Modelling recursive calls with UML state diagrams

Jennifer Tenzer* and Perdita Stevens**

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

Abstract. One of the principal uses of UML is the modelling of synchronous object-oriented software systems, in which the behaviour of each of several classes is modelled using a state diagram. UML permits a transition of the state diagram to show both the event which causes the transition (typically, the fact that the object receives a message) and the object's reaction (typically, the fact that the object sends a message). UML's semantics for state diagrams is "run to completion". We show that this can lead to anomalous behaviour, and in particular that it is not possible to model recursive calls, in which an object receives a second message whilst still in the process of reacting to the first. Drawing on both ongoing work by the UML2.0 submitters and recent theoretical work [1, 6], we propose a solution to this problem using state diagrams in two complementary ways.

1 Introduction

The Unified Modelling Language [10] has been widely adopted as a standard language for modelling the design of (software) systems. One diagram type within UML is the state diagram, an object-oriented adaptation of Harel statecharts.

The use to which state diagrams are put varies with the type of project and the modeller's preferences, but a typical use is as follows. The modeller decides that some or all of the classes which are to appear in the system should be modelled with state diagrams; typically, classes which are perceived to have "interesting" state change behaviour will be so modelled, whereas those which are considered to be stateless or almost so will not be. For a given class, the modeller identifies the abstract states, which will be represented as *states* in the state diagram. This involves deciding which aspects of state are interesting, in that they may affect behaviour; it can be seen as choosing an equivalence relation on the set of (concrete, fully-detailed) possible states of objects of the class. Next, s/he considers which *events* may happen to an object of this class, and what effect those events have on the abstract state of the object.

* Email: J.N.Tenzer@sms.ed.ac.uk Fax: +44 131 667 7209
** Email: Perdita.Stevens@ed.ac.uk Fax: +44 131 667 7209

In sequential single-threaded systems on which we concentrate in this paper a typical event is the receipt of a message which requests the synchronous invocation of an operation. The modeller may record that certain state transitions happen only in particular circumstances; that is, s/he may add *guards* to the transitions. Finally, s/he may record how an object reacts to a given event happening in a given state by showing *actions* on the transitions (and/or within the states, but for simplicity we omit that possibility here). Typically, the modeller will not record every detail of the object's reaction, since that might involve placing the whole of an eventual method implementation as annotation on a transition in a diagram. A common compromise is to show only the messages which the object may *send* as part of its reaction to an event such as receiving a message, but not to show internal computation such as variable assignments.

Details vary, but essentially this process is recommended in many reputable sources, including for example Booch et al.'s *UML User Guide* [3] and the second author's own *Using UML*[14].

Unfortunately, this standard way of using UML is incoherent, given the semantics for UML state diagrams laid down in the UML standard. That is, following the UML semantics may yield nonsense for an apparently sensible collection of state diagrams. These are not pathological cases, either: there are common situations which cannot be modelled, with their intended semantics, using UML state diagrams as described above, and there are simple state diagrams, correct according to the UML standard, which cannot be interpreted. A very simple example is shown in Figure 1. Suppose an object represented by this diagram is in state $S1$ and receives message $f()$, which causes the object to send itself the message $g()$.[1] What should the resulting state of the object be?

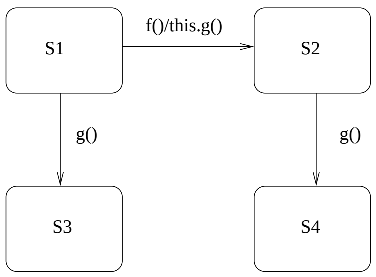

Fig. 1. Simple problem situation

[1] More precisely, $f()$ is a *call event* and $g()$ a *call action* in the transition from $S1$ to $S2$. Call events are caused by call actions and are distinct from them [10] (3-148). In our example $g()$ on the transitions from $S1$ to $S3$ and $S2$ to $S4$ is a call event which is caused by the corresponding call action. Both call events and call actions are associated with an operation in UML, i.e. call events and actions in a state diagram model invocations of the operation of the object.

The UML semantics does not adequately cover cases like this. Probably the designer intended $S2$, with the idea that the implementation of $f()$ would involve extra actions besides the invocation of $g()$, but if we must consider the diagram as a complete machine, there is no good answer.

