Dynamic Logic Semantics for UML Consistency

Greg O'Keefe

Research School of Information Science and Engineering, Australian National University, Canberra, ACT 0200, Australia. `greg.okeefe@anu.edu.au`

Abstract. The Unified Modelling Language (UML) is intended to describe systems, but it is not clear what systems satisfy a given collection of UML diagrams. Stephen Mellor has described a small collection of diagrams which appear to be inconsistent, yet are "cool" according to UML. We describe an approach to defining semantics for UML diagrams using dynamic logic, and show that Mellor's example is inconsistent, given a reasonable assumption. Our approach interprets all diagrams, static and dynamic, in a single semantic space. The modeller specifies how the meaning of a model is made up from the meanings of its diagrams, thus the "viewpoint" taken by each diagram is made explicit. This composition is achieved through formation of the dynamic logic formulae. It is therefore very flexible, and we propose it as a means for defining semantics for domain specific languages, and for specifying "bridges" or "weaving" model transformations used in aspect oriented modelling.

One approach to modelling is to begin with use-cases, and aim to reproduce them as sequence diagrams executed by the model being developed. Whether or not a model can execute a sequence diagram is therefore a question which will be asked frequently when working this way. We want the question to have a definite answer, and we want our tools to give us that answer.

Any multi-view approach to modelling will have similar questions about the relative consistency of its parts. The way to make these questions precise and amenable to automatic solution is by defining formal semantics for our language. Formal semantics are usually associated with formal verification for safety critical, or other trusted systems. Formal semantics are also usually associated with incomprehensible symbolic mumbo-jumbo.

The ability to formally certify the products of model driven development would certainly be beneficial. However, if the semantics could be understood by at least some modelling practitioners, the result would be clearer thinking and greater consensus in the modelling community. Modelling of problem domains would produce better understanding, and hence better solutions.

Although endowing parts of UML with formal semantics has become quite an industry, we do not know of any work that gives uniform semantics for class, state machine and sequence diagrams, as well as UML actions. Reggio and coworkers suggest an analogy between UML diagrams and logical axioms [13]. We promote the idea from analogy to practice, translating the diagrams and actions into formulae of dynamic logic [7]. Object diagrams can also be formalised, and adding OCL to the repetoire would be straightforward, following [3].

Some would argue that UML 2.0 is already well enough defined to resolve the kind of consistency question we study here. From this point of view, dynamic logic or any other rigorous mathematics are a complicated waste of time, since OCL and the UML action semantics provide all that is needed. Far from being the solution, we consider OCL and the action semantics to be a part of the problem. We believe that the current official definition is not adequate to unequivocally demonstrate the inconsistency of apparently inconsistent collections of diagrams. Attempting to do this would be an interesting test, but it is beyond our present scope. Here, we take UML model consistency to be a mathematical question, and tackle it using mathematical techniques.

By translating into a well studied formalism like dynamic-logic, we obtain precise semantics, along with a wealth of metatheory. Our example model attaches actions to states, and state diagrams to classes. We achieve this by making an action a subformula of a state machine diagram axiom, which is in turn a subformula of a class axiom. This suggests a general approach to specifying "weaving" of diagrams and models, by defining how their translations are combined to form model axioms. The idea of a "semantic variation point" used throughout the UML definition [11] can also be made precise in this way. One would simply use some parts of the translation output and not others, according to the required interpretation of the diagrams.

In [8], Stephen Mellor gives an example where UML blindly accepts an intuitively erroneous model. We use this example to demonstrate the style of semantics we propose, and their application to model consistency problems.

The first section briefly introduces dynamic logic, and then we introduce a version of Mellor's example inconsistent model, using the conventions of the Executable UML method [10]. In the following section we systematically translate each diagram into dynamic logic formulae. A system specification is then formed using these diagram formulae as subterms. We then search for a trace which satisfies this specification and establish that none exists. The conclusion compares this approach with related work, and considers the next steps towards a useful formal semantics for UML.

1 Dynamic Logic

In this section we briefly introduce logic, beginning with simple propositional logic. Then we consider two different extentions: modal logics and first order logic. Finally, we combine these extentions and obtain the form of dynamic logic we need to complete our formalisation.

A logic consists of syntax, semantics and a deductive calculus. The syntax defines a set of formulae, which we call the language of the logic. The formulae are just symbolised statements. The semantics defines a range of possible situations, each of which assigns either $true$ or $false$ to each formula of the language. A deductive calculus defines proofs, each of which derives a formula from some set of formulae. We will not say much more about deduction.

