



Formality, Evolution, and Model-driven Software Engineering

Jim Davies^a Charles Crichton^a Edward Crichton^a
David Neilson^b Ib Holm Sørensen^b

^a *Oxford University Computing Laboratory, Parks Road, Oxford, OX1 3QD, UK*

^b *B-Core (UK) Limited, Kings Piece, Harwell, Oxfordshire OX11 0PA, UK*

Abstract

This paper introduces an approach to software development in which a series of working implementations are generated automatically from a series of formal specifications. The implementations are data stores, communicating through standard protocols. The specifications are precise object models, in which operations are described in terms of pre- and post-conditions. The approach is evolutionary, in the sense that the specification may evolve while the system is in use, in response to changes in requirements, and any changes to the specification are automatically reflected in the structure of the implementation, and in the representation of any data currently stored.

Keywords: model-driven, object modelling, formal methods

1 Introduction

The term *model driven architecture* (MDA) will be familiar to most software developers. It is often described as ‘the future of software engineering’. A popular, and sufficient, characterisation of the term is this:

“MDA is about using modelling languages as programming languages rather than merely as design languages” [1]

To make this characterisation a little more precise: MDA is about programming in languages that are, at present, used only for the purposes of specification and design.

The word *architecture* reflects the origins of the term. It was first applied in the context of embedded systems, where requirements are precisely deter-

mined, the design of components is highly regular, and there is a significant degree of re-use. It is not unusual for several different embedded systems of the same type to be built from the same model, simply by changing parameters, or by modifying a state transition diagram.

The software that controls the brakes in most modern cars is a good example of MDA in its original context: the equations of a mathematical model determine the values used to label the transitions on a state diagram, which is then automatically translated into C, and compiled using a set of standard libraries. The same basic model may be re-used for many different kinds of car, simply by changing the equations, or the diagram.

As the requirements upon other types of system become better understood, and as their designs become more regular—through increasing standardisation of languages, protocols, and interfaces—the model-driven approach is being extended to other application domains: most notably, web services and enterprise computing systems.

The paper begins with a brief exploration of a slightly more general approach—*model-driven software engineering*—in which abstract models are used to generate many different kinds of artifact: programs, test suites, prototypes, documentation, interfaces, and other abstract models. In Section 2, we show how models may be classified according to purpose, and discuss the potential for automatic model transformation.

If they are to be treated as programs and compiled, or used as the basis for automatic generation of any kind, then the abstract models must have a precise, *formal* semantics. Existing examples of model-driven development use precise sub-languages of the Unified Modeling Language (UML) [2], or related notations such as Harel’s StateCharts [3]. In Section 3, we introduce a new, object-based notation, drawing inspiration from formal methods such as Z [8], B [6], and the Refinement Calculus [4].

In Section 6, we will show how this language supports an iterative style of development in which models are repeatedly updated to incorporate additional functionality, or to take account of changes in requirements. With each update, a new version of the system is generated, and any data from the previous version is automatically transferred: as a result, the system appears to continually *evolve* as it is being used. The paper ends with a brief discussion of the implications.

2 Model-driven software engineering

Software engineering, as defined by Bauer [5], is

“the establishment and use of sound engineering principles in order to

obtain, economically, software that is reliable and works efficiently on real machines”

These principles include the description of intended functionality, without which the essential engineering activities of evaluation and validation would be impossible. Such a description would normally involve the creation of one or more models of the system.

We may classify these models according to purpose:

sketch models These contain information about design features and system requirements. They should be seen as informal models—even though the languages used may admit a formal semantics—because they are not designed to be taken literally. For example, a state diagram in a sketch model might include transitions which, after further consideration, would be redirected or removed.

design models These differ from **sketch** models in that they are intended as a faithful record of the design, to be referenced in development, testing, and subsequent maintenance. They need not explain how the features described are to be implemented; neither do they need to explain every feature or behaviour of the system.

analysis models These are faithful representations of the system, created for the purposes of simulation, verification and validation. They will typically contain far less information than a **design** model, and the orientation of this information may be quite different. For example, a sequence of interactions may be represented as a single event, or a complex data structure may be represented as a single integer.

testing models These are similar to **analysis** models in terms of information content and orientation. Here, however, the intention is to guide the construction of tests for the implementation. For example, the model might explain exactly how the system should respond to a particular sequence of messages.

program models These are structured presentations of the implementation; they contain all of the information necessary for the automatic generation of the final working system. These are the models presented in programming tools, or interactive development environments (which often describe the return journey between structured representation and executable as ‘round-trip software engineering’).

