USING RELATIONAL AND BEHAVIOURAL SEMANTICS IN THE VERIFICATION OF OBJECT MODELS

Christie Bolton
Oxford University Computing Laboratory
Parks Road, Oxford OX1 3QD, UK
Christie.Bolton@comlab.ox.ac.uk

Jim Davies
Oxford University Computing Laboratory
Parks Road, Oxford OX1 3QD, UK
Jim.Davies@comlab.ox.ac.uk

Abstract This paper shows how a combination of relational and behavioural semantics might be used in the creation and verification of object models. Specifications written in UML may be expressed in terms of abstract data types and processes; different notions of refinement may then be used to establish consistency between diagrams, or to verify that a design is faithful to the specification.

Keywords: specification; verification; refinement; consistency; analysis; UML

1. INTRODUCTION

The Unified Modeling Language (UML) [19] is a visual language for the specification of object-oriented systems. It can be used to describe both structure and behaviour at different levels of rigour and abstraction, using a variety of graphical notations. As [15] observe, it has a precisely-defined syntax and static semantics, but only an informally-specified dynamic semantics.

A considerable amount of work has been done on the formal semantics of UML, notably: a type semantics of class models [11]; a dynamic semantics for state diagrams [15]; and the combined work of the precise UML group [17]. Also important is the work of [2, 1], using LOTOS [14] to reason about Use Case Maps [6].

In this paper, we build upon this work using formal description techniques to reason about UML specifications; we show how individual UML diagrams may be used to construct formal descriptions of system behaviour in terms of abstract data types and processes. The relational and behavioural semantics of these formal descriptions can then be used to support the further development and analysis of an object model.

Our immediate objective is not the definition of an adequate or complete formal semantics for UML. We are concerned instead with the practical issue of how projections of these semantics might be used for analysis and verification. We restrict our attention to diagrams and class descriptions formulated at a particular level of abstraction: one at which the names of methods and navigability of associations have been identified.

We show how a formal representation of a class description can be used to verify a proposed implementation against specification diagrams. The class description itself can be used to construct a single composite abstract data type, whose semantics can be compared with those of the individual specification diagrams.

Once we have constructed our formal descriptions, we are able to compare them using notions of refinement for abstract data types and processes. Since both static and dynamic information is present, our method of comparison requires the definition of a behavioural semantics for data types.

The particular semantics we employ is a *blocking* semantics, in the sense of [5]; operations or methods cannot occur outside their domain of definition. This might seem, at first, a surprising choice for object modelling, given that the public methods of an individual class are always available. However, in examining the behaviour of *combinations* of classes, we observe that in context, methods are not always available, either because of synchronisation constraints or because no active object is ready to call them. With a blocking semantics, we can encode information about availability within our data types.

We present two semantic models: one *relational*, the other *behavioural*. Each model admits a notion of refinement, and a corresponding proof technique. In the relational model, refinement can be established inductively, using simulation rules [22]. In the behavioural model, refinement can be established by model-checking [18].

The behavioural semantics, one of those identified in [10], is consistent with the relational semantics, in that the two refinement orderings coincide: a data type is refined in the behavioural model precisely when it is refined in the relational model [4]. This equivalence allows us to move freely between state-based and behavioural views, using whichever mode of description, and whichever proof technique, is the most directly applicable. For example, abstract data types are a natural target for class descriptions, but sequence diagrams and activity diagrams are more easily formalised in terms of processes.

The paper begins with a brief introduction to the two semantic models, and to our notation for data types and processes. In Section 3, we describe the construction of a data type based upon a class model, and show how the static and dynamic content of various UML diagrams may be tested against this description. In Section 4, we discuss the prospects for extending this work to a larger subset of UML, examine how the various transformations may be automated, and suggest a strategy for the verification of implementation-level diagrams and byte code. We assume some knowledge of Z [21], and in particular, the schema calculus.

2. SEMANTICS

State-based and behavioural specification techniques can be used together to create and reason about formal descriptions of object-oriented systems. As [5] details, a considerable amount of work has been done in linking these two paradigms: notably [13], [9] and [20].

2.1. ABSTRACT DATA TYPES

An abstract data type combines a notion of state with a collection of named operations, modelled as relations, that may involve input and output. Two of these operations are distinguished, representing initialisation and finalisation of the data type.

In modelling data types, we may choose either to associate each operation with input from, and output to, the context of the data type, or alternatively, to maintain the notion of a local environment that stores inputs and outputs from initialisation to finalisation. These two approaches are equivalent: see [22] and [3].

In this paper, we adopt the second approach, defining our data types in terms of three generic parameters, representing the internal state space, the set of possible operation names, and the local environment.

$$\begin{array}{|l}
\hline
ADT[Local, Name, Env] \\
state : \mathbb{P}\, Local \\
init : Env \leftrightarrow Local \\
op : Name \rightarrowtail\!\!\!\!\!\!\!\!+\;\, (Local \leftrightarrow Local) \\
final : Local \leftrightarrow Env \\
\hline
\end{array}$$

To ensure the adequacy of the local environment approach, we insist that both *init* and *final* are total relations.

