Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 195 (2008) 151–170
www.elsevier.com/locate/entcs

Domain-specific Semantics and Data Refinement of Object Models

Jim Davies[1], David Faitelson[2] and James Welch[3]

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

Abstract

This paper shows how a domain-specific semantics for object models can be used to support the development of transformations that reflect a particular implementation strategy. The semantics captures model constraints and domain assumptions in terms of abstract data types, and a transformation is correct if and only if it corresponds to a data refinement. The transformations represent development steps, involving the completion of method descriptions, and validity checks, addressing issues of definedness and consistency. The paper shows how compositions of transformations may be used for the automatic generation of working systems from formal, object-oriented designs.

Keywords: formal methods, object modelling, data refinement, automatic programming, model-driven development, Z notation

1 Introduction

Object-orientation is an effective means of controlling complexity in software systems design. Object-oriented languages provide rich, semantic support for the classification and association of data, operations, and constraints. However, the richness of this semantics means that the consequences of design decisions can be difficult to determine; in particular, it may be difficult for a designer to establish whether the description of an operation is consistent with the specified constraint information.

In this paper, we present a formal semantics for object models that will allow us to calculate the consequences of our design decisions, under two, simple assumptions: that the effect of each operation can be adequately described in terms of the relationship between data values and associations before and after it takes place; and

[1] email: Jim.Davies@comlab.ox.ac.uk
[2] email: David.Faitelson@comlab.ox.ac.uk
[3] email: James.Welch@comlab.ox.ac.uk

doi:10.1016/j.entcs.2007.08.031

that the specification describes a component whose interface will allow concurrent execution only of operations that involve disjoint sets of data values.

We show how this semantics can be extended to include additional assumptions about the domain of application, and used to establish a simple criterion for the correctness of model transformations with respect to a particular implementation strategy: a transformation is correct if and only if it corresponds to a data refinement in the extended semantics. We show also how a sequence of model transformations can be used in the automatic generation of correct implementations of object-oriented designs.

The paper begins with a brief account of object modelling, and an overview of Booster, a formal, object-oriented language that will enable a concise exploration of the issues explored in this paper. In Section 3, we present a semantics for object models, sufficient for the analysis of global constraint information. In Section 4, we show how this semantics may be extended to reflect assumptions about the domain of application, expressed as an implementation strategy for postconditions.

We present a correctness criterion for model transformations: a transformation is correct with respect to a particular implementation strategy if and only if it corresponds to a data refinement within the extended semantics. In the final section, we examine two particular aspects of the semantics: the distinction between classes and components, and the treatment of inheritance. We discuss related work in the domain of formal methods, and outline possible directions for future research.

2 Object models

Object-oriented design arose as extension of object-oriented programming, taking the concepts of classes, associations, and methods into the initial stages of systems analysis and design. The representation of a system as a collection of objects, each with an intuitive explanation in the real world, made it easier for designers to capture requirements and develop understanding—just as the developers of programming languages such as Simula [3] had intended.

Object-oriented modelling languages support the presentation of a design in terms of an object (or class) model, describing the features of each class of objects, and the associations between them. A constraint language can be used to specify pre- and postconditions for operations, and invariant properties for classes and associations. Any further description of operations is usually given in terms of code, written in the target language of the implementation.

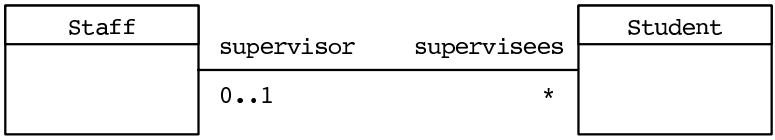
2.1 Association invariants

In realistic applications, we will often see constraints whose scope extends beyond that of a single class. For example, we might wish to insist that, whenever an object of class `Student` refers to an object of class `Staff` as its `supervisor`, then that object should refer to the student object as one of its `supervisees`, and vice

versa:

$$\forall\, s_1 : \mathit{Staff};\ s_2 : \mathit{Student} \bullet s_2 \in s_1.\mathit{supervisees} \Leftrightarrow s_2.\mathit{supervisor} = s_1$$

We refer to such a constraint as an *association invariant*.




```
context Staff
  inv supervisees -> forall( s | s.supervisor = self)

context Student
  inv supervisor.supervisees -> includes(self)
```

Fig. 1. Supervision

In Unified Modeling Language [17] models, the Object Constraint Language (OCL) [21] can be used to specify association invariants. In Fig. 1, the text below the class diagram tells us that the two associations—represented as a single line—are related in the way specified above: `supervisor` and `supervisees` are inverses of one another.

The existence of global constraints is an inevitable consequence of the distribution of related information across multiple classes and associations. However, the complex nature of a large object model, and the phenomenon of aliasing—the fact that there may be more than one means of referring to the same object—may make it extremely difficult for a designer to take proper account of global constraints when specifying an operation. This leads to inconsistencies in design, and errors in implementation.

The demand for more precise object models is increasing with the adoption of model-driven development, in which a wide variety of system artifacts—source code, configuration files, interface definitions—are generated automatically from the current version of the model. If this generation process is to produce correct results, then any global constraints must be included explicitly in the model, making the overall design harder to understand, and more likely to be inconsistent.

This problem can be addressed through the application of formal methods. By giving an appropriate semantics to object models, and providing a formal description of the intended effect of each operation, we can calculate any additional pre- and post-conditions required for the consistency of the overall design. Furthermore, if we extend the semantics with an explicit implementation strategy, we may use the completed model as a basis for the generation of a correct implementation.

```
CLASS Staff
  ASSOCIATIONS
    supervisees : SET(Student . supervisor)
  METHODS
    assign( true
          | student_in : supervisees )
END;

CLASS Student
  ATTRIBUTES
    account : NAT
  ASSOCIATIONS
    supervisor : [Staff . supervisees]
    registered : SET(Course . reglist)
    waiting : SET(Course . waitlist)
  METHODS
    register( account >= course_in.fee &
            | course_in : registered &
              account = account_0 - course_in.fee)
END;
```

Fig. 2. Staff and Student

2.2 The Booster notation

The Booster notation is an object-oriented modelling language developed with these objectives in mind. It includes features of three different formal methods—the B method [2], the Z notation [19], and the Refinement Calculus [16]—and is supported by an enhanced version of the B toolkit. An overview of its design, and the default semantics of the method language, have been presented at earlier SBMF conferences [4,6].