The diagram of Figure 1 shows how these five classes of models are related in terms of construction: the arrows indicate that models of the source class can be used as a basis for generating models of the target class.

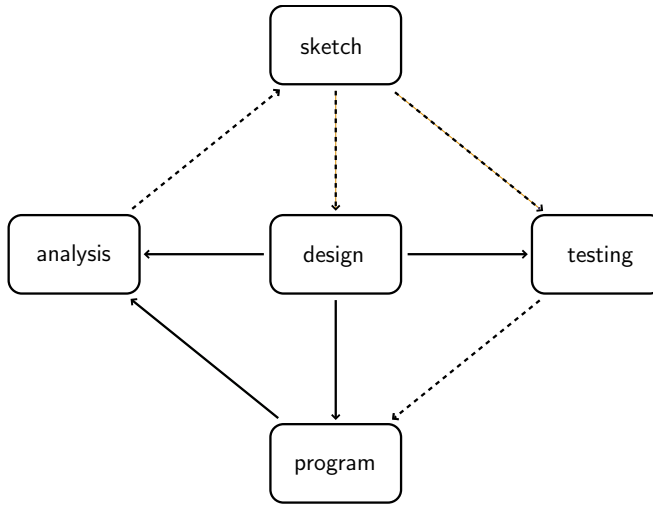


Fig. 1. Model-driven/model-based software engineering

It is worth observing that the **analysis** and **testing** classes differ only in terms of application:

- an **analysis** model is constructed on the basis of the **design** or **program** models; the results of the analysis may inform further **sketch** models, and lead to changes in the design.
- a **testing** model is constructed on the basis of the **design** and **sketch** models; the results of the tests may result in changes to **program** models.

These two classes are not disjoint: the same model *could* be used for analysis and testing. However, the difference in purpose makes this is unlikely: a testing model is used to check that a program meets the design; an analysis model is used to check a design against requirements.

In the diagram, we distinguish between situations in which the construction of one model is *based* on information contained in another, and those in which construction is *driven*. In the former, significant creative input may be required. In the latter, the course of construction is largely determined, and automatic generation should be possible.

We will use the term *model-driven software engineering* to describe any software engineering activity that is largely determined by the information content of a model. In the special case of *model driven architecture*, the activity is the construction of the program model, or the production of the system itself.

```

SYSTEM NEWSYS

SETS                                DEFINITIONS
SET = ... ;                        def == ... ;

ENUMERATIONS                       UTILITIES
CLASS ENUM ... END;               util = ... ;

IS                                END

CLASS Class1[dim1]
...
END;
...
CLASS ClassN[dimN]
...
END

```

Fig. 2. An outline of a **Booster** model

3 Programming with specifications

Model-driven software engineering requires assumptions about the context in which the development takes place: the construction of each new model will involve a large number of choices; these assumptions will allow us to make these choices automatically. The particular approach that we present here makes the following assumptions:

- the system under development behaves as an object-based data store;
- all external communication will be through standard protocols;
- there is no requirement for concurrent access.

The first of these is reflected in the syntax of the modelling language, which is itself object-based. The second is a practical concern: that the model need not concern itself with the design of external protocols. The third is quite fundamental: the modelling language cannot be used to describe the effects of concurrent access to data.

The modelling language employed, tentatively named **Booster**, has many of the features of an object-oriented programming notation: *classes*, *associations*, *attributes*, *methods*, *primitives*, and *enumerations*. The way in which these features are described, however, is more reminiscent of the formal notations of *Z* and the Refinement Calculus [4].

Figure 2 shows the structure of a specification written in the language. A model consists of a single **SYSTEM**, described as a collection of classes, in the context created by defining sets, enumerated types, type synonyms, and external utilities. A strict limit is placed upon the number of objects in each class, although this limit is easily increased as the system evolves.

