abstract methods inherited from *Account*. The method bodies respectively add, or subtract an *amount* from the inherited *balance* field. Whereas *deposit* returns no result, *withdraw* returns the amount withdrawn. Each method body consists of a single *Sequence*, containing further nested expressions. All expressions are typed (resolved by the process that builds the OOP model) and some identifiers have scope attributes. So, the OOP model has the flavour of a machine-readable annotated parse tree that also supports visual inspection by humans.

Listing 2. Translation of the OOP *SavingsAccount* class model into the Java programming language. This text is saved as the file *SavingsAccount.java*.

```
package example.finance;

import example.people.Person;

class SavingsAccount extends Account {

  public SavingsAccount(Person holder, int amount) {
    openWith(holder, amount);
  }

  public void deposit(int amount) {
    balance = balance + amount;
  }

  public int withdraw(int amount) {
    balance = balance – amount;
    return balance;
  }
}
```

The translation of the example OOP model into the Java programming language is given in listing 2. This shows obvious mappings into Java, such as the translation of method names and types, and the recursive translation of expressions, all emitted by generators visiting each OOP node in turn, as described in section 2.2. Canonical OOP types are mapped to Java types using a *JavaSymbolTable* (see fig. 1), for

example, canonical *Integer* maps to the primitive Java type *int*. The listing also illustrates the generation of a package namespace and the selective import of a type from outside this namespace (the locations are supplied by the OOP package model).

3.2 Adding Assertions to the *SavingsAccount* Class

In the next stage, we supplement the model with assertions. We add a class invariant, stating that the balance of a *SavingsAccount* must always remain in credit. We also add a precondition to the *deposit* method, requiring this to accept only positive amounts of money. Likewise, we add a postcondition to the *withdraw* method, ensuring that the prior balance is equal to the sum of the amount and the current balance. Listing 3 illustrates how these assertions are expressed in the OOP model. In general, many assertions of each kind may be added to the model.

Listing 3. Adding assertions to the OOP *SavingsAccount* model from listing 1. Only the extra elements are shown in full; other elements from listing 1 have been elided, for brevity.

```
<Class name="SavingsAccount" visible="public">
  ...
  <Assert contract="balance in credit" when="always">
    <Operator symbol="noLessThan" type="Boolean">
      <Identifier name="balance" type="Integer"
        scope="object"/>
      <Literal value="0" type="Integer"/>
    </Operator>
  </Assert>
  <Method name="deposit" type="Void" visible="public">
    <Variable name="amount" type="Integer"/>
    <Assert contract="amount is positive" when="before">
      <Operator symbol="moreThan" type="Boolean">
        <Identifier name="amount" type="Integer"/>
        <Literal value="0" type="Integer"/>
      </Operator>
    </Assert>
    ...
  </Method>
  <Method name="withdraw" type="Integer" visible="public">
    <Variable name="amount" type="Integer"/>
    <Assert contract="amount was debited" when="after">
      <Operator symbol="equals" type="Boolean">
        <Identifier name="balance" type="Integer"
            scope="object" old="true"/>
        <Operator symbol="plus" type="Integer">
          <Identifier name="balance" type="Integer"
              scope="object"/>
          <Identifier name="amount" type="Integer"/>
        </Operator>
      </Operator>
    </Assert>
    ...
```

```
  </Method>
</Class>
```

Each *Assert* node represents an individual contract, a named semantic property requested in the design, which must be translated faithfully into code. The *Assert* node may stand for an invariant, pre- or postcondition, respectively depending on the *when*-attribute's value of *always*, *before* or *after*. *Assert* nodes are attached as children to the *Class* or *Method* nodes they constrain.

The asserted property is any single (possibly nested) *Boolean* expression, the only child of the *Assert* node. Whereas the invariant may only refer to class members, preconditions and postconditions may also refer to method arguments. Postconditions may also refer to variables in their prior states (indicated using the *old* attribute), or to the distinguished *result* identifier.

