Practical Model-to-Code Transformation in Four Object-Oriented Programming Languages

Anthony J H Simons, Ahmad F Subahi and Stephen M T Eyre,

Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello, Sheffield S1 4DP, UK
{A.Simons, A.Subahi, aca06se}@dcs.shef.ac.uk

Abstract. Model-driven engineering (MDE) seeks to raise the level of abstraction at which software systems are constructed. While recent work has focused on high-level model-to-model transformations, less attention has been devoted to the final model-to-code transformation step. This paper describes a practical framework that generates idiomatic code in four different object-oriented languages, Java, C++, Eiffel and C#, starting from an intermediate model expressed as an XML parse tree. Effort was directed towards ensuring that the generated systems executed with identical semantics. The code generators are provided as an object-oriented framework, following a compositional strategy pattern, which is readily adapted for new target languages. This framework constitutes the bottom tier in a future architecture for MDE by layered representational transformations.

Keywords: Model-driven engineering, model-driven architecture, model-to-code, code generation, object-oriented programming, ReMoDeL.

1 Introduction

Model-driven engineering (MDE) is an ambitious, fast-developing strategy in software engineering for synthesizing software systems from high-level models that represent the abstract structure and behaviour of those systems. Specific incarnations of the approach include the Object Management Group's drive towards a 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) and Object Constraint Language (OCL) for bootstrapping its syntax and semantics [3, 4] and the rule-based graph-matching Query-Value-Transformation (QVT) strategy for its model transformations [5]. This standards-driven approach has the advantage of leveraging existing conceptual design architectures, but the disadvantage that it offers would-be adopters a very steep learning curve before they can master the wealth of dense technologies.

The work reported in this paper is in some ways a reaction to the large-scale, standards-driven approach, in that it seeks to demonstrate proofs of MDE using simple, available technologies such as XML and Java. The over-arching ambition of the project, which is called *ReMoDeL: Reusable Model Design Languages* [6], is to

develop a complete path of representational transformations, from high-level reusable business models to low-level system models and executable code, by a layered series of transformations, which bring different design constraints to bear at each level. We consider it an open research question regarding what kinds of representations (models) should exist at each level, what kinds of system view they should represent and how the different views should be folded together (perhaps in the style of aspect-oriented programming) to yield a consistent, executable system. So, rather than posit the need for specific architectural layers top-down (such as the OMG's CIM, PIM and PSM), we thought it more profitable to motivate models bottom-up, by identifying first what kind of content might be needed to support full, idiomatic code generation in a variety of popular object-oriented languages.

The result of this experiment was the specification for an abstract object-oriented programming model known as ReMoDeL OOP [7], not so much a domain-specific language as an annotated parse-tree stored as XML data (to be created from higher-level models in the longer term). The advantages of using XML were clear: not only were there many existing tools available to parse and save the models, but also the content and structure of the OOP model could readily be adapted as the information requirements for full, idiomatic code generation were discovered.

As the OOP model stabilized, a particular architecture for code generation emerged as the "obvious" or most elegant strategy for building code generators for different programming languages. The code generator framework was constructed in Java, consisting of a compositional hierarchy of generators, each having the specific knowledge to generate a fragment of the target code. The framework exemplifies a fusion of the *Strategy*, *Composite* and *Abstract Factory* Design Patterns [8]. Translation rules were expressed directly as imperative algorithms, encoded as the methods of the generators. This is in marked contrast to other current work [9, 10, 11], which emphasizes the development of novel declarative languages for graph-based pattern matching and graph transformation, according to the agenda set by QVT [5]. Our decision was motivated by the desire to accomplish some significantly complex transformations via a more direct route. In this way, effort was invested more in identifying interesting transformations, and less on expressing each transformation declaratively within a standard meta-modelling framework.

In the rest of this paper, section 2 discusses the notion of a Common Semantic Model for code generation in different object-oriented programming languages. Section 3 describes the generator framework and how this acts on the OOP model. Section 4 illustrates some examples of code generation in Java, Eiffel and C++. Section 5 concludes with a comparison between our approach and the contrasting approaches taken by ATL [9], Kermeta [10] and UML-RSDS [11].

2 Common Semantic Model

Models of software systems should have, in their own terms, a precise and unambiguous semantics, such that systems generated from them should behave in predictable ways, no matter in what language or on what platform they execute. We therefore wanted to define a single, unambiguous operational semantics for the OOP

model. In support of this, we surveyed the kinds of language features offered in popular, strongly-typed object-oriented programming languages that contrasted strongly in style (in particular, Java, C++ and Eiffel) and tried to determine the largest overlapping subset of features that could either be directly supported, or simulated by smart code generation, in all of the intended target languages.

This became the basis for the *Common Semantic Model* (CSM) for the core features of OOP, a safe zone in which models could be guaranteed to translate with the same operational behaviour into every target language. OOP would also support an *Extended Semantic Model* (ESM), in which optional features might have realizable translations in a subset of the target languages, but not in all. The CSM is in some ways the dual of Microsoft's Common Language Infrastructure (CLI) model for the .NET platform [12], in that whereas the CLI defines the least superset of all language features supported by the .NET platform, the CSM defines the greatest common subset of the target languages' features.

2.1 Dynamic Objects with Reference Semantics

We determined at the outset that the OOP model should support strongly typed objects, dynamic object creation and automatic recycling, coupled with reference semantics for all object variables. This is standard in Java and C#, and the default in Eiffel. To achieve the same in the C++ translation required a special treatment, since by default all variables have value semantics in C++. The eventual C++ translation relies on a base library containing a reference-counted generic smart pointer, *Ref*<*T*>. All object references are translated as instantiations of this pointer type, which is specialized for each type *T*. Smart pointers manage the lifetime of dynamic objects, which are created freely on the heap using the C++ *new* operator.

OOP also supports a distinction between *Basic* types and *Class* types, but only from the viewpoint that the *Basic* types are atomic. Instances of the canonical OOP types *Integer* or *Decimal* are considered first-class objects, but with immutable content. This allows translations to optimise whether these types are implemented with value or reference semantics, and to perform automatic boxing and un-boxing (viz. promoting values to objects, demoting objects to values).

2.2 Multiple versus Single Inheritance and Interfaces

Different target languages support different models of inheritance. Whereas classes in Java and C# inherit from at most one superclass, but may satisfy multiple interfaces, C++ and Eiffel support multiple inheritance, with no need for distinct interface types. The largest area of overlap was therefore to adopt the Java and C# viewpoint for the *Common Semantic Model*, and have the translations for Eiffel and C++ mimic the behaviour of pure interfaces (as found in Eiffel's *Marriage of Convenience* code idiom [13]), using wholly abstract classes with deferred methods (in Eiffel) or *pure virtual* functions (in C++). Every class in the OOP model may inherit from at most one concrete superclass, but may satisfy many abstract interfaces.

Subclasses may override and redefine methods, so long as the redefined method's signature matches the original method's signature.