The fundamental problem seems to be that UML – like the UML community – is ambivalent about whether its state diagrams are intended to be *machines*, capable of being executed, or loose specifications, constraining a later implementation.[2] The UML semantics strongly suggests the former, but this does not always accord with how UML is used. In particular, when state diagrams are used to model synchronous message passing between objects in a sequential single-threaded system UML's run-to-completion semantics causes anomalies. Situations as shown above can be modelled by UML sequence diagrams and implemented in an object oriented programming language although the execution of corresponding UML state machines would result in a deadlock according to the default UML run-to-completion semantics (see example in section 2.1). It is interesting to notice that Harel and Gery [7] were aware that recursive operation calls are problematic but apparently considered them unimportant. They wrote:

> ...when the client's statechart invokes another object's operation, its execution freezes in midtransition, and the thread of control is passed to the called object. Clearly, this might continue, with the called object calling others, and so on. However, a cycle of invocations leading back to the same object instance is illegal, and an attempt to execute it will abort.

We do not consider that the problem can be so easily dismissed. In object oriented design recursive calls occur frequently: for example, whenever any method is recursive, or when the Visitor or Observer pattern is used. In this paper we discuss the problem and propose a solution, drawing on both ongoing work by UML2.0 submitters and recent theoretical work [1, 6].

The paper is structured as follows. The remainder of this section includes a note on standards and terminology. In Section 2 we explain the problem with the UML's current understanding of state diagrams. In Section 3 we introduce our solution, making use of two kinds of state diagrams. Section 4 formalises these diagrams and defines a suitable notion of consistency. In Section 5 we revisit the example introduced above; Section 6 discusses related work, and Section 7 concludes.

1.1 Note on standards and terminology

This paper is based on UML1.4, the current standard at the time of writing. It also draws on drafts of the new UML2.0 standard. The reader is assumed to be

[2] Even the terminology in the standard shows this tension: the terms state diagram, statechart diagram, statechart and state machine are not always used consistently (compare for example [10] (3-137) and (3-141)). In this paper we use "state diagram" as a general term, reserving "state machine" for an executable version.

familiar with UML; it is important to note that the definition of UML is the OMG standard [10], not what is contained in any UML book.

In sequence diagrams, we show expressions on return arrows to indicate the value returned – this is common and harmless, but not actually described in [10].

We make several simplifying assumptions. We only consider sequential systems, so our state diagrams are assumed not to make use of concurrent substates. We treat rolenames identically with attributes; for example, our attribute environments include rolenames. Such an attribute has a class type, and its value will be an object identifier.

2 State Diagrams in UML

According to run-to-completion semantics, the action on a transition must have been completed before the transition is finished. If the action involves an object o of the class modelled by the state diagram making a call to another object p (and perhaps using the result of that call in some calculation), the action, and therefore the transition, does not complete until the call has been received by p, processed, and the result returned to o.

However, as soon as we consider the case that o and p might be the same object (recursion) or that part of p's reaction to the message from o might be to send o a new message (callback), it becomes clear that we cannot model situations with recursion or callbacks with UML state diagrams in which call events and actions involving calls are recorded on transitions. The next subsection demonstrates this using an example which we will use throughout the paper.

2.1 An example of callbacks with UML

The following example is used in different variants to illustrate several problems with callbacks in UML. The class diagram is given in the left part of figure 2.

Consider now an interaction between two objects as shown in the sequence diagram in the right part of figure 2. As response to an invocation of f object a calls method g of b and during the execution of g object b performs a callback to a. When the message setA2 arrives at a the execution of f is still in progress.

An interaction like that can be implemented in Java without problems. One possible implementation is shown in figure 3.

The internal behaviour of objects of classes A and B can be modelled by state machines. In this example the state of an object depends on the signs of its attribute values. An object of A has four different states, an object of B two. The state machines in figure 4 show how methods affect the object states. An object of A is in one of the states in the top part of figure 4 if its value of a1 is negative, in one of the states at the bottom otherwise. Similarly the object is in one of the states on the left side of the diagram if a2 is negative, and on the right side if it is positive. Notice that these state machines intuitively correspond to the code in figure 3 under the assumption that internal computations are omitted

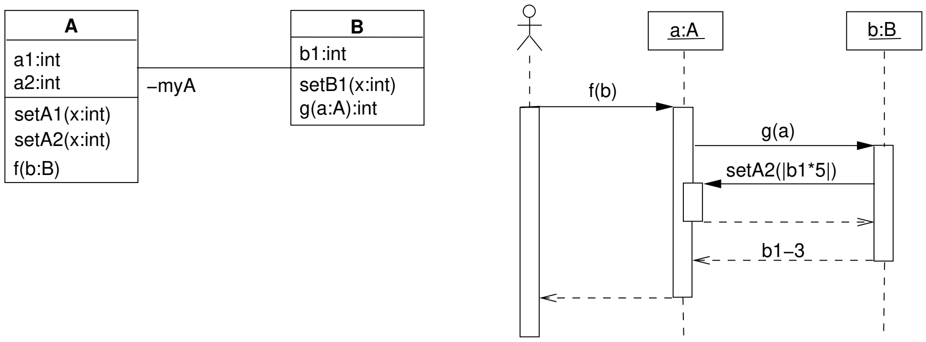

Fig. 2. Class diagram (left) and sequence diagram (right)

```
public void f(B b) {
    int y;
    y=b.g(this);
    if(a1*y>=0) {
        a1=y;
    }
}
```

```
public int g(A a) {
    myA=a;
    myA.setA2(Math.abs(b1*5));
    return(b1-3);
}
```

