Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 184 (2007) 171–187
www.elsevier.com/locate/entcs

From Predicates to Programs: The Semantics of a Method Language

David Faitelson, James Welch and Jim Davies

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD UK

Abstract

This paper explains how a declarative method language, based upon the formal notations of Z and B, can be used as a basis for automatic code generation. The language is used to describe the intended effect of operations, or methods, upon the components of an object model; each method is defined by a pair of predicates: a precondition, and a post-condition. Following the automatic incorporation of model invariants, including those arising from class associations, these predicates are extended—again, automatically—to address issues of consistency, definition, and dependency, before being translated into imperative programs. The result is a formal method for transforming object models into complete, working systems.

Keywords: formal methods, declarative programming, object modelling, weakest preconditions

1 Introduction

The practice of generating code from higher-level specifications, sometimes called automatic programming, dates from the beginning of the 1970s [10]. Although the development of arbitrary, combinatorial programs—such as algorithms for sorting and searching—remains too complex to be mechanised effectively [6], considerable progress can be made within specific application domains.

An early example was the Model II language [13] for stream processing, in which programs could be generated from a description of the intended relation between incoming and outgoing records. A more recent example is the Descartes language [9], which has been used to generate part of a control system from a formal description of shutdown requirements.

We may expect to see many more examples in the near future: the generation of code from precise descriptions in domain-specific languages is the essence of the 'software factories' approach being developed at Microsoft [7]; it is also the practical realisation of the 'model-driven architecture' being promoted by the UML/models community [8].

doi:10.1016/j.entcs.2007.03.021

The method language explained here, one aspect of an approach described in general at a previous SBMF workshop [5], is intended for the generation of object databases: data stores in which information is organised as a collection of objects, and acted upon by associated methods. Its syntax is based upon aspects of Z [16] and B [1] languages; its semantics is based upon that of Z [16] and the Refinement Calculus [11].

A formal basis for the first phase of the generation process, in which methods are *expanded* to take account of invariants and references, has been established [18]. The contribution of this paper is to give a formal semantics for the language of expanded methods, and thus explain precisely how the expanded predicates are transformed into program statements.

The paper begins with an explanation of the *booster* approach. We then introduce a language of primitive methods and combinators. Section 4 presents a formal semantics for this language, using a notion of weakest precondition similar to that proposed for Z [4], and a modified version of Dijkstra's language of guarded commands [6,12]. In Section 5, we explain how methods are analysed and transformed to take account of issues of consistency, dependency, and definedness within postconditions.

2 A formal, domain-specific modelling language

The *booster* language, first described in [5], is intended for the generation of software components whose design is:

- *transformational*—the intended effect of an operation can be described in terms of values of inputs, outputs, and attributes immediately before, and immediately after, the operation has been performed.
- *sequential*—at most one operation may be acting upon the data within the component at any one time; the current operation must finish reading or updating the data before the next can begin.

In applications of the language, components are described as object models, in which data is organised into classes, and acted upon by associated methods.

Each method is declared as a triple: a *precondition* that must be satisfied before the method can be called; a *change list* of attributes that may be updated; a *postcondition* that describes the intended effect. The first of these is a predicate upon attribute values and inputs. The last is a predicate upon inputs, outputs, and values before and after the operation.

For example, the declaration below introduces a method `M`, which is applicable only if the current value of `a` is less than `1`. The method may alter the value of `b`, and should ensure that `b` and `c` have the same (after) value.

```
M(a < 1 | b | b = c)
```

```
CLASS Conference
  ATTRIBUTES
    attendees : SET(Person . attending)
    presenters : SET(Person . presenting)
    schedule : Schedule . conference
    capacity : NAT
  METHODS
    AddPresenter(
      person_in : attendees | presenters | person_in : presenters)
    ReplacePresenter(
      existing_in : presenters & new_in : attendees | presenters |
          existing_in /: presenters & new_in : presenters)
    ScheduleTalk(
      true | schedule.AddTalk AND AddPresenter)
    AddExtraCapacity(
      extra_in > 0 | capacity | capacity = capacity_0 + extra_in)
```

Fig. 1. A class declaration in booster

The value of a data attribute may be a primitive—a number, or a string—or it may consist of one or more references to objects (a `SET` or an `OSET`). In the declaration of a reference-valued attribute, we identify a corresponding attribute in the target class: associations between classes are bidirectional.

Fig. 1 shows part of the declaration for the class `Conference`. Each object of this class contains attributes to represent the people registered as `attendees` or as `presenters`; in each case, the value of the attribute is a set of references to objects of the type `Person`. There is also a reference to a `Schedule` object, describing the schedule of talks and events, and a `capacity` figure: the maximum number of people who may be registered to `attend`.