A Booster model is described as a collection of classes, each with a collection of attribute, association, and method declarations. Each association is declared in the context of its source class, and takes the type of its target. Basic association invariants—those linking matching pairs of associations—are incorporated in the association declarations; other constraints may be introduced as invariants, declared in the context of an appropriate class.

Fig. 2 presents a Booster model that includes the constraints of the UML model of Fig. 1, adding two further associations: `registered`, and `waiting`. The `supervisor` association is optional: a student may not have a supervisor. The other three associations are set-valued: each staff member may have a number of students as supervisees; each student may be registered or waiting for many courses.

In Booster, each method is declared as a pair of constraints: the intended pre- and post-conditions for that operation. These constraints may refer to attributes and associations declared in other classes. The postcondition constraint may refer also to other methods; this allows the construction of compound operations, and the

```
CLASS Course
  INVARIANTS
    reglist /\ waitlist = {}
    reglist.card <= capacity
  ATTRIBUTES
    fee, capacity : NAT
  ASSOCIATIONS
    reglist : SET(Student . registered)
    waitlist : SET(Student . waiting)
END;

CLASS TA EXTENDS Staff, Student
  INVARIANTS
    supervisor /= this
  METHODS
    register( course_in.waitlist = {}
            | true )
END;
```

Fig. 3. Course and Teaching Assistant (TA)

delegation of aspects of functionality to methods declared in the context of other, associated classes.

The annotation `_in` denotes an input value, which may be a reference to an object of another class; `_0` denotes a value held by an attribute before the operation takes place. In Fig. 2, the method `register` may be performed only if sufficient funds are available and the size of `reglist` for the course in question is less than the specified capacity. The intended effect is that the course should be added to the `registered` list, and that the student's account should be debited by the appropriate fee.

The declaration of an association can be extended to give the name of a matching association in the opposite direction; this is an economical means of expressing symmetry properties. For example, the declaration of `supervisor` within `Student` includes the information that `supervisees` is the name of the matching association within `Staff`, and vice versa. This captures exactly the property described at the beginning of this section.

Other constraints may be declared simply as class invariants. In Fig. 3, the constraints `reglist /\ waitlist = {}` and `reglist.card < capacity` express the requirements that: no student should be on both the waiting list and the registration list for the same course; and the number of students registered on a course should not exceed its stated capacity.

The class `TA` extends `Staff` and `Student`, inheriting the features of both classes, and adding two additional constraints: the `supervisor` should not be a reference to the current object (`this`); and the `register` operation should be available only if the waiting list for the course is empty. A teaching assistant cannot be his or her

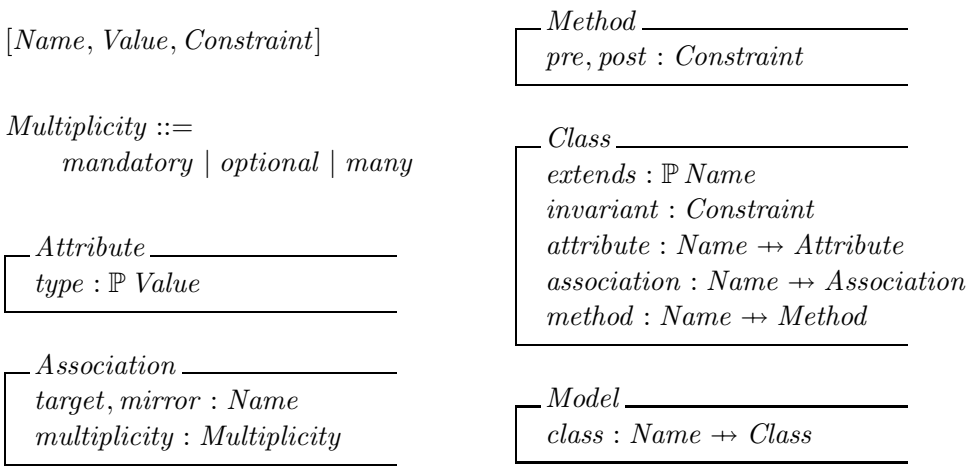


Fig. 4. Model structure

own supervisor, and can register on a course only if there are no students waiting.

Method declarations may be re-used by including method names as part of a postcondition, and composed using one of four method combinators: `AND`, `OR`, `THEN`, and `ALL`: for example, we might declare

```
wait( true
    | course_in : waiting )

signup( register OR wait )
```

The method `signup` has the intended effect of registering a student on a course, or adding them to the waiting list if this is not possible.

3 Component semantics

To define a semantics for object models, we introduce an abstract syntax, representing the structure of our models using names, sets, and relations. In the syntax of Fig. 4: a model is described as a collection of named classes; each class has a list of superclasses, an invariant property, and named methods, attributes, and associations; each method has two named components, both constraints, denoting its pre- and post-conditions; each attribute has a type, corresponding to a range of primitive values. For the purposes of this paper, we will take *Name*, *Constraint*, and *Value* as given sets.

An association has a target type and a multiplicity, which may be optional, mandatory, or many. It may be connected to a matching association in the opposite direction by naming that association as its *mirror*: this provides an economical means of specifying the basic form of association invariant. The multiplicity of the two associations will determine the nature of the underlying relationship.

We will map each instance of this structure to an abstract data type, comprising

a set of states, and a collection of named operations upon these states. The abstract data type will represent the semantics of a *component*, specified by the collection of associated classes in the object model. If we use *ObjectID* to denote a set of references, then the set of all possible component states may be described by the following schema:

$$
\begin{array}{|l}
\textit{ComponentState} \\
\hline
\textit{extent} : \textit{Name} \pfun \mathbb{P}\, \textit{ObjectID} \\
\textit{link} : \textit{Name} \pfun \textit{Name} \pfun (\textit{ObjectID} \rel \textit{ObjectID}) \\
\textit{value} : \textit{Name} \pfun \textit{Name} \pfun (\textit{ObjectID} \pfun \textit{Value}) \\
\hline
\end{array}
$$

For each class name, we have an *extent*, modelled as a set of references; for each association, we have a relation between references; for each attribute, we have a function from references to values.