Listing 4. Translation of the extended OOP *SavingsAccount* model from listing 3 into Java, showing the result of folding in assertion monitoring code. Java comments were inserted by the code generators, for tracing purposes.

```
package example.finance;

import example.people.Person;

class SavingsAccount extends Account {

  public SavingsAccount(Person holder, int amount) {
    openWith(holder, amount);
  }

  protected void assertInvariant() {
    super.assertInvariant();
    if (balance < 0)
      brokenContract("invariant: balance in credit");
  }

  public void deposit(int amount) {
    if (amount <= 0)
      brokenContract("deposit: amount is positive");
    balance = balance + amount;
    assertInvariant();  // Check before exit
  }

  public int withdraw(int amount) {
    int result;
    int oldBalance = balance;
    balance = balance – amount;
    result = amount;  // Save the result
    if (oldBalance != (balance + amount))
      brokenContract("withdraw: amount was debited");
    assertInvariant();  // Check before return
    return result;  // Return saved result
  }
}
```

The Java translation of the model, including the assertions, is given in listing 4. From inspection, it is clear that both code synthesis and expression re-ordering has occurred. Below, we sketch the algorithm for folding in assertions:

- Class-level invariant assertions trigger the synthesis of *assertInvariant*, a protected method to check the invariant, to be generated with other members; this contains boilerplate to check the inherited invariant, and includes local assertions;
- Method-level postcondition expressions are scanned for the presence of references to the *result*, or to *old* variables in their prior state; this triggers the synthesis of extra local state variables, to be generated early in method bodies;
- Methods that contain assignments to local fields trigger an invariant check, placed last in (each branch of) the method body, unless a return-expression is found;
- Methods that contain return-expressions and invariant or postcondition checks trigger the synthesis of a local *result* variable to store the return-expression;
- Methods are generated in order of: method signatures, preconditions, local state variables with initialisation, method body, saved method result, postconditions, invariant check, and returned result; or *pro rata*, where applicable;
- Assertions are converted into guarded exceptions: each asserted contract property is converted into its logical complement, which acts as the guard condition. If the guard is satisfied, the *brokenContract* utility method will raise an exception.

This algorithm is not wholly declarative, but is partly dependent on the states of the generators. Tricky interactions occur, according to the combination of field assignments, posterior checks and return expressions encountered. For this, the code generator may need to reset multiple times to its starting state, especially when processing each branch of a *Select*-node, to ensure that the right result is temporarily saved and returned when all posterior checks are complete.

3.3 Adding an Exception Handler to the *withdraw* Method

In the next stage, we illustrate the addition of an exception handler to the model. In order to implement programming-by-contract [11] correctly, failed preconditions should be handled by the current method's caller, whereas failed postconditions may be cleaned up by the current method. Listing 5 shows a more developed version of the OOP *withdraw* method, declaring a precondition and an exception handler.

Listing 5. Revised OOP method model for *withdraw*, with a precondition and an exception handler. We assume that an invariant was also declared for the owning class.

```
<Method name="withdraw" type="Integer" visible="public">
  <Variable name="amount" type="Integer"/>
  <Assert contract="amount is positive" when="before">
    <Operator symbol="moreThan" type="Boolean">
      <Identifier name="amount" type="Integer"/>
      <Literal value="0" type="Integer"/>
    </Operator>
  </Assert>
```

```
  <Rescue type="Integer" attempts="2">
    <Sequence type="Integer">
      <Assign symbol="assign" type="Integer">
        <Identifier name="balance" type="Integer"
            scope="object"/>
        <Operator symbol="minus" type="Integer">
          <Identifier name="balance" type="Integer"
              scope="object"/>
          <Identifier name="amount" type="Integer"/>
        </Operator>
      </Assign>
      <Return type="Integer">
        <Identifier name="amount" type="Integer"/>
      </Return>
    </Sequence>
    <Sequence type="Void">
      <Assign symbol="assign" type="Integer">
        <Identifier name="balance" type="Integer"
            scope="object"/>
        <Operator symbol="plus" type="Integer">
          <Identifier name="balance" type="Integer"
              scope="object"/>
          <Identifier name="amount" type="Integer"/>
        </Operator>
      </Assign>
    </Sequence>
  </Rescue>
</Method>
```

The method body is now a *Rescue* node, denoting a protected environment, whose two *Sequence* children are the operational body code, and some recovery code, whose job is to roll back the *SavingsAccount* object to its prior state. The *Rescue* node may optionally specify the number of times a method may be attempted. Reattempted methods may refer to the distinguished *attempts* variable, to track their progress.

Listing 6 shows the translation into Java, assuming that *SavingsAccount* also declared an invariant to ensure that the balance never goes negative. This generates code with both prior and posterior checks, so that we may observe the context of raised exceptions.