The additional component of each attribute declaration—the name following the period ‘ . ’—identifies the corresponding attribute in the target class. For example, each object of class `Person` will have an attribute `attending`, whose value records the set of conferences that they plan to attend.

`AddPresenter` is applicable only if `person_in` is registered to attend, and it has the effect of adding them to the list of presenters. `ReplacePresenter` replaces an existing presenter with a new presenter, provided that the new presenter is attending the conference.

`ScheduleTalk` adds a talk to the schedule and ensures that a person is recorded as one of the presenters; it is defined in terms of a method of the `Schedule` class, referred to as `schedule.AddTalk`, together with the method `AddPresenter` from `Conference`.

The definition of `AddExtraCapacity` demonstrates that the postcondition may make reference to the values of an attribute before the operation takes place: `capacity_0` denotes the value of the `capacity` attribute beforehand, and `capacity` denotes the value afterwards.

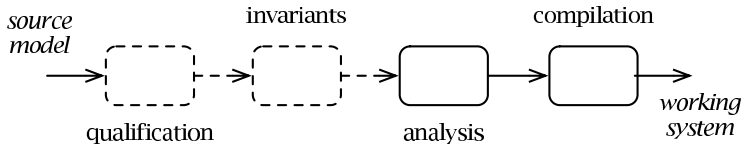

Fig. 2. The generation process

The text of Fig. 1, in combination with suitable declarations of `Person` and `Schedule`, is enough to define a complete working system. The process of model transformation and compilation—from declarative model to executable code—is shown in Fig. 2.

This process of code generation is divided into four stages: in the first, attribute and method names are fully qualified, to take account of the scope in which they are declared; in the second, method pre- and post-conditions are extended to ensure that model invariants are maintained. The formal basis for the second stage is the subject of an earlier paper [18].

In the third stage of the process, the method definitions are analysed to determine necessary conditions for:

- the postconditions of each method to be consistent, in terms of the requirements placed upon after values and outputs;
- there to be no mutual dependency between the after values of different, changed attributes;
- any expressions that appear in pre- and post-conditions to be well-defined.

These conditions are then added to the existing preconditions of the methods concerned. Finally, in the fourth stage, the fully-expanded definitions are compiled into executable code. The formal basis for these two stages—the semantics of the method language—is the subject of the current paper.

3 The method language

The precondition of a method may be any predicate upon a combination of inputs and before values, formed using a range of boolean operators. The change list is simply a list of attributes. However—and this is an essential aspect of the approach—the postcondition must be formed from the primitives

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

together with the trivial postcondition `skip`, using conjunction (`&`), implication (`=>`), and universal quantification (`forall`), as follows:

```
CLASS MethodsExample
  METHODS
    Method1(a /= 0  |  a, b  |  a = a_0 + 1  &  b = b_0 + a)
    Method2(pre2 | Method3 AND Method4(d_in = e))
  ATTRIBUTES
    a : NAT
    b : NAT
    c : SET(Other . example)
      METHODS
        Method5(Method6 THEN Method7)
        Method7(Method8 OR SKIP)
END
```

Fig. 3. Methods in Booster

- `post1 & post2`: both `post1` and `post2` should hold;
- `cond => post1`: if condition `cond` is true, then `post` should hold;
- `forall b_each : B . post1`: the postcondition `post1` should be true for every value of `b_each` drawn from set `B`.

In the above, `cond` may be any predicate, but `post1` and `post2` should be formed as (the conjunction of) primitive postconditions.

Mandatory decorations are used to identify inputs, dummy variables, and 'before values'.

- `_in`: an input from the context of the method;
- `_each`: a dummy variable used in a quantification;
- `_0`: the 'before value' of an attribute.

The first two may appear in any condition; `_0`, however, is seen only in postconditions: see, for example, method `Method1` of Fig. 3, which will have the effect of incrementing `a`, and increasing the value of `b` by the new value of `a`.

Some methods, such as `Method1` above, will be defined in extension, with explicit pre-conditions, post-conditions, and change lists. Others will be defined using *method combinators*: for methods `M1` and `M2`,

- `M1 AND M2` is a method that will be available whenever both `M1` and `M2` are available; it has the combined effect of both methods;
- `M1 OR M2` is available whenever either `M1` or `M2` is available; it has the effect of `M1` if that method is available; otherwise it has the effect of `M2`.
- `M1 THEN M2` has the effect of `M1` followed by that of `M2`; it is available whenever `M1` is available *and* the effect of `M1` would make `M2` available.
- `ALL C_each :  C WHERE P DO M` has the combined effect of all the methods that are instantiated from `M` for each binding of `C_each` that satisfies the predicate `P`. This combinator is a generalisation of the `AND` combinator.