Propositional logic has atomic formulae $P, Q, R, \ldots$ and if φ and ψ are formulae, then so are $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$ and $\varphi \longrightarrow \psi$. These symbols stand for not, or, and, and if ... then ... respectively. The possible situations of propositional semantics are functions from the atomic formulae to the truth values $\{true, false\}$. We will call these functions propositional interpretations. These are extended to the whole language by assigning each of the connectives $\neg, \vee, \wedge$, $\longrightarrow$ the obvious truth function. We write $\top$ as a formula that is always true, and $\bot$ for one which is always false.

Propositional modal logics add some one-place connectives. Typically we add to the above syntactic rules that if φ is a formula, then so are $\Box\varphi$ and $\Diamond\varphi$. Some intuitive interpretations of these connectives are: necessarily and possibly, always and sometimes, obligitory and permissible. We are interested in temporal interpretations, where $\Box\varphi$ means that φ is true at all possible future situations, and $\Diamond\varphi$ is true in some possible future situation. Semantics for a propositional modal logic are given by introducing a binary relation R between the propositional interpretations. Then $\Box\varphi$ is true at w if φ is true at every situation R-related to w, and $\Diamond\varphi$ is true at w if φ is true at some situation R-related to w.

This is already a useful formal language, because if R captures the possible evolution of a system, and we have formulae $Init$ and Bad which represent the acceptable initial states of the system, and undesirable situations respectively, then the formula $Init \longrightarrow \neg\Diamond Bad$ is true if and only if it is impossible for the system to evolve from an acceptable initial state into an undesirable situation.

Propositional dynamic logic PDL has a pair of modal operators for each program in a simple programming language. There are atomic programs, $\alpha, \beta, \gamma, \ldots$ and if ρ and σ are programs then so are $\rho;\sigma$, $\rho \cup \sigma$ and ρ^*. These are the regular expressions over the atomic programs. Also, if φ is a formula, then $\varphi?$ is a program. Each atomic program denotes a relation over the situations, $\rho \cup \sigma$ denotes the union of the two relations (non-deterministic choice) and ρ^* denotes the reflexive transitive closure of the relation denoted by ρ (non-deterministic repetition). The program $\varphi?$ relates a situation to itself when φ is true there. This can be used to place guards on programs, and to write conditionals, such as $(\varphi?;\alpha) \cup (\neg\varphi?;\beta)$ for `if` φ `then` α `else` β. In propositional modal logic, the semantics for the modal operators $\Box$ and $\Diamond$ were given using a binary relation. In propositional dynamic logic, each program ρ corresponds to a binary relation, and the semantics of the modal operators $[\rho]$ and $\langle\rho\rangle$ depend on this relation.

We can write for example $\langle\alpha\rangle\top$, to mean that the program α runs successfully (terminates), or $\langle\alpha\rangle\top \longrightarrow \varphi$ to mean that α only runs successfuly in situations satisfying φ.

First order logic also extends propositional logic. Where the basic formulae of propositional logic are unanalysed propositions P, Q etc, first order logic formulae assert properties of individuals or assert relationships between individuals. For example, a two place relation symbol L might be interpreted as "... loves ..." the name a might mean "Aaron" and b "Belinda" then Lab would be read as "Aaron loves Belinda." The logic includes equality, so we may write $a = b$ meaning "Aaron is Belinda." Names are one kind of *term*, that is expressions which refer to an individual. Variables $x, y, z, \ldots$ are another kind of term, and terms can be formed by applying n-place function symbols, $f, g, \ldots$ to n terms, for $n = 1, 2, \ldots$. For example if the 1-place function symbol f is read as "the father of $\ldots$," then $Lxf(b)$ should be read as "x loves Belinda's father." First order logic also has *quantifiers* $\forall$ and $\exists$ so that $\forall x, Lxf(b)$ means everybody loves Belinda's father, and $\exists x, Lxf(b)$ that somebody does.

The semantic situations for first order logic (which logicians call "*models*") consist of a set of individuals, called the semantic *domain*, and an *interpretation* which takes each name to an individual and each n-place relation/function symbol to a n-place relation/function. To evaluate variables and quantifiers, we also need a *valuation*. This takes each variable to an individual in the semantic domain. The formal definition of truth of a formula in an interpretation just says that, in the interpretation, things are as the formula says they are. The frightening notation required to state this precisely is unhelpful in the current context.

The semantics of quantified formulae are defined using the idea of *variants* of the valuation. An x variant of w is a valuation that is the same as w for all inputs except for x. This is worth explaining, because we will use these ideas again soon. We introduce some notation for a function the same as w, except that it takes x to q. Define $w \oplus x \mapsto q$ by

$$(w \oplus x \mapsto q)(y) = \begin{cases} q & \text{if } x = y \\ w(y) & \text{otherwise} \end{cases}$$

Then $\exists x, \varphi(x)$ is satisfied by the model and valuation $\mathcal{M}, w$ iff $\varphi(x)$ is satisfied by $\mathcal{M}, (w \oplus x \mapsto q)$ for some q. And similarly for $\forall$ formulae.

