JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2005

Vol. 4, No. 1, January–February 2005

A unified superstructure for UML

Andy Evans, Paul Sammut, and James S. Willans
Xactium Ltd., Sheffield, U.K.

Alan Moore
ARTiSAN Software Tools Ltd, Chelthenham, U.K.

Girish Maskeri
Tata Consultancy Services, India

A key aspect of successfully using UML is understanding the semantics of the notations. UML 2 will increase the already substantial collection of notations supported by UML 1.x. At the same time, this will augment the difficulty users experience in understanding semantics. In this paper we propose that while the diverse notations may render concepts differently, the concepts can often be considered semantically equivalent. This gives rise to an architecture where two single abstract syntaxes (structure and behaviour) underpin UML 2's seven concrete syntax. Because there a fewer semantically distinct concepts, this makes UML both easier to understand and substantially easier to implement.

1 INTRODUCTION

A characteristic of languages is that the richer they become, the more difficult they are to use. UML 2 promises to delivery a number of new language constructs that enable a broader range of specifications to be constructed. A key issue in successfully using UML 2 is understanding the semantics of the augmented language. For example, what is the meaning of a class diagram in terms of a component diagram, or what is the meaning of a state machine in terms of an activity diagram. These are not easy questions to answer and involve understanding the semantics of each individual construct.

The current approach to defining UML is to define an abstract syntax for each of the notations (see [3]). Each abstract syntax is associated with a concrete syntax that defines the rendering of the notations to the user, as illustrated in figure 1. This approach is also followed in the dominant submission to UML 2 superstructure [4]. Because every concrete syntax construct has a construct in the abstract syntax, there are as many semantic *units* to understand as there are concrete constructs. Consequently, comprehending the semantics of UML is (and will be) non-trivial.

The approach proposed in this paper is to generalise the abstract syntax of UML 2 such that there is no longer an abstract syntax for each concrete syntax, rather a

Cite this article as follows: Andy Evans et al: "A unified superstructure for UML", in *Journal of Object Technology*, vol. 4, no. 1, January–February 2005, pages 165–181,
http://www.jot.fm/issues/issues_2005_01/article6

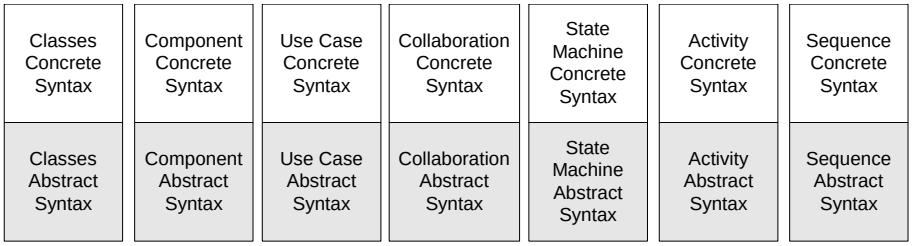

Figure 1: The current approach to defining UML

single abstract syntax is shared by many concrete syntaxes. This generalisation is motivated by the observation that there are many semantically equivalent constructs across diverse UML notations, for example *join* action is common to both state machines and activity diagrams. An overview of the resulting architecture is shown in figure 2. From this it can be seen that two abstract syntaxes underpin the spectrum of concrete syntaxes. Structure abstract syntax deals with notations such as class diagrams and component diagrams and behaviour abstract syntax deals with notations such as state machines activity diagrams and sequence diagrams. The consequence of this generalisation is that overall size of the abstract syntax definition is reduced. This means that there are fewer distinct semantics to understand, and a significant smaller definition to implement.