We require that the two arguments to `AND` are independent, in the sense that variables appearing in the change list of one do not appear in the pre- or post-condition of the other.

Input variables may be obtained from the environment at the time of execution, or instantiated using any expression that has meaning within the current scope. In the example, `Method4` is instantiated: every occurrence of input `d_in` is replaced by the value of `e`.

A method expression may be 'preconditioned' with an additional constraint; this has the effect of adding that constraint to the overall precondition. In the example, `Method2` is defined by preconditioning a combination of `Method3` and (an instantiated version of) `Method4`.

The primitive method `SKIP` is always available, and its execution has no effect upon the state of the system. It may be used, in combination with `OR`, to make a composite method that is always available. `Method7` is always available, but has no effect outside the precondition of `Method8`, and (consequently) the availability of `Method5` is constrained only by that of `Method6`. As this example demonstrates, methods may be declared at the level of classes, but also at the level of individual attributes.

4 The semantics of methods

4.1 A language of partial programs

We give our language of methods a semantics using a language of partial programs, a variant of Dijkstra's language of guarded commands [6]. Our language is similar to that defined by Nelson [12], but with a different notion of weakest precondition.

The if ... fi and do ... od operators are no longer required; any program may be prefixed with an arbitrary guard; and any pair of programs may be used as arguments to the generalised choice operator □. The syntax of this language is given by:

$$
\begin{aligned}
\langle \mathit{command} \rangle ::= {} & \text{“}\mathit{skip}\text{”} \mid \langle \mathit{assignment} \rangle \mid \langle \mathit{guard} \rangle \text{ “}\rightarrow\text{” } \langle \mathit{command} \rangle \mid \\
& \langle \mathit{command} \rangle \text{ “}\Box\text{” } \langle \mathit{command} \rangle \mid \\
& \langle \mathit{command} \rangle \text{ “ ; ” } \langle \mathit{command} \rangle \mid \\
& \text{“}\mathsf{input}\text{” } \langle \mathit{variable\ list} \rangle \text{ “}\bullet\text{” } \langle \mathit{command} \rangle \mid \\
& \text{“}\mathsf{var}\text{” } \langle \mathit{variable\ list} \rangle \text{ “}\bullet\text{” } \langle \mathit{command} \rangle \mid \\
& \text{“}\mathsf{all}\text{” } \langle \mathit{variable} \rangle \text{ “:” } \langle \mathit{variable} \rangle \text{ “}\bullet\text{” } \langle \mathit{command} \rangle
\end{aligned}
$$

Sequential composition has the conventional interpretation, as does the declaration of local variables: the special declarations input and output indicate that values are to be obtained from, and passed to, the environment of the method (normally the transaction manager, but inputs may also be supplied through instantiation).

The weakest precondition semantics for this language is defined in part by the following clauses:

$$wp(skip, p) \;=\; p$$

$$wp(a := e, p) \;=\; p[a := e]$$

$$wp(g \rightarrow c, p) \;=\; g \wedge wp(c, p)$$

$$wp(c1 \;\square\; c2, p) \;=\; wp(c1, p) \vee wp(c2, p)$$

$$wp(c1;\; c2, p) \;=\; wp(c1, wp(c2, p))$$

$$wp(\mathsf{dec}\ dec \bullet c, p) \;=\; \forall\ dec \bullet wp(c, p)$$

where dec may be either var or input, and $p[a := e]$ denotes the predicate produced by substituting e for a within p.

This notion of *wp* reflects our assumption that a method will be unavailable when the guard is false: any attempt at execution will be blocked. The weakest precondition for $g \rightarrow c$ to achieve p is thus the *conjunction* of the guard with the weakest precondition for the remainder of the command to achieve p:

$$wp(g \rightarrow c, p) \;=\; g \wedge wp(c, p)$$

This is quite different from the notion adopted by Nelson [12], in which:

$$wp(g \rightarrow c, p) \;=\; \neg\, g \vee wp(c, p)$$

There is a corresponding difference in the semantics of the choice operator: with no requirement for angelic nondeterminism; the constraints of any subsequent guards will be included in the *wp* semantics of each alternative.

