On giving a behavioural semantics to activity graphs

Christie Bolton and Jim Davies

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

{christie,jdavies}@comlab.ox.ac.uk

We have been exploring how formal description techniques may be used to reason about UML specifications; how descriptions of system behaviour, expressed in terms of abstract data types and processes, may be constructed from individual UML diagrams. Given these formal interpretations—and the associated behavioural and relational semantics—we have shown how existing refinement techniques may be applied to support the further development and analysis of object models.

Our intention is not, however, to give a complete or formal semantics to the whole of UML. Indeed, perhaps that is not an appropriate thing to do. Before we embark on the formalisation process it is important to consider the role of the various notations within the language.

The use of UML in software specification and design can be divided into three, not necessarily distinct, phases:

1. Initial brainstorming period;
2. Development process;
3. Implementation-level specification.

Diagrams constructed in the first phase—for instance CRC cards and use case diagrams—are a means of getting ideas flowing and opening up a dialogue between various members of the team. A formal semantics is less useful here.

The focus of our work has been on diagrams constructed during the development process and in particular those, such as activity diagrams and sequence diagrams, describing the *behaviour* of the system. We are concerned not only with giving these notations a formal semantics but with how that semantics may be used to support analysis and verify consistency.

We begin by outlining the work presented in [BD00a]. We present a formal semantics for activity graphs, describing them in terms of the process language CSP [Hoa85]. We go on to explain how this semantic interpretation may be used to verify that the final class description of a system is consistent with the activity graphs constructed in its development process. We end by identifying possible extensions to the work.

Activity graphs

An activity graph is constructed from a combination of action states, (sub)activity states, start and finish states and pseudostates merge, decision, fork and join. We define:

$$\begin{aligned} \mathit{Type} \ ::= \ & \mathit{action} \langle\!\langle \mathit{Action} \rangle\!\rangle \mid \mathit{activity} \langle\!\langle \mathit{ActivityLabel} \rangle\!\rangle \\ \mid \ & \mathit{start} \mid \mathit{finish} \mid \mathit{merge} \mid \mathit{decision} \mid \mathit{fork} \mid \mathit{join} \end{aligned}$$

where *Action* and *ActivityLabel* are given types. Each *Action* corresponds to a method on the system and an *ActivityLabel* is the name of another graph within the specification. We define the function *Env* which maps each activity label onto its associated graph.

$$\mathit{Env} \ == \ \mathit{ActivityLabel} \nrightarrow \mathit{Graph}$$

Each state within an activity graph records the type of its contents, the set of incoming lines it requires and the set of outgoing transitions it enables

$$\begin{array}{|l} \mathit{State} \\ \hline \mathit{lines} : \mathbb{P}\, \mathit{Line} \\ \mathit{transitions} : \mathbb{P}\, \mathit{Transition} \\ \mathit{type} : \mathit{Type} \\ \hline \end{array}$$

where a transition records the name of the line as well as any associated guard or action.

$$\begin{array}{|l} \mathit{Transition} \\ \hline \mathit{guard} : \mathit{Guard} \\ \mathit{action} : \mathit{Action} \\ \mathit{line} : \mathit{Line} \\ \hline \end{array}$$

We impose well-formedness conditions on our states; for instance a start state can have no incoming lines and the only states which can have self-transitions are action states and (sub)activity states. A graph is built up from a finite set of well-formed states.

$$\mathit{Graph} \ ::= \ \mathit{states} \langle\!\langle \mathbb{F}\, \mathit{WellFormedState} \rangle\!\rangle$$

Semantic interpretation

The process corresponding to each *state* within an activity graph is essentially the sequential composition of the process corresponding to its incoming lines, the process corresponding to its type and the process corresponding to its outgoing transitions. The resulting process is then interleaved with itself to facilitate multiple (possibly synchronous) executions of the same state.[1]

$$\begin{array}{lll} \mathit{StateP}(\mathit{state}) \;\approx\; \mathit{LinesInP}(\mathit{state}) \fatsemi (& \mathit{TypeP}(\mathit{state}) \fatsemi \mathit{TransOutP}(\mathit{state}) \\ & \;||| \\ & \mathit{StateP}(\mathit{state})\,) \end{array}$$

The definitions of the processes corresponding to the incoming lines and outgoing transitions, *LinesInP* and *TransOutP* respectively, depend on the type of the state. For instance *join* states require all in-coming lines to be enabled whereas all other states (with the exception of the start state which has no incoming lines) require precisely one of their incoming lines to be enabled.

Similarly, the process corresponding to each type, *TypeP*, depends on the type of the state; for (sub)activity states it is the process corresponding to the associated graph, for action states it is the event corresponding to the given action followed by successful termination and for all other states it is simply successful termination.

The process corresponding to a *graph* is then the parallel composition of the processes corresponding to each of its constituent states each synchronising on its own alphabet, or set of associated events.

$$\mathit{GraphP} \;=\; \parallel\; \mathit{state} : \mathit{States} \;\bullet\; [\,\mathit{alphabet}(\mathit{state})\,]\;\; \mathit{StateProcess}(\mathit{state})$$

Our semantic function takes an activity label corresponding to a particular graph and the environment function, *Env*, mapping activity labels onto their associated graphs and returns the process which models the *behaviour* of the given graph. This is the process obtained by hiding all the line events in the process *GraphP* corresponding to the given graph.