We will use schemas to introduce names to represent particular instances of classes, associations, attributes, and methods. The same schemas will declare the identifiers of the corresponding schema type, and relate their values to those of identifiers in the enclosing scope: for example,

$$
\begin{array}{|l}
\textit{IdentifyModel} \\
\hline
\textit{Model} \\
\textit{thisModel} : \textit{Model} \\
\hline
\textit{thisModel} = \theta \textit{Model} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{IdentifyClass} \\
\hline
\textit{IdentifyModel};\ \textit{Class} \\
\textit{thisClass} : \textit{Name} \\
\hline
\textit{thisClass} \mapsto \theta \textit{Class} \in \textit{class} \\
\hline
\end{array}
$$

The second schema relates the two scopes *Model* and *Class*, linking the values of identifiers so that, for example, $\textit{extends} = (\textit{class}\ \textit{thisClass}).\textit{extends}$.

We define a semantic function $\mathcal{C}$ to translate each model constraint into the semantic notions of *extent*, *link*, and *value*. If

$$\textit{Condition} == \mathbb{P}\, \textit{ComponentState}$$

then this function has the type

$$\mathcal{C} : \textit{Model} \pfun \textit{ObjectID} \pfun \textit{Constraint} \pfun \textit{Condition}$$

The translation of the constraint is parameterised by the identity of the current object: any relative paths in the constraint expression will begin from this point.

Our model semantics will be parameterised by an implementation strategy $\mathcal{P}$, expressed as a function from constraints to programs:

$$\mathcal{P} : \textit{Model} \pfun \textit{ObjectID} \pfun \textit{Constraint} \pfun \textit{Program}$$

For the purposes of this section, we need not explore the representation of programs, requiring only that two other functions can be defined: the first describes a notion of weakest preconditions,

$$wp : Program \times Condition \nrightarrow Condition$$

and the second maps each program to the corresponding effect upon the state of the component:

$$\mathcal{R} : Program \nrightarrow (ComponentState \leftrightarrow ComponentState)$$

We will emphasise the key arguments of $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{R}$ by enclosing them within double brackets.

In this account of the semantics, we will assume that we have already 'flattened' any inheritance hierarchies. Within each class declaration, we may add the conjunction of the invariants, and the union of the attribute lists, declared in any superclasses. Within each method declaration, we may add the conjunction of the preconditions, and the conjunction of the postconditions, declared for the same method in any superclasses. The justification for this, and the implications for object modelling, will be discussed at length in Section 5.

The conditions upon the component state are derived from information declared within classes, most notably the class invariants:

InvariantConstraint

$$IdentifyClass;\ ComponentState$$

$$\forall\, thisObject : extent\ thisClass \bullet$$
$$\quad \textbf{let}\ \mathcal{C}' == \mathcal{C}\ thisModel\ thisObject \bullet \theta ComponentState \in \mathcal{C}'\ [\![\ invariant\ ]\!]$$

For each class, the invariant constraint must be satisfied when instantiated for each of the current objects: that is, for each object reference in the extent of the class, the current component state (written as $\theta ComponentState$) must satisfy the instantiated condition $\mathcal{C}\ thisModel\ thisObject\ [\![invariant]\!]$, abbreviated as $\mathcal{C}'\ [\![\ invariant\ ]\!]$.

Similarly, the value of each attribute must be of the appropriate type:

AttributeConstraint

$$IdentifyAttribute;\ ComponentState$$

$$\forall\, thisObject : extent\ thisClass \bullet$$
$$\quad thisAttr \in \mathrm{dom}(value\ thisClass) \wedge$$
$$\quad\quad value\ thisClass\ thisAttr\ thisObject \in type$$

Just as attribute declarations constrain the *value* part of the component state, association declarations constrain the *link* relation. We record link information for exactly those association names declared in a class, and the information recorded must reflect the type and multiplicity constraints.

AssociationConstraint

IdentifyAssociation; *ComponentState*

$\mathrm{dom}(\mathit{link}\ \mathit{thisClass}) = \mathrm{dom}\ \mathit{association}$

let $\mathit{thisLink} == \mathit{link}\ \mathit{thisClass}\ \mathit{thisAssoc} \bullet$

$\quad \mathit{thisLink} \in \mathit{extent}\ \mathit{thisClass} \leftrightarrow \mathit{extent}\ \mathit{target} \land$

$\quad \mathit{thisLink} = (\mathit{link}\ \mathit{target}\ \mathit{mirror})^{\sim} \land$

$\quad \mathit{multiplicity} = \mathit{mandatory} \Rightarrow \mathrm{dom}\ \mathit{thisLink} = \mathit{extent}\ \mathit{thisClass} \land$

$\quad \mathit{multiplicity} \neq \mathit{many} \Rightarrow \mathit{thisLink} \in \mathit{extent}\ \mathit{thisClass} \nrightarrow \mathit{extent}\ \mathit{target}$

These constraints are combined in the following schema, for each of the classes in the model, and for each of the attributes and associations in each class:

ModelConstraints

IdentifyModel; *ComponentState*

$\mathrm{dom}\ \mathit{class} = \mathrm{dom}\ \mathit{link} = \mathrm{dom}\ \mathit{extent} = \mathrm{dom}\ \mathit{value}$

$\forall\ \mathit{thisClass} : \mathrm{dom}\ \mathit{class} \bullet \exists\ \mathit{Class} \bullet$

$\quad \mathit{InvariantConstraint} \land$

$\quad (\forall\ \mathit{thisAttr} : \mathrm{dom}\ \mathit{attribute} \bullet \exists\ \mathit{Attribute} \bullet \mathit{AttributeConstraint}) \land$

$\quad (\forall\ \mathit{thisAssoc} : \mathrm{dom}\ \mathit{association} \bullet \exists\ \mathit{Association} \bullet \mathit{AssociationConstraint})$