4.2 From predicates to programs

To obtain the program semantics of a method, we transform the postcondition to a command, and the precondition to an additional guard. The first transformation is complicated by the need to consider the guards of both components in a disjunction (`OR`); the second by the need to consider the program semantics of the first component in a sequential composition (`THEN`).

The postcondition of a method written in extension corresponds to a series of (possibly conditional) assignments. In the following, we write *assign* ⟦ p ⟧ to denote the series of assignments corresponding to postcondition p:

$$assign\,[\![\,\texttt{x = e}\,]\!] \;=\; x := e$$

$$assign\,[\![\,\texttt{skip}\,]\!] \;=\; skip$$

$$assign\,[\![\,\texttt{e : s}\,]\!] \;=\; s := s \cup \{e\}$$

$$assign\,[\![\,\texttt{e /: s}\,]\!] \;=\; s := s \setminus \{e\}$$

$$assign\,[\![\,\texttt{c => p}\,]\!] \;=\; c \rightarrow assign\,[\![\,p\,]\!] \;\Box\; \neg\, c \rightarrow skip$$

$$assign\,[\![\,\texttt{p \& q}\,]\!] \;=\; assign\,[\![\,\texttt{p}\,]\!]\,;\, assign\,[\![\,\texttt{q}\,]\!]$$

$$assign\,[\![\,\texttt{forall c\_each : C . p}\,]\!] \;=\; \mathsf{all}\ c_{each} : C \bullet assign\,[\![\,\texttt{p}\,]\!]$$

4.3 Universal quantification and iteration

In our original version of the semantics, presented at the SBMF conference, we placed a constraint upon the use of the universal quantifier, requiring—in effect—that any quantified expression should correspond to a sequence of assignments to different attributes. This meant that we could not, for example, implement the following postcondition

```
forall a_each : A . p => a_each /: B
```

which expresses the requirement that any element of A that satisfies predicate p should be removed from B, as this postcondition corresponded to a sequence of assignments to the same attribute B.

However, by considering the quantified predicate as an equality upon the corresponding set expressions, we can allow quantification over such postconditions. We first observe that if the body of the quantification is a conjunction, then the quantified expression may be considered as a conjunction:

```
forall i : I . conj1 & conj2
    ≡ (forall i : I . conj1) & (forall i : I . conj2)
```

Thus we need only consider quantifications in which the body is a possibly-guarded primitive postcondition.

If the body is a guarded set-membership predicate (or its negation), then it is equivalent to the statement that the characteristic set is a subset of (or does

not intersect with) the set-valued attribute in question; in set-theoretic terms, we observe that

$$\forall\, i : I \bullet P \Rightarrow e \in s \Leftrightarrow \{i : I \mid P \bullet e\} \subseteq s$$

$$\forall\, i : I \bullet P \Rightarrow e \notin s \Leftrightarrow \{i : I \mid P \bullet e\} \cap s = \emptyset$$

where indexing variable i may appear free in expression e, but not in set-valued expression s.

Thus these conditions can be safely implemented as simple assignments in our language of guarded commands:

$$\mathit{assign}\,[\![\,\texttt{forall i : I . P => e : s}\,]\!] =$$

$$\quad s := s \cup \{i : I \mid P \bullet e\}$$

$$\mathit{assign}\,[\![\,\texttt{forall i : I . P => e /: s}\,]\!] =$$

$$\quad s := s \setminus \{i : I \mid P \bullet e\}$$

The set expressions on the right-hand side of each assignment can in turn be implemented as iterations over the set in question. In either case, the weakest precondition is easily calculated, as a simple substitution.

4.4 The semantics of primitive methods

We write *execute* $[\![\,\texttt{M}\,]\!]$ to denote the guarded command corresponding to the post-condition of a method `M`. If `M` is written in extension, then this may be obtained by prefixing the corresponding sequence of assignments with a local variable declaration:

$$\mathit{execute}\,[\![\,\texttt{(pre | change | post)}\,]\!] \;=$$

$$\quad \texttt{var}\ \mathit{change}_0 \bullet \mathit{change}_0 := \mathit{change}\ ;\ \mathit{assign}\,[\![\,\texttt{post}\,]\!]$$

The temporary array of variables is used to hold any values that might change, so that the subsequent assignment statements may refer to both before and after values of the variables concerned.