Fig. 3. Java implementation of methods f and g

in actions: the value of a2 is always positive after the completion of g due to the usage of the absolute value, and the sign of a1 is not changed in f, since the assignment of y to a1 is only carried out if y and a1 have the same sign.

For readability, an arrow may represent more than one transition with the same source and target states. The different transition labels are separated by commas. For example there are two transitions attached to the arrow from S1 to S2.

According to the UML semantics these state machines, considered together, do not behave in a sensible way. When a call of f arrives at an object a of A which is in state S1, the transition from S1 to S2 labelled by f is triggered, which invokes g. This causes a transition from S5 to itself and leads to a callback of setA2 to a, but a cannot react to this call because it is not in a stable state. The UML run-to-completion semantics prescribes that a can only process the call of setA2 after the transition from S1 to S2 has been completed, which will never happen.

Note that the same problem arises even if the callback is a query method, i.e. does not change any state.

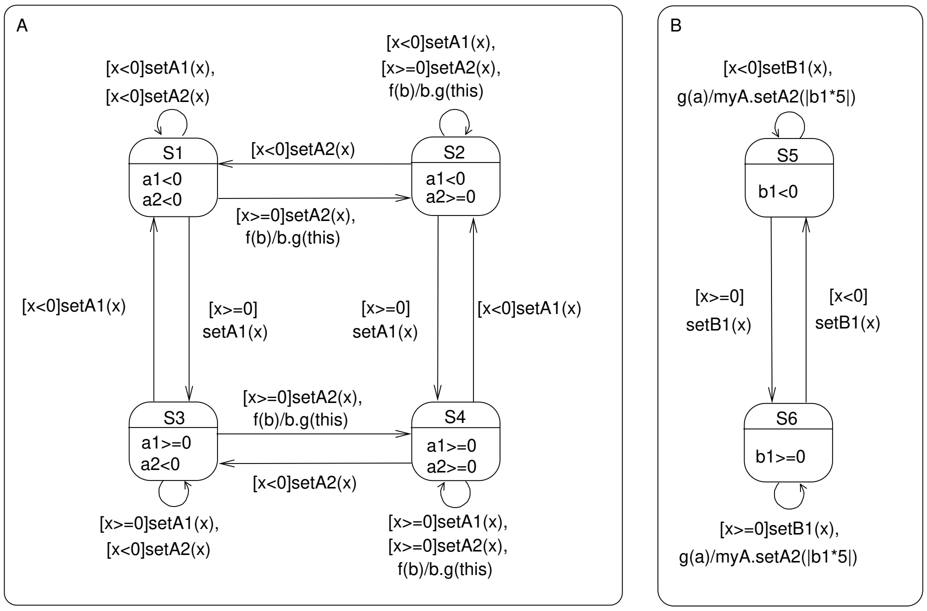

Fig. 4. State machines for A and B

3 Two kinds of state diagrams

We suggest handling the problem of callbacks by using two different kinds of state diagrams, one to model the overall effect of a method on the state of an object and the other to model the execution of actions of which this method consists. Thus we resolve the issue of whether state diagrams are loose specifications or executable machines by providing both, for use in clearly defined different contexts.

3.1 Protocol state machines (PSMs)

In UML1.4 [10] (2-170) PSMs are introduced as a state diagram variant, defined in the context of a classifier. We keep UML's terminology (PSM), but in our opinion these diagrams are best thought of as loose specifications, not as executable machines. They specify the permissible sequences of method calls on an object (the protocol), but not how the object will react to each method call. Their transitions (protocol transitions) are *allowed*, but not expected, to have action expressions. Here we only consider PSMs for classes and in order to enforce the separation between the diagram types, we follow a recent proposal for UML2.0 [11] where actions at protocol transitions are explicitly forbidden. The definition given below is a formalisation of a simplification of PSMs as presented in [11]. As notation for PSMs we use the standard UML state diagram notation as described in [10] (3.74.2). Thus a PSM is a simple form of UML state diagram, without actions: e.g., removing all actions from Figure 4 yields PSMs. The designer would develop a PSM for classes that s/he considers to have interesting state change behaviour, as in current practice. The only difference is that instead of recording actions on the transitions, s/he will choose when it is worthwhile to record how reactions to events are implemented using another diagram type, as follows.