Adequacy

We observed in Section 1 that in order to facilitate multiple, possibly synchronous, executions of each state we must interleave the process *StateP* with itself. However, the simplified definition we presented could diverge. To eliminate the possibility of divergence we must, in Petri net terms, put an upper limit on the number of tokens allowed in any place; that is, for each action state and activity state we must only allow a finite number of events corresponding to an incoming line to occur before an event corresponding to an outgoing transition occurs.

We extend the definition of *State* to incorporate the variables *isDynamic* and *dynamicMultiplity* as defined in the UML specification [OMG99]. The Boolean *isDynamic* is *True* for action states and activity states in which the state's actions may be executed concurrently. If this Boolean is *True* then the integer

[1] This definition is a simplification for the brevity and clarity of the paper. We take care to handle self-transitions separately; for more details see [BD00a]. To prevent the process from diverging, we put bounds on the number of synchronous executions of any given state; we discuss this further in Section 1.

dynamicMultiplicity limits the number of parallel executions of the actions of the state. If the Boolean *isDynamic* is *False* then each execution of the state may occur only if the last execution has been completed. Hence *for practical purposes* we use the following process to model the behaviour of each state[2]

```
$PracticalStateP(state) \approx$
    if ( $isDynamic = False$ ) then
        let
            $PSP(state) = LinesInP(state) \fatsemi TypeP(state) \fatsemi$
                                $TransOutP(state) \fatsemi PSP(state)$
        within
            $PSP(state)$
    else
        $StateP(state)$
        $[\![ LinesIn(state) \cup LinesOut(state) ]\!]$
        $ConstraintP(state)$
```

where the constraining process *ConstraintP* enforces the invariant:

$$0 \leq \#InLineEvents - \#OutLineEvents \leq dynamicMultiplicity$$

Analysis

In order to compare the final class description of a model with the activity graphs constructed during the development process we first translate the final class description to its corresponding process. This translation is based on results presented in [FBLPS97] and a translation from abstract data types to processes described in [BDW99] and is explained in greater detail in [BD00b].

Having translated all activity graphs—and indeed message sequence charts—and the final class description to CSP processes, it is simply a question of refinement to confirm that the final object model is consistent with its specification. We use the model-checker FDR to perform this verification.

But the nature of the refinement check we do is not always immediately obvious. For instance, consider the message sequence chart illustrated in Figure 1. Does this diagram mean that whenever *methodA* is called, *methodB* must be called and then *methodC* must be called? Or does it mean that whenever *methodA* is called *methodC* cannot be called before *methodB* has been called? Or does it simply mean that *methodA* followed by *methodB* followed by *methodC* is a possible sequence of interaction?

At present the information content of each diagram is conveyed (possibly tacitly) by its context within the specification. But this information content dictates the nature of the refinement check we make: do we use the traces model which simply checks that the behaviour described in the diagram is a possible sequence of interaction or do we perform our refinement check using the failures

[2] Once more this is a slightly simplified version for brevity and clarity. For full details see [BD00a].

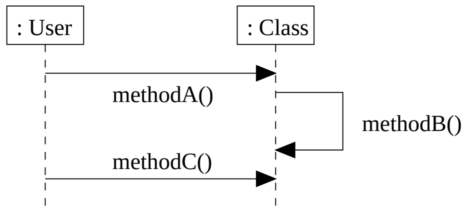

Fig. 1. A sequence diagram

model thereby checking availability information. Before we are in a position to automate the process of verifying the consistency of object models we must settle the issue of information the content of each of the diagrams.

Extensions

The semantic interpretation of activity graphs presented above contains the machinery for actions having guards, but at present these are non-variable guards. We need to incorporate object flow states into our semantics and consider how best to model systems with multiple threads in which more than one object is able to change state variables.

It is necessary to model access to data components in activity graphs; in an implementation objects may be locked, methods may be synchronised. It is relatively easy to extend the CSP model to treat this. The issue that requires clarification is the matter of event queues and event handling. As [BCR00] say, this is an area of the UML specification that remains somewhat opaque.

One of the strengths of UML—why we hope it should lead to a general improvement in the quality of program design—is that it is increasingly being adopted by software developers with no background in formal methods, those who might otherwise skip the specification process and move directly to the coding phase.

It is therefore crucial in giving a formal semantics to various parts of UML that we consider the needs of these users. To expect them to have to learn to use more traditional formal methods in order to verify the consistency of their UML specification is defeating the object of the exercise.

Our long term objective would be for a modelling tool to automatically:

– Translate the final class description to its corresponding process;
– Translate all behavioural descriptions to their corresponding processes;
– Use a model checker to verify consistency;
– Give the user intelligible reports about where/if the specification is violated.

We hope that the suggested behavioural semantics for activity graphs presented here is a step in the right direction.

References

[BCR00] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T. Rus, editor, *Proceedings of AMAST'00*, LNCS. Springer, 2000.

[BD00a] C. Bolton and J. Davies. Activity graphs and process. In W. Grieskamp, T. Santen, and W. Stoddart, editors, *Proceedings of IFM '00*, LNCS. Springer, 2000. To appear.

[BD00b] C. Bolton and J. Davies. Using relational and behavioural semantics in the verification of object models. In C. Talcott and S. Smith, editors, *Proceedings of FMOODS '00*. Kluwer Academic Publishers, 2000. To appear.

[BDW99] 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.

[FBLPS97] 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.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[OMG99] Object Management Group. *Unified Modeling Language Specification, version 1.3*. Rational Software Corporation, Santa Clara, CA 95051, USA, June 1999.