4.5 The semantics of combinators

The guarded command semantics of `AND` and `THEN` are both given in terms of sequential composition. The difference between these two combinators is that the arguments to `AND` are required to be mutually independent: neither may require an update to any attribute that appears in the pre- or post-condition of the other.

$$execute\,[\![\,\texttt{SKIP}\,]\!] \;=\; skip$$

$$execute\,[\![\,\texttt{(pre | M)}\,]\!] \;=\; execute\,[\![\,\texttt{M}\,]\!]$$

$$execute\,[\![\,\texttt{M(i = e)}\,]\!] \;=\; (execute\,[\![\,\texttt{M}\,]\!])[i := e]$$

$$execute\,[\![\,\texttt{M1 AND M2}\,]\!] \;=\; execute\,[\![\,\texttt{M1}\,]\!]\,;\, execute\,[\![\,\texttt{M2}\,]\!]$$

$$\begin{aligned} execute\,[\![\,\texttt{M1 OR M2}\,]\!] \;=\;& guard\,[\![\,\texttt{M1}\,]\!] \rightarrow \\ & \quad execute\,[\![\,\texttt{M1}\,]\!] \\ & \Box \\ & \neg\, guard\,[\![\,\texttt{M1}\,]\!] \wedge guard\,[\![\,\texttt{M2}\,]\!] \rightarrow \\ & \quad execute\,[\![\,\texttt{M2}\,]\!] \end{aligned}$$

$$execute\,[\![\,\texttt{M1 THEN M2}\,]\!] \;=\; execute\,[\![\,\texttt{M1}\,]\!]\,;\, execute\,[\![\,\texttt{M2}\,]\!]$$

$$execute\,[\![\,\texttt{ALL C\_each : C WHERE P DO M}\,]\!] \;=$$
$$\mathsf{all}\ C_{each} : C \bullet execute\,[\![\,\texttt{M}\,]\!]$$

The semantics of `OR` is a deterministic choice: the sequence of assignments corresponding to each component is prefixed by the negated guards of any earlier components, reading from left to right.

The program semantics of a method `M` is obtained by prefixing $execute\,[\![\,\texttt{M}\,]\!]$ with a declaration of inputs, and a guard calculated from the precondition:

$$program\,[\![\,\texttt{M}\,]\!] \;=$$
$$\mathsf{input}\ inputs\,[\![\,\texttt{M}\,]\!] \bullet$$
$$guard[\![\,\texttt{M}\,]\!] \rightarrow execute\,[\![\,\texttt{M}\,]\!]$$

The input variables of a method may be determined by recursive inspection of the method syntax, taking the union of component inputs—for example,

$$inputs\,[\![\,\texttt{M1 AND M2}\,]\!] \;=\; inputs\,[\![\,\texttt{M1}\,]\!] \cup inputs\,[\![\,\texttt{M2}\,]\!]$$

—in every case but that of instantiation:

$$inputs\,[\![\,\texttt{M(i = e)}\,]\!] \;=\; (inputs\,[\![\,\texttt{M}\,]\!]) \setminus \{\texttt{i}\}$$

where the substituted input is encapsulated.

The guard corresponding to a method in extension is simply the transliteration of the stated precondition. The guard of a conjunction or disjunction is simply the conjunction or disjunction of component guards, respectively:

$$guard\,[\![\,\texttt{(pre | change | post)}\,]\!] \;=\; pre$$

$$guard\,[\![\,\texttt{SKIP}\,]\!] \;=\; true$$

$$guard\,[\![\,\texttt{(pre | M)}\,]\!] \;=\; pre \wedge guard\,[\![\,\texttt{M}\,]\!]$$

$$guard\,[\![\,\texttt{M(i = e)}\,]\!] \;=\; (guard\,[\![\,\texttt{M}\,]\!])[i := e]$$

$$guard\,[\![\,\texttt{M1 AND M2}\,]\!] \;=\; guard\,[\![\,\texttt{M1}\,]\!] \wedge guard\,[\![\,\texttt{M2}\,]\!]$$

$$guard\,[\![\,\texttt{M1 OR M2}\,]\!] \;=\; guard\,[\![\,\texttt{M1}\,]\!] \vee guard\,[\![\,\texttt{M2}\,]\!]$$

$$guard\,[\![\,\texttt{M1 THEN M2}\,]\!] \;=\; wp(program\,[\![\,\texttt{M1}\,]\!], guard\,[\![\,\texttt{M2}\,]\!])$$

$$guard\,[\![\,\texttt{ALL C\_each : C WHERE P DO M}\,]\!] \;=$$
$$\forall\, c_{each} : C \;\bullet\; P \Rightarrow guard\,[\![\,\texttt{M}\,]\!]$$

To obtain the guard of a sequential composition (`THEN`), we must consider the program semantics of the first component, and then calculate the weakest precondition for this to achieve the guard of the second component.