3.1 Attributes

The language has two built-in types: **STRING** and **NAT**; other primitive types may be introduced directly, as sets, or indirectly, via enumeration classes or type synonyms. An attribute of primitive type is declared simply by writing **pAtt** : **PTYPE**. Whenever **PTYPE** is **STRING**, an additional argument is used to give the expected length.

Association information is described by means of reference- or set-valued attributes. An additional argument gives the maximum size of the set, which may be ordered (**OSET**) or ordered with possible repetition (**SEQ**). If the association is bidirectional, then the name of the opposing attribute is also given. For example, we write **sAtt** : **SET(Other.oAtt)** [**num**] to introduce an attribute **sAtt** denoting a set of references to objects of class **Other**. The expected cardinality of this set is **num**, and **sAtt** contains a reference to an object **other** if and only if **other.oAtt** contains a reference to the current object.

Optional attributes are a special case of set-valued attributes, corresponding to an association with multiplicity 0..1. *Derived* attributes are introduced with an equality symbol =, and their value is entirely determined by other values reachable from the current object.

3.2 Methods

Methods may be associated with classes, and also with set-valued attributes. A *primitive method* is defined as a triple:

Method(**pre** | **change_list** | **post**)

This should be read as follows: “given precondition **pre**, achieve postcondition **post**, changing only the values of classes, objects, and attributes in the **change_list**”.

Composite methods may be constructed as logical combinations of other methods, including any *local methods* declared for the current class or attribute. The method combinators in **Booster** correspond closely to schema operators of the Z notation: **ALSO**, to sequential composition; **ALL**, to universal quantification; and **ANY**, to existential quantification. A fourth combinator, **ORELSE**, corresponds to disjunction for operations with disjoint preconditions: it has the same effect as the first operation (reading left-to-right) whose precondition is satisfied.

Figure 3 shows the structure of a class, with two class-level methods **Method1** and **Method2**, and five attributes. **att1** is a primitive; **att2** and **att3** are reference-valued. **att2** is a mandatory reference to an object of class **OtherA**, whose attribute **oAttA** refers back to this object; **att3** is an optional

```

CLASS Name
METHODS
  Method1,
  Method2
ATTRIBUTES
  att1 : PTYPE1 ;
  att2 : OtherA.oAttA ;
  att3 : [ OtherB.oAttB ] ;
  att4 : SET(OtherC.oAttC) [num]
  METHODS
    Method3,
    Method4
  LOCAL_METHODS
    Method5,
    Method6 ;
  att5 = att2.oAttS ^ att2.oAttT
END

```

Fig. 3. A class in Booster

reference to an object of class `OtherB`.

`att4` is a set-valued attribute, containing references to objects of class `OtherC`. This attribute has two methods of its own, `Method3` and `Method4`; the definitions of these methods may include references to local methods `Method5` and `Method 6`. The expected size of `att4` is `num`. `att5` is a derived attribute, whose valued is obtained by concatenating the values of the two (string-valued) attributes `att2.oAttS` and `att2.oAttT`.

3.3 Pre- and post-conditions

A precondition may be any logical expression, but—to facilitate automatic generation—the syntax of postconditions is quite constrained. There are three primitive postconditions,

- `att = exp`: the value of attribute `att` should be equal to the value of the expression `exp`;
- `att : sAtt`: the attribute `att` should be an element of the set-valued attribute `sAtt`;
- `att /: sAtt`: the attribute `att` should *not* be an element of the set-valued attribute `sAtt`.

An expression `exp` may be any combination of reachable values—values of attributes in objects linked to the present object by navigable associations—and calls to external utilities.

These primitive conditions may be combined using `&` (\wedge) and `=>`; in an implication, the antecedent may be any logical expression. Restricted forms of universal and existential combination are also possible; however, their use is discouraged; the corresponding method combinators produce descriptions that are more elegant, and far easier to maintain.

The decoration `_each` is used to introduce a bound variable of object type; other decorations are used as follows:

- `_0`: the ‘before’ value of an attribute (this corresponds to the ‘ $_0$ ’ decoration in the Refinement Calculus);
- `_this`: the current, enclosing object of a particular class (this corresponds to the ‘ θ ’ operator of the Z notation);
- `_in`: a value input from the context of the method (this corresponds to the ‘?’ decoration in Z);
- `_out`: a value output to the context of the method (this corresponds to the ‘!’ decoration in Z);
- `_new` a newly-created object of a particular class.