The semantics of a method will be given as an operation: a relation between component state, parameterised by an object identifier. We will define an operation for each named method in each named class:

$$\mathit{Operations} == \mathit{Name} \nrightarrow \mathit{Name} \nrightarrow \mathit{ObjectID} \nrightarrow (\mathit{ComponentState} \leftrightarrow \mathit{ComponentState})$$

The operation corresponding to a method with precondition *pre* and postcondition *post* is obtained by restricting the relation produced by the postcondition to a suitable subset of its domain. The restriction ensures that whenever the operation is applied, the component invariant—the sum total of constraint information presented in the class declarations—is maintained; it is calculated as the weakest precondition for the intended program to achieve the component invariant:

MethodSemantics

$$
\begin{array}{l}
\mathit{IdentifyMethod} \\
\mathit{modelInvariant} : \mathit{Condition} \\
\mathit{thisObject} : \mathit{ObjectID} \\
\mathit{operations} : \mathit{Operations} \\
\hline
\textbf{let } \mathcal{C}' == \mathcal{C}\ \mathit{thisModel}\ \mathit{thisObject};\ \mathcal{P}' == \mathcal{P}\ \mathit{thisModel}\ \mathit{thisObject} \bullet \\
\quad \mathit{operations}\ \mathit{thisClass}\ \mathit{thisMethod}\ \mathit{thisObject} = \\
\quad\quad \mathcal{C}' [\![\mathit{pre}]\!] \cap \mathit{wp}(\mathcal{P}' [\![\mathit{post}]\!],\ \mathcal{C}' [\![\mathit{post}]\!] \cap \mathit{modelInvariant}) \\
\quad\quad \lhd \\
\quad\quad (\mathcal{R} \circ \mathcal{P}') [\![\mathit{post}]\!]
\end{array}
$$

where $\lhd$ is the domain restriction operator of the Z notation, restricting the relation to those pairs whose first element lies within the specified set.

We may now define our semantics, relating each instance of *Model* to an instance of an abstract data type, represented as an element of the schema type

ADT

$$
\begin{array}{l}
\mathit{states} : \mathbb{P}\ \mathit{ComponentState} \\
\mathit{operations} : \mathit{Operations}
\end{array}
$$

by applying the constraint of *MethodSemantics* to each method, within each class, with the component invariant defined by *ModelConstraints*:

ComponentSemantics

$$
\begin{array}{l}
\mathit{Model};\ \mathit{ADT} \\
\hline
\mathrm{dom}\ \mathit{operations} = \mathrm{dom}\ \mathit{class} \\
\textbf{let } \mathit{modelInvariant} == \{\mathit{ComponentState} \mid \mathit{ModelConstraints}\} \bullet \\
\quad \mathit{states} = \mathit{modelInvariant} \land \\
\quad \forall\, \mathit{thisClass} : \mathrm{dom}\ \mathit{class} \bullet \exists\, \mathit{Class} \bullet \\
\quad\quad \forall\, \mathit{thisMethod} : \mathrm{dom}\ \mathit{method} \bullet \exists\, \mathit{Method} \bullet \\
\quad\quad\quad \forall\, \mathit{thisObject} : \mathit{ObjectID} \bullet \mathit{MethodSemantics}
\end{array}
$$

For any pair of abstract data types A and B, we say that B is a refinement of A if every behaviour of B, in the context of a sequential program, is a possible behaviour of A. In the simplest case, where the two data types share the same underlying state space, and have the same initialisation and finalisation, B is a refinement of A if and only if

$$\mathrm{dom}\ \mathit{op}\ A \subseteq \mathrm{dom}\ \mathit{op}\ B \quad \land \quad (\mathrm{dom}\ \mathit{op}\ A \lhd \mathit{op}\ B) \subseteq \mathit{op}\ A$$

for any operation name *op*, where *op A* and *op B* denote the relations corresponding to *op* in A and B respectively.

If two models M_A and M_B differ only in terms of invariant properties and method descriptions, then the data type semantics of M_A will be refined by that of M_B if and only if for each method

$$pre'_A \subseteq pre'_B \quad \wedge \quad pre'_A \lhd (\mathcal{R} \circ \mathcal{P}')\,[\![\, post_B \,]\!] \subseteq (\mathcal{R} \circ \mathcal{P}')\,[\![\, post_A \,]\!]$$

where pre'_A and pre'_B denote the completed preconditions of the method in models M_A and M_B respectively, and are given by

$$pre'_A \;=\; \mathcal{C}'\,[\![\, pre_A \,]\!] \cap wp(\mathcal{P}'\,[\![\, post_A \,]\!],\ \mathcal{C}'\,[\![\, post_A \,]\!] \cap inv_A)$$
$$pre'_B \;=\; \mathcal{C}'\,[\![\, pre_B \,]\!] \cap wp(\mathcal{P}'\,[\![\, post_B \,]\!],\ \mathcal{C}'\,[\![\, post_B \,]\!] \cap inv_B)$$

In the domain-specific context of *Program*, *wp*, and $\mathcal{P}$, we may produce a data refinement by weakening the *completed* precondition, or by strengthening the *completed* postcondition of any operation.

This semantics for object models, together with the corresponding notion of refinement, gives us a correctness criterion for model transformations. If the implementation strategy is fully deterministic, in the sense that $\mathcal{P} \fatsemi \mathcal{R}$ is functional, then a transformation is correct if and only if it preserves the model semantics exactly. If not, then we have a more general requirement: that the transformation should correspond to a data refinement of the completed model semantics.

4 A specific application domain

4.1 Target language

To demonstrate the value of the semantics, we will consider a possible application domain for the Booster notation, expressed as an implementation strategy directed at the following language of guarded commands:

$$\begin{aligned}\langle command\rangle ::= {}& \text{“skip”} \mid \langle assignment\rangle \mid \\ & \langle guard\rangle \text{“}\rightarrow\text{”} \langle command\rangle \mid \langle command\rangle \text{“}\Box\text{”} \langle command\rangle \mid \\ & \langle command\rangle \text{“ ; ”} \langle command\rangle \mid \langle command\rangle \text{“}\|\text{”} \langle command\rangle \mid \\ & \text{“all”} \langle variable\rangle \text{“:”} \langle variable\rangle \text{“.”} \langle command\rangle \mid \\ & \text{“any”} \langle variable\rangle \text{“:”} \langle variable\rangle \text{“.”} \langle command\rangle\end{aligned}$$

Here, ‘all’ is implemented as iteration, and ‘any’ will choose an object identifier for which the specified predicate is true, provided that there is at least one available.