The schema notation of Z, used above in its generic form, can also be used to describe the components of the data type. We may use schemas to represent the state space as a set of bindings—mappings from named identifiers to values—satisfying any specified data type invariant. Using the schema calculus, we can define operations as relations on the state space without necessarily having to refer to the components of the state schema. We can work at the level of schema names, using θ, the characteristic binding operator; this allows us to refer to the collection of identifiers in a particular schema without introducing a named instance of the schema type. State and operation schemas can be factorised using logical and relational operators.

An alternative approach to the description of data types is offered by Object-Z [8], an object-oriented extension of Z that includes notions of class, instance, inheritance and polymorphism. Both paradigms share the same underlying interpretation in terms of data types.

2.2. RELATIONAL SEMANTICS

We may give a relational semantics to a data type by considering the visible effects of finite sequences of named operations. In our approach, the possible effects of a given sequence may be seen as a relation between environment states, obtained from the sequential composition of the individual operations.

To record the fact that a sequence of operations might be blocked—it might require the performance of an operation outside its domain of definition—we augment the environment space with a distinguished element $\perp$. To ensure that the possibility of blocking is propagated to the end of a sequence, we augment the local state space in the same way, and consider *totalised* versions of the operations.

If X is a set then we write $X_\perp$ to denote the augmented set $X \cup \{\perp\}$, and if r is a relation with source X we write $r^\top$ to denote the totalised relation $r \cup ((X_\perp \setminus \operatorname{dom} r) \times \{\perp\})$ which maps every state outside the domain of r onto $\perp$. For example, if X were the set $\{a, b, c\}$ and r were the relation $\{a \mapsto a, b \mapsto a\}$, then the totalised version $r^\top$ would be the relation $\{a \mapsto a, b \mapsto a, c \mapsto \perp, \perp \mapsto \perp\}$.

Other totalisations are possible. For example, [22] defines one in which every element outside the domain is mapped to every element in the augmented state. This leads to a *non-blocking* semantics in which calling an operation outside its domain can leave you in any state.

Under our chosen totalisation the relational semantics of a data type is then given by

$$
\begin{array}{|l}
[Local, Name, Env] \\
\mathcal{R} : ADT[Local, Name, Env] \rightarrow (\operatorname{seq} Name \nrightarrow (Env_\bot \leftrightarrow Env_\bot)) \\
\hline
\mathcal{R} = (\lambda\, ADT[Local, Name, Env] \bullet \\
\quad (\lambda\, p : \operatorname{seq}(\operatorname{dom} op) \bullet \\
\quad\quad init^\top \mathbin{;} run^\top\; p\; op \mathbin{;} final^\top))
\end{array}
$$

where $run^\top$ maps any sequence of names to a relation upon the augmented local state. It describes the effect of calling the named operations one after the other and is the sequential composition of the *totalised* relations corresponding to each of the named operations.

We say that data type A is refined by data type C if, for each sequence of named operations, the effect upon C is more deterministic than the effect upon A. For any sequence p, the relation $\mathcal{R}\; C\; p$ must be a subset of the relation $\mathcal{R}\; A\; p$. When this is the case, we write $A \sqsubseteq_\mathcal{R} C$. Formally,

$$
\begin{array}{|l}
[Abs, Con, Name, Env] \\
\_ \sqsubseteq_\mathcal{R} \_ : ADT[Abs, Name, Env] \leftrightarrow ADT[Con, Name, Env] \\
\hline
\forall A : ADT[Abs, Name, Env];\ C : ADT[Con, Name, Env] \bullet \\
\quad A \sqsubseteq_\mathcal{R} C \Leftrightarrow \forall p : \operatorname{seq} Name \bullet \mathcal{R}\; C\; p \subseteq \mathcal{R}\; A\; p
\end{array}
$$

Note that the two data types must have the same *interface*, that is, the same combination of *Name* and *Env* that defines the signatures of the operations.

Rather than establishing refinement by means of a generalisation argument within the semantics, we can establish it inductively, using a set of simulation rules. A sound and complete set of rules for simulation in the blocking relational semantics is derived in [4].

2.3. PROCESSES

A process, as defined in [12], is a pattern of communication. We may use processes to represent components in terms of their communicating behaviour, building up descriptions using the standard operators of the CSP language.

Processes themselves are defined in terms of events: synchronous, atomic communications between a process and its environment. Compound events may be constructed using ' . ' – the dot operator. A family of compound events is called a channel. Channels may be used to represent the passing of a value between components.

The atomic process *Skip* denotes successful termination: the end of a pattern of communication. If P is a process and a is an event, then

$a \rightarrow P$ is a process that is ready to engage in a, and if this event occurs it subsequently behaves as P. If P and Q are processes then the process $P \fatsemi Q$ first behaves as P and then, if P successfully terminates, behaves as Q.

If P and Q are processes, then $P \sqcap Q$ represents an *internal* choice between P and Q. This choice is resolved by the process without reference to its environment. An internal choice over a set of indexed processes $\{i : I \bullet P(i)\}$ is written $\sqcap\, i : I \bullet P(i)$.

An *external* choice between two processes, written $P \,\square\, Q$, may be influenced by the environment. This choice is resolved by the first event to occur. An external choice over a set of indexed processes is written $\square\, i : I \bullet P(i)$; if each begins with a different event, then this is a menu of processes for the environment to choose from.