In the *Extended Semantic Model*, it is possible to construct OOP models with multiple inheritance, so long as code generation is restricted to target languages supporting multiple inheritance (such as C++ and Eiffel). In this case, a class should inherit the distributed union of the features of its parents. It should merge identical declarations inherited multiple times (via fork-join paths in the class hierarchy), but force the designer to select how to combine non-identical inherited features, derived from a common abstract base feature, by explicit method combination.

2.3 Overloading Names and Dynamic Binding

Languages like Java and C++ support within-class overloading of methods and constructors, which are distinguished on the types of their arguments. Eiffel requires every class feature (attribute, constructor or method) to be uniquely named within the class's namespace. The CSM adopts Eiffel's position as the area of greatest common agreement. No further resolution need be performed on the types of arguments when selecting methods; and it is easier to determine which methods are being redefined, when dynamic binding is required. Duplicate names only occur in OOP when a method is being overridden in a subclass. If translators had global access to the class hierarchy, this would allow the automatic detection of the need for keywords influencing the binding strategy [14], such as *virtual* (in C++), *final* (in Java) and *redefine* (in Eiffel). Here, local translation units are assumed, with no global analysis.

2.4 Object Construction and Method Invocation

Target languages differ markedly in their style of object construction. The two main groups include languages that treat constructors as initialisation methods called after object allocation (Eiffel, Delphi) and those that treat constructors as distinguished functions returning new objects (Java, C++, C#). Eventually, special methods called *Creators* were designed for OOP, which were capable of being interpreted according to the preferred style of the target language. Having unique names, they translate into the creation procedures and init-methods (of Eiffel and Delphi). Having types, they translate into constructor functions (in Java, C# and C++).

Creation expressions mimic the syntax for method invocation in OOP. In the same way that a method invocation requires a target object (the receiver), but may sometimes omit the target, if the method is invoked implicitly on *self*, a creation expression in OOP expects a target variable to initialise, but may leave this implicit, in which case the result of creation is passed outward to the next enclosing expression.

Translators are free to choose the most efficient way to construct object fields (*assignment* in Java, but cheaper *initialisation* in C++). Similarly, the way in which values are passed back to superclass constructors can be treated as renamed creation calls in Eiffel, super-invocation in Java and superclass constructor calls in C++. In the *Extended Semantic Model*, multiple super-invocations are supported by naming

each inherited super-object explicitly, so that all super-invocations may be suitably resolved against the correct parent.

2.5 Encapsulation, Namespaces and Sharing

Target languages differ in how they encapsulate classes at the package level and class members at the class level. The greatest area of agreement was to adopt the three visibility levels *private, protected* and *public* for class members in the CSM, rather than offer selective visibility to different clients (as in Eiffel). The ESM also supports *package* visibility for members, which translates directly in Java, but commutes to *protected* visibility in C++ with special *friend* access granted to classes in the same package. A similar treatment for Eiffel can be applied using selective exports.

Packages in OOP have unique names and locations. The name translates directly into a *namespace* in C++, whereas the location identifier serves a similar role in Java and C#. In Eiffel, package namespaces can only be simulated by prefixing class names with package names. At package-level, the CSM allows classes to be declared *public* (exported from the package) or *private* (kept secret within the package), which in Eiffel must be folded in with feature visibility declarations.

When considering shared values and class constants, OOP supports only shared (static) fields in the CSM, but not static methods (invoked against class objects), since some languages do not support runtime class objects. Interface types may declare constants as shared fields with initial values, implemented as static attributes in most languages and as *once functions* in Eiffel.

2.6 Assertions, Exception Handling and Recovery

One of the goals of the wider project is to have self-validation and testing procedures at every level of model transformation and code generation. Part of this obligation involves run-time monitoring of the generated software's correct behaviour. For this, the CSM adopts Eiffel's *programming by contract* metaphor [13], according to which software may only succeed completely, or fail gracefully, handing the failure back to the caller. This decision was informed by a comparison of exception handling strategies [15], which convinced us that the Java and C++ *try-catch* model was open to abuse as an alternative kind of control structure.

Correctness in OOP is monitored through *Assert* clauses, which, depending on their location at class-, signature- or code body level, are treated respectively as invariants, preconditions or postconditions. Recovery may only clean up and fail gracefully, or reattempt the failed method and succeed. This translates directly into Eiffel, but requires special treatments in other languages. In Java, for example, a class invariant translates into a protected method, called in every other method that modifies object states. Breaking a contract either raises an exception in the executing method (for invariants and postconditions), or in the caller (for preconditions). This, together with the possibility of recovery, leads to interesting transformations that generate *try*- and *catch*-blocks, *while*-loops, *if*-statements and *throw*-statements.

3 The OOP Code Generation Framework

A generic model for object-oriented programming, called *ReMoDeL* OOP [7], was developed and cast into a dialect of XML, sufficient to capture the kinds of information required by the CSM and ESM (see section 2). The generator framework was designed around the hierarchical structure of the OOP model, which conforms to a common meta-model, used by the whole *ReMoDeL* family of models.

3.1 Conceptual Design of the OOP Model

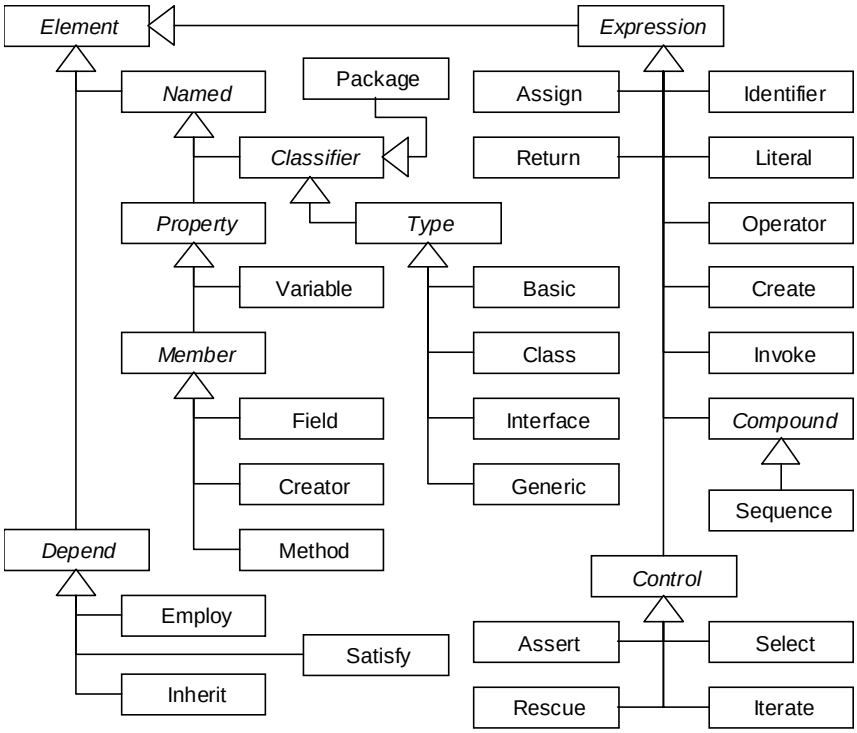

Fig. 1. Hierarchy of the major OOP concepts in the *ReMoDeL* family meta-model. Terminal nodes correspond to XML elements used in OOP models to describe object-oriented programs.