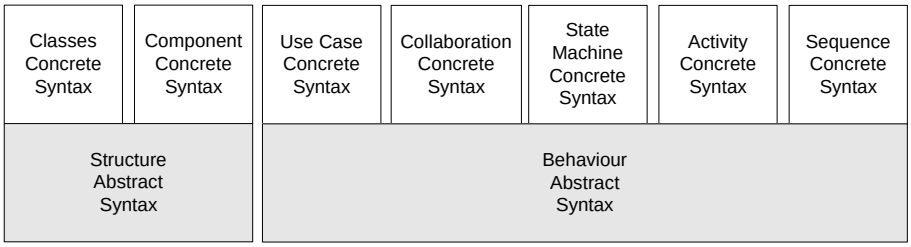

Figure 2: The proposed approach to defining UML

It should be noted that it is still necessary to define the semantics of the fewer abstract syntax concepts. The approach we have taken to this is described in [1]. In this, the abstract syntax is mapped down to a kernel of primitive concepts [2] each of which has an associated model-based semantics. Although this detail is outside of the scope of the paper, we observe that this mapping is greatly simplified by having fewer abstract syntax concepts to define the mapping for.

This paper describes the two abstract syntax models, the (almost complete) concrete syntax for five of the concrete syntaxes. Due to space limitations we do not describe the concrete syntax for collaboration and use case diagrams and only the important constraints are included in presented models. The paper is now structured as follows. Sections 2 and 3 presents the structure and behaviour abstract and concrete syntax model respectively. Finally in section 4 we summarise our

conclusions.

2 STRUCTURE

Currently UML enables the definition of structural models using class diagrams. UML 2 augments this with the capability of modelling component diagrams. Component diagrams are used to express architectural level definitions of systems and are particularly used within the context of systems engineering. In the following sub-sections we demonstrate how a unification can be leveraged between class diagrams and component diagrams. This unification is enabled by the treatment of components as classes and a component's ports as attributes of a class.

Abstract Syntax

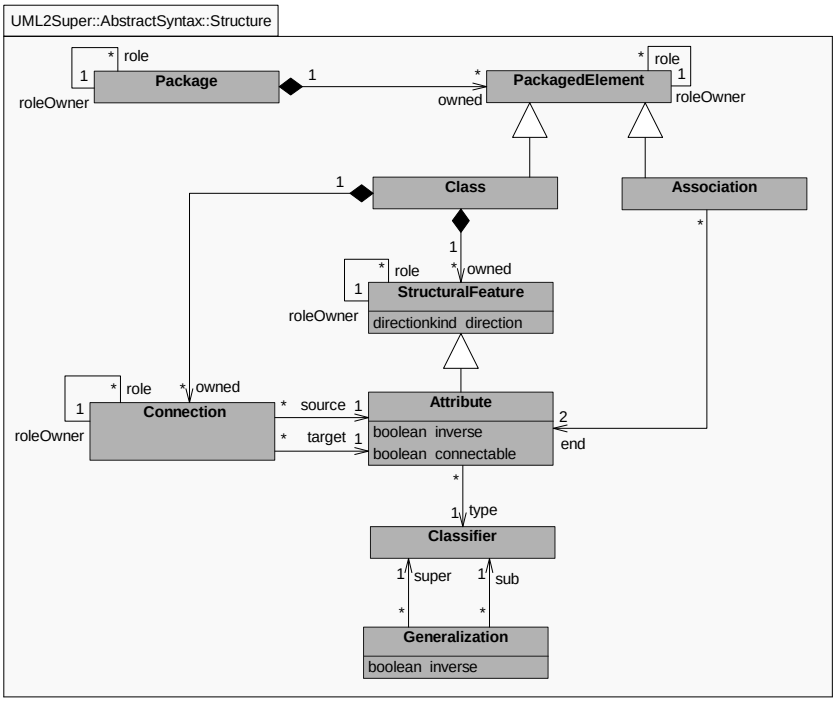

Figure 3: Structure abstract syntax

Figure 3 illustrates the abstract syntax for structure. The treatment of *Class* and its *Features* is conventional, the important difference is the addition of *Connection* which relates two *Attributes*. Syntactically, a *Connection* binds the type of the related *Attributes*:

```
context Connnection inv:
```

```
self.source.type = self.target.type
```

This constraint is mirrored in the semantic domain (not shown) and ensures that the values of the two *Attribute*s are appropriately bound (see [2] for more details of the semantic domain and the role it plays in our approach to modelling). The semantic domain constraint must also takes into account the direction of *Connection* and whether the *Connection* is inverse to that direction.