Processes may be defined by sets of mutually-recursive equations, which may be indexed to allow parametrised definitions. Parameters may be used to represent aspects of the process state, and may appear in *guards*: we write $B \,\&\, P$ to denote the process that behaves as P if B is true, and can perform no events otherwise.

If A is a set of events, then the parallel combination $P \,|[\, A \,]|\, Q$ is a process in which components P and Q can evolve independently but must synchronise upon every occurrence of any event from A. Furthermore, the combination cannot terminate until both processes are ready to do so. We use $P \,|||\, Q$ as a synonym for $P \,|[\, \emptyset \,]|\, Q$.

Finally, if P is a process and A is a set of events, then $P \setminus A$ is a process which behaves as P except that both the requirement to synchronise upon, and the ability to observe events from the set A, has been removed.

2.4. BEHAVIOURAL SEMANTICS

Several standard semantic models exist for the process language of CSP: see, for example, [18]. For the purposes of this paper we will employ the traces model and the stable failures model.

In the traces model, each process is associated with a set of traces, or finite sequences of events. The presence of a trace *tr* in the semantic set of a process indicates that it is *possible* for that process to engage in that sequence of events.

In the stable failures model, each process is associated with a set of failures where a failure is a pair in which the first element is a possible trace, and the second is a refusal. The presence of a failure (tr, ref) in the semantic set of a process indicates that it is *possible* for the process to engage in the trace *tr* and then refuse every event from the set *ref*.

Letting Σ denote the set of all event names and *CSP* denote the syntactic domain of process terms, we may define a pair of semantic functions $\mathcal{T}$ and $\mathcal{F}$ which each take a CSP process and return respectively the set of all traces and the set of all failures of the given process:

$$\begin{aligned}
&\mathcal{T} : CSP \nrightarrow \mathbb{P}(\mathrm{seq}\,\Sigma)\\
&\mathcal{F} : CSP \nrightarrow \mathbb{P}(\mathrm{seq}\,\Sigma \times \mathbb{P}\,\Sigma)
\end{aligned}$$

We may use the stable failures model to give a behavioural semantics to a data type by exhibiting a *process equivalent*: the behavioural semantics of a data type A is the failures semantics of its process equivalent, $\mathcal{F}(process\ A)$. The same approach is taken in [10].

The function *process*, from data types to process terms is given by

```
[Local, Name, Env]
  $process : ADT[Local, Name, Env] \to CSP$
  ----------------------------------------
  $process =$
    $(\lambda\, ADT[Local, Name, Env] \bullet$
      let
        $P(a) \;=\; \Box\, n : \mathrm{dom}\, op \mid a \in \mathrm{dom}(op\ n) \bullet$
                  $\mathrm{op}.n \to$
                      $\sqcap\, a' : state \mid a \mapsto a' \in (op\ n) \bullet P(a')$
                $\Box$
                $\sqcap\, e' : Env \mid a \mapsto e' \in final \bullet \mathrm{final}.e' \to Skip$
      within
        $\Box\, e : Env \bullet$
          $\mathrm{init}.e \to \sqcap\, a' : state \mid e \mapsto a' \in init \bullet P(a'))$
```

We observe that there are three channels of events: init and final which take an argument of type *Env* and op which takes the name of an operation.

Both the traces and the stable failures models admit refinement orderings, based upon reverse containment:

```
  $\_ \sqsubseteq_{\mathcal{T}} \_\,,\ \_ \sqsubseteq_{\mathcal{F}} \_ \ : CSP \leftrightarrow CSP$
  ----------------------------------------
  $\forall P, Q : CSP \bullet$
    $P \sqsubseteq_{\mathcal{T}} Q \Leftrightarrow \mathcal{T}(Q) \subseteq \mathcal{T}(P) \land$
    $P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow \mathcal{F}(Q) \subseteq \mathcal{F}(P)$
```

These two notions of refinement are consistent, in that $\sqsubseteq_{\mathcal{F}} \subseteq \sqsubseteq_{\mathcal{T}}$. In each case, refinement may be established through a combination of structural induction [7], data-independence [16], and exhaustive, mechanical model-checking [18].

Crucially, the failures refinement ordering coincides with the refinement ordering in our relational model. For any data types A and B,

$$A \sqsubseteq_{\mathcal{R}} B \Leftrightarrow \textit{process } A \sqsubseteq_{\mathcal{F}} \textit{process } B.$$

A proof of this result is presented in [4].

3. OBJECT MODELLING

In object modelling, we may use a variety of tools and techniques, such as class models, use cases, scenarios, activity diagrams, interaction diagrams, sequence diagrams, and state diagrams, to arrive at a suggested class description.

From the class description, we can produce a mathematical model of the system in terms of data types. If necessary, we can extend this model using additional information from activity diagrams and interaction diagrams, either by extending the data type or by placing a constraint process in parallel.

Using the relational and behavioural semantics of these data types, we can verify that the design exhibits the intended behaviour, and check that the various parts of the specification are consistent. Using the refinement orderings, we can compare our data type description to the information content of the various use cases and diagrams.