With the exception of `_0`, these decorations may be applied to class names (to indicate a particular object) or to attribute names (to indicate a particular attribute). Where attribute names are unique within the scope of a method, any root qualification may be omitted: for example, `Class.att1.att2` may be written as `att2`, and `Class_this.att3` may be written as `att3`.

4 Model expansion

The first step of the automatic generation process is *model expansion*. The model is extended to include the definitions of any default methods that have been referred to, and preconditions are strengthened, where necessary, to ensure that links—association instances—are preserved. The expanded model, also in *Booster*, will typically be 4 or 5 times larger than the original.

4.1 Default methods and associations

Every class has two default methods:

- **Create**, to create a new object;
- **Destroy**, to destroy an existing object.

every optional attribute has two:

- **Set**, to set the attribute;
- **Clear**, to clear it;

and every set-valued attribute has four:

- **Add**, to add a reference to an existing object;
- **Remove**, to remove a reference to an existing object;

- **New**, to add a reference to a new object, creating the object;
- **Erase**, to remove a reference to an existing object, destroying it.

Each default method has a basic definition: for example, the **Create** method for class **C** would be defined as **Create(true | C | C_new : C)**. It has no precondition (**true**), it may change the set **C**—the set of all objects of that class—and it should ensure that the new object **C_new** is an element of **C**. Unless **C_new** is already in scope, the use of the decoration **_new** will result in a new object being created.

The default **Add** method on a set-valued attribute **S** of class **C**, containing references to objects of class **D**, would be defined as follows:

Add(C_this : C & S_in : D & S_in /: S | S | S_in : S)

The precondition insists that the current object is an object of class **C**; that input **S_in** refers to an object of class **D**; and that **S_in** is not already an element of **S**. The change list indicates that the method may change the value of **S**. The postcondition states that, afterwards, **S_in** should be in **S**.

The cardinality constraints given in the model are added to the preconditions of the default **Create** and **New** methods: for example, if class **C** were introduced with a dimension of **N**, then the **Create** method for that class would expand to

Create(C.card < n | C | C_new : C)

where **.card** is the built-in cardinality operator.

The pre- and post-conditions of **Destroy** and **Erase** are extended to take account of the association information in the model. If we are to destroy an object **O**, we must ensure that there is no associated object with an attribute **att** whose value includes a reference to **O**. If the associated object is **P**, of class **D**, then this can be achieved in three different ways:

- if **att** is set-valued or optional, the postcondition is extended to require that **att** is changed, if necessary, to remove the reference to **O**;
- if **att** is mandatory and **D** is not in the change list, the precondition is extended to require that **att** contains no reference to **O**;
- if **att** is mandatory and **D** is in the change list, the postcondition is extended to require that **P**, too, is destroyed.

The default semantics is given by **i** and **ii**. We may choose **iii** instead by adding **D** to the change list when referring to either method.

The default methods may be used by adding their names to the **CLASS** declaration: **Create** and **Destroy**, at the class level; **Set** and **Clear**, to an optional attribute; the others to set-valued attributes. When referring to a

default method, we may insert additional pre- or post-conditions, and add to the change list: for example, the declarations

```
attA : Other
sAttB : SET(Other)[5]
METHODS
  Add,
  Remove (sAttB\_this /= attA | skip)
```

introduce two attributes, `attA` and `sAttB`, together with a pair of methods on `attB`. The first method is the default `Add`; the second extends the default `Remove` with an additional precondition, requiring that the object to be removed is not the same as the object currently referred to in `attA`.

As a single predicate could denote either a pre- or a post-condition, a method definition with no change list must include both or neither. Even if just one is required—as in the example above—we must include the other to disambiguate the expression: `true` denotes an empty precondition; `skip`, an empty postcondition (with an empty change list).

Each method has a corresponding decoration. Within a class `C`, the methods `Create` and `Destroy` can use `C_new` and `C_this` to refer to the object that is to be created or, respectively, destroyed. For an optional attribute `att`, the methods `Set` and `Clear` can use `att_in` and `att_this` to refer to the object that is to be inserted or, respectively, removed.

For a set-valued attribute `sAtt`: the method `New` can use `sAtt_new` to refer to the object being created; `Add` can use `sAtt_in` to refer to the object being added; `Remove` and `Erase` can use `sAtt_this` to refer to the object being removed or deleted. Additional input and output objects can be introduced using the `_in` and `_out` decorations.