4.6 Correctness

There is a natural correctness criterion for our semantics: that the calculated guard should be strong enough to ensure that the calculated program will achieve the stated postcondition, viewed as a predicate upon the before and after values of attributes. This is exactly what is required of the method language semantics if the output of the generation process is to be consistent with the specification given in the source model.

We write $post\,[\![\,\texttt{M}\,]\!]$ to denote the postcondition of method `M`, evaluated as a predicate. In this evaluation, `AND`, `OR`, and `ALL` are replaced with their logical equivalents; `THEN` corresponds to a conjunction in which the after values of the first method are identified with the before values of the second, and then concealed using existential quantification.

Our formulation of weakest precondition *wp* reflects the assumption that the (automatically generated) interface to our software component will prevent a method from being invoked unless its (published) guard is satisfied. As a result, the correctness criterion for our semantics may be stated as:

$$guard\,[\![\,\texttt{M}\,]\!] \;\Rightarrow\; wp(program\,[\![\,\texttt{M}\,]\!], post\,[\![\,\texttt{M}\,]\!])$$

for any method `M` produced by the expansion process.

The requirement that the arguments to `AND` and `ALL` should be independent is enough to ensure that each of the combinators is correctness-preserving, and the reader may be reassured—by the following calculation—that the correctness result holds for simple examples of methods written in extension. Consider the methods `M1` defined by

```
M1(b = 0 | a | a = 1)
```

We may calculate that $guard\,[\![\,\texttt{M1}\,]\!] = b = 0$ and that

$$\begin{aligned}
&wp(program\,[\![\,\texttt{M1}\,]\!], post\,[\![\,\texttt{M1}\,]\!])\\
&\quad \Leftrightarrow wp(b = 0 \;\rightarrow\; a := 1, a = 1)\\
&\quad \Leftrightarrow b = 0 \wedge wp(a := 1, a = 1)\\
&\quad \Leftrightarrow b = 0
\end{aligned}$$

confirming that the required implication holds. It is interesting to compare the semantics of this method to that of `M2`, defined by

```
M2(true | a | b = 0 => a = 1)
```

Here, we obtain $guard\,[\![\,\texttt{M2}\,]\!] = true$ and

$$\begin{aligned}
&wp(program\,[\![\,\texttt{M2}\,]\!], post\,[\![\,\texttt{M2}\,]\!])\\
&\quad \Leftrightarrow wp(b = 0 \;\rightarrow\; a := 1 \;\square\; b \neq 0 \;\rightarrow\; skip, b = 0 \Rightarrow a = 1)\\
&\quad \Leftrightarrow wp(b = 0 \;\rightarrow\; a := 1, b = 0 \Rightarrow a = 1) \vee\\
&\qquad\qquad wp(b \neq 0 \;\rightarrow\; skip, b = 0 \Rightarrow a = 1)\\
&\quad \Leftrightarrow b = 0 \wedge wp(a := 1, b = 0 \Rightarrow a = 1) \vee\\
&\qquad\qquad b \neq 0 \wedge wp(skip, b = 0 \Rightarrow a = 1)\\
&\quad \Leftrightarrow b = 0 \wedge true \vee b \neq 0 \wedge (b = 0 \Rightarrow a = 1)\\
&\quad \Leftrightarrow true
\end{aligned}$$

The two methods have the same effect if $b = 0$: the value of attribute a will be set to 1. However, if $b \neq 0$, then `M1` (and any method that requires its invocation) will be blocked, whereas `M2` will remain available.

However, the correctness result does not hold for every method written in extension. It may be that the postcondition of a method includes an impossible requirement, or one for which we have no means of calculating the necessary sequence of assignments. Furthermore, it may be that there is a combination of input and attribute values for which the value of some expression within the postcondition is undefined.

5 Consistency, dependency, and definedness

To ensure that the implementation produced is correct, the compilation phase of the generation process—the formal basis of which is the program semantics defined above—is preceded by an analysis phase, in which preconditions are strengthened to ensure that the corresponding methods are not invoked unless their postconditions can be achieved.

5.1 Consistency

There are two ways in which a postcondition may express an impossible requirement: it may insist that an attribute takes two different values, or it may insist that some value is both an element of, and not an element of, the same set. If this requirement is unconditional, then we have no alternative but to set the precondition of the method to `false`.

Otherwise, we may determine a set of sufficient conditions for the postcondition to be *consistent*—for the impossible requirement not to arise—and add these to the precondition of the method. We do this by considering the existing program semantics for the method, and calculating the weakest precondition for this to satisfy the postcondition.