4.2 Implementation strategy

In this example, we will consider a simple strategy in which primitive conditions—equality, set membership, set non-membership—are mapped to guarded assignments, and other predicates are either decomposed, or mapped to guard conditions. This strategy may map methods to programs that have non-trivial initial guards:

the resulting operation is not always available; it is thus most appropriate for the development of data components with interfaces that can block the invocation of methods in circumstances where they are not applicable.

The effect of $\mathcal{P}'$ upon composite postconditions is exactly as one might expect:

$$\begin{aligned}
\mathcal{P}' \llbracket\ \mathit{true}\ \rrbracket &= \mathrm{skip} \\
\mathcal{P}' \llbracket\ p \wedge q\ \rrbracket &= \mathcal{P}' \llbracket\ p\ \rrbracket \parallel \mathcal{P}' \llbracket\ q\ \rrbracket \\
\mathcal{P}' \llbracket\ p \vee q\ \rrbracket &= \mathcal{P}' \llbracket\ p\ \rrbracket \,\Box\, \mathcal{P}' \llbracket\ q\ \rrbracket \\
\mathcal{P}' \llbracket\ p \Rightarrow q\ \rrbracket &= (\mathcal{C} \llbracket\ p\ \rrbracket \rightarrow \mathcal{P}' \llbracket\ q\ \rrbracket) \\
&\quad\ \Box \\
&\quad\ (\neg\, \mathcal{C} \llbracket\ p\ \rrbracket \rightarrow \mathrm{skip}) \\
\mathcal{P}' \llbracket\ \forall\, a : s \bullet p\ \rrbracket &= \mathrm{all}\ a : s \,.\, \mathcal{P}' \llbracket\ p\ \rrbracket \\
\mathcal{P}' \llbracket\ \exists\, a : s \bullet p\ \rrbracket &= \mathrm{any}\ a : s \,.\, \mathcal{P}' \llbracket\ p\ \rrbracket
\end{aligned}$$

When it comes to the primitive conditions, $\mathtt{a} = \mathtt{b}$, $\mathtt{a} \in \mathtt{s}$ and $\mathtt{a} \notin \mathtt{s}$, the effect of $\mathcal{P}'$ depends upon whether $\mathtt{a}$ and $\mathtt{s}$ are reference-valued—denoting associations—or attributes of primitive type. If they are reference-valued, then we must consider the multiplicity and symmetry properties of the corresponding association: for example, if $\mathtt{a}$ and $\mathtt{b}$ correspond to an optional-to-optional association between classes $\mathtt{B}$ and $\mathtt{A}$, then $\mathcal{P}' \llbracket\ \mathtt{a} = \mathtt{a1}\ \rrbracket$ is interpreted as follows:

$$\begin{aligned}
\mathcal{P}' \llbracket\ \mathtt{this.a} = \mathtt{a1}\ \rrbracket = \mathtt{a1.b} = \{\} \rightarrow\ & \\
& \mathtt{a} := \mathtt{a1};\ \mathtt{a1.b} := \mathtt{this}; \\
& \quad (\mathtt{a}_0 = \{\} \vee \mathtt{a}_0 = \mathtt{a1} \rightarrow \mathrm{skip} \\
& \quad\ \Box \\
& \quad\ \mathtt{a}_0 \neq \{\} \wedge \mathtt{a}_0 \neq \mathtt{a1} \rightarrow \mathtt{a}_0\mathtt{.b} := \{\})
\end{aligned}$$

Provided that the matching optional attribute $\mathtt{b}$—denoting the reverse association—is not already set, this method will assign the prescribed value to $\mathtt{a}$, update $\mathtt{b}$ to match. Finally, if $\mathtt{a}$ already had a different value, then the original matching optional attribute, identified as $\mathtt{a}_0\mathtt{.b}$, needs to be unset to maintain the invariant.

Another strategy that we could have adopted here is described by

$$\begin{aligned}
\mathcal{P}' \llbracket\ \mathtt{this.a} = \mathtt{a1}\ \rrbracket = \mathtt{a1.b} = \{\} \rightarrow\ & \\
& \mathtt{a} := \mathtt{a1};\ \mathtt{a1.b} := \mathtt{this}; \\
& \quad (\mathtt{a}_0 = \{\} \vee \mathtt{a}_0 = \mathtt{a1} \rightarrow \mathrm{skip} \\
& \quad\ \Box \\
& \quad\ \mathtt{a}_0 \neq \{\} \wedge \mathtt{a}_0 \neq \mathtt{a1} \rightarrow \mathtt{a}_0\mathtt{.b} := \{\}) \\
\Box & \\
\mathtt{a1.b} \neq \{\} \rightarrow \mathrm{skip} &
\end{aligned}$$

in which the method is always available, but will have no effect if the matching optional attribute is already set when the method is invoked.

The following definition of $\mathcal{P}'$ would give us a more insistent strategy, in which the intended change will be effected, and any other assignments required to maintain the model invariants will be carried out—even if they affect a part of the system not directly associated with the current object.

$$\begin{aligned}\mathcal{P}'\,[\![\,\texttt{this.a} = \texttt{a1}\,]\!] = \texttt{a} := \texttt{a1};\ \texttt{a1.b} := \texttt{this};\\ (\texttt{a}_0 = \{\} \vee \texttt{a}_0 = \texttt{a1} \to \text{skip}\\ \Box\\ \texttt{a}_0 \neq \{\} \wedge \texttt{a}_0 \neq \texttt{a1} \to \texttt{a}_0\texttt{.b} := \{\});\\ (\texttt{a1.b}_0 = \{\} \vee \texttt{a1.b}_0 = \texttt{this} \to \text{skip}\\ \Box\\ \texttt{a1.b}_0 \neq \{\} \wedge \texttt{a1.b}_0 \neq \texttt{this} \to \texttt{a1.b.a} := \{\})\end{aligned}$$

The current version of Booster compiler adopts the first implementation strategy: it is being used for the development of data components with interfaces that can effectively block methods outside their guard (or their completed precondition with respect to the invariants, which amounts to the same thing); it assumes that users can anticipate the effects of their intentions upon directly connected objects, but that effects upon others should be blocked by default—any intent to change attributes elsewhere in the model must be added explicitly.