Like all the models within the *ReMoDeL* family, the *Object-Oriented Programming* model (OOP) was designed according to a conceptual hierarchy, or ontology, in which similar concepts appear as siblings, derived from common parent concepts (fig. 1). For example, all the elements representing object-oriented types (*Basic*, *Class*, *Interface* or *Generic*) specialise a common element *Type*, which is also

a primary concept in other models, such as the *Database and Query* model (DBQ) [16], which offers a different, intersecting family of types (*Basic*, *Record* and *Table*). In this way, different sets of concepts may be present in different *ReMoDeL* models, but all concepts may be described within a common meta-model.

Our earliest modelling experiments built an explicit meta-model in Java, with classes corresponding to the abstract syntax tree (AST) nodes in fig. 1. However, we found that this approach was fragile, due to the constant evolution of the conceptual ontology. Eventually, we found it much quicker to build models directly in XML. The loss of strongly typed AST nodes was more than compensated by the ability to adjust the model frequently, as the need arose. Essentially, this moved the burden of type checking from the AST onto the transformation tools.

3.2 Concrete Design of the OOP XML Syntax

The design of the XML language to express OOP was determined by the conceptual meta-model, in that this provided a rationale for choosing between XML *elements* or XML *attributes* to express the information in the model. XML writing styles vary widely, with some preferring elements over attributes [17]. In this style, the declaration of a typed variable might appear as:

```
<Variable>
  <Name>count</Name>
  <Type>Integer</Type>
</Variable>
```

However, this makes it difficult to distinguish between primary and dependent concepts, since all are expressed equally as element nodes. Furthermore, the saved format (of a realistically large model) is excessively vertical in presentation and somewhat harder for a human reader to check. Instead, we restricted the use of XML elements to meta-model node concepts, and described dependent properties as XML attributes, in the style:

```
<Variable name="count" type="Integer"/>
```

This yielded a style in which contained elements really did correspond to logical substructure in the meta-model. The main guidelines were therefore to distinguish structures that were logically contained (using element subtrees), from information that was either attributive, or cross-referenced other structures (using attributes).

Apart from this, Occam's razor was applied to invent as few new elements as possible. For example, the same *Variable* element was used to model both local variables and method parameters. The same *Assert* element was used, within different contexts, to model invariants, preconditions and postconditions. A single *Select* element was used to model both *if*-statements and multi-branching selection; likewise a single *Iterate* element to model both conditional and deterministic iteration. Since the OOP model is intended primarily for machine processing, it resembles an annotated parse tree, rather than the direct rendering of the syntax of a programming language into XML, such as JavaML [18]. The latter contains more elements as syntactic sugar and more levels of structure to partition definitions for the sake of human readers. A larger working example of OOP is shown in section 4 below.

3.3 The Generator Framework Architecture

The architecture of the generator framework (fig. 2) was influenced directly by the hierarchical structure of the OOP model, which consists of *Packages*, containing *Types* (*Class*, *Basic* and *Interface* types), which in turn contain *Members* (*Fields*, *Creators* and *Methods*), which in turn contain *Expressions* (field initialisers, and body code sequences).

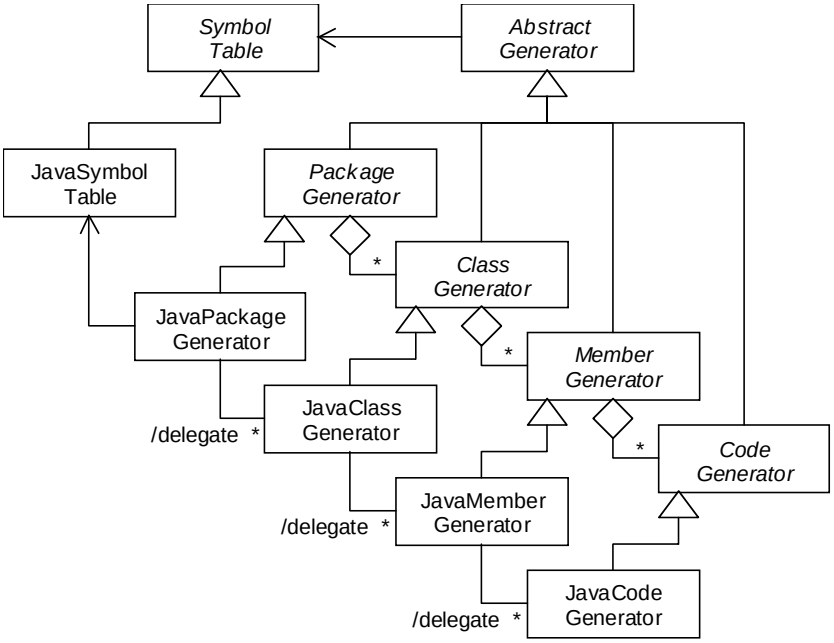

Fig. 2. The generator framework, with specialised components for Java generation. Generators delegate to subcontractors responsible for handling the next level of detail in the OOP model. Generators for specific languages specialise the abstract classes in the framework, which share common resources, such as symbol tables.

The architecture of the framework exhibits a number of well-attested Design Patterns [8]. Following the *Strategy* Design Pattern, different generator subclasses apply different strategies for code generation in different target languages. Following the *Abstract Factory* Design Pattern, specialised generators, such as *JavaPackageGenerator*, spawn a related family of language-specific generators (viz. *JavaClassGenerator, JavaMemberGenerator* and *JavaCodeGenerator*). Following the *Composite* Design Pattern, the command to *generate()* propagates through a bespoke compositional hierarchy of delegate generators, specific to each target language. The generators for C++, Eiffel and C# were provided in a similar way to the Java generators illustrated in fig 2.

3.4 Operation of the Framework

The root *AbstractGenerator* declares references to common resources that are eventually shared among many kinds of generator. These include the current output stream and the language symbol table. Every generator also refers to the OOP model segment it is translating. The root *SymbolTable* declares the protocol for mapping canonical OOP type and operator names into types and operators in the target languages and is specialised for each target language.

PackageGenerator is the entry-point into the translation framework, in that specific subclasses initiate the translation of an OOP *Package* model, which is the main translation unit, into a given target language. It is locally responsible for creating directory locations within which source files will be placed. It will spawn one *ClassGenerator* delegate for each class, interface or other type found in the OOP *Package* model. *ClassGenerator* is locally responsible for creating the file in which the generated source code will be placed, and for saving this file in the directory provided by its parent generator. Specific subclasses generate the type declaration for a *Class*, *Interface* or *Basic* type in a given target language, including any dependencies on a superclass, interfaces and component types. A *ClassGenerator* will spawn one delegate *MemberGenerator* for each *Field*, *Creator* (see section 2.4) or *Method* found in the OOP type model.