The language of dynamic logic includes that of first order logic plus modal operators similar to those of PDL. The atomic programs of DL are assignments of the form $x := t$ for some variable x and some term t. For each interpretation $\mathcal{M}$, the atomic program $x := t$ relates each valuation w to $w \oplus x \mapsto t^{\mathcal{M},w}$, where $t^{\mathcal{M},w}$ is the value of the term t under the interpretation $\mathcal{M}$ and valuation w. For example, the formula $a = 5 \longrightarrow \langle x := a \rangle x = 5$, says that if $a = 5$ then after you set x to a, you get $x = 5$. This is always true, because if $a = 5$ in $\mathcal{M}, w$, that is $a^{\mathcal{M},w} = 5$, then $x = 5$ in $\mathcal{M}, (w \oplus x \mapsto a^{\mathcal{M},w})$.

Our objective is to reason about object oriented systems, where objects retain their identity over time, but have attributes whose values may change. To achieve this, we have object identifiers as individuals, and object attributes as functions, so that the familiar $o.a$ notation becomes short-hand for $a(o)$. Then what we want is the ability to update these functions. An extension of dynamic logic studied in [7] allows such updateable functions, called array variables. Indeed, a similar system has been used to formalise parts of UML 1.1 in [17].

The syntax of DL is extended by n-place array variables for each $n = 1, 2, \ldots$. These can occur wherever an n-place function symbol can occur, but also on the left hand side of an assignment statement. The semantics are adapted so that now the valuations also assign an n-place function to each n-place array variable. For a fixed model $\mathcal{M}$, this assignment denotes a relation that relates each valuation

w to $w \oplus h \mapsto (h^w \oplus t^{\mathcal{M},w} \mapsto s^{\mathcal{M},w})$. That is, w is related to an updated form of w, which maps the array variable h to the same function w maps it to, except updated so that it sends the value of t to the value of s.

This is the form of dynamic logic that we use to precisely express our interpretations of the UML diagrams and actions.

2 The Problem Model

We begin this section with Mellor's description of the problem which we aim to a resolve.

Consider this. We have two state chart diagrams, one of which sends a single signal X to the other. In the same system, we have a sequence diagram that shows lifelines for two objects whose behaviour is captured by the state chart diagrams, one of which sends a single signal Y to the other. Both diagrams are intended to describe the same behaviour; that is, a single message being sent between them. Which of these two - contradictory - models is correct? Astonishingly, UML's answer is Yes. So long as the syntax of each of the two diagrams is correct, UML is cool.

-Stephen Mellor in [8]

There should be some definite meaning attached to UML diagrams, so that tools can detect that no system can satisfy all the diagrams in Mellors' example.

We could formulate the problem as Mellor has, using two state machine diagrams one sequence diagram and no class diagram, but this would require more "weaving" logic (Section 4, Page 10) than the Executable UML [10] style of model we present here.

A class diagram (Fig. 1, Page 5) declares the two classes. The association between them will be used to target the signal.

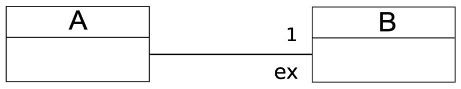

Fig. 1. Class Diagram

We only give one state machine diagram (Fig. 2, Page 6), which describes the behaviour of class A. Since the other state machine is completely arbitrary, there is no point in specifying it. The join between the state machine and class A will be made explicit in Section 4.

The only actions which we consider are signal send and receive. These are only approximations of the official actions described in [11, §11.3.44 and §11.3.2],

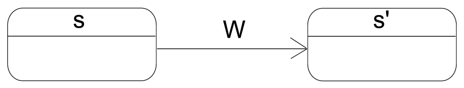

Fig. 2. State Machine for Class A

since we do not attempt to capture ideas such as values travelling across "pins," and we use a stricter message queueing policy. The entry action for state s' is given in an action language of our own devising in Figure 3. Again, the join will be given formally in Section 4.

```
send X to self.ex
```

Fig. 3. Entry Action for State s'

Finally, the sequence diagram (Fig. 4, Page 6) shows an object of class A accepting a W signal from an external entity, enterning state s' and (erroneously?) sending a Y signal to a B object (presumably its ex).