Only *connectable* attributes can be connected:

```
context Connection inv:
  self.source.connectable and self.target.connectable
```

*Package*s, *PackagedElement*s and *Connection*s have a self association called *role*. This is a generalised version of UML 1.x role mechanism which enables candidate scenarios of the model element to be specified [5][195-203]. All model elements in the structure (and behaviour) abstract syntax have a role mechanism either directly or derived through generalisation[1]. The relationship between a *role* and and a *roleOwner* is that the former can only constrain the semantics of the latter. The precise constraint for this association is dependent on the model element, in the case of class this is:

```
context Class inv:
self.role->forAll(role | role.memberStructuralFeature->excluding(
  role.memberStructuralFeature->select(s | s.isKindOf(Constraint))->forAll(
    feature | self.memberStructuralFeature->exists(feature.roleOwner))) and
      self.role->forAll(role | self.memberStructuralFeature->select(s |
        s.isKindOf(Constraint))->forAll(constraint |
          role.memberStructural->exists(constraint)))
```

*Role*s effectively create a parallel model mirroring the composition hierarchy of the *roleOwner*. A *Package* role must have *PackagedElement* roles for its role owners *PackagedElement*s, and a *Class* role must have *StructuralFeature* roles for its role owners *StructuralFeature*s, and so on.

Class diagram

The concrete syntax for *Class diagram* is shown in figure 4 and describes how a *ClassBox* can be the target of *AttributeLine*s,*AssociationLine*s and a *GeneralisationLine*s. A *ClassBox* contains compartments for its *Operations*'s and *Attribute's* strings. The mapping between the *Class diagram* model and *Structure* model is depicted in 5.

[1]The repeating role pattern alludes to templates which are the means by which the definition presented in this paper has been specified, see [2] for more details.