3.2 Method state machines (MSMs)

Both the current UML specification [10] and the proposal [11] allow the definition of a state diagram in the context of an operation, but do not provide detail about the particular features and behaviour of this kind of state diagram. We propose MSMs which are a simplified variant of *sequential class machines* as presented in [6], which in turn are a variant of *recursive state machines* as introduced in [1]; they allow recursion.

Figure 5 shows the three MSMs for f, setA2 and g which correspond to the Java implementation given in figure 3. Each MSM is represented by a box with rounded edges and is labelled by the class which owns the method, the method name and its parameter. The MSM for f consists of an *entry state* Fe, an *invocation box*, two *internal states* F1 and F2, and two *return states* Fr1 and Fr2. Most of the notation is standard UML. Entry states contain variable declarations and in the state diagram for g a return expression is shown within the return state Gr. Invocation boxes are shown as boxes with a double borderline and include a method call. Each box has an entry and an exit point, represented as shown. In the MSM for f a *return variable* is attached to the exit point of the invocation box. States and boxes are connected by transitions which are labelled by guards and actions, but not by events. In terms of the UML metamodel two kinds of events are relevant in MSMs: implicit completion events which cause normal transitions (as commonly used in UML activity diagrams), and return events, which permit the MSM to move on from a method invocation box. Since an MSM models how an activity is performed, it is bound to have similarities with an activity diagram. We add a precise semantics for MSMs in section 4, especially, semantics for invocation of methods represented by other diagrams, which is not defined in UML activity diagrams. In this paper we will not discuss how our proposal sits inside the UML metamodel, since it would raise no interesting issues: instead, we focus on description and formalisation.

4 Formal definitions and consistency

Clearly, the designer will need to satisfy him/herself that the state diagrams, both PSMs and MSMs, contained in a given model are consistent: that is, that it is possible to implement methods according to the MSMs and have the resulting classes act in accordance with the PSMs. In this section we specify what this consistency means.

First we consider some informal examples:

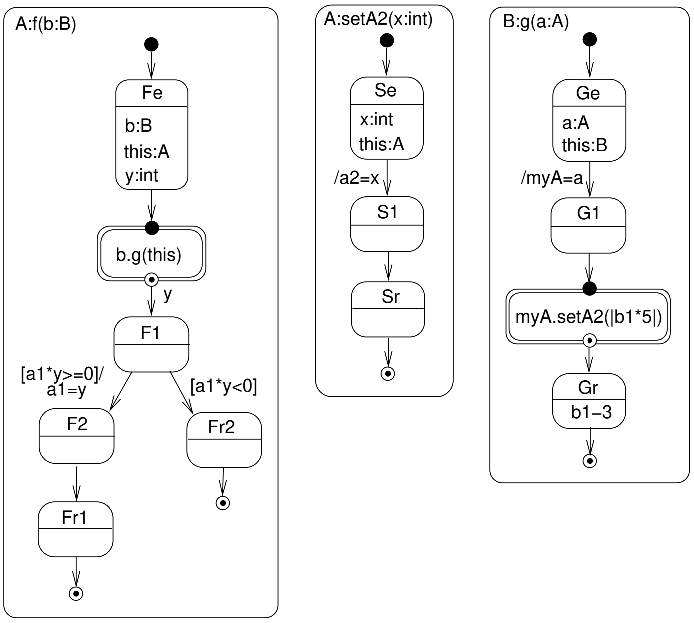

Fig. 5. Method state machines for f, setA2 and g

- The MSMs in Figure 5 are intuitively consistent with the PSMs obtained by deleting actions from Figure 4; the transitions in the PSMs accurately reflect the state changes that occur when the MSMs are followed through in a natural way.
- However, if we change the invocation in the MSM for g from myA.setA2(|b1*5|) to myA.setA2(b1*5), we destroy consistency because if b1 is negative then a2 is set to a negative value. That means a call of f on a does not always lead to an object configuration of a where a2 is positive, contradicting the PSM.
- Alternatively, if the MSM for f is altered so that a1 is always set to a1 * y (i.e. remove the guard on the transition from F1 to F2 and delete Fr2) we get a slightly more complicated inconsistency example, involving the states of two objects. If for instance a1 is positive and b1 is negative, then an invocation of f on a results in a configuration where a1 is negative. This is again a contradiction to the PSM for A which specifies that the sign of a1 is not supposed to change during the execution of f.