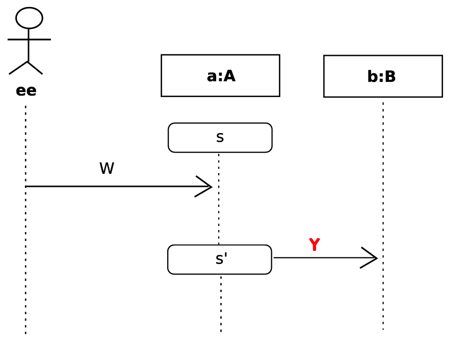

Fig. 4. Sequence Diagram

3 The Model as Dynamic Logic Formulae

How can we represent this model using formulae of dynamic logic? We begin this section with some general considerations about the relationship between dynamic logic and the small UML subset used for this model. In each of the following subsections, we will discuss one of the diagrams, and give its meaning as a dynamic logic formula. These meanings are influenced by the use made of the diagrams in the Executable UML [10] method. In the next section we will combine the class diagram, state machine diagram and action formulae to specify the system, and then show that this specification is inconsistent with the sequence diagram formula.

We give rather weak interpretations of each diagram, for example, assuming that there might be objects in the system that do not belong to any class on the class diagram. These interpretations are debatable, indeed it is possible that the weak interpretations are more appropriate in an analysis phase, whilst stronger ones might serve the design phase better. Indeed, we may want to keep several interpretations available to cater for UML's numerous "semantic variation points." The particular interpretations are not the main point though. Rather we aim to demonstate that dynamic logic provides a simple and useful way of giving the meanings of the diagrams. The reader should not overlook another important virtue of the weak interpretations: they are shorter!

3.1 System Snapshots and Evolution

In Section 1 we said that the possible situations of the semantics of dynamic logic consist of an interpretation and a valuation. We will consider all model specific vocabulary to be variables, evaluated by the valuation, while the interpretation takes care of the global UML vocabulary such as OCL library functions. Hence, a system snapshot is a valuation, and the interpretation can be largely ignored, because it is fixed.

Each DL program relates pairs of these snapshots, but not every DL program corresponds to a legal evolution of the system. What we need is a formal definition of legal evolution. So, what can happen in a system defined by our subset of UML? Only two things really: objects can send messages, and they can accept them. There are conditions though, an object can only send a message if that action is at the head of its todo list, although an external entity can send whatever it wants whenever it wants. An object can only accept a message if it has a message to accept, and it is not currently activated. The non-deterministic DL program ε which describes how these systems can evolve is defined as follows.

$$\varepsilon \equiv ((\mathit{sendCond}(x, M, y)?;\ x.\mathtt{send}\ M\ \mathtt{to}\ y) \cup (\mathit{acceptCond}(x)?;\ x.\mathtt{accept}))^*$$

where $\mathit{sendCond}(x, M, y)$ is a formula, defined below, stating the conditions under which it is OK for x to send an M message to y, $\mathtt{send}\ M\ \mathtt{to}\ y$ is a DL program, also defined below, which does what the send action is meant to do, and similarly for *acceptCond* and `accept`.

$$\mathit{sendCond}(x, M, y) \equiv \mathit{class}(x) = \mathit{EE} \vee (\mathit{head}(\mathit{todo}(x)) = \mathtt{send}\ M\ \mathtt{to}\ y)$$
$$\mathit{acceptCond}(x) \equiv \mathit{todo}(x) = ()\ \wedge\ \mathit{size}(\mathit{intray}(x)) > 0$$

where *head* and *size* are library functions with the obvious meaning, and *todo* and *intray* are array variables used to represent an objects outstanding actions and messages respectively. The special class name *EE* is introduced so that external entities can be treated as objects having this class.

Note that *todo* takes objects to programs, which on the face of it is a category error, because programs are part of the syntax, not individuals in the semantic domain. By adding function symbols corresponding to $*, :=, \cup$ and ;, we can copy the program language into the semantic domain. Whenever you see a program on the right hand side of an equals sign, it is shorthand for a term formed using this vocabulary.

Now we define the actions `send` and `accept`. When an object x sends a message M to the object y, the message is placed in y's intray, and the `send` action is removed from x's *todo* list.

```
$x$.send $M$ to $y$ $\equiv$
    $intray(y)$ := $append(intray(y), M)$;
    $todo(x)$ := $tail(todo(x))$
```

where *append* and *tail* are library functions.

When an object accepts a message, it makes the state transition specified in its state machine, loads the entry procedure of the new state, and removes the message from its *intray*.

```
$x$.accept $\equiv$
    $state(x)$ := $nextState(state(x), head(intray(x)))$;
    $todo(x)$ := $entryProc(state(x))$;
    $intray(x)$ := $tail(intray(x))$
```

Where *state* is an array variable used to record an objects state. This definition depends on the functions *nextState* and *entryProc*, which in turn depend on the state machine diagrams of the model. The definitions of these functions will turn out to be a consequence of the formulae we extract from the state machine diagram. We will consider them to be array variables in order to keep the interpretation general, but any attempt to assign them values that do not agree with the state machine diagram(s) would result in an inconsistency.