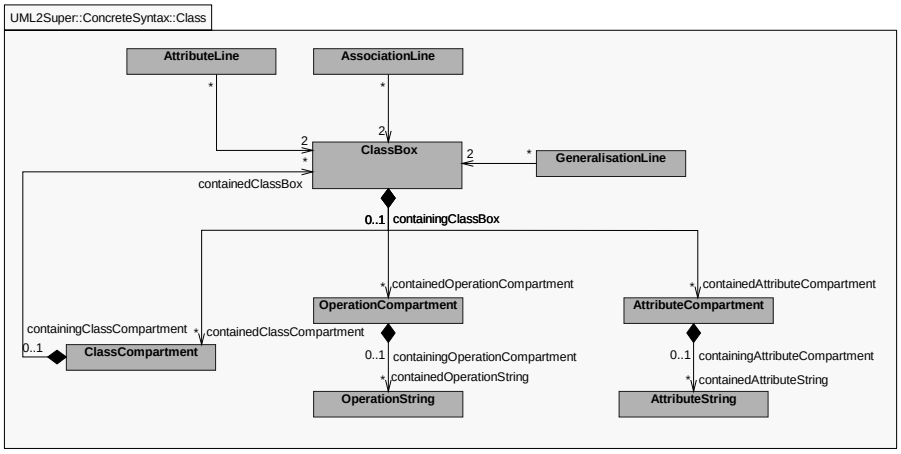

Figure 4: Class diagram concrete syntax

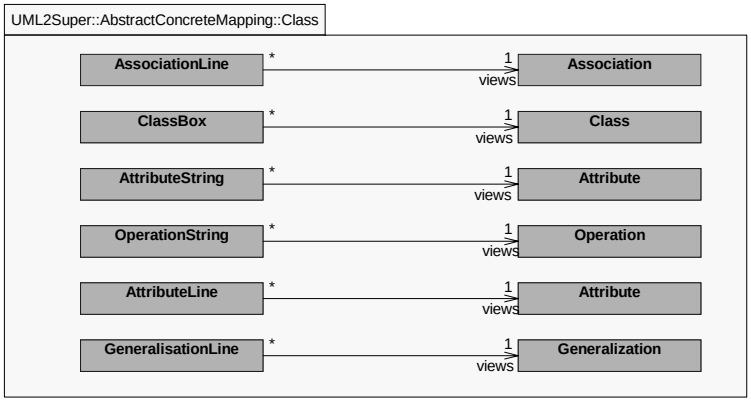

Figure 5: Class diagram abstract syntax mapping

Figure 6 describes a vending machine as an instance of the class concrete syntax definition. As can be observed from this example, our class model conforms to the conventional UML 1.x definition.

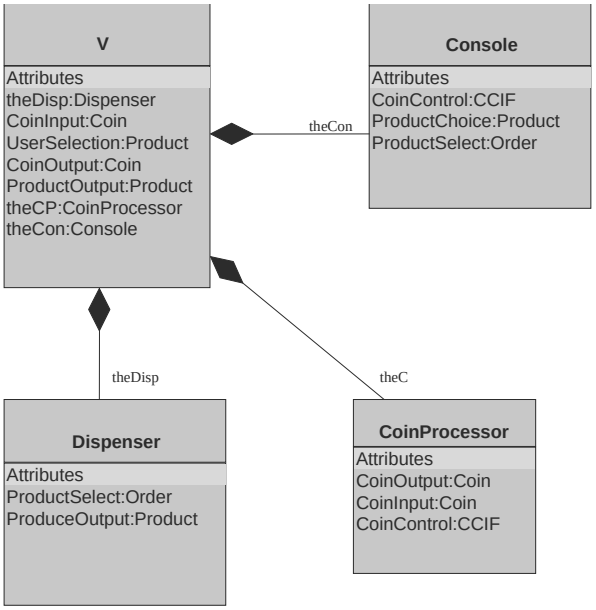

Figure 6: Class diagram of a vending machine

Component diagram

Component diagrams describe systems as components and the relationship between the systems as connectors. When two systems are related via a connector, this specifies that data sent by one system is available to the other system (depending on the direction of connection). The concrete syntax model for component diagrams is shown in figure 7 and the mapping to the abstract syntax model is shown in figure 8. *PortBoxes* must map onto attributes which are *Connectable*:

```
context PortBox inv:
  self.views.connectable = true
```

The nesting of components is unfolded onto classes and their attributes. Given a *ComponentBox* A which contains a *ComponentBox* B, A and B are mapped to *Classes* and the class representing A must have an attribute of type B:

```
context ComponentBox inv:
  self.containedComponentBox->forAll(ccb | self.views.owned->includes(sf |
    ccb.views = a and a.isKindOf(Attribute)))
```

The equivalence of component and class models is concretely demonstrated by the component diagram illustrated in figure 9 which is semantically equivalent to the vending machine shown in figure 6. Those attributes in figure 6 which are shown