3.1. CLASS DESCRIPTIONS

In this paper, we restrict our attention to class descriptions in which all class names, attributes and methods have been identified. In addition, we consider only those descriptions in which multiplicity and navigability information is expressed directly: that is, without the use of association classes.

Information about the availability and effect of methods can be drawn either from model annotations using the Object Constraint Language (OCL), or from an accompanying collection of state diagrams.

We can use state diagrams to provide information about the availability of methods if we regard the transition information as *complete*: that is, a method is available precisely when there is a suitably-labelled transition from the current state. This assertion applies only to the subset of the class methods, and the projection of the class state, presented in the current state diagram. Other diagrams may present information about other methods, or about the effects of the same subset of methods upon other components of the state.

To reason about state diagrams in which the transition information is intentionally *partial*, we might employ a non-blocking relational se-

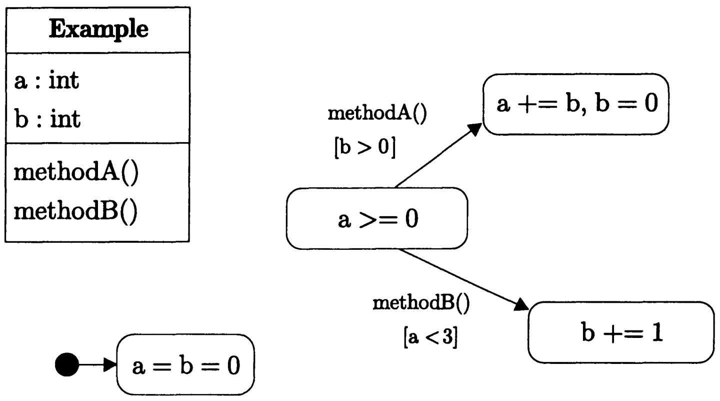

Figure 1 Fragments of a class description

mantics, with a matching behavioural semantics, in which the result of performing an operation outside its domain is left undefined. With such a semantics, we may *refine* the information content of a diagram by adding transitions. However, no availability information can be inferred.

As an example, consider the simple class description presented in Figure 1. The class diagram describes a single class with two data members, a and b, of types A and B, respectively. It also introduces two methods, *methodA* and *methodB*, but reveals only that they expect no arguments.

The accompanying state diagram describes the methods in greater detail, presenting information about their availability and their effect upon the state. Each of the boxes contains a predicate written in OCL describing a constraint upon the combination of values taken by the data members of the state.

Using this information, we may define a simple data type using Z schemas to describe class state and methods.

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{State}} \\ a, b : \mathbb{Z} \\ \hline \end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{\mathit{Init}} \\ \hline \mathit{State}' \\ \hline a' = b' = 0 \\ \hline \end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{\mathit{OpA}} \\ \hline \Delta \mathit{State} \\ \hline a \geq 0 \ \wedge \ b > 0 \\ a' = a + b \ \wedge \ b' = 0 \\ \hline \end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{\mathit{OpB}} \\ \hline \Delta \mathit{State} \\ \hline a \geq 0 \ \wedge \ a < 3 \\ a' = a \ \wedge \ b' = b + 1 \\ \hline \end{array}$$

After initialisation, the state will satisfy $a = b = 0$. For either method to be called, the state must satisfy $a \geqslant 0$; if *methodA* is to be called, then an additional constraint $b > 0$ must also be satisfied; a similar condition applies to *methodB*. If *methodA* is called, then the resulting state, described by values a' and b', will satisfy $a' = a + b \wedge b' = 0$; if *methodB* is called, then the resulting state will satisfy $b' = b + 1$.

In this example, we have no information about the accessibility of data members. We will assume that either member could be accessed, and choose an injective finalisation: such a finalisation can be used to export any state information to the environment. The data type corresponding to this example is as follows

$$\begin{array}{|l}
\textit{Example} \\
\hline
\mathit{ADT}[\mathit{State}, \mathit{Name}, \mathit{Env}] \\
\hline
\mathit{state} = \mathit{State} \\
\mathit{init} = \{\, \mathit{Env};\ \mathit{Init} \bullet \theta \mathit{Env} \mapsto \theta \mathit{State}' \,\} \\
\mathit{op} = \{\, \mathit{methodA} \mapsto \{\mathit{OpA} \bullet \theta \mathit{State} \mapsto \theta \mathit{State}'\}, \\
\qquad\quad \mathit{methodB} \mapsto \{\mathit{OpB} \bullet \theta \mathit{State} \mapsto \theta \mathit{State}'\,\}\,\} \\
\mathit{final} \in \mathit{State} \rightarrowtail \mathit{Env} \\
\hline
\end{array}$$

where the local environment *Env* is any set into which the state can be embedded. With a schema to match the relation *final*,

$$\begin{array}{|l}
\textit{Final} \\
\hline
\mathit{State};\ \mathit{Env} \\
\hline
\theta \mathit{Env} = \mathit{final}\, \theta \mathit{State} \\
\hline
\end{array}$$

we may define a process equivalent for our data type, and hence obtain a behavioural semantics for our class description:

$$\begin{array}{l}
\mathit{ClassDesc} = \\
\quad \text{let} \\
\qquad \mathit{Proc}(\mathit{State}) = \\
\qquad\quad \text{pre } \mathit{OpA}\ \&\ \text{methodA} \rightarrow \sqcap\, \mathit{OpA} \bullet \mathit{Proc}(\theta \mathit{State}') \\
\qquad\quad \Box \\
\qquad\quad \text{pre } \mathit{OpB}\ \&\ \text{methodB} \rightarrow \sqcap\, \mathit{OpB} \bullet \mathit{Proc}(\theta \mathit{State}') \\
\qquad\quad \Box \\
\qquad\quad \sqcap\, \mathit{Final} \bullet \text{final}.\theta \mathit{Env} \rightarrow \mathit{Skip} \\
\quad \text{within} \\
\qquad \sqcap\, \mathit{Init} \bullet \text{init} \rightarrow \mathit{Proc}(\theta \mathit{State}')
\end{array}$$

3.2. ACTIVITY GRAPHS

To verify that a class description satisfies the requirements captured by other diagrams, we need to construct formal descriptions for these diagrams. In this section, we show how this can be done for a particular language of activity graphs, corresponding to a strict subset of UML.

The terms in the language, which we will refer to as *activities*, correspond to the action states of a UML activity graph. The atomic terms are expressed purely in terms of methods; by identifying methods with events, we may express these activities as CSP processes. We consider three distinct types of activities: *Act*, *TAct* and *Graph*.

Each element of *Act* is a basic activity with neither an external starting point nor an external stopping point. Two of these basic activities can be combined either sequentially or in parallel to produce another basic activity; we use *then* and *parallel* to describe these graphical operators. In addition, an activity may be turned into a terminating activity using *stop*. These three cases are illustrated in Figure 2.

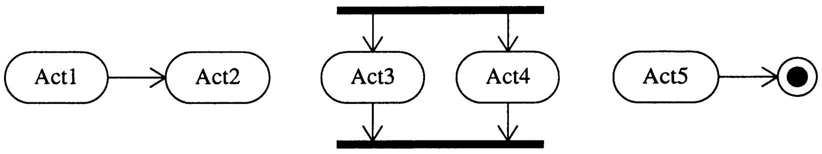

Figure 2 Combining simple activities and terminating a simple activity: $then(Act1, Act2)$, $parallel(Act3, Act4)$ and $stop(Act5)$.

Terminating activities can be combined with explicit forks, or with decision boxes as illustrated in Figure 3. The branches of a decision are guarded with Boolean expressions. We may obtain the effect of an unguarded decision box by setting the guard to *true*. In Figure 3 we also illustrate how we may attach a starting point to a terminating activity, in which case the result is an activity of type *Graph* and how our diagrams can be nested: a complete graph can itself be used as a basic activity using *include*.

We describe loops within activity graphs by using *declare* and a *Label* to mark where the loop begins and using *use* to mark where we return back to the beginning of the loop. We show how these operators may be used in the example illustrated in Figure 4.

Assuming suitable definitions of *Label* and *Bool*, our language has the following abstract syntax:

$$\begin{aligned} Act \quad ::= \quad & atom \langle\!\langle CSP \rangle\!\rangle \mid then \langle\!\langle Act \times Act \rangle\!\rangle \mid parallel \langle\!\langle Act \times Act \rangle\!\rangle \mid \\ & declare \langle\!\langle Label \times Act \rangle\!\rangle \mid include \langle\!\langle Graph \rangle\!\rangle \end{aligned}$$

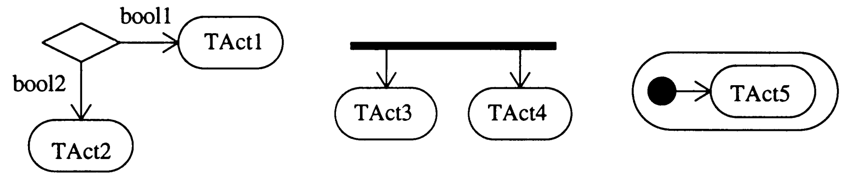

Figure 3 Combining terminating activities: *decide*(*bool*1, *bool*2, *TAct*1, *TAct*2) and *fork*(*TAct*3, *TAct*4) and *include*(*start*(*TAct*5)).

$$\begin{aligned} TAct \; &::= \; stop \, \langle\!\langle Act \rangle\!\rangle \mid fork \, \langle\!\langle TAct \times TAct \rangle\!\rangle \mid \\ &\qquad decide \, \langle\!\langle Bool \times Bool \times TAct \times TAct \rangle\!\rangle \mid use \, \langle\!\langle Label \rangle\!\rangle \\ Graph \; &::= \; start \, \langle\!\langle TAct \rangle\!\rangle \end{aligned}$$