As an example, consider the method `M` defined by

```
M(true | a | a = 0 & (c = 0 => a = 1))
```

The program semantics of `M` is given by

$$a := 0;\ (c = 0 \rightarrow a := 1, c \neq 0 \rightarrow \mathit{skip})$$

and the weakest precondition for this to achieve $a = 0 \wedge (c = 0 \Rightarrow a = 1)$ is $c \neq 0$. In the analysis phase, the precondition of `M` would be strengthened to produce the following declaration:

```
M(c /= 0 | a | a = 0 & (c = 0 => a = 1))
```

As another example, consider the definition of `ReplacePresenter` given in Fig. 1. Here, the postcondition is impossible to achieve if the two inputs `existing_in` and `new_in` have the same value. In the analysis phase, this will be detected, and the constraint `existing_in /= new_in` will be added to the precondition. The generated interface will not allow this method to be invoked with the same value for both inputs.

A term-rewriting system may be used to increase the readability of the method definitions, and to improve the efficiency of the subsequent implementation. For example, the postcondition

```
x : s & y /: s
```

with the program semantics $s := s \cup \{x\};\ s := s \setminus \{y\}$, gives rise to the additional, calculated precondition:

$$x \in ((s \cup \{x\}) \setminus \{y\}) \wedge y \notin ((s \cup \{x\}) \setminus \{y\})$$

A suitable term-rewriting system would simplify this to $x \neq y$.

5.2 Dependency

If a method may change the values of two different attributes `a` and `b`, then our program semantics may fail to achieve any postcondition in which the after values of these attributes are related. For example, the program semantics of the method `M`, defined by

```
M(true | a, b | a = b & b = c)
```

given by $a := b;\ b := c$, will fail to achieve the postcondition unless the initial value of b is equal to that of c.

As before, we may ensure the correctness of any implementation by calculating the weakest precondition (in this case, $b = c$) for the program semantics to achieve the stated postcondition, and adding this to the precondition part of the method declaration. However, for methods such as `M` above, the results may fall short of the original expectations.

We may improve upon this by constructing a dependency graph, performing a topological sort, and reordering the expressions in the postcondition so that, as far as possible, the value of each attribute will be determined before it is used to determine the value of any other attribute. For example, the definition of `M` may be transformed to produce

```
M(true | a, b | b = c & a = b)
```

for which the program semantics is guaranteed to achieve the postcondition, and no additional precondition will be required.

5.3 Definedness

It may be that the value of an expression within a postcondition will be undefined for some combinations of input and attribute values. For example, in the postcondition

```
a = b/c
```

the value of the expression `b/c` will be undefined when the value of `c` is `0`.

To ensure the correctness of a generated implementation, we may add additional constraints to the postcondition, asserting that the value of each expression lies properly within the range allowed. In the above example, we would add the constraint `integer(a)`, asserting that the after value of `a` is an integer. These constraints make no direct contribution to the program semantics or the generated code.

However, their inclusion enables us to determine whether a stronger precondition is required.

For example, the program semantics of `a = b/c & integer(a)` is simply $a := b/c$, but the weakest precondition for this to achieve the postcondition, including the additional constraint, is $b/c \in integer$. This may be simplified, by a suitable term-rewriting system, to $c \neq 0$, and the condition `c /= 0` may be added to the precondition part of the method declaration.

6 Discussion

6.1 Applications

In this paper, we have shown how statements in a formal, predicate notation may be automatically translated to produce executable programs. This is possible because of the restricted form of postconditions, which ensures that the value of any updated attribute is completely determined; this might seem to reduce the value of the notation as an abstract specification language, but experience suggests that this is not the case.

Before calculating the program semantics of a method, we expand its definition to include contextual information: invariant properties described elsewhere in the model, and assumptions regarding the nature of the application being developed. For example, if an attribute `s` is declared within class `A` by

```
s : SET(B . t)
```

and method `M` requires that a reference to object `b` of class `B` is added to `s`, then the postcondition of `M` will be extended, if necessary, to include the requirement that a reference to the current object will be added to the value of the corresponding attribute `a` of object `b`.

If `t` is an optional attribute, then the precondition of `M` will be extended to include the condition that `t` is not currently set. Furthermore, both pre- and post-condition may be extended to take account of any global invariants that refer to either `s` or `t`, and any preconditions declared in class `B` for the addition of references to `t`.

It is not unreasonable to expect the author of a specification to include sufficient information to determine the values of any attributes deemed important enough to include in the description of the component state: at least, not if the specification is intended to describe an abstract *design*.