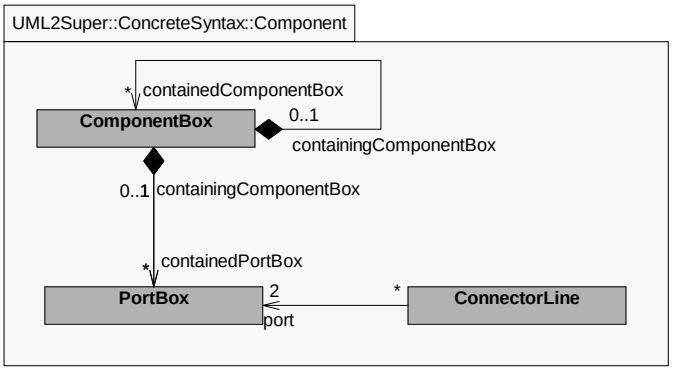

Figure 7: Component diagram concrete syntax

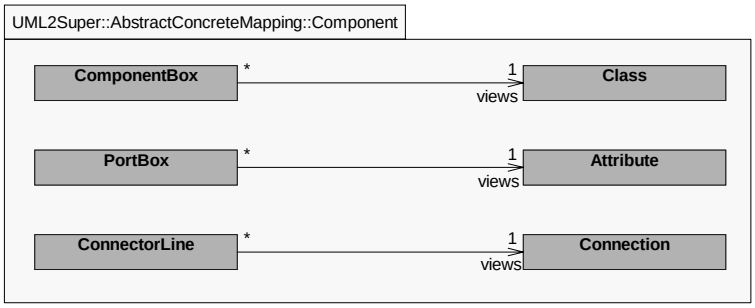

Figure 8: Component diagram abstract syntax mapping

as ports in figure 9 have their boolean *connectable* value set to false (not concretely rendered).