4.2 A library example

Figure 4 shows a fragment of a **booster** model of a library system. The class `Person` has a dimension value of 10: no more than 10 of these objects may exist in a system built from this model. The `Create` method for this class is an extension of the default `Create`: the addition of `personName` to the change list indicates that the value of this attribute should be set when a new person object is created.

`personName` is a primitive attribute: a person's name will be represented as a string, with an expected length of 30. The other attributes, `personLoans` and `personRequests` both denote sets of references to objects of class `Book`. In each case, the association is bidirectional: for each person `p`

- whenever a reference to a book `b` appears in the set `p.personLoans`, a

```

CLASS Person[10]
  METHODS
    Create(personName), Destroy,
  ATTRIBUTES
    personName : STRING[30];
    personLoans : SET(Book.bookLoanedTo)[3]
  METHODS
    Add(personLoans.card < 3 | skip), Remove ;
    personRequests : SET(Book.bookRequestedBy)[3]
  METHODS
    Add(personRequests_in /: personLoans | skip),
    Remove
END;

CLASS Book[50]
  METHODS
    Create(bookTitle,bookNumber), Destroy
  ATTRIBUTES
    bookTitle : STRING[20];
    bookNumber : NAT;
    bookLoanedTo : [Person.personLoans] ;
    bookRequestedBy : [Person.personRequests]
END;

```

Fig. 4. Library model (fragment)

reference to **p** will appear in the set **b.bookLoanedTo**;

- whenever a reference to a book **b** appears in the set **p.personRequests**, a reference to **p** will appear in the set **b.bookRequestedBy**.

The default **Add** method for **personLoans** has been extended with the precondition that no person may borrow more than 3 books at a time; that for **personRequests** has the additional constraint that no person may request a book that they already have.

Figure 5 shows part of the class **Person** in the expanded model. The addition of **personName** to the change list of the **Create** method has resulted in an additional precondition: that **personName_in** is a valid string. There is also an additional postcondition: that the value of the **personName** in the new object should be equal to the value of this input string.

The associations between **Person** and **Book** lead to additional postconditions on the **Destroy** method. The postcondition

```

forall(Book).(Book_each : Person_this.personLoans =>
  Person_this /: Book_each.bookLoanedTo) &

```

requires that if a book appears in the **personLoans** attribute, then the current person should be removed from the corresponding **bookLoanedTo** attribute. (Observe that the antecedents of logical implications in a postcondition are evaluated before the operation is performed.) The two additional postconditions remove any references to the current object at the same time that it is destroyed.

The **Add** method of **personLoans** includes several additional preconditions.

```

CLASS Person[10]
METHODS
  Create(Person.card < 10 &
    personName_in : STRING
    | Person , personName |
    Person_new : Person &
    Person_new.personName = personName_in),
  Destroy(Person_this : Person
    | Person , bookLoanedTo , bookRequestedBy |
    forall(Book).(Book_each : Person_this.personLoans =>
      Person_this /: Book_each.bookLoanedTo) &
    forall(Book).(Book_each : Person_this.personRequests =>
      Person_this /: Book_each.bookRequestedBy) &
    Person_this /: Person ),
ATTRIBUTE
  personName : STRING[30];
  personLoans : SET(Book.bookLoanedTo) [3]
METHODS
  Add(0 < Book.card &
    Person_this : Person &
    Person_this.personLoans.card < 3 &
    personLoans_in /: Person_this.personLoans &
    personLoans_in : Book &
    personLoans_in.bookLoanedTo.card = 0
    | bookLoanedTo , personLoans |
    Person_this : personLoans_in.bookLoanedTo &
    personLoans_in : Person_this.personLoans),
...

```

Fig. 5. Library model, expanded (fragment)

Three of these are sanity checks: that the references point to objects of the expected types, and that the input type is non-empty:

- `Person_this : Person`
- `personLoans_in : Book`
- `0 < Book.card`

One has been introduced explicitly:

- `Person_this.personLoans.card < 3`

Another illustrates our strict interpretation of `Add`: that it should be blocked if the object is already an element of the set.