We may now give a behavioural semantics to the language, defining a semantic function $\mathcal{S}$ by structural recursion:

$$\begin{aligned} \mathcal{S}(atom\, p) &= p \\ \mathcal{S}(then\,(a,b)) &= \mathcal{S}(a) \fatsemi \mathcal{S}(b) \\ \mathcal{S}(parallel\,(a,b)) &= \mathcal{S}(a) \,|||\, \mathcal{S}(b) \\ \mathcal{S}(declare\,(x,a)) &= \text{let } x = a \text{ within } a \\ \mathcal{S}(include\, g) &= \mathcal{S}(g) \\ \mathcal{S}(stop\, a) &= \mathcal{S}(a) \\ \mathcal{S}(fork\,(a,b)) &= \mathcal{S}(a) \,|||\, \mathcal{S}(b) \\ \mathcal{S}(decide\,(p,q,a,b)) &= (p \,\&\, a) \,\Box\, (q \,\&\, b) \\ \mathcal{S}(use\, x) &= \mathcal{S}(x) \\ \mathcal{S}(start\, a) &= \mathcal{S}(a) \end{aligned}$$

To see the effect of this function consider the activity graph presented in Figure 4.

This graphical representation corresponds to the following syntactical definition:

$$\begin{aligned} & start(then(A, \\ & \qquad declare(X, \\ & \qquad\qquad then(B, \\ & \qquad\qquad\qquad then(parallel(C,D), \\ & \qquad\qquad\qquad\qquad decide(p,q,use(X),stop(E))))))). \end{aligned}$$

Expressing the processes corresponding to the individual action states in terms of the inverse mapping $atom^{\sim}$, and applying our semantic function

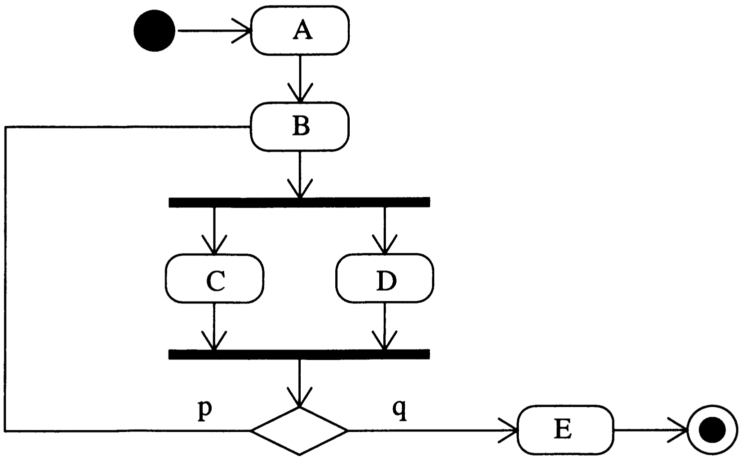

Figure 4 An activity graph

we see that the following process expression describes the behavioural semantics of the activity graph.

$$
\begin{array}{l}
atom^{\sim}A \fatsemi \\
\quad \text{let} \\
\quad\quad X = atom^{\sim}B \fatsemi (atom^{\sim}C \;|||\; atom^{\sim}D) \fatsemi \\
\quad\quad\quad\quad ((p \;\&\; X) \;\square\; (q \;\&\; atom^{\sim}E)) \\
\quad \text{within} \\
\quad\quad X
\end{array}
$$

3.3. ANALYSIS

In attempting to analyse a UML specification, we must take account of the context in which each diagram is presented. For example, it may be that a particular activity graph was never intended to convey *availability* information for the methods that appear. The choice of the two behavioural models allows us to treat the information obtained from a diagram in two different ways; having derived a collection of process descriptions, we may compare them using the failures model if availability information is present, or the traces model if the availability information is incomplete.

In this section we give examples of three simple UML diagrams and suppose that they had been constructed during the development process of *Example*, as described in Section 3.1. We explain how these fragments of the specification might be compared against the data type and process corresponding to the final class description.

First consider the simple activity graph shown in Figure 5.

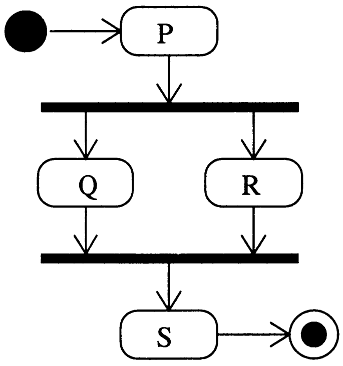

Figure 5 Activity graph for *Example*

The processes corresponding to the given action states may be expressed in terms of events corresponding to the given methods on the class. Suppose, for instance, that action states corresponded to the following processes

$$\begin{aligned} atom^{\sim}P &= methodB \rightarrow methodA \rightarrow Skip \\ atom^{\sim}Q &= methodB \rightarrow methodB \rightarrow Skip \\ atom^{\sim}R &= methodA \rightarrow Skip \\ atom^{\sim}S &= methodB \rightarrow Skip \end{aligned}$$

then *Activity*, the process corresponding to this graph, determined by applying the semantic function $\mathcal{S}$ to the syntax of the graph, would be as follows.

$$\begin{aligned} Activity = {} & methodB \rightarrow methodA \rightarrow Skip \ ⨾ \\ & \quad (methodB \rightarrow methodB \rightarrow Skip \ ||| \ methodA \rightarrow Skip) \ ⨾ \\ & \quad methodB \rightarrow Skip \end{aligned}$$

This activity graph is intended only to illustrate a particular use case and so it would be inappropriate to infer availability information; we use the traces model to compare *Activity* with *ClassDesc*, the process representing the class description.

$$ClassDesc \sqsubseteq_{\mathcal{T}} (\text{init} \rightarrow Activity) ⨾ Stop$$

This refinement check tells us that every sequence of methods allowed by the activity graph is a possible behaviour of the class description. The use of *Stop* to end the activity process avoids the unwanted requirement that *ClassDesc* should be able to terminate.