In order to formalise these intuitions we introduce formal definitions of both kinds of state diagrams. For purposes of exposition we use simplified forms of the diagrams; we believe, however, that most of the missing features of UML state diagrams could be added without serious problems. Our definitions of MSMs and their execution are adapted from definitions in [6] and [1].

We assume that there is a class diagram which defines a set of classes C. Each $c \in C$ is associated with a finite set $A_c = \{a_1 : T_1, \ldots, a_n : T_n\}$ of typed attributes and a finite set M_c of methods, where each method $m \in M_c$ has a type $T_{cm} = cT_{cm} \times rT_{cm}$ defining the call and return type of the method[3].

Note that in the current work we are eliding the difference between the *operations* of classes and their *methods*. In UML these concepts are distinguished in order to allow for inheritance: classes in an inheritance hierarchy may all have the same operation, inherited from a base class, which they implement using different methods. We do not consider inheritance here, since this raises many interesting questions about the appropriate inheritance of behaviour, and so the distinction between operation and method is unimportant. One possibility to explore would be that the protocol state machine would be written in terms of operations, and that the PSM defined for a base class would be inherited by subclasses. Then when a subclass provided its own method implementing an inherited operation, the designer could draw a new MSM for that method. This would open the door to considerations of behavioural subtyping: we could ask to what extent the MSMs for different methods implementing an operation were compatible.

A PSM is unsurprisingly simply a labelled transition system with guards:

Definition 1. *A protocol state machine (PSM) for a class c consists of*

- *a set S_c of states*
- *a labelled transition relation $\gamma \subseteq S_c \times L \times S_c$ where each label $l \in L$ is a tuple $(g, m(x))$ where $m \in M_c$ is a method name, x is a formal parameter of type cT_{cm}, and g is a Boolean expression over $A_c \cup \{x\}$ specifying the condition under which the transition may be taken. We do not prescribe the expression language used for g but we assume it can be evaluated to true or false given values for $A_c \cup \{x\}$.*

We will now define MSMs more formally. For a set $X = \{x_1 : T_1, \ldots, x_n : T_n\}$ of typed variables, a *variable environment* σ over X is a function $[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$ where $a_i \in T_i \cup \bot_{T_i}$ for all i. The set of all variable environments over X is denoted by Σ_X. Attributes and attribute environments are treated in a similar way.

Moreover let $A = \bigcup_{c \in C} A_c$ be the set of all attributes and O the set of all object identifiers. An *object environment* is defined as a partial function $\omega : C \to (O \to \Sigma_A)$ and the set of all object environments is denoted by Ω.

We do not prescribe an action language: we only specify that, given an object environment ω and variable environment σ over X, an action is syntactically an expression over X, suitably extended with attribute selectors, for which an evaluation function $[\![\ ]\!]_{\sigma\omega}$ exists. We will later assume that the same evaluation function can be used to evaluate the guards used in PSMs. An action may not involve the invocation of methods or the creation or deletion of objects. Semantically an action α is a partial function $\alpha : (\Sigma_X \times \Omega) \to (\Sigma_X \times \Omega)$

[3] For simplicity we allow only one parameter and return value

expressing the effect the action has on the variable environment and object environment.

Definition 2. *A method state machine (MSM) for a method* $m \in M_c$ *consists of*

- *a set of local variables* $X_{cm} = \{x_1 : T_1, \ldots, x_n : T_n\}$*, including those mentioned below*
- *a set* B_{cm} *of invocation boxes as defined below*
- *a set of states* S_{cm} *partitioned into*
  - *a set* I_{cm} *of internal states*
  - *an entry state* e_{cm} *with formal parameters* $x : cT_{cm}$ *and* this : c *in* X_{cm}
  - *A set of return states* R_{cm} *where each state* $r \in R_{cm}$ *has attached a return expression* re *over* X_{cm} *of type* rT_{cm}
  - *a set of box entry points* $Entry_{cm}$ *and a set of box exit points* $Exit_{cm}$
- *a transition relation* $\delta_{cm} \subseteq F \times Act \times T$ *where* $F = \{e_{cm}\} \cup I_{cm} \cup Exit_{cm}$*,* $T = R_{cm} \cup I_{cm} \cup Entry_{cm}$ *and* Act *is a set of actions* $\alpha : (\Sigma_{X_{cm}} \times \Omega) \rightarrow (\Sigma_{X_{cm}} \times \Omega)$

Notice the "incompleteness" of the transition relation of an individual MSM: if the MSM reaches a box entry point, it cannot go further based on the definition of this MSM alone. This makes sense because we cannot know what the effect of the call on the environments should be. Later we will show how several MSMs interact to "complete" the transition relation.