The most convenient way of expressing the intention to update unrelated attributes is through the use of combinators. The implementation strategy interprets these in the obvious way:

$$\mathcal{P}'\,[\![\,\texttt{M1 AND M2}\,]\!] = \mathcal{P}'\,[\![\,\texttt{M1}\,]\!] \parallel \mathcal{P}'\,[\![\,\texttt{M2}\,]\!]$$

$$\mathcal{P}'\,[\![\,\texttt{M1 OR M2}\,]\!] = \mathcal{P}'\,[\![\,\texttt{M1}\,]\!] \,\Box\, \mathcal{P}'\,[\![\,\texttt{M2}\,]\!]$$

$$\mathcal{P}'\,[\![\,\texttt{M1 THEN M2}\,]\!] = \mathcal{P}'\,[\![\,\texttt{M1}\,]\!] \,;\, \mathcal{P}'\,[\![\,\texttt{M2}\,]\!]$$

Finally, a simple reference to another method is interpreted as a guarded program, where the guard is that method's declared precondition, and the remainder of the program is obtained by applying our strategy to the postcondition:

$$\mathcal{P}'\,[\![\,\texttt{M}\,]\!] = \mathcal{C}\,[\![\,\texttt{M.pre}\,]\!] \to \mathcal{P}'\,[\![\,\texttt{M.post}\,]\!]$$

For the object model of Fig. 2, the association denoted by `supervisees` and `supervisor` is many-to-optional, from the perspective of the `Staff` class. Our implementation strategy interprets the given postcondition of `assign` as follows:

$$\begin{aligned}\mathcal{P}'\,[\![\,\texttt{student\_in : supervisees}\,]\!] \;=\\ \texttt{student\_in.supervisor} = \{\} \;\to\\ \texttt{supervisees} := \texttt{supervisees} \cup \{\texttt{student\_in}\}\ ;\\ \texttt{student\_in.supervisor} := \texttt{this}\end{aligned}$$

Similarly, the association denoted by `registered` and `reglist` is many-to-many,

and the strategy interprets the first postcondition of `register` as follows:

$$\mathcal{P}'\,[\![\,\texttt{course\_in : registered}\,]\!] \;=\; \texttt{registered} := \texttt{registered} \cup \{\texttt{course\_in}\}\;;\; \texttt{course\_in.reglist} := \texttt{course\_in.reglist} \cup \{\texttt{this}\}$$

The fact that `supervisor` is an optional attribute means that the first program is guarded, given our insistence that no change is made, automatically, to an object at more than one remove; a many-to-many association, on the other hand, requires no additional guard.

4.3 Weakest preconditions

For an object *thisObject* of class *thisClass*, the semantic effect of an assignment to association *as* is described by the substitution

$$\mathit{link\ thisClass\ as} := \mathit{link\ thisClass\ as} \oplus \{\mathit{thisObject} \mapsto e\}$$

where $\oplus$ is the relational overriding operator of the Z notation. The weakest precondition for the assignment to achieve condition *cond* is then given by

$$\mathit{wp}(\mathit{as} := e, \mathit{cond}) \;=\; \mathit{cond}\,[\mathit{link\ thisClass\ as} := \mathit{link\ thisClass\ as} \oplus \{\mathit{thisObject} \mapsto e\}]$$

If *at* represents a primitive-valued attribute, then the weakest precondition for the assignment is given by

$$\mathit{wp}(\mathit{at} := e, \mathit{cond}) \;=\; \mathit{cond}\,[\mathit{value\ thisClass\ at} := \mathit{value\ thisClass\ at} \oplus \{\mathit{thisObject} \mapsto e\}]$$

If we regard object identifiers simply as values, then *link* and *value* may be replaced by a single function; it is useful, however, to maintain a distinction.

This formulation of weakest preconditions illustrates the adequacy of the model semantics with regard to the phenomenon of aliasing. If for example, x and y are two different ways of referring to the same object, then the weakest precondition for the assignment $x.a := e$ to establish the condition $y.a = e$ should be simply *true*, and this is the condition obtained from the above calculation. If, on the other hand, our semantics permitted direct substitution, we would obtain

$$\mathit{wp}(x.a := a, y.a = e) \;=\; y.a = e$$

which, as x is aliased to y, would be the wrong result.

If we calculate the weakest precondition for the program $\mathcal{P}'\,[\![\,\texttt{Student.assign}\,]\!]$ to achieve the component invariant, we obtain the constraint

$$\texttt{student\_in.supervisor} = \{\} \wedge \texttt{student\_in} \in \texttt{Student} \wedge \texttt{this} \in \texttt{Staff}$$

together with the requirement that the invariant holds before the operation is invoked. The precondition of the related program $\mathcal{P}'\,[\![\,\texttt{TA.assign}\,]\!]$ has the additional constraint

$$\texttt{student\_in} \neq \texttt{this} \;\wedge\; \texttt{this} \in \texttt{TA}$$

Similarly, the precondition of $\mathcal{P}'\,[\![\,\texttt{Student.register}\,]\!]$ includes

$$\begin{array}{l}\texttt{course\_in.reglist.card} < \texttt{course\_in.capacity} \wedge \\ \texttt{course\_in} \notin \texttt{waiting} \wedge \\ \texttt{course\_in} \in \texttt{Course} \wedge \texttt{this} \in \texttt{Student}\end{array}$$

The final semantics of `assign` and `register` is obtained by restricting the proposed programs with these preconditions. If we use this semantics to direct our implementation, we are guaranteed to produce a system that correctly implements both the method intentions and the model constraints.