Each *MemberGenerator* subclass is locally responsible for translating the signature of each member and enforcing the consistency of visibility rules (e.g. public signatures in interface types). It spawns *CodeGenerators* as needed, to translate expressions such as field initialisers, precondition sequences or code body sequences. *CodeGenerator* subclasses generate fully idiomatic, executable code in the chosen target languages. Code generation assumes a small standard library for each target language, which maps the canonical OOP interfaces onto native classes.

4 Translations into Target Languages

A series of OOP models of increasing complexity were developed to validate the operation of the code generators [6]. These included: *Greeter*, a "hello world" style main program; *People*, a library of person-classes related by inheritance; *Finance*, a library of banking concepts, demonstrating interfaces and assertions; and *Sorting*, a generic sorted list type with a binary sorting algorithm. Generators were developed initially for Java and C++ and later for Eiffel and C#.

4.1 OOP *SavingsAccount* Example

The OOP *Finance* package model [6] provides a reasonably complete example that demonstrates most of the features of the code generators. Altogether, it defines an enumerated type, *Status*, describing the status of an account; an interface type *Asset* standing for the notion of value; an abstract class *Account* that satisfies the interface *Asset*; and a concrete class *SavingsAccount* that inherits from *Account*. Listing 1 shows a fragment of this model, depicting the *SavingsAccount* class:

List. 1. A fragment of the OOP *Finance* package model, defining a *SavingsAccount* class. A complete version of the *Finance* model may be viewed on the website [6].

```
<Class name="SavingsAccount">
  <Inherit from="Finance" location="example.finance">
    <Class name="Account"/>
  </Inherit>
  <Employ from="People" location="example.people">
    <Class name="Person"/>
  </Employ>
  <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>
  <Creator name="makeWith" type="SavingsAccount"
      visible="public">
    <Variable name="holder" type="Person"/>
    <Variable name="amount" type="Integer"/>
    <Sequence type="Void">
      <Invoke method="openWith" implicit="true">
        <Identifier name="holder" type="Person"/>
        <Identifier name="amount" type="Integer"/>
      </Invoke>
    </Sequence>
  </Creator>
  <Method name="deposit" type="Void" visible="public"
      override="true">
    <Variable name="amount" type="Integer"/>
    <Assert contract="account currently active"
        when="before">
      <Operator symbol="equals" type="Boolean">
        <Identifier name="status" type="Status"
            scope="object"/>
        <Literal value="active" type="Status"/>
      </Operator>
    </Assert>
    <Assert contract="positive deposit amount" when="before">
      <Operator symbol="moreThan" type="Boolean">
        <Identifier name="amount" type="Integer"/>
        <Literal value="0" type="Integer"/>
      </Operator>
    </Assert>
    <Sequence type="Void">
      <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"
    override="true">
  <Variable name="amount" type="Integer"/>
  <Assert contract="account currently active"
      when="before">
    <Operator symbol="equals" type="Boolean">
      <Identifier name="status" type="Status"
        scope="object"/>
      <Literal value="active" type="Status"/>
    </Operator>
  </Assert>
  <Assert contract="positive withdrawal amount"
      when="before">
    <Operator symbol="moreThan" type="Boolean">
      <Identifier name="amount" type="Integer"/>
      <Literal value="0" type="Integer"/>
    </Operator>
  </Assert>
  <Sequence type="Integer">
    <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>
    <Rescue retry="true">
      <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>
    </Rescue>
  </Sequence>
</Method>
</Class>
```

This illustrates the style of the XML syntax used in the programming language-neutral OOP model. The *SavingsAccount* class depends on two other classes, the inherited superclass *Account*, from the same package, and *Person*, which it employs

from a different package. It has a creator *makeWith* that creates a *SavingsAccount* by invoking the inherited method *openWith*, supplying a *Person* holder and *Integer* amount as arguments. It has an invariant asserting that the balance is always in credit. The methods *deposit* and *withdraw* override abstract versions inherited from *Account*. Each of these has a number of preconditions, asserting that the *SavingsAccount* is currently active, and that the amount is positive. The method bodies respectively add, or subtract the amount from the balance. Whereas *deposit* returns no result, *withdraw* returns the amount withdrawn. *Withdraw* also provides a rollback facility in its rescue clause, in case deducting the amount from the balance breaks the invariant.

Each method and creator body consists of a single *Sequence* expression, containing further expressions. All expressions are typed (expression types are resolved by the process that builds the OOP model) – this gives the flavour of an annotated parse tree and allows human readers to validate the initial models by inspection. Identifiers also carry scope information (a field has *object* scope; the default scope is *local*), permitting the reuse of the same names for both object fields and method arguments.

4.2 Translation from OOP into Java

The translation of the OOP model from listing 1 into the Java programming language is given in listing 2. This shows a number of obvious mappings to Java, but also illustrates a highly sophisticated treatment of assertions, explained below. The major achievement is in creating fully idiomatic Java code that faithfully executes the desired model semantics (see section 2). Any Java comments appearing in listing 2 were inserted by the generators, to highlight particular decisions.

When generating the illustrated code fragment, a *JavaClassGenerator* declares the Java package identifier for the *SavingsAccount* class based on the package's location. It inserts import statements only for those types that come from outside the current package. It generates the inheritance relationship with *Account*. It then delegates to a separate *JavaMemberGenerator* to generate each member.

List. 2. Translation of the OOP fragment corresponding to a *SavingsAccount* class into the Java programming language. This illustrates in particular the intelligent processing of assertions and recovery by the generator framework.

```
/**
 * SavingsAccount : generated on Tue Dec 07 22:58:15 GMT 2010
 * by ReMoDeL.
 */

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)
      throw new BrokenContract("balance in credit");
  }

  public void deposit(int amount) {
    if (status != Status.ACTIVE)
      throw new BrokenContract("account currently active");
    if (amount <= 0)
      throw new BrokenContract("positive deposit amount");
    balance = balance + amount;
    assertInvariant();  // Check before exit.
  }

  public int withdraw(int amount) {
    if (status != Status.ACTIVE)
      throw new BrokenContract("account currently active");
    if (amount <= 0)
      throw new BrokenContract("positive withdrawal amount");
    int methodAttempts = 2;  // Including 1 retry attempt(s).
    while (methodAttempts > 0) {
      try {
        balance = balance - amount;
        assertInvariant();  // Check before returning.
        return amount;
      }
      catch (BrokenContract broken) {
        balance = balance + amount;
        if (--methodAttempts == 0)
          throw broken;  // Fail.
      }
    }
  }
}
```

The Java constructor is created using the type information in the OOP creator to name the constructor. Each method name is generated from the OOP method name, and the OOP type is used to determine the result type. Canonical OOP types are mapped to suitable Java types using a *JavaSymbolTable* – canonical *Integer* maps to *int*, but canonical *Natural* (an unsigned type) maps to a *long int* in Java (which has no unsigned types) to ensure that 32 bit unsigned numbers may be represented.