This program ε allows us to say things about model dynamics. Adapting the example in Section 1, we can say that nothing bad will happen so long as the model starts within acceptable initial conditions: $Init \longrightarrow \neg\langle\varepsilon\rangle Bad$. Note ε is a $*$ program, so the program under the $*$ can run 0 times. This means that for $[\varepsilon]\varphi$ to be true in some situation w, φ has to be true there and in every situation reachable by legal model evolution. Therefore, if we want to say that in our model, φ is always true (an invariant), we can assert $[\varepsilon]\varphi$.

This sets the general framework for systems defined by models of our tiny UML subset. Now we are ready to look at the diagrams that define Mellor's example model.

3.2 Class Diagram

This class diagram (Fig. 1, Page 5) does not tell us a lot. The association however, does tell us something. It says that each object of class A, has exactly one ex which is an object of class B. Since this is all the information we can obtain from the class diagram, we will name the formula CD.

$$\begin{aligned} CD \equiv [\varepsilon](\forall x, class(x) = A \longrightarrow \\ size(x.ex) = 1 \ \wedge \ (\forall y,\ y \in x.ex \longrightarrow class(y) = B)) \end{aligned}$$

Notice that we have taken some vocabulary from the class diagram. The class names A and B are variables, ex and $class$ are array variables.

3.3 State Machine Diagram

The state machine diagram (Fig. 2, Page 6) does not specify which objects it applies to, so the state machine diagram formulae contain a free variable. In Section 4 we will use this variable to connect the state machine diagram to the class A.

For an object to conform to this state machine, it must be in one of the diagrams' states.

$$SM_s(x) \equiv [\varepsilon](state(x) = s \ \vee \ state(x) = s')$$

The transition in the state machine says that if an object x is in state s and it has a W message at the top of its $intray$, then after it does an `accept`, it will be in state s'.

$$\begin{aligned} SM_t(x) \equiv [\varepsilon](state(x) = s \ \wedge \ head(intray(x)) = W \\ \longrightarrow [x.\texttt{accept}]\ state(x) = s') \end{aligned}$$

We will not combine the state and transition formulae yet, but will combine them with the entry procedure for state s' and the class A in Section 4.

3.4 Sequence Diagram

The sequence diagram (Fig. 4, Page 6) partly specifies an initial model state, and lists some occurrences in the order that they are meant to happen. It is satisfied by model execution traces that begin in a state that satisfies the diagrams intitial conditions, and in which all the occurrences happen legally, in the given order.

Note that other things are allowed to happen in between the occurrences given in the diagram. Read, write, link and unlink actions are not shown in sequence diagrams. An object or external entity not shown in the diagram might send one of the participants a message. Indeed, we might allow participants of the sequence to exchange messages not shown on the diagram. If the diagram is intended as a high-level summary, we might choose to omit some of these details.

The following formula captures our interpretation of the sequence diagram. It says that ee is an external entity, a has class A, b has class B, and that it is possible for some stuff to happen, and then for ee to legally `send` a W to a, and then for some more stuff to happen followed by a legally doing an `accept` (activation) after which some stuff can happen and then a can legally `send` a Y message to b.

$$\begin{aligned} SEQ \;\equiv\; & class(ee) \;=\; EE \;\wedge\; class(a) = A \;\wedge\; class(b) = B \;\wedge \\ & \langle\varepsilon\rangle(\; sendCond(ee, W, a) \;\wedge\; \langle ee.\texttt{send}\ W\ \texttt{to}\ a\rangle \\ & \langle\varepsilon\rangle(\; acceptCond(a) \;\wedge\; \langle a.\texttt{accept}\rangle \\ & \langle\varepsilon\rangle(\; sendCond(a, Y, b) \;\wedge\; \langle a.\texttt{send}\ Y\ \texttt{to}\ b\rangle\ \top))) \end{aligned}$$

As usual, we use variables for all the model specific vocabulary.

4 Weaving and Consistency

Having specified the meaning of each of the parts of the model, we now turn to the task of connecting these parts to specify the required system. Because we have interpreted the diagrams using logical formulae, we can join them using logical connectives.

We want to make the send action the entry action of state s'. That is, if an object is in state s', then immediately after it does an accept, we want its *todo* list to contain only this action.

$$SM_p(x) \equiv [\varepsilon][x.\texttt{accept}](state(x) = s' \longrightarrow todo(x) = \texttt{send}\ X\ \texttt{to}\ x.\texttt{ex})$$

Putting x.`ex` rather than `self.ex` saves us the trouble of evaluating `self`.