Furthermore, we could easily generalise the method notation to allow nondeterministic postconditions, and define a relational semantics that would allow models to be compared using a familiar notion of sequential data refinement. In practice, we have found that the notation is sufficiently abstract for the concise description of complex requirements within the intended application domain.

We are confident of the wider applicability of the approach, as most of the programming problems currently encountered in the development of large applications do not require the invention of sophisticated algorithms [9]. Instead, they require

the consistent implementation of—often quite complex—combinations of simple requirements.

6.2 Further developments

There are several options for the further development of the basic method language. We are exploring its use as an action language within UML class diagrams, as an alternative to the Object Constraint Language OCL [17]. We are considering the removal of the change list: in most situations (see below) the information this contains can be inferred from the postcondition. We wish to allow nested universal and existential quantifications in the postcondition.

The greatest potential for further development, however, is in the design of other aspects of the modelling language. Here, we are aiming to provide theoretical and practical support for a wider range of global invariants. We are also considering the use of state diagrams to describe intended workflow, and the automatic incorporation of the information they contain, in the style of [15].

An early objective is the extension of the notion of association to include some indication of whether a linked object should be deleted when a particular reference to it is removed. At present, this may be inferred from the contents of the stated change list of any method that would remove a reference: if the class of the linked object is present, then the object will be deleted.

However, this is an awkward means of recording intentions, and thus increases the risk that valuable data could be inadvertently deleted; it is also the only reason why the method declarations of the source model must include explicit change lists. It would be better if this information were expressed more directly, as a property of the association.

Even if the change list were removed from the method language, we would wish to extract the corresponding information from the postcondition: it provides valuable confirmation to the designer that a method will change only the expected list of attributes; and it allows the compiler to determine which combinations of methods may be permitted to execute concurrently.

6.3 Related work

In this paper, we have given a semantics to a method language whose design is based in part upon the Z notation. In doing so, we have encountered many of the issues described by Cavalcanti and Woodcock in their weakest precondition semantics for Z [4]. The restricted nature of our language allows us to resolve these issues in a different fashion: in particular, it is easier for us to reason about conjunction and sequential composition. However, their work could be used as a basis for extending the semantics presented here.

The *booster* notation has features in common with Object-Z [14]: the use of reference types; the scoping of operations within classes; the ability to refer to attributes and methods using the dot ‘ . ’ notation. However, the approach to the definition of a semantics is quite different, reflecting the sequential nature of the

system under consideration.

The work presented here fits well with recent work on the Spec# programming system [2], and the extension of the *Boogie* methodology described by Barnett and Naumann [3]. Combining techniques for generation and verification would allow us to address a wider range of specifications: the predicates that do not become programs can be included in the code as assertions.

References

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, LNCS. Springer, 2004.

[3] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC 2004*, LNCS. Springer, 2004.

[4] A. Cavalcanti and J. Woodcock. A Weakest Precondition Semantics for Z. *The Computer Journal*, 41(1), 1998.

[5] J. Davies, C. Crichton, E. Crichton, D. Neilson, and Ib H. Sørensen. Formality, evolution, and model-driven software engineering. In Alexandre Mota and Arnaldo Moura, editors, *Proceedings of SBMF 2004*, ENTCS, 2005.

[6] E. W. Dijkstra and W. H. Feijen. *A Method of Programming*. Addison-Wesley, 1988.

[7] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: assembling applications with patterns, models, frameworks, and tools*. Wiley, 2004.

[8] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained, The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[9] J.-Y. Lucas, J.-L. Dormoy, B. Ginoux, C. Jimenez-Dominguez, and L. Pierre. How to reconcile formal specifications and automatic programming: The Descartes system. In *APSEC '98*. IEEE Press, 1998.

[10] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3), 1971.

[11] C. C. Morgan. *Programming From Specifications (2nd ed.)*. Prentice Hall, 1998.

[12] G. Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4), 1989.

[13] N. Prywes, S. Amir, and S. Shastry. Use of a nonprocedural specification language and associated program generator in software development. *ACM Trans. Program. Lang. Syst.*, 1(2), 1979.

[14] G. Smith. *The Object-Z Specification Language*. Kluwer, 2000.

[15] C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. Technical report, Electronics and Computer Science, Southampton, 2004.

[16] J. M. Spivey. *The Z Notation (second edition)*. Prentice Hall, 1992.

[17] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, 2003. 2nd edition.

[18] J. Welch, D. Faitelson, and J. Davies. Automatic maintenance of association invariants. In *Proceedings of SEFM 2005*. IEEE Press, 2005. To appear.