- `personLoans_in /: Person_this.personLoans`

The last precondition is particularly interesting: it is a consequence of the precondition of the `Set` method on the opposing attribute:

- `personLoans_in.bookLoanedTo.card = 0`

An additional postcondition for this method insists that `bookLoanedTo` be updated with a reference to the current person.

5 Code generation

The expanded model serves as source code in a subsequent compilation process. The model is translated into abstract machine notation, and then translated into C source by means of a B rulebase, before being compiled using the standard B toolkit libraries.

5.1 Method implementations

The method implementations are quite straightforward: preconditions become boolean expressions to be evaluated; postconditions become program statements. In the *simplest case*,

- the primitive postcondition $\mathbf{a} = \mathbf{e}$ is implemented as a simple assignment $\mathbf{a} := \mathbf{e}$;
- the primitive postconditions $\mathbf{a} : \mathbf{s}$ and $\mathbf{a} / : \mathbf{s}$ are implemented using functions that add or remove elements from sets;
- logical implication is implemented using a conditional `if ... then` construction;
- logical conjunction is implemented using a sequential composition of statements: $\mathbf{c1} ; \mathbf{c2}$

Each operation is wrapped with code that—efficiently—restores the original state, should any part of the operation fail.

Contextual information is used to resolve ambiguity: for example, the postcondition in the following definition might appear to admit two alternative implementations:

```
Method( $\mathbf{a} < 10 \ \& \ \mathbf{b} > 3 \mid \mathbf{b} \mid \mathbf{a} = \mathbf{b}$ )
```

We might satisfy the requirement—that the ‘after’ values of \mathbf{a} and \mathbf{b} are the same—with one of two assignments: $\mathbf{a} := \mathbf{b}$ or $\mathbf{b} := \mathbf{a}$. However, \mathbf{a} does not appear in the change list, so the first of these is disallowed; the postcondition must be implemented as $\mathbf{b} := \mathbf{a}$.

Where ambiguity cannot be resolved, or where definitions appear to be inconsistent, the model will need to be modified. For example, if the above definition were replaced with

```
Method( $\mathbf{a} < 10 \ \& \ \mathbf{b} > 3 \mid \mathbf{a}, \mathbf{b} \mid \mathbf{a} = \mathbf{b}$ )
```

then the compilation process would halt, reporting that the presence of both \mathbf{a} and \mathbf{b} in the change list would appear inconsistent with the use of $\mathbf{a} = \mathbf{b}$ as a postcondition.

It would be possible for the compiler to choose a implementation—either of the two assignments—but it is better that it should signal an error: it seems

likely that at least one part of this definition contains a mistake. The author of the model can remedy the situation by removing one of the variables from the change list, or by decorating one of them with `_0`: the postcondition `a = b_0` states that the value of `a` after the operation should be equal to value of `b` before the operation was called; this condition would then be implemented using the assignment `a := b`.

5.2 Interfaces and documentation

The method implementations constitute a programming interface to the underlying data store. This interface is not accessed directly: instead, requests are queued for sequential processing by a decoder component. For a combination of pre- and post-conditions to serve as an adequate description of a method `m`, there must be no possibility of another method updating variables in the pre- or post-condition while `m` is still executing.

Requests and responses are presented in a structured language, using tags that correspond to the operations of the model. The decoder logic is generated at the same time as the programming interface and the data store. If the system has a user interface, then components of this interface may also be generated, together with user documentation.

Although the expanded model is quite readable, it is sometimes useful to augment an interface with a transliteration of preconditions: for example, the unavailability of an operation `Enrol`, perhaps corresponding to the ‘greying-out’ of a button on a web page, might be explained as

“The operation `Enrol` is unavailable: either the input `name` is not an element of `registeredStudents`, or the size of `enrolment` is not less than `classCapacity`”

Relatively little effort has been expended on this aspect of the `booster` technology: at present, such a message would read

```
"precondition:
  name : registeredStudents &
  enrolment.card < classCapacity"
```

Nevertheless, the calculation and presentation of such information can be extremely valuable in providing feedback for model development.

6 Model evolution

In model-based development, even if the initial program design corresponds exactly to the model, this correspondence will often weaken as development

proceeds. Additional insights gained during implementation or maintenance may lead to changes that are not properly incorporated in the model: once code has been produced, it becomes the focus of attention, and the model is neglected.