The processing of assertions goes through a number of stages, which are encoded as algorithms of the various generator classes. Firstly, the class invariant assertion is picked up by the *JavaClassGenerator* and transformed, via a model-to-model transformation, into a *protected* method, having the standard name *assertInvariant()* and a boilerplate code body that invokes the super-invariant, to which the invariant assertions are added. This method is passed to a *JavaMemberGenerator* like any other method, whose body is processed by a *JavaCodeGenerator*.

When translating any assertions, a *JavaCodeGenerator* performs another model-to-model transformation, to convert the positive assertion of each contract property into a guard against violating that property, in which event an exception should be raised.

This transformation improves the idiomatic quality of the generated Java (rather than simply wrap the assertion with a layer of negation), and constructs the logical negation of the original asserted expression. This replaces inequality operators by their logical complement, applies de Morgan's law to *Boolean* operator expressions and simply negates any *Boolean*-valued method invocation.

The next step of the algorithm relies on the states of the *JavaCodeGenerator* as it processes expressions in a method body *Sequence*. If it encounters an assignment that alters object states (assigns to variables with *object* scope), this raises an obligation to check the class invariant before the method terminates. In the normal course, an invocation of *assertInvariant()* is placed last in the method body (see *deposit* in listing 2). However, if the generator encounters a return-statement, the obligation to check the invariant must be discharged immediately (see *withdraw* in listing 2), before the return expression. When processing multiple returning branches of a *Select* expression, the generator may reset multiple times.

The last layer of sophistication encodes Meyer's programming-by-contract [13] rule in Java. Any method that attempts to recover from failure (see *withdraw*) has a rescue-clause in OOP, containing the rollback code to restore the object's stable state. The *JavaCodeGenerator* wraps the protected method body in a *try*-block, with the rollback code placed in the *catch*-block. If the rescue-clause specifies that the method may be reattempted (by default once), then a *while*-loop is created around the *try-catch* construction. The method may nonetheless fail again after cleaning up (as is likely in this example), in which case the Java exception is passed back to the caller. Following the programming-by-contract rule, broken preconditions raise exceptions in the caller, whereas broken postconditions (here, the invariant check) raise an exception in the executing method [13].

4.3 Translation from OOP into Eiffel

The assertion handling behaviour described above is native to Eiffel, which expresses this directly in its *require*, *ensure* and *invariant* clauses. However, other aspects of the model have to be encoded specially in Eiffel, in order to obtain the semantics agreed in the CSM (see section 2). In particular, this requires a more sophisticated handling of package namespaces, member visibility and access methods. Listing 3 illustrates the translation of the OOP model into Eiffel.

The *EiffelPackageGenerator* examines all the dependencies expressed by each of the types in the package on other types, and from this creates an *Eiffel Configuration File*, an XML file used by the Eiffel compiler, directing it to import specific clusters (sets of classes) and identifying the system root (the entry point). While a *cluster* is analogous to a package, Eiffel does not support the notion of package namespaces: clusters exist in a flat namespace. To distinguish classes from different OOP namespaces, the *EiffelClassGenerator* prefixes all class names by their logical package names (following idiomatic "Eiffel case" conventions [13]), such that the OOP class *SavingsAccount* becomes *FINANCE_SAVINGS_ACCOUNT*.

List. 3. Translation of the OOP fragment corresponding to a *SavingsAccount* class into the Eiffel programming language. This illustrates in particular the special treatment for package scoping and access methods, which are not natural idioms in Eiffel.

```
note
  description: "SavingsAccount"
  author: "ReMoDeL"
  date: "Wed Dec 08 18:33:18 BST 2010"

class
  FINANCE_SAVINGS_ACCOUNT

inherit
  FINANCE_ACCOUNT
    redefine
      deposit,
      withdraw
    end

create
  make_with

feature {ANY} -- Public members

  make_with (holder: PEOPLE_PERSON; amount: INTEGER) is
    do
      initialise
      open_with(holder, amount)
    end

  deposit (amount: INTEGER) is
    require else
      account_currently_active: status = status_active
      positive_deposit_amount: amount > 0
    do
       balance := balance + amount
    end

  withdraw (amount: INTEGER): INTEGER is
    require else
      positive_withdrawal_amount: amount > 0
      account_currently_active: status = status_active
    local
      retry_attempts: INTEGER  -- Initial value 0
    do
      balance := balance – amount
      Result := amount
    rescue
      balance := balance + amount
      if retry_attempts < 1 then
        retry_attempts := retry_attempts + 1
        retry
      end
```

```
    end

invariant
  balance_in_credit: balance >= 0

end
```

Eiffel requires all redefined methods to be declared in the class header. This is accomplished by the *EiffelClassGenerator*, which detects the *override* attribute in the OOP method models. Initially, we had hoped to detect method overriding without needing a model attribute; however, this would potentially have required a global analysis of other OOP packages outside the current compilation unit [14].

Eiffel's idiom for enumerated types is to create, within a class, a set of uniquely named integer constants. An *EiffelClassGenerator* will have translated the OOP *Status* type into the Eiffel class *FINANCE_STATUS*, which enumerates the constants: *status_active, status_closed* and *status_frozen*. This class is inherited by the parent of the current class, *FINANCE_ACCOUNT*, and so transitively by the current class.

An *EiffelMemberGenerator* handles each of the OOP models for field, creator or method members handed down by the *EiffelClassGenerator*. The members are sorted into batches of different visibility levels. Public visibility is encoded by exporting the features to *ANY*, private visibility by exporting to *NONE* and protected visibility by exporting the feature to its owning class, whence it is also visible in subclasses.

Creation procedures in Eiffel are named after the OOP creator names (c.f. Java, which uses the creator types). All creation procedures invoke *initialise* first, which performs default field initialisation (in OOP, *Fields* may be declared with default initial values). This version of *initialise* was defined in the deferred (abstract) parent class *FINANCE_ACCOUNT*, to set the inherited *status* field to *status_closed*, but is first invoked in the effective (concrete) subclass.

The translation of OOP *Fields* and *Methods* poses a more sophisticated problem in Eiffel. Whereas the coding idiom in many target languages is to prevent external access to fields and control public access via "get"-methods, the natural idiom in Eiffel is to provide read-only attributes. Eiffel has no need for separate access methods. The *EiffelMemberGenerator* therefore has to block code generation for all OOP methods commencing with "get". Instead, when generating an attribute, it must determine, via its parent *EiffelClassGenerator*, whether any related public access method was supplied, in which case the attribute's export status is changed to *ANY* (so long as the class is also exported from its package – see section 2.5 above).

Later, *EiffelCodeGenerators* must take this into account when invoking OOP access methods. The "get"-method invocation is replaced by a direct access to the read-only public attribute. This may be further complicated by rules governing valid preconditions in Eiffel, which insists that all tested expressions in preconditions use exported features of the class. For this reason, the export status of a read-only attribute may change, to allow it to be tested in a precondition.