4.4 Model transformations

In practice, calculating the weakest precondition for a program to achieve the whole of the component invariant may not be the most effective way to proceed. Instead, we prefer to develop a series of model transformations, consistent with the implementation strategy for a specific domain, and apply these to complete the pre- and post-conditions of methods in a step-wise fashion, until the method description corresponds to the final semantics.

For example, the Booster compiler employs a model transformation that would have the effect of extending the declaration of `assign` from

```
assign( true
      | student_in : supervisees )
```

to one that is closer to the completed semantics:

```
assign( student_in.supervisor = {}
      | student_in : supervisees &
        student_in.supervisor = this )
```

Note that this transformation strengthens the specified precondition of the method, and yet corresponds to a data refinement of the model semantics, in the context of the implementation strategy expressed by $\mathcal{P}'$.

Another transformation, applicable when a postcondition includes a set membership condition for a many-to-many association, extends the declaration of `register`

```
register( account >= course_in.fee &
        | course_in : registered &
          account = account_0 - course_in.fee )
```

to produce

```
register( account >= course_in.fee &
```

```
    course_in.reglist.card < course_in.capacity &
    course_in /: waiting
  | course_in : registered &
    this : course_in.reglist &
    account = account_0 - course_in.fee )
```

Again, this corresponds to a data refinement in the semantics; in fact, in this case, the new semantics is exactly equivalent.

5 Discussion

5.1 Related work

We have presented a semantics for object models that addresses association constraints, as well as global constraints included in class invariants and operation specifications. Although others have identified the shortcomings of semantic approaches that do not address this constraint information—[20] gives a particularly good account—we are not aware of another approach in which it has been successfully incorporated.

Most approaches to the theory of objects have been focussed upon the features of programming language implementations: the models are seen as a way of organising and abstracting object-oriented programs, rather than as specifications in their own right. The key contributions, such as that of [1] have been in terms of logics of programs, rather than abstract semantics of models.

As a consequence of the focus upon programming, many authors have suggested a semantics for models in which associations are treated as attributes of the source class, and classes are then treated as self-contained components. The model semantics is then presented simply as a combination of class semantics, with the semantics of each class determined separately.

While this is consistent with the view of models as collections of classes, and that of classes as implementations of abstract data types:

> *A class—you may have heard this quite a few times by now—is an implementation of an abstract data type, whether formally specified or (as in many cases) just implicitly understood.* [15]

the resulting semantics—although appropriate for programming implementation, where other logics and tools may be applied—will prove inadequate for the analysis of object models; we cannot even express the requirement that each reference should point to an existing object of the appropriate class.

This point is acknowledged in the design of the Unified Modeling Language (UML): in class diagrams, modellers are encouraged to distinguish between attributes and associations; associations are considered at the same level as classes; and the constraint language OCL, now a fundamental component of the UML, is explicitly intended for the description of constraints involving attributes and associations from different classes.

It is acknowledged also in the design of the refinement calculus for object systems presented in [10], where the authors write:

> *We design each use case... to delegate its partial responsibilities to other classes in the class diagram according to what information a class maintains or knows via its associations with other classes*

In that calculus, functionality is expressed by means of use cases, and the published work has yet to be extended to an explicit treatment of pre- and post-conditions. It seems, however, that the authors are moving towards a similar approach to inheritance as that adopted in Booster (see below).

5.2 Objects and Z

The potential impact of global constraints, and the issue of aliasing, is addressed in an early attempt at object-oriented modelling using the Z notation [9], in which the author explores the use of relations to capture reference information (although rejects this for modelling purposes) and observes that the specification of an operation may address attributes and associations outside the scope of the current class.

Other attempts at presenting object-oriented designs using extensions of the Z notation—in particular, early accounts of Object-Z [18]—have not taken global constraints into account. As a result, these languages were not able to support the formal analysis of object models in which associations play an important role. In the case of Object-Z, this limitation has been recognised, initially in

> *... encapsulation is not sufficient to render a system fully modular. That is, the operations defined in a class may prescribe the invocation on one of the referenced objects. As such, the meaning of such operations is not confined to the referencing object—it is not modular.*
>
> [8]

and again in

> *... Object-Z allows coupling constraints between classes which, on the one hand, facilitate specification at a high level of abstraction, but, on the other hand, make class refinement non-compositional.*
>
> [14]

However, the suggested solution is to transform and complete the specification, losing the abstraction, and much of the value of the model. Yet without this, Object-Z descriptions that include class invariants such as

$$\forall s : \mathit{supermarkets} \bullet \\ \quad \mathrm{dom}(s.\mathit{database}.\mathit{itemRec}) \subseteq \mathrm{dom}(\mathit{warehouse}.\mathit{stock})$$

[5]

do not have a formal semantics adequate for the analysis of method specifications in the context of model constraints; neither do they fully support the semantic comparison of two versions of the same model.

5.3 Inheritance

The semantics of inheritance presented in this paper differs from the usual formulation in one important respect: the precondition of a method in a particular class may be stronger than that given for the same method in any superclasses. The usual approach is to weaken preconditions, using disjunction: see, for example, the second 'Assertion Redeclaration Rule' presented in [15], which insists upon the use of the keyword `require else` when redeclaring method constraints.

This approach is seen as a necessary consequence of the principle of *polymorphic substitutability*, outlined in [12]

> *If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T*

In programming terms, this suggests that a method must be defined, or available, in the context of a particular class whenever it would be available in the context of any superclasses: that is, its precondition must be the same, or weaker.

However, while this principle may apply to types in programming languages, it is not clear that it should apply to classes in object models. In an object model, we should not assume that the specified precondition of a method will correspond exactly to its availability in an implementation: other model constraints may restrict this availability; alternatively, the method may be offered more widely, but have no effect when called outside its specified precondition.

As an example, consider the method `assign` defined in the context of `Staff`, and inherited by the subclass `TA`. The subclass includes the additional invariant that the assigned `supervisor` should not be a reference to the current object. It does not seem appropriate to insist that the `TA.assign` method should be allowed when called with argument `this`, just because the method `Staff.assign` would be available in the same circumstances.