The model-driven approach eliminates this problem, but another remains: each change to the model produces a new version of the system, but what of the data that the previous version may have collected? The **booster** compiler has been designed to address exactly this problem.

In generating each new version of the system, the compiler will compare the new model with the model used to generate the previous version. If it can determine how the model has been transformed, it can apply a corresponding transformation to any data that has already been collected.

If the transformation is a minor one, then this *data upgrade* process will be completely automatic. Code is generated to create objects, and to set attribute values, in the new version of the system; the data from the previous version is stored in a form that matches the new model; the integrity of the data is checked against the association constraints; finally, the code is then executed to create the new system and install the data.

Minor transformations include:

- renaming classes and attributes;
- removing classes and attributes;
- adding new classes, sets, or enumerations;
- adding set-valued or optional attributes;
- adding primitive attributes.

In some cases, no data upgrade is necessary: the data from the previous version can be reloaded without modification. This will be the case when the only change to the model involves:

- adding, removing or modifying methods;
- adding, removing or modifying derived attributes.

Other transformations may make a direct, automatic upgrade impossible; there are then two strategies that may be employed:

- (i) express the transformation as a sequence of minor transformations, each of which admits an automatic upgrade;
- (ii) introduce a third, intermediate model and specify the required upgrade transformation as a method in that model.

In some cases, only the second strategy will succeed. For example, to introduce a mandatory reference-valued attribute **att** to a class **C**, in the absence of a

default object value, we would need to introduce an intermediate model in which the attribute was optional. This model could contain a method

```
Upgrade(ALL C THEN Set_att(...) END)
```

which has the effect of setting each instance of the attribute. After calling this method, an automatic upgrade to a model in which `att` is mandatory becomes possible.

7 Discussion

In this paper, we have described a model-driven approach to software engineering, based around the use of a new object-based language. This approach has developed through application: a number of large case studies, including two systems for commercial customers, have been produced, and are being maintained. The language is inspired by Z, and implemented using B, in a combination of formal techniques and code generation that has proved particularly effective.

The largest system produced thus far is an online database system with 1000 users: the model for the system is 3000 lines long, and remains perfectly readable; the implementation consists of 300000 lines of (generated) C program, the equivalent of perhaps 30000 lines of hand-written code. The requirements upon this system are still changing, and the current model is Version 88.03. The system has never failed; it has stopped, but only when the model said that it should do so.

There is considerable potential for further development: for example, a **booster** system has been created to manage developments, updating **booster** models as objects; with this approach, it may be possible to automate a wider range of data upgrades. The compiler may be retargeted to produce Java, or C#, and libraries added to build other classes of systems.

Although a great deal of work has been done on integrating formal methods—and, in particular, on combining Z with B, and B with UML [9]—we are not aware of any work in this area that has also addressed the question of how to generate method implementations from pre- and post-conditions; the emphasis instead is upon verification of proposed implementations.

An important point of reference—as should be clear from the introduction to this paper—is the work on the Model Driven Architecture[1], and the further development of the Object Constraint Language (OCL) [2]. Our use of **booster** presents it as a higher-level programming language, another step in the progression from machine code, to Cobol, to C, and then to J2EE and .NET. For a particular class of application, at least, we are ready to program

in formal methods.

References

- [1] D. S. Frankel, *Model Driven Architecture*. Wiley 2003.
- [2] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley 2003.
- [3] D. Harel and E. Gery. *Executable Object Modeling with Statecharts*. IEEE Computer 30. 1997.
- [4] C. C. Morgan. *Programming from Specifications*. Prentice Hall 1990. (available from <http://users.comlab.ox.ac.uk/carroll.morgan/PfS/>).
- [5] P. Naur and B. Randall. Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee, 1969.
- [6] S. A. Schneider. *The B Method*. Palgrave 2001.
- [7] C. Snook and M. Butler. *U2B - UML to B translation tool*. Available at <http://www.ecs.soton.ac.uk/cfs/U2Bdownloads/U2Bdownloads.htm>
- [8] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, second edition, 1992.
- [9] H. Treharne. Supplementing a UML Development Process with B. In the Proceedings of *FME 2002: Formal Methods – Getting IT Right*. Springer LNCS 2391. 2002.