4.4 Translation from OOP into C++

In the C++ translation, occurrences of each OOP class identifier are rendered either as a C++ class identifier (appearing after *new* in object creation expressions), or as an instantiation of the smart pointer *Ref*<*T*> (for all typed variable declarations). Apart from this, the other main difference from Java is in the generation of a separate C++ header file and C++ source code file for each implemented class type (enumerated and interface types only need header files). However, there are further sophisticated aspects of the translation that generate fully idiomatic C++ code styles.

List. 4. Translation of the OOP fragment corresponding to a *SavingsAccount* class into a C++ header file: *SavingsAccount.h*. This illustrates the intelligent processing of type dependencies, the use of namespaces and the generation of smart pointers.

```
/**
 * SavingsAccount : generated on Fri Dec 10 14:12:15 GMT 2010
 * by ReMoDeL.
 */

#ifndef SAVINGS_ACCOUNT_H
#define SAVINGS_ACCOUNT_H

#include "Account.h"

namespace Finance {

    class Person;

    class SavingsAccount : public Account {
    public:
        SavingsAccount (Ref<Person> holder, int amount);
        virtual ~SavingsAccount ();
    protected:
        virtual void assertInvariant ();
    public:
        virtual void deposit (int amount);
        virtual int withdraw (int amount);
    };

} // namespace Finance

#endif
```

Listing 4 illustrates the translation of the OOP model into a C++ header file. To avoid multiple inclusions of the same header file, the *CplusHeaderGenerator* wraps the contents of the file in a C++ conditional compile instruction. The treatment of dependencies between the *SavingsAccount* class and other types is subtle: any inherited type (such as *Account* in listing 4) must be fully defined before the *SavingsAccount* type, so is imported using the *#include* directive. The same applies to any satisfied interface types, or used enumerated types (transitively, the superclass header file *Account.h* includes both *Asset.h* and *Status.h*). By contrast, any other

referenced class type need only be declared forward (such as *Person* in listing 4). This strategy not only reduces the number of file inclusions (increasing compile-time efficiency for the C++ compiler), but also is absolutely necessary where classes have circular usage dependencies. When generating *#include* directives, the generator must take into account the relative locations of all the included files with respect to the *current* file's location (c.f. in Java, where a relative path from the *package root* is assumed). In listing 4, the inherited *Account* class is in the same package directory, so is included using a short pathname. In listing 5, the referenced *Person* class is included via a longer pathname that must navigate up the directory tree, out of the *Finance* package, and down the tree into the *Person* package.

The OOP package identifier *Finance* is used to define a C++ *namespace*, within which all types belonging to the package are scoped. This later has implications on how types from outside the package are imported and on how types from this package will be used elsewhere. Listing 5 shows how the forward-declared *Person* type is eventually bound to the *Person* type obtained from the *People* package, via a *using* declaration inside the current package.

List. 5. Translation of the OOP fragment corresponding to a *SavingsAccount* class into a C++ source code file: Sav*ingsAccount.cc*. This illustrates the selective generation of class identifiers or smart pointers, the inclusion and qualification of a component from another package, the scoping of local methods and the special treatment of field names.

```
/**
 * SavingsAccount : generated on Fri Dec 10 14:12:15 GMT 2010
 * by ReMoDeL.
 */

#include "SavingsAccount.h"
#include "..\..\example\people\Person.h"

namespace Finance {

  using People::Person;

  SavingsAccount::SavingsAccount(Ref<Person> holder,
      int amount) {
    openWith(holder, amount);
  }

  SavingsAccount::~SavingsAccount() {
  }

  void SavingsAccount::assertInvariant() {
    Account::assertInvariant();
    if (my_balance < 0)
      throw Ref<BrokenContract>(
        new BrokenContract("balance in credit"));
  }

  void SavingsAccount::deposit(int amount) {
```

```
    if (my_status != ACTIVE)
      throw Ref<BrokenContract>(
        new BrokenContract("account currently active"));
    if (amount <= 0)
      throw Ref<BrokenContract>(
        new BrokenContract("positive deposit amount"));
    balance = balance + amount;
    assertInvariant();  // Check before exit.
  }

  int SavingsAccount::withdraw(int amount) {
    if (my_status != ACTIVE)
      throw Ref<BrokenContract>(
        new BrokenContract("account currently active"));
    if (amount <= 0)
      throw Ref<BrokenContract>(
        new BrokenContract("positive withdrawal amount"));
    int methodAttempts = 2;  // Including 1 retry attempt(s).
    while (methodAttempts > 0) {
      try {
        my_balance = my_balance - amount;
        assertInvariant();  // Check before returning.
        return amount;
      }
      catch (Ref<BrokenContract> broken) {
        my_balance = my_balance + amount;
        if (--methodAttempts == 0)
          throw broken;  // Fail.
      }
    }
  }
}  // namespace Finance
```

The *CplusHeaderGenerator* generates the class declaration, which in listing 4 includes the base class *Account* in the inheritance list. When an OOP class also satisfies further OOP interfaces, the generated C++ inheritance list includes further class names, denoting the abstract interface types. These must be prefixed by *virtual public*, to ensure that only one copy of each interface type is inherited, since it is technically possible to create fork-join paths among specialised interfaces, causing an abstract base interface to be inherited multiple times, via different paths.

The *CplusSignatureGenerator* creates the signatures for the declared members. Listing 4 illustrates how members are grouped according to their visibility. This is achieved through the state machine of the generator, which emits a new visibility specifier, whenever the visibility level is changed. This allows members to be generated in the same order as they appeared in the OOP model.

All methods in the C++ translation must be prefixed as *virtual*, since any method could later be overridden in a subclass (the decision to use package-based compilation units makes it impossible to perform a global analysis of dynamic, versus static binding [14]). A destructor is automatically added to the list of members to be generated for the class, as boilerplate. Technically, none of the generated classes manages further dynamic memory directly; this is performed by the smart pointers,

such as *Ref<Person>* in listing 4. However, in idiomatic C++ programming style, it is usual to see "default" automatic destructors declared explicitly.

Listing 5 demonstrates how a *CplusClassGenerator* generates the source code for the C++ implementation file. The coding style is very similar to the Java example in listing 2, apart from the special generation of smart pointers. This is seen especially in the throwing and catching of exceptions, where the thrown object must be a *Ref<BrokenContract>* for the catcher to recognise this type.

The *CplusMemberGenerator* prefixes each defined member with the name of the owning class, to scope the member appropriately. The processing involved can be more complex if the owning class is generic (yielding a *template class* declaration in C++), in which case type parameter names must first be obtained from the parent *CplusClassGenerator*. OOP field members are renamed in C++ by prefixing with the tag "*my_*", to distinguish these from method parameter names, which may have the same canonical OOP name.