As a further example, consider the sequence diagram shown in Figure 6. In terms of possible behaviours, the information content of this

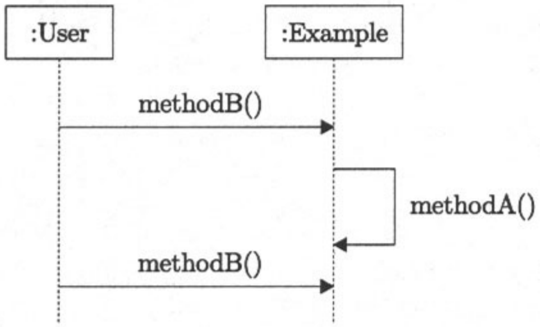

Figure 6 Sequence diagram for *Example*

diagram might be represented by the following process

$$Sequence = methodB \rightarrow methodA \rightarrow methodB \rightarrow Skip$$

This is a process that can engage in events *methodB*, *methodA*, and *methodB*, in that order, before terminating successfully.

The question of whether the sequence diagram is consistent with the class model and state diagrams of Section 3.1 could be phrased as a refinement check in the traces model:

$$ClassDesc \sqsubseteq_T (\text{init} \rightarrow Sequence) \fatsemi Stop.$$

For this to be true, $\langle methodB, methodA, methodB \rangle$ must be a trace of *ClassDesc*.

Alternatively, if the sequence diagram is intended to express the requirement that *methodA* should be possible after a single occurrence of *methodB*, then we might perform a refinement check in the failures model:

$$(\text{init} \rightarrow Sequence) \fatsemi Chaos \sqsubseteq_F ClassDesc.$$

For such a check, we compose the *Sequence* process not with *Stop*, but with *Chaos*, a process that can exhibit any behaviour on the current alphabet: no restriction should be placed upon *ClassDesc* once the specified requirement has been met.

As a final example, we consider a state diagram. Such a diagram may present only a part of the transition information for the methods that it describes. If this is the case, or if the availability of these methods depends upon state components that are not present, then we cannot infer availability information.

In the absence of this information, we may reason about the effects of methods by totalising the operations of a data type *before* considering its semantics, mapping any state outside the domain of definition to every possible after state. An equivalent effect could be obtained using a non-blocking semantics, provided that each method is guaranteed to terminate normally.

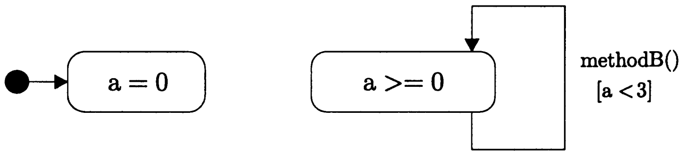

Figure 7 State diagram for *Example*

If the transition information is complete, and the state components considered are enough to characterise the availability of the methods, then we may reason using the failures model. As an example, consider the state diagram shown in Figure 7; this gives rise to the following data type:

$$\begin{array}{|l}\multicolumn{1}{l}{\textit{AbsState}}\\ \hline a : \mathbb{Z} \\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{\textit{AbsInit}}\\ \hline \textit{AbsState}' \\ \hline a' = 0 \\ \hline\end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{\textit{AbsOpB}}\\ \hline \Delta \textit{AbsState} \\ \hline a \geq 0 \wedge a < 3 \\ \hline\end{array}$$

and a process equivalent, defined by

$$\begin{array}{l} \textit{Abs} = \\ \quad \text{let} \\ \quad\quad \textit{Proc AbsState} = \\ \quad\quad\quad \text{pre } \textit{AbsOpB} \ \& \\ \quad\quad\quad\quad \text{methodB} \rightarrow \sqcap \textit{AbsOpB} \bullet \textit{Proc}(\theta \textit{AbsState}') \\ \quad\quad\quad \Box \\ \quad\quad\quad \sqcap \textit{AbsFinal} \bullet \text{final}.\theta \textit{Env} \rightarrow \textit{Skip} \\ \quad \text{within} \\ \quad\quad \sqcap \textit{AbsInit} \bullet \text{init} \rightarrow \textit{Proc}(\theta \textit{AbsState}') \end{array}$$

where the schema *AbsFinal* describes an embedding of the new state into the local environment.

As the two data types have different interfaces—different sets of named operations—we cannot compare them without first *hiding* the methods that are not mentioned in *Abs*: we may perform the refinement check

$$Abs \sqsubseteq_{\mathcal{F}} (ClassDesc \setminus methodA)$$

to confirm that this state diagram is consistent with the class description of Section 3.1.

In reasoning about state diagrams, it may be advantageous to reason entirely within the relational semantics. Here also, we hide any components that are not mentioned in the more abstract description: any operation that is not hidden is composed with the reflexive transitive closure of those that are.