Recall that in Section 3.3 we defined two formulae from the state machine diagram, $SM_s(x)$ for the states and $SM_t(x)$ for the transitions. Now, we attach the state machine diagram augmented by the entry procedures, to the class A.

$$SM \equiv [\varepsilon](\forall x,\ class(x) = A \longrightarrow SM_s(x) \;\wedge\; SM_t(x) \;\wedge\; SM_p(x))$$

We are now in a position to ask whether our model is consistent. In other words, is there an execution trace which can satisfy CD, SM and SEQ?

Semantic tableaux deductive calculi [4] do a systematic search for an interpretation which satisfies their input formulae. Their purpose is to show that an argument is valid, that is, that it is impossible for the premises of the argument to all be true in the same situation where the conclusion is false. To test this, the conclusion is negated, and together with the premises, input to the search procedure. If a situation that satisfies these inputs is found, it is a counter-example to the argument, because it makes the premises and the negated conclusion all true, hence it makes the premises true and the conclusion false. It is important that the search procedure is exhaustive, because then if no counter-example is found, we may conclude that the argument is valid. Some logics, such as first order logic and DL are undecidable, so for some inputs, the procedure will not terminate. However, when it does terminate without finding a counter-example, we still know that the argument is valid.

Our goal is different: we want to show that a collection of formulae are consistent. We can therefore simply enter our formuale into the search procedure. If it finds an interpretation which satisfies the inputs, our model is consistent. If it terminates without finding one, it is inconsistent. Since the required execution traces are unlikely to be difficult to find, if the process runs for a long time without terminating, this would be a fair indication that the model has problems.

For now, we will outline a manual search. We will assume that there is a situation which satisfies SEQ, CD and SM, and we will call it w_0. Recall that these situations are valuations, which take variables to individuals, and array variables to functions. Recall also that programs denote relations between these valuations. So our search will consist in breaking down the formulae until we have a collection of explicitly specified valuations, related as required by DL programs.

The three formulae, and some harmless assumptions tell us that w_0 satisfies: $class(ee) = EE, class(a) = A, class(b) = B, a.ex = \{b\}, state(a) = s, intray(a) = (), intray(b) = ()$ and also the dynamic part of SEQ

$$\langle\varepsilon\rangle(sendCond(ee, W, a) \wedge \langle ee.\mathtt{send}\ W\ \mathtt{to}\ a\rangle \ldots \top) \ \text{ at } w_0 \tag{1}$$

But ee is an external entity which can send as it pleases, and ε can run 0 times, so it is sufficient to have

$$\langle ee.\mathtt{send}\ W\ \mathtt{to}\ a\rangle \ldots \top \ \text{ at } w_0 \tag{2}$$

For this to be true, we need the send action to relate w_0 to another valuation which we will call w_1. We write $w_0 \xrightarrow{\mathtt{send}\ W\ \mathtt{to}\ a} w_1$ to express this. Satisfaction of (2) now reduces to

$$\langle\varepsilon\rangle(acceptCond(a) \wedge \langle a.\mathtt{accept}\rangle \ldots \top) \ \text{ at } w_1 \tag{3}$$

Nothing needs to happen before a accepts this message, so we turn our attention to w_2 where $w_1 \xrightarrow{a.\mathtt{accept}} w_2$. Now the definition of `accept`, on Page 8, depends on the array variables $nextState$ and $entryProc$, so its not immediately obvious what the situation will be in w_2. A little reasoning, for which we do not have space, shows that the state machine formula SM determines that these variables behave as we expect, and so we require w_2 to satisfy $state(a) = s', todo(a) = \mathtt{send}\ X\ \mathtt{to}\ b$ and

$$\langle\varepsilon\rangle(sendCond(a, Y, b) \wedge \langle a.\mathtt{send}\ Y\ \mathtt{to}\ b\rangle\top) \text{ at } w_2 \tag{4}$$

and now, it would appear that we are stuck, because it seems nothing that can possibly happen is going to put $\mathtt{send}\ Y\ \mathtt{to}\ b$ into a's todo list, which we need in order to satisfy $sendCond(a, Y, b)$. But recall that X and Y are variables. Hence each valuation maps them both to individuals in the semantic domain. If w_2

maps X and Y to the *same* individual, then *sendCond* is satisfied, the `send` can proceed and the sequence successfully completes, showing that it is consistent with the model defined by the class and state machine diagrams.