The *CplusMemberGenerator* handles constructor definition in a particular way, to implement the default initial value declarations for OOP fields. A table of default initial values is created for the class, and each generator then emits the C++ initialisation syntax for these in every constructor, but permits explicit construction arguments to override these default values. Default values were provided for *my_holder*, *my_balance* and *my_status* in the default base constructor *Account()*, which is invoked implicitly by *SavingsAccount(Ref<Person>, int)* in listing 5, which overrides these with a new value for the holder and initial balance.

The *CplusCodeGenerator* functions in a similar way to the *JavaCodeGenerator* (parts of the algorithm are shared in the generator class hierarchy). One difference is seen in the way that super-methods are invoked (see *assertInvariant()* in listing 5). This requires access to the context two levels up in the related *CplusClassGenerator*, to discover the immediate base class of the current class. Some further optimisations of the generated C++ may yet be possible, for example the boilerplate destructor in listing 5 could be declared inline in listing 4. Inlining is a tricky issue in general, since moving methods to the header file might break the rules that generate efficient forward declarations.

4.5 Translation from OOP into C#

The principles of translation from OOP into C# are almost identical to those applied to the Java translation, except for trivial differences, where C# follows a syntax style closer to C++. The C# translation also supports namespaces, like the C++ translation. For reasons of space, the C# example is not elaborated further here.

5 Comparison with Other Approaches

The state of the art in Model-Driven Engineering is evolving rapidly, with much work being carried out by large software companies and tool vendors. For commercial reasons, few details of this work are publicly available, apart from strategic overviews [19]. A considerable number of position papers have appeared on how the OMG's

MDA proposal could be made to work, focusing either on aspects of pattern-driven transformation [20] or on the overall plan of work necessary to realise MDA [21].

Tool vendors seek to appear "MDA-compliant", where in practice this often means (no more than) the kind of skeleton code generation that CASE tools have achieved for over a decade. While some tools can capture complete information for full, idiomatic code generation, this is often done via the filling of endless property boxes, or the drawing of UML *Sequence Diagrams*, which offers no higher abstraction than writing the source code. The best tools offer round-trip engineering, in which full source code is parsed and later may be visualised either as diagrams or text, with editing operations affecting a common model (such as *EiffelBench* in the 1990s, renamed *EiffelStudio* since 2001 [22]). Such features are excellent in a programmer's IDE, but do not, in our opinion, constitute Model-Driven Engineering.

5.1 Meta-CASE and Meta-Modelling Approaches

The earliest complete working example of fully general model-to-code transformation of which the authors are aware was achieved by the tool XMF Mosaic, currently the property of Ceteva [23]. This tool, originally developed in Java, but later recompiled from its own self-describing models, offered users the ability to create their own domain-specific languages (DSLs) in either graphical or textual format (or both). XMF Mosaic supported the creation of domain-specific editors, with which users could develop their own models, which compiled down to executable systems. XMF Mosaic is therefore perhaps best characterised as a meta-CASE tool.

The tool was based around a core abstract programming language XOCL, an extended version of OCL [4] that also included state-modifying statements. The process of creating a DSL was complex: it involved defining a syntax for the language, a graphical mapping to diagram elements in the style of UML (if desired), an underlying semantics for the models (in XOCL), and rules for how the models should be translated into executable code (in XOCL). While the major building blocks of the DSL could be constructed rapidly as instances of a MOF-style metamodel [3], considerable effort had to be invested by the designer in creating rules for well-formedness, and translation rules that described how model elements should be executed. As a consequence, while the customised DSL editors were of considerable value to their end-users, and raised the level of abstraction at which they developed software systems, the process of designing the DSL was best left to expert architects, who understood the full capabilities of the XMF Mosaic tool.

MDE tools will eventually have to be much simpler for end-users to understand, customise and use. It is our belief that the majority of information systems may be described by just a few kinds of standard, abstract model, representing different functional, structural and time views of the system. Different constraints naturally apply within each model (for example, the normalisation of data schemas from conceptual data models can be fully automated [16]), and also affect how these models should be "folded together" to produce the system. This kind of information could be captured as part of the translation architecture, and need not be exposed to end-users as something they should have to design or customise.

5.2 Model-to-Model Transformation Approaches

Influenced by the MDA agenda set by the OMG [1], much recent effort has been invested in the Query/Value/Transformation (QVT) approach [5]. At the time of writing, QVT is only concerned with model-to-model transformations. In this approach, model transformation rules are expressed as a pair of related graphs, mapping from the prior to the posterior model state. According to [20], the rules' antecedents may contain positive patterns that must match in the model graph and negative patterns that may not match. The rules' consequents may contain structures to be inserted, indicate structures to be deleted or modified, and highlight structures that should remain unchanged. Eventually, the QVT rule language itself should also be describable within the MOF metamodel [3].

One early concrete realisation of this approach is the *Eclipse*-hosted ATL project [9]. This uses a combination of declarative languages to achieve pattern-driven transformations. MOF-style class diagrams are expressed as metamodels in KM3, a textual syntax for declaring meta-types and their relationships. The KM3 (*Kernel Meta-Meta-Modelling*) language conforms to the *Eclipse ECore* framework, a Java implementation of *EMOF* (the essential MOF [3]), in the sense that metamodels may be implemented directly as instances of *ECore* types. Instances of KM3 meta-types, known as models, are expressed in XML according to the OMG XMI schema [24]. A third language, ATL (*ATLAS Transformation Language*), is used to describe the transformations from one model to the other. Rule expressions in ATL refer to source and target elements, whose types are given in the respective KM3 metamodels for the source and target. Rules may require further "helper" functions, which are written in OCL [4]. Rules map each source element to the appropriate target element, by matching source model elements to the rule guard patterns and firing the rules to build the target model. This achieves the declarative QVT-style model transformation, but at the cost of introducing multiple novel languages. Nonetheless, the ATL website [9] has an impressive selection of case studies. ATL supports only model-to-model transformations; and a separate translation procedure using *MOFScript* is required to convert target models to native text.

A more direct approach is taken by the Triskell group in the design of their metamodelling language, *Kermeta* [10]. This has the flavour of a high-level object-oriented language, enriched with OCL-style constraints and set comprehensions. *Kermeta* is a single, uniform language that is used to represent both models (as object-oriented structures) and model transformations (as imperative algorithms). Built on top of the *Eclipse* framework, *Kermeta* is also designed to conform to the *EMOF* metamodel [3] realised in the *Eclipse ECore* implementation. It styles itself as an "extension to MOF", adding executable behaviour to the MOF static structures. Since model transformations are encoded in an imperative style, *Kermeta* is closer to XOCL than to ATL's pattern-driven transformations. Though it can be criticised as "just another programming language", its tight integration with *Eclipse* offers the possibility of model creation and manipulation through Eclipse's visual editors. Like XMF Mosaic, abstraction only comes through the libraries developed by experts.