If a subclass describes a specialisation, with additional constraints in terms of invariant properties and postconditions, then it seems natural to allow that preconditions, too, can be specialised: whether directly, or through the addition of new attributes, associations, or invariants. We should observe that this view may be reconciled with the generalised version of the principle [13]:

> *Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.*

If our notion of what is provable about a method is restricted to what may be established of the states of the component before and after its execution, given that it is being executed within its precondition, then any property provable of a method remains provable when that method is re-used in the context of a subclass.

With our interpretation, inheritance does not correspond exactly to the refinement of classes as separate abstract data types. This is consistent with our decision to regard association invariants and other global constraints as essential aspects of object modelling, and to consider refinements of models or components, rather than

individual classes.

5.4 Future work

The semantics presented in this paper can be extended to allow the comparison of object models with different class structures, by adding a notion of encapsulation at the component level. Provided that models M_A and M_B present the same collection of classes, attributes, and methods to the environment, we can use the theory of data refinement to compare their semantics. In this way, we may provide formal support for the notion of *refactoring* proposed by [7].

Another application of the semantics involves the incremental extension of implementation strategies. If we extend a strategy $\mathcal{P}$ to a more ambitious strategy $\mathcal{Q}$, and the two strategies are consistent in the sense that for any program *prog*,

$$\begin{aligned}&\mathrm{dom}\,(\mathcal{R} \circ \mathcal{P})\,[\![\, prog \,]\!] \subseteq \mathrm{dom}\,(\mathcal{R} \circ \mathcal{Q})\,[\![\, prog \,]\!] \;\wedge \\ &\quad (\mathrm{dom}\,(\mathcal{R} \circ \mathcal{P})\,[\![\, prog \,]\!] \lhd (\mathcal{R} \circ \mathcal{Q})\,[\![\, prog \,]\!]) \subseteq (\mathcal{R} \circ \mathcal{P})\,[\![\, prog \,]\!]\end{aligned}$$

then the semantics of any model under $\mathcal{Q}$ will be a data refinement of the semantics of the same model under $\mathcal{P}$.

For example, we might extend our existing strategy for Booster to take account of invariants of the form `A /\ B = {}`, where `A` and `B` are associations. A postcondition requiring that `x` be added to `A` could be implemented as `A` $:=$ `A` $\cup$ {`x`}; `B` $:=$ `B` $\setminus$ {`x`}. With this new strategy, the semantics of the `register` method, considered in the context of the invariant `reglist /\ waitlist = {}`, would have a weaker precondition: the constraint that `course_in` $\notin$ `waiting` would disappear.

In selecting an implementation strategy, we must draw a balance between the degree of automation required and the extent to which a modeller can be expected to anticipate the consequences of their design decisions. In the implementation strategy chosen for Booster, we expect the modeller to take into account the values of attributes in the current object, and in any directly-linked object, but any change to an object further removed must be specified explicitly within the postcondition.

The current version of the Booster system executes this strategy automatically, as a series of model transformations; the final method specification is translated directly into functions in C, using library calls to access a data store component (generated using the C libraries of the B toolkit). Each transformation could be proved correct with respect to the strategy presented in this paper: outline proofs for the treatment of basic association invariants are presented in [22]. One of our objectives in developing the next version is to make proofs of correctness more straightforward.

The guarded command language employed here is a development of the one presented at a previous conference [6]: we have extended the weakest precondition semantics to address conditions upon component state, rather than constraints upon attributes; this allows an appropriate treatment of aliasing. Our intention is to further develop this language as a platform-independent notation [11], and to produce automatic translations into C# and Java.

Our work on Booster has been targeted at the development of sequential data components, or object databases, where the maintenance of integrity constraints is particularly important. It would be instructive to see whether the same approach—generating implementations from object models using transformations based upon a formal, domain-specific semantics—would offer the same value when applied to other domains.

References

[1] Abadi, M. and Leino, K. R. M. (2003). A logic of object-oriented programs. In *Verification: Theory and Practice*, pages 11–41.

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

[3] Dahl, O. and Nygaard, K. (1966). Simula, an Algol-based simulation language. *Communications of the ACM*, 9:671–678.

[4] Davies, J., Crichton, C., Crichton, E., Neilson, D., and Sørensen, I. H. (2005). Formality, evolution, and model-driven software engineering. In Mota, A. and Moura, A., editors, *SBMF 2004*, volume 130 of *ENTCS*, pages 39–55.

[5] Duke, R. and Rose, G. (2000). *Formal Object-Oriented Specification Using Object-Z*. Macmillan Press.

[6] Faitelson, D., Welch, J., and Davies, J. (2006). From predicates to programs. In Sampaio, A., editor, *Proceedings of SBMF 2005*. to appear.

[7] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[8] Griffiths, A. (1995). An Extended Semantic Foundation For Object-Z. Technical report, SVRC, University of Queensland.

[9] Hall, A. (1990). Using Z as a Specification Calculus for Object-Oriented Systems. In *Proceedings of VDM '90*, pages 290–318. Springer-Verlag.

[10] Jifeng, H., Li, X., and Liu, Z. (2006). rcos: a refinement calculus of object systems. *Theor. Comput. Sci.*, 365(1):109–142.

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

[12] Liskov, B. (1987). Data abstraction and hierarchy. In *Proceedings of OOPSLA '87*, pages 17–34. ACM Press.

[13] Liskov, B. and Wing, J. (1994). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841.

[14] McComb, T. and Smith, G. (2006). Compositional class refinement in object-z. In *FM 2006*, volume 4085 of *LNCS*. Springer.

[15] Meyer, B. (1997). *Object-Oriented Software Construction, Second edition*. Prentice Hall.

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

[17] OMG (2005). UML 2.0 superstructure specification. http://www.omg.org/cgi-bin/doc?ptc/05-07-04.

[18] Smith, G. (2000). *The Object-Z Specification Language*. Kluwer.

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

[20] van den Berg, J., Breunesse, C.-B., Jacobs, B., and Poll, E. (2001). On the role of invariants in reasoning about object-oriented languages. In *Proceedings of ECOOP'2001*.

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

[22] Welch, J., Faitelson, D., and Davies, J. (2005). Automatic maintenance of association invariants. In *Proceedings of SEFM 2005*. IEEE Press.