To see how the refinement above could be established using the relational semantics, observe that the effect of hiding operation *OpA* in data type *Example* is described by the following data type

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Con}} \\
ADT[Local, Name, Env] \\
\hline
state = State \\
init = \{\, Env;\ Init \fatsemi OpA^{*} \bullet \theta Env \mapsto \theta State' \,\} \\
op = \{\, methodB \mapsto \{\, OpB \fatsemi OpA^{*} \bullet \theta State \mapsto \theta State' \,\} \,\} \\
final = Final \\
\hline
\end{array}$$

We may demonstrate a simulation based upon the obvious retrieve relation

$$Retrieve \mathrel{\widehat=} AbsState \land State$$

using the simulation rules presented in [4]. We have only to show that

$$\forall\, ConState';\ AbsState' \bullet ConInit \Rightarrow AbsInit \fatsemi Retrieve'$$

$$\begin{array}{l}
\forall\, \Delta AbsState;\ \Delta ConState \bullet \\
\quad (Retrieve \land ConOpB \Rightarrow AbsOpB \land Retrieve') \\
\quad \land \\
\quad (Retrieve \land \text{pre } AbsOpB \Rightarrow \text{pre } ConOpB)
\end{array}$$

If these two conditions hold, then we may conclude that the class description presented in Section 3.1 is faithful to the state diagram of Figure 7.

4. DISCUSSION

In this paper, we have shown how relational and behavioural semantic models might be used in the verification of object models. Using abstract data types and processes based upon class models and diagrams, we are able to check that two different parts of a specification are consistent, or to verify that a particular requirement has been satisfied.

The language of diagrams used is a strict subset of UML, and only simple forms have been considered here. In particular, the abstract syntax for activity diagrams excludes the possibility of swimlanes, cross-synchronisation, and joins; these require the definition of an additional category of activity—starting activities—and a mapping from synchronisation bars to internal synchronisation events.

We have said nothing about the translation of class models with multiple classes and associations. However, as [11] shows, formal descriptions of such models can be constructed by promoting the data types that model the individual classes. The process of reasoning about these descriptions can then be simplified: [22] shows that refinement distributes through promotion.

A considerable amount of work remains to be done, both in the application of relational and behavioural semantics and in the wider context of the formalisation of languages such as UML. A particularly promising area for research is the mechanical translation of diagrams and the automatic verification of static and behavioural properties; it is our hope that the work presented in this paper may be useful in that regard.

Acknowledgements

We would like to thank Geraint Jones, Perdita Stevens, and Jim Woodcock for their helpful and insightful comments.

References

[1] D. Amyot and L. Logrippo. Use case maps and lotos for the prototyping and validation of a mobile group call system. *Computer Communications*, 23(8), 2000.

[2] D. Amyot, L. Logrippo, R.J.A. Buhr, and T. Gray. Use case maps for the capture and validation of distributed systems requirements. In *Proceedings of RE '99*, 1999.

[3] C. Bolton. ioData types and processes. Technical Report PRG-TR-01-00, University of Oxford, 2000.

[4] C. Bolton, J. Davies, and J. Woodcock. On the refinement and simulation of data types and processes. In K. Araki, A. Galloway,

and K. Taguchi, editors, *Proceedings of IFM'99*. Springer, 1999.

[5] H. Bowman and J. Derrick. A junction between state-based and behavioural specification. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proceedings of FMOODS '99*. Kluwer, 1999.

[6] R. J. A. Buhr and R. S. Casselman. *Use case maps for object-oriented systems*. Prentice-Hall International, 1996.

[7] S. J. Creese and A. W. Roscoe. Verifying an independent family of inductions simultaneously using data independence and fdr. In *Proceedings of FORTE/PSTV '99*. Kluwer Academic Press, 1999.

[8] R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z*. Macmillan, 2000. To appear.

[9] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proceedings of FMOODS '97*, volume 2. Chapman and Hall, 1997.

[10] C. Fischer. How to combine Z with a process algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *Proceedings of ZUM '98*, volume 1493 of *LNCS*. Springer-Verlag, 1998.

[11] R. B. France, J.-M. Bruel, M. M. Larrondo-Petrie, and M. Shroff. Exploring the semantics of UML type structures with Z. In H. Bowman and J. Derrick, editors, *Proceedings of FMOODS '97*, volume 2. Chapman and Hall, 1997.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[13] C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 1987.

[14] ISO. *LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. IS 8807, ISO, Geneva, Switzerland, 1989.

[15] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In A. Fantechi P. Ciancarini and R. Gorrieri, editors, *Proceedings of FMOODS '99*. Kluwer, 1999.

[16] R. Lazic. *A semantic study of data independence with applications to model checking*. PhD thesis, University of Oxford, 1999.

[17] precise UML group. http://www.cs.york.ac.uk/puml/, 2000.

[18] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, 1998.

[19] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual*. Addison-Wesley, 1997.

[20] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In *Proceedings of ICFEM'97*. IEEE Computer Press, 1997.

[21] J. M. Spivey. *The Z notation: a reference manual.* Prentice-Hall International, 1992.

[22] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science, 1996.