A formal approach to specifying declarative QVT-style transformations is currently being developed at Kings College, London using a tool that acts on UML-RSDS [11], a controlled subset of UML specialised for reactive systems design,

whose semantics can be described using Real time Action Logic (RAL). UML-RSDS supports UML *Class Diagrams*, *State Machine Diagrams* and OCL. The approach taken to model transformations specifies preconditions and postconditions on the source and target models, and constraints on the translation rules, which are naturally expressed in OCL as dependent quantifications ($\forall s \in$ source, $\exists t \in$ target). This is perhaps the most abstract approach, which is both mathematically elegant and also offers the prospect of proving the correctness of transformations.

What emerges from all of these QVT-inspired approaches is how much machinery is required to define even a simple declarative transformation. Considerable effort is invested in defining *EMOF*-conforming models and rule-based languages for pattern-driven transformations. The eventual vision of deriving QVT rules themselves from first principles out of core MOF concepts (a rule can be seen as a kind of map) only adds to this machinery. We believe that this route to MDE, while worthy, will be slow and hampered by the weight of its own infrastructure.

5.3 The ReMoDeL Layered Transformation Approach

In the work reported here, we exposed a lightweight approach to MDE that uses well-known, available technologies, such as Java and XML. Rather than invest effort in defining further novel transformation languages, we wanted to show that interesting and subtle transformations could be performed using a simple object-oriented framework for model transformation and code generation, acting on flexible models expressed directly in XML. Our long-term interest is in discovering what kinds of model representations and what kinds of transformations might best support the complete MDE cycle, from business analysis to executable systems.

In contrast with the other approaches reported above, we started with the final model-to-code transformation step. This was partly by way of ensuring that we would always have an "existence proof" of working generated systems, which executed with a known operational semantics; and partly in order to bootstrap the content of the lowest-level model, *ReMoDeL* OOP, needed to generate idiomatic, object-oriented software. The code generation step described in this paper is still only one small step, translating from an abstract model of object-oriented programming into executable code in four contrasting languages (Java, C++, Eiffel and C#). But, as earlier sections have shown, there are many subtle issues involved in generating software for different platforms and languages that executes in an identical way.

The experience of crafting the generation algorithms was also informative. In general, idiomatic code generation is not a simple mapping from source to target models. While some of the generation rules were pattern-driven transformations (e.g. transforming the invariant into a method; or inverting the logic of assertions in preparation for exception guards), others were complex, state-based transformations that required a re-structuring of the logical content (e.g. determining when and how to invoke the class invariant method in Java; or fusing access methods and attributes in Eiffel). While some transformations required access just to local information, others required access to contextual information further afield (e.g. examining preconditions and access methods before deciding the export status of Eiffel attributes; or looking up the correct class scope for a C++ super-method invocation).

We believe that, for MDE to deliver on the promise of designing systems at an abstract level, model transformation tools must be smart enough to synthesise the missing details, rather than require designers to supply these via customisation interfaces. This is much more than simply inserting boilerplate designs or code for particular target architectures. We expect there to be standard solutions that result from "folding together" abstract models of the processing, data and time views of the system. Working out how to weave these models together is still an open research question. It depends critically on the particular representations chosen for each model, and the kinds of constraint that these expose. Eventually, we envisage a layered system of gradual transformations, from model-to-model and model-to-code, which exploits different constraints at each transformation step, analogous to the representational transformations used in computer vision [25].

Acknowledgments. The ReMoDeL team for 2009-10 included: Anthony Simons (project director, Java and C++ leader), Ahmad Subahi (C# team leader) and Stephen Eyre (Eiffel team leader), with Manal Alghamdi, Liam Farrell, Andrew Whitehead, Joseph Gooding and Swathi Nadella.

References

1. Object Management Group: MDA Guide, Version 1.0.1, Miller, J., Mukerji, J. (eds.), 12 June (2003). http://www.omg.org/cgi-bin/doc?omg/03-06-01
2. Object Management Group: Unified Modeling Language (OMG UML) Superstructure, Version 2.3, 5 May (2010). http://www.omg.org/spec/UML/2.3/Superstructure/PDF/
3. Object Management Group: Meta Object Facility (MOF) Core Specification, Version 2.0, 1 January (2001). http://www.omg.org/spec/MOF/2.0/PDF/
4. Object Management Group: Object Constraint Language, Version 2.0, 1 May (2006). http://www.omg.org/spec/OCL/PDF/
5. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0, 3 April (2008). http://www.omg.org/spec/QVT/1.0/PDF/
6. ReMoDeL: Reusable Model Design Languages, http://www.dcs.shef.ac.uk/~ajhs/remodel/
7. Simons, A.J.H.: ReMoDeL Object-Oriented Programming Model, Version 10 March 2010. Technical Report, Department of Computer Science, University of Sheffield (2010). http://www.dcs.shef.ac.uk/~ajhs/remodel/OOP/
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Design. Addison Wesley, Reading MA (1995)
9. ATL: A Model Transformation Technology, http://www.eclipse.org/atl/
10. Kermeta: Triskell Metamodelling Kernel, http://www.kermeta.org/
11. Lano, K.: A Compositional Semantics of UML-RSDS. Software and Systems Modeling 8, 85—116 (2009)
12. ECMA International: Standard ECMA-335, Common Language Infrastructure (CLI). ISO/IEC 23271:2006, June (2006)
13. Meyer, B.: Object-Oriented Software Construction, 2nd ed., Prentice Hall, Redwood CA (1997).
14. Simons, A.J.H., Ng, Y.M.: An Optimising Delivery System for Object-Oriented Software. Object-Oriented Systems 1 (1), 21—44 (1994)

15. 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)
16. Subahi, A.F.: ReMoDeL Database Generators. MSc Dissertation, Department of Computer Science, University of Sheffield (2010)
17. W3Schools: XML Attributes, http://www.w3schools.com/XML/xml_attributes.asp
18. Badros, G. J.: JavaML: a Markup Language for Java Source Code. In: 9th. Int. World-Wide Web Conf., Elsevier Science, May (2000). http://www9.org/w9cdrom/
19. Brown, A.W., Iyengar, S., Johnston, S.: A Rational Approach to Model-Driven Development. IBM Systems J., 45 (3), 463—480 (2006)
20. Tratt, L.: Model Transformation and Tool Integration. Software and Systems Modeling 4, 112—122 (2005)
21. France, R., Rumpe, B.: Model-driven Development of Complex Software: a Research Roadmap. In: 29th IEEE Int. Conf. on Softw. Eng. Workshop on Future of Software Engineering, pp. 37—54. IEEE Press, New York (2007)
22. Interactive Software Engineering: ISE to Release EiffelStudio 5.1. Press Release, 2 December (2001). http://www.eiffel.com/general/news/pdf/12-02-2001.pdf
23. Clark, A., Sammut, P., Willans, J.: Applied Metamodelling – a Foundation for Language-Driven Development, 2nd ed., Ceteva (2008)
24. Object Management Group: MOF 2.0/XMI Mapping, Version 2.1.1, 1 December (2007). http://www.omg.org/spec/XMI/2.1.1/PDF/
25. Marr, D.: Vision: A Computational Investigation into the Human Representation and Processing of Visual Information. Freeman, New York (1982)