This is clearly not how the story is supposed to end. We are developing a formal theory of our intuitive understanding of the diagrams, and our intution says they are inconsistent, because X is X, Y is Y, and they can not be the same thing. That is "X" is not just a label we stick on that message from outside the system, but rather something essential to its identity. We can capture this idea formally by retrospectively asserting the following naming invariant

$$NAMES \equiv [\varepsilon](name(X) = \text{“}X\text{”} \wedge name(Y) = \text{“}Y\text{”} \wedge \cdots \wedge name(s') = \text{“}s'\text{”})$$

which makes $X = Y$ impossible. It also prevents the classes A and B from being the same, and also weird possibilities like $A = s$, identifying a class and a state. (Although Mellor [9] has made suggestions that are almost this strange when discussing how domain models might be woven together.)

So now there really is no way to find a situation with `send` Y `to` b on a's todo list, so the sequence diagram is inconsistent with the model. This could be proven by induction on the definition of ε.

5 Related Work, Future Work, Conclusions

Our work is influenced by that of Roel Weiringa and collaborators. Executable UML's predecessor, the Shlaer-Mellor method [15] is studied in [16]. Another shorter paper [17] studies a subset of UML 1.1, but the work is so similar that we focus on the earlier contribution. The visual modelling language of the Shlaer-Mellor method is given formal semantics using their own language LCM, which is based on dynamic logic. This logic is first order, but the programs are not explicit assignments, but rather unanalysed atomic programs like those of PDL. To ensure that these programs have the desired effect, guard axioms $\langle\alpha\rangle\top \longrightarrow \varphi$ and effect axioms $\varphi \longrightarrow [\alpha]\psi$ are employed, but assumptions not expressible as axioms are also required.

Shlaer-Mellor is subjected to a "methodological analysis" resulting in many suggested revisions, some of which appear in Mellor's newer method, Executable UML [10]. Some of the revisions though, seem too extreme.

> Our methodological analysis has led to a system transaction concept that is simple to formalize: Each system transaction is a synchronous execution of a non-empty finite set of local object transactions. Because each object transaction has a local effect only, this composition is harmless and we can simply conjoin the local effects.

Thus tracing a timeline of object interactions, as we have done with our sequence diagram formalisation, would be impossible. Weiringa's goal is requirements engineering, which is not concerned with the internal behaviour of a system, only its interaction with its environment. Model driven development asks more than this of its models. We require the ability to fully describe a system, including its internal operations.

Another body of work using a type of dynamic logic to formalise parts of UML is that of the KeY project [1], led by Peter Schmitt and Bernhard Beckert. Their KeY tool verifies programs written in a subset of the Java programming language against OCL constraints and their contextual class diagram, using a specialised dynamic logic [2] with over 250 rules. This tool is tightly integrated with a commercial CASE/modelling tool. This combination of UML class diagrams, OCL and Java which they have formalised [2,3], can be seen as an executable modelling language like UML, or various executable subsets of it. The difference is that the Java code is much more complex and less flexible than UML actions, but can stand alone as an implementation. To formalise such a system and build a practical tool from it is a remarkable accomplishment, which we find encouraging for our own project. Java, although complex, is according to Beckert [2, §4] quite well defined. Almost everybody, except for Bran Selic [14], would argue that this is not the case with UML.

Algebraic specification extended with "generalised labelled transition systems," is used by Gianna Reggio, Maura Cerioli and Egidio Astesiano to formalise parts of UML in [13] and earlier papers by the same authors. They do this by translating UML diagrams into the language Casl-LTL, though they emphasise that the particular language is immaterial. This work, like our own, explicitly aims for a way of giving useful formal semantics to the whole of UML, and as the title suggests, they take seriously the idea that the different diagrams combine to specify a single system. Our translation seems, to us at least, to more closely resemble the original model, although the Casl-LTL specifications are perhaps more readable for software developers. We agree with Reggio and her coauthors that

> It is worth noting that to state the behavioural axioms we need some temporal logic combinators available in Casl-Ltl that we have no space to illustrate here. The expressive power of such temporal logic will also be crucial for the translation of sequence diagrams...

A small executable subset of UML suitable for real-time systems is defined in [5], and given formal semantics using symbolic transition systems. These are due to Amir Pnueli, who is one of the papers authors. While we and Reggio's group translate diagrams into an existing language with semantics of its own, this work starts with a "blank slate." The metamodel abstract syntax of the official definition is ignored, and a traditional formal syntax is given for the selected UML subset. For tools based on this language definition to interoperate with metamodel based tools, a translation will be required. Also, in order to do verification, one needs to express ones proof goals. Ideally, OCL should be used for this, since it is the official constraint language of UML. However, even if you can not use OCL, if you translate the diagrams into some other language, you can use that to express your constraints. This work has neither semantics for OCL nor any alternative constraint language, so more work must be done before they can state what they want to prove. Much of the complexity of this work comes from the need to model hard real-time systems. It may be best to spare the general UML semantics from this complexity, and save it for specialised efforts like this one.