Definition 3. *A method invocation box* $b \in B_{cm}$ *specifies*

- *an object expression* oe*, determining the target object*
- *a class identifier* $d \in C$ *determining the class*[4] *of* oe
- *a method identifier* $m \in M_d$
- *an argument expression* $ae : cT_{dm}$

A box is not itself considered to be a state in the MSM: instead it has two associated states:

- *an entry point* $c_b \in Entry_{cm}$.
- *an exit point* $r_b \in Exit_{cm}$

A return variable $y : rT_{dm}$ *from* X_{cm} *is defined to hold the value returned from the method invocation.*

Notice that any MSM is by definition well-typed, and that all method invocations occur in boxes.

---

[4] As mentioned, we do not consider inheritance in this work, so polymorphism is not allowed: the class of the target object must be given statically.

4.1 Execution of MSMs

Suppose that we have a *closed set* $\mathcal{MSM}$ of MSMs: that is, each method invoked in an invocation box of an MSM in $\mathcal{MSM}$ is itself defined by an MSM in $\mathcal{MSM}$. We can then define the execution of MSMs in terms of a global state machine.

Let the sets of states and of boxes for each MSM be pairwise disjoint and let N be the set of all states, X the set of all variables, and B the set of all boxes. Before we specify a global state machine, we give definitions of a call stack and a global environment.

Definition 4 (Call stack). *A call stack $cs \in B^*N$ specifies the current position in each active MSM. It is a stack $b_1 : \ldots : b_k : n$ of boxes b_i and a state n on top. It must satisfy a coherence condition as follows. Suppose that box b_j contains method identifier m_{b_j} and class identifier c_{b_j}. Then box b_{j+1} if $j < k$ (respectively the state n if $j = k$), must belong to the MSM for method m_{b_j} in class c_{b_j} (which must exist, by the assumption that we have a closed set of MSMs).*

Definition 5 (Global environment). *Given a call stack $cs = b_1 : \ldots : b_k : n$, a global environment $ge = \sigma_0 : \ldots : \sigma_k \in \Sigma_X^*$ associated with cs is a stack of variable environments. It must satisfy a coherence condition as follows. For each $j \leq k$, σ_j is the local variable environment of the MSM containing box b_{j+1} if $j < k$, or of the MSM containing n otherwise.*

Definition 6 (Global state machine).

A state of a global state machine (GSM) consists of a call stack $cs \in B^*N$, a global environment ge associated with cs, and an object environment ω.

There are three kinds of transitions: as in a pushdown system, the applicable transitions are always determined by the state at the head of the stack. Suppose $cs = b_1 : \ldots : b_k : n$ where n is a state in MSM, and let the global environment be $ge = \sigma_0 : \ldots : \sigma_k$ and the object environment be ω.

1. *If n is an entry state, an internal state or a box exit state, the only possible transitions are* internal transitions *which are induced by transitions of MSM. Formally, suppose that $n \xrightarrow{\alpha} n'$ is a transition in MSM. Then $(b_1 : \ldots : b_k : n, \sigma_0 : \ldots : \sigma_k, \omega) \rightarrow (b_1 : \ldots : b_k : n', \sigma_0 : \ldots : \sigma'_k, \omega')$ is a transition in the global state machine, provided that $\alpha(\sigma_k, \omega) = (\sigma'_k, \omega')$. (Note that if (σ_k, ω) is not in the domain of the partial function α, there is no transition.)*
2. *If n is a box entry state, the only possible transition is a* call transition, *pushing a new invocation onto the stack. If $n = c_{b_{k+1}}$, the entry state for a box which we now call b_{k+1}, let the object expression, class, method and argument expression specified in b_{k+1} be oe, c, m and ae respectively. Then let σ_{k+1} be a new variable environment over X_{cm} in which the formal parameter of m and* this *are bound to $[\![ae]\!]_{\sigma_k\omega}$, $[\![oe]\!]_{\sigma_k\omega}$ respectively. (If either evaluation fails, there is no transition). Let e_{cm} be the entry state of the MSM for method m in class c. Then $(b_1 : \ldots : b_k : c_{b_{k+1}}, \sigma_0 : \ldots : \sigma_k, \omega) \rightarrow (b_1 : \ldots : b_k : b_{k+1} : e_{cm}, \sigma_0 : \ldots : \sigma_k : \sigma_{k+1}, \omega)$ is a transition of the global state machine.*