Listing 6. Translation of the OOP method model for *withdraw*, from listing 5, into Java. The generated code obeys the programming-by-contract principle [11] in the way exceptions are handled, refusing failed preconditions, but rescuing failed postconditions and invariants.

```
public int withdraw(int amount) {
  if (amount <= 0)
    brokenContract("withdraw: amount is positive");
  int result;
  int attempts = 2;
  while (attempts > 0) {
    try {
      balance = balance – amount;
```

```
        result = amount;  // Save the result
        assertInvariant();  // Check before return
        return result;  // Return saved result
      }
      catch (BrokenContract broken) {
        balance = balance + amount;
        if (--attempts == 0)
          throw broken;  // Fail eventually
      }
    }
  }
```

From inspection, it is clear that substantial code synthesis is required to obtain the desired behaviour. We sketch the actions of the folding algorithm below:

- The presence of a rescued method body triggers the synthesis of a Java *try…catch* construction, in which the operational body code is placed in the *try*-block and the recovery code is placed in the *catch*-block;
- If multiple attempts are specified, this triggers the synthesis of a Java *while*-loop surrounding the *try...catch*, and an associated *attempts* loop control variable;
- Reattempted operational body code must either *return*, or else set the *attempts* control variable to zero, the loop exit condition, upon normal termination;
- Reattempted recovery code must decrement the *attempts* control variable after cleaning up and re-throw the exception when zero is reached; if not reattempted, it must simply re-throw the exception after cleaning up;
- Precondition checks are inserted at the head of the method, before the try-block; whereas postcondition and invariant checks are placed last inside the try-block, but before any return statement.

Notice how violated preconditions must now be handled by this method's caller, whereas violated postconditions and invariants fall under the protection of this method's own handler, satisfying the programming-by-contract principle.

4 Conclusions

In this paper, we presented a folding approach to generating self-monitoring software, which recovers from failure according to the programming-by-contract rule [11]. The monitored properties and related exception handlers are expressed in a modular way, in an abstract programming model. The folding algorithms were developed as methods of the code generator classes, some of which are quite complex. Whereas some steps could be encoded as pattern-driven rules, others depend mostly on the states of the generation process. We believe that this kind of example should motivate interest in *direct manipulation* approaches to model transformation [7, 10], which outperform other approaches in terms of the ease with which they can produce quite sophisticated, multi-layered and idiomatic transformations. Folding approaches to model transformation are clearly needed to develop strategies for combining partial and orthogonal abstract views of software systems, to advance the state-of-the-art in model-driven engineering.

References

1. Object Management Group: MDA Guide, Version 1.0.1, Miller, J., Mukerji, J. (eds.), 12 June (2003)
2. Object Management Group: Unified Modeling Language (OMG UML) Superstructure, Version 2.3, 5 May (2010)
3. Object Management Group: Meta Object Facility (MOF) Core Specification, Version 2.0, 1 January (2001)
4. Object Management Group: Object Constraint Language, Version 2.0, 1 May (2006)
5. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0, 3 April (2008)
6. ATL: A Model Transformation Technology, http://www.eclipse.org/atl/
7. Kermeta: Triskell Metamodelling Kernel, http://www.kermeta.org/
8. Lano, K.: A Compositional Semantics of UML-RSDS. Software and Systems Modeling 8, 85--116 (2009)
9. Beugnard, A.: Une Comparaison de Langages Objet Relative au Traitement de la Redéfinition de Méthode et à la Liaison Dynamique. L'Objet, 8 (1-2), 99--114 (2002)
10. ReMoDeL Project, http://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/
11. Meyer, B.: Object-Oriented Software Construction, 2nd ed., Prentice Hall, Redwood CA (1997)
12. Kinriy, J.: Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application. In: Doney, C., Knudsen, J.L., Romanovsky, A.B., Tripathi, A. (eds.) Exception Handling 2006, LNCS, vol. 4119, pp. 288--300. Springer, Heidelberg (2006)
13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes C. V., Loingtier, J., and Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997, LNCS, vol. 1241, pp. 220--242. Springer, Heidelberg (1997)
14. Bratanis, K., Dranidis, D. and Simons, A. J. H.: An Extensible Architecture for the Run-time Monitoring of Web Services. In: Karastoyanova, D., Kazhamiakin, R., Metzger, A. (eds.) Proc. 3rd. Int. Workshop on Monitoring, Adaptation and Beyond, ACM, New York, pp. 9--16 (2010)
15. Mens, T. and van Gorp, P.: A Taxonomy of Model Transformations. ENTCS, vol. 152, Elsevier, 125--142 (2006)
16. Biehl, M.: Literature Study on Model Transformations. Technical Report, Embedded Control Systems, KTH, Stockholm (2010)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Design. Addison Wesley, Reading MA (1995)
18. ECMA International: Standard ECMA-335, Common Language Infrastructure (CLI). ISO/IEC 23271:2006, June (2006)