Finally, for something completely different, [6] uses UML collaboration diagrams as graph transformation rules which specify the semantics for object and state machine diagrams. The state of a UML specified system is actually an object diagram, and its evolution is determined by these rules. This is certainly a very direct kind of formal semantics which would have great advantages if it can be applied to more of the language.

How is one to compare these different approaches? None yet comes close to covering a convincing selection of UML's huge repetoire. None can demonstrate an actual proof of a plausible model requirement. Without careful presentation, only the graph rewriting approach is likely to be understood by practitioners. Only KeY is implemented in a form useable by practicing developers. It seems to us that it is more urgent to find justifiable and testable criteria for UML formal semantics proposals, than it is to find more proposals.

Our formalisation has its weaknesses. Dynamic logic lacks types and specialisation. The DL variant used in [16,17] has ordered sorts for this purpose. We have no concurrency, though concurrent dynamic logics have been studied [12]. We consider only a tiny fragment of UML, with less than perfect fidelity. Readability must also count as a serious weakness of dynamic logic formulae. An attractive possible solution is to obtain a dynamic logic from a graph-based logic, rather than from the usual text-based logics. The atomic programs of such a logic would be graph transformations.

On the positive side, we have shown that our approach can solve a practically motivated problem that no previous effort can manage. The translation into dynamic logic has made the meaning of each diagram quite precise, and made explicit the role they play in the meaning of the model. We have tackled the problem of consistency in UML using the much better developed theory and techniques from logic. Expressing models as logical formulae has the additional benefit that they are ready for use in formal proofs.

A suitable variant of dynamic logic, perhaps graphical, could we believe, serve as the foundation for UML. A definitive translation from the UML metamodel to the syntax of this language would make it absolutely clear what any given model means. By applying tools and techniques from logic, model consistency could then be checked automatically.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, 2005.
2. Bernhard Beckert. A dynamic logic for the formal verification of java card programs. In *Java on Smart Cards: Programming and Security*, number 2041 in LNCS, pages 6–24. Springer, 2001.

3. Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the object constraint language into first-order predicate logic. In *Proceedings of VERIFY, Workshop at Federated Logic conferences (FLoC)*, 2002.
4. M D'Agostino, D Gabbay, R Haehnle, and J Posegga, editors. *Handbook of Tableau Methods*, chapter Tableau methods for modal and temporal logics, by Rajeev Goré. Kluwer, 1999. `http://rsise.anu.edu.au/~rpg/Publications/Handbook-Tableau-Methods/TR-ARP-15-95.ps.gz`.
5. Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A formal semantics of concurrency and communication in real-time UML. In *Formal Methods for Components and Objects, Proceedings 2002*, volume 2852 of *LNCS*. Springer, 2003.
6. Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioural diagrams in UML. In *Proceedings of UML*, volume 1939. LNCS, 2000.
7. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.
8. Brian Henderson-Sellers. UML - the good, the bad or the ugly? perspectives from a panel of experts. *Software and System Modeling*, 4(1):4–13, 2005.
9. Stephen J. Mellor. A framework for aspect-oriented modelling. In *The 4th AOSD Modeling With UML Workshop*, 2003.
10. Stephen J. Mellor and Marc J. Balcer. *Executable UML, A Foundation for Model-Driven Architecture*. Object Technology Series. Addison-Wesley, 2002.
11. Object Management Group. Unified modeling language: Superstructure. Technical report, Object Management Group, August 2005. `http://www.omg.org/docs/formal/05-07-04.pdf`.
12. David Peleg. Concurrent dynamic logic. *Journal of the ACM*, 34(2):450–479, April 1987.
13. Gianna Reggio, Maura Cerioli, and Egidio Astesiano. Towards a rigourous semantics of UML supporting its multiview approach. In H. Hussmann, editor, *FASE 2001*, volume 2029 of *LNCS*, pages 171–186. Springer, 2001.
14. Bran V. Selic. On the semantic foundations of standard UML 2.0. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, number 3185 in LNCS, 2004.
15. Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modeling the World in States*. Yourdon Press, 1992.
16. R.J. Wieringa and G. Saake. A formal analysis of the Shlaer-Mellor method: Towards a toolkit of formal and informal requirements specification techniques. *Requirements Engineering*, pages 106–131, 1996.
17. Roel Wieringa and Jan Broerson. Minimal transition system semantics for lightweight class and behaviour diagrams. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universitaet Muenchen, TUM-I9803, April 1997.