3. *If n is a return state, the only possible transition is a* return transition, *popping the stack. If $n = r \in R_{cm}$, then $(b_1 : \ldots : b_k : r, \sigma_0 : \ldots : \sigma_{k-1} : \sigma_k, \omega) \to (b_1 : \ldots : b_{k-1} : r_{b_{k-1}}, \sigma_0 : \ldots : \sigma'_{k-1}, \omega)$ is a transition of the global state machine, where $r_{b_{k-1}}$ is the exit state for box b_{k-1}, and σ'_{k-1} is the environment σ_{k-1} updated by binding the return variable of box b_{k-1} to $[\![re]\!]_{\sigma_k\omega}$ where re is the return expression associated with r (again, if re fails to evaluate, there is no transition).*

Note that the only non-determinacy in the global state machine is that arising from non-determinacy inside individual MSMs: if they are deterministic, so is the GSM. Notice also that the behaviour of the GSM respects the stack discipline. We will be most interested in how the GSM implements a particular method call. We write $s \overset{c,m}{\twoheadrightarrow} t$ when for some class c and method m, $s = (b_1 : \ldots : b_k : e_{cm}, \sigma_0 : \ldots : \sigma_k, \omega)$, $t = (b_1 : \ldots : b_k : r, \sigma_0 : \ldots : \sigma'_k, \omega')$ for some return state $r \in R_{cm}$ and there is some sequence of GSM transitions $s \to \ldots t$, in which no intermediate state whose call stack contains at most k boxes has e_{cm} at the head of the call stack. Without the restriction on intermediate states we might inadvertently "catch" more than one invocation of m from within MSMs that have been activated earlier. Notice that if m is recursive the call stack grows each time m is invoked so that e_{cm} is allowed to appear as head of the call stack in this case.

4.2 Consistency between protocol and method state machines

So far MSMs and PSMs are only connected by method names which are used to label transitions in PSMs and represent the context of a MSM. In this section we define what it means for a MSM to conform to a protocol which is specified by a PSM.

There can be no formal connection unless the designer has specified the precise meanings of the states in the PSM.[5] Accordingly, we assume that along with any PSM P for class c having states S_c we are given a function $h : \Sigma_{A_c} \to S_c$ which maps an attribute environment to a state in P.

Definition 7 (Consistency). *Let G be a GSM defined by a closed set $\mathcal{MSM}$ of MSMs, and let P be a PSM for class c. G conforms to P with respect to a given initial global state gs if and only if whenever*

$$gs \to^* gs' \overset{c,m}{\twoheadrightarrow} gs''$$

(that is, a method is executed from some global state which is reachable from the initial state), where $gs' = (cs, \sigma_0 : \ldots : \sigma, \omega)$ and $gs'' = (cs', \sigma_0 : \ldots : \sigma', \omega')$ we have $h(\omega(c)(\sigma(\mathsf{this}))) = s$ and $h(\omega'(c)(\sigma'(\mathsf{this}))) = t$ where $s \xrightarrow{[g]m(x)} t$ is a transition of P and $[\![g]\!]_{\sigma\omega} = \mathsf{true}$.

[5] This is sometimes done in practice by adding constraints to the states of a state diagram.

Note that because the PSM plays no role in the execution of the global state machine, but acts as an independent specification of what it should achieve, it suffices to specify consistency with one PSM at a time. Note also that we are *not* requiring that every transition in the PSM has some counterpart in the GSM. This is deliberate: the PSM for a reusable class will specify all the capabilities of the class, not all of which may be used in a particular system (GSM).

An obvious question to ask is whether consistency is decidable. The answer depends on the choice of action language, but for any reasonable action language Turing Machines can be coded as MSMs, in which case it is easy to reduce the Halting Problem to a consistency problem, which must therefore be undecidable. Nevertheless, some tool support is possible. For example, a tool might construct representative object configurations for the different combinations of abstract states in the PSMs, symbolically execute the MSMs and check against the PSM transitions. Even if the tool's checking was not exhaustive, it might find useful counterexamples, helping the user to develop the design.

5 Introductory example revisited

Finally we revisit the example discussed in the introduction, Figure 1. If the designer modelled with PSMs and MSMs as introduced in this paper, the state diagram would be split into a PSM and MSMs for f and g. The PSM specifies unequivocally that an object in state S1 is in state S2 after f has been executed, whatever further invocations are performed during f.

Under the assumption that the set containing the MSMs for f and g is consistent with the PSM, an object calls g during the execution of the MSM for f when it is in an appropriate state, i.e. either in S1 or in S2. According to the specification of g in the PSM, the object is either in S3 or S4 after the execution of the MSM for g has finished, depending on which state it was in at the time of g's invocation. In either case the MSM for f has to perform further actions to guarantee that the object is in S2 after its completion, as specified in the PSM.

Variants of the notation might be considered. For example, we might permit annotation of transitions to show what callbacks were *expected* to happen. However, the designer of the class will not always be in a position to know what callbacks might happen, if other classes in the system are designed by other people. (This need not prevent the designer knowing what the state after the transition will be, given adequate contracts for called methods.)

6 Related work

We have used work by others for our definitions of PSMs and MSMs. PSMs are mentioned in UML1.4 [10] and specified in more detail in U2 Partners' proposal for UML2.0 [11]. Since this proposal contains some remarks on inheritance, operations and methods are differentiated there.

The formalism for MSMs presented in this paper is based on recursive state machines, which have been defined in [1], and sequential class machines as introduced in [6]. Recursive state machines are extensions of ordinary state machines where a state can correspond to a possibly recursive invocation of a state machine. They can be used for modelling sequential imperative programs with recursive procedure calls. Besides a definition [1] also contains a complexity analysis of recursive state machines concentrating on reachability and cycle detection. Similar results have been achieved independently in [2].

Class machines are an object oriented extension of recursive state machines. The semantic definition of their sequential variant in [6] covers exceptions, inheritance and object creation in addition to what we have presented as MSMs. Adding a mechanism for multi-threading leads to the definition of concurrent class machines. In contrast to our work class machines are considered in isolation, not in conjunction with a more abstract modelling technique.

There is much work on formalisation of UML state diagrams; we only present a small subset. All of the approaches differ from ours in that they do not consider the problem of recursive calls. In [12] a formalisation with labelled transition systems and algebraic specifications written in the specification language CASL is presented. Labelled transition systems are also suggested as formalism in [13], where a structured operational semantics for UML state diagrams is introduced. Both in [5] and in [8] graph transformations are used as basis for state diagram formalisation, but these works differ in detail; [4] uses ASMs. In [9] state diagrams are first mapped to extended hierarchical automata and then a semantics for these specific automata is defined in terms of Kripke structures.

7 Conclusions and further work

We have pointed out that the current UML semantics for state diagrams is not sensible for situations involving recursive method calls. After showing this problem on an example we have presented an alternative approach for modelling the internal behaviour of objects using UML. In contrast to the current version of UML we differentiate between a loose specification of the effect of a method on an object and an executable machine representing an implementation of a method. We have introduced PSMs and MSMs for these purposes and defined what it means for a set of MSMs to be consistent with a PSM.

In future we would like to consider tool support; indeed we undertook this work because the recursive call problem prevented us from making progress with work on providing tool support for the concurrent development of state and sequence diagrams. For practical use, more complex MSMs would be needed, allowing for object creation for example. Most of that work would be routine; the exception would be adding inheritance, which as briefly mentioned in Section 4 would raise both theoretical questions and issues in practical modelling.

Acknowledgements We are grateful to the British Engineering and Physical Sciences Research Council for funding (GR/N13999/01, GR/A01756/01), and to the referees for helpful comments on presentation.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Computer Aided Verification*, pages 207–220, 2001.

[2] M. Benedikt, P. Godefroid, and T.W. Reps. Model checking of unrestricted hierarchical state machines. In *Automata, Languages and Programming*, pages 652–666, 2001.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman, 1998.

[4] E. Börger, A. Cavarra, and E. Riccobene. A precise semantics of UML state machines: making semantic variation points and ambiguities explicit. In *Proc. of Semantic Foundations of Engineering Design Languages (SFEDL), Satellite Workshop of ETAPS 2002*, 2002.

[5] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

[6] R. Grosu, Yanhon A. Liu, S.A. Smolka, S.D. Stoller, and J.Yan. Automated software engineering using concurrent class machines. In *Proc. of the 16th IEEE International Conference on Automated Software Engineering, ASE'01*. IEEE, 2001.

[7] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30:7:31–42, 1997.

[8] S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 241–256, 2001.

[9] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999*. Kluwer, 1999.

[10] OMG. *Unified Modeling Language Specification version 1.4*, September 2001. OMG document formal/01-09-67 available from http://www.omg.org/technology/documents/formal/uml.htm.

[11] U2 Partners. Unified Modeling Language 2.0 proposal, version 2 beta R1, September 2002.

[12] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated state machines — A lightweight formal approach. In *FASE 2000 - Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 127–146, 2000.

[13] M. von der Beeck. Formalization of UML-Statecharts. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 406–421, 2001.

[14] Perdita Stevens with Rob Pooley. *Using UML: software engineering with objects and components*. Addison Wesley Longman, 1999 (updated edition).