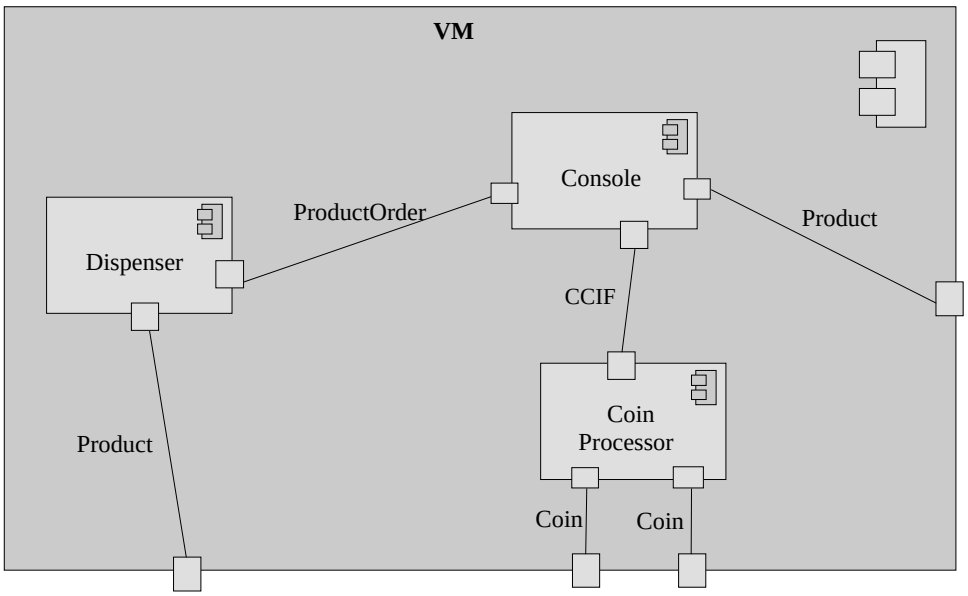

Figure 9: Component diagram of the vending machine

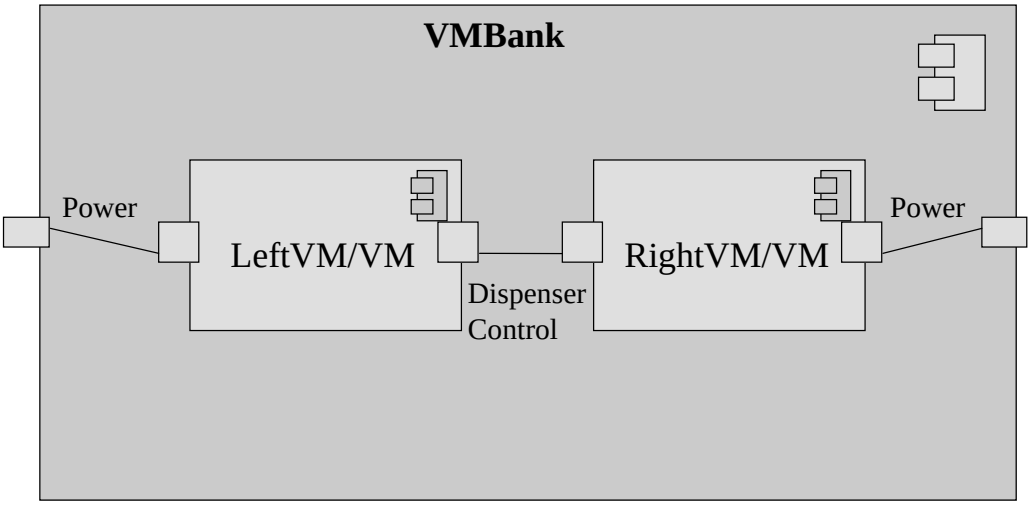

Figure 10: A Scenario of the vending machine's component diagram

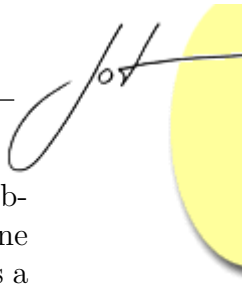

Figure 10 presents an example of the usefulness of the *role* mechanism for describing scenarios. In this, a bank of two vending machines is described, each machine being a particular scenario of the original machine. Figures 9 and 10 demonstrates a powerful aspect of the recursive treatment of *role*s, which is that the same concrete syntax can be used to describe both types and *role*s of types since they are the same model element (the basic type is a *role* whose *roleOwner* is self).

3 BEHAVIOUR

In our definition of behaviour, a single abstract syntax model is used as a basis for use case, collaboration, state machines, activity diagrams and sequence diagrams. In this paper we deal only with state machines and activity diagrams.

Abstract Syntax

The abstract syntax of behaviour illustrated in figure 11 can be broadly considered in two parts. The first part is a hierarchy of actions (which is made explicit in the lower half of figure 11) some of which have sub-actions. The second part is relations between actions such as *Flow* and *Transition*.

At first sight, treating *State* as a type of *Action* may seem unconventional since the conventional interpretation is that it represents a static property of a system (i.e. the value of slots and objects) at a particular point in time. Our approach extends this property with dynamic aspects by defining a state to be an action for two reasons:

1. In UML, state machine states have dynamic properties. For example, a state *state* specifies an entry action, exit action and may continuously execute a do action until the exit action is invoked. Thus, it is natural and convenient to treat a state as an action.

2. State machines and activity diagrams share many concepts (for example *join* and *fork*) the our superstructure proposal provides a common abstract syntax model for both these and other behavioural notations. This unification is leveraged by treating state machine states as actions.

The formal definition of the semantics of behaviour is outside of the scope of this paper. However, informally the semantics is built around the ability to pre-empt execution. Each *Action* has a boolean flag which denotes whether it can be prematurely terminated. When a pre-emptable action is related by a type of flow, the action does not need to complete its execution before the flow can be executed. A transition's effect *Action* cannot be pre-empted, an additional constraint captures this:

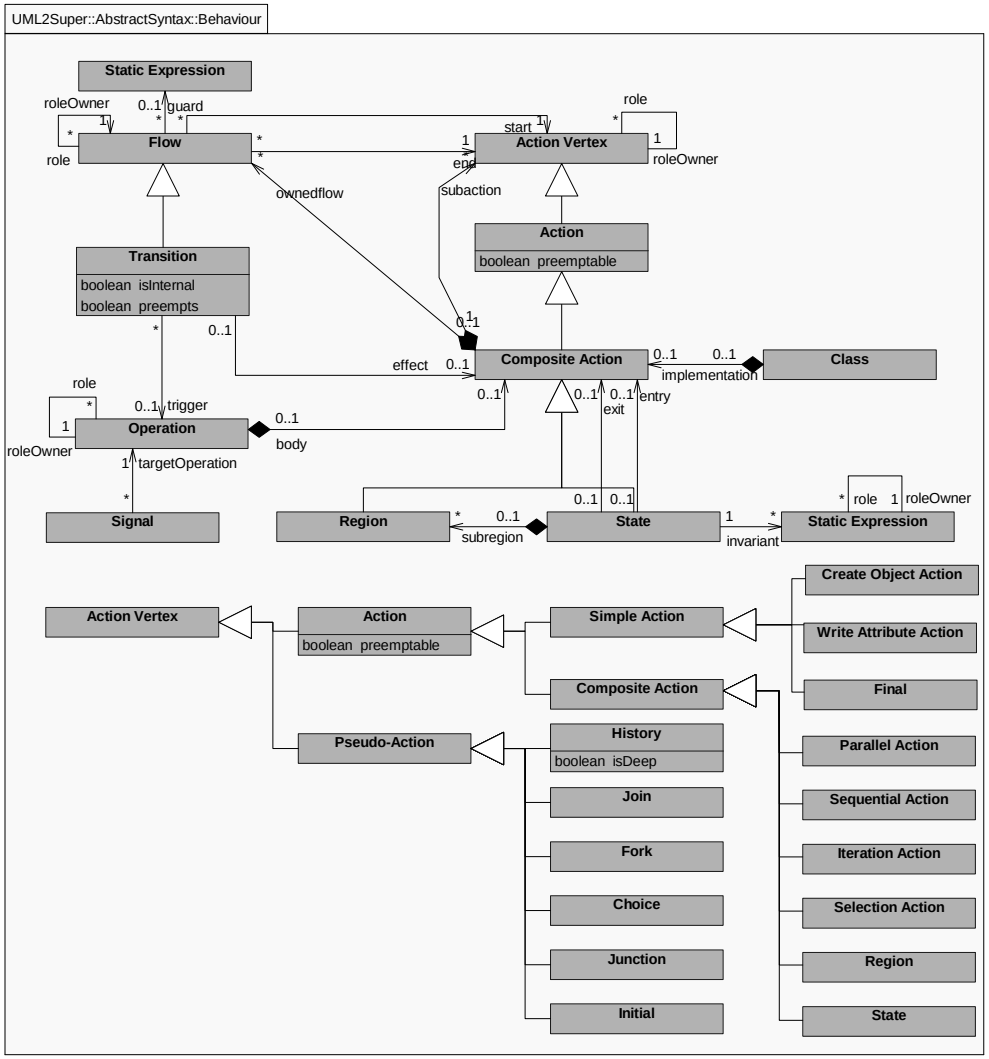

Figure 11: Behaviour abstract syntax

```
context Transition inv:
  self.effect.preemptable = false
```

As with the definition of structure, figure 11 includes a generalised version of UML 1.x's role mechanism. This enables scenarios to be described over any behavioural model based on the abstract syntax.

State machine diagram

The concrete syntax for state machines is shown in figure 12 and the associated abstract syntax mapping in figure 13. A top level state machine *State* is owned by a *Class*, only *Region*s can own sub-*State*s:

```
context CompositeAction inv:
  self.subaction->forAll(sa | sa.isKindOf(State) implies
    self.isKindOf(Region)
```

A *State* has *invariant*s which must hold true during the lifetime of its execution. The nesting of *State*s results in the propagation of *invariant*s down the state hierarchy:

```
context State inv:
  self.subregion->forAll(sr | sr.subaction->forAll(sa | sa.isKindOf(State)
    implies (self.invariant->forAll(i | sa->includes(i)))))
```

A *Transition* is semantically a type of guarded *Flow* (the guard is denoted by the *trigger* in figure 11). Because of the pre-emption semantics, *State*s are not concerned with checking whether a *Transition* can fire. It is the responsibility of the *Transition* to ensure that when it fires that source *State*s are terminated. The *entry* and *exit* actions of *State*s are not pre-emptable:

```
context State inv:
  self.entry.preemptable = false and self.exit.preemptable = false
```

Therefore when a *Transition* pre-empts a *State*, the *entry* and *exit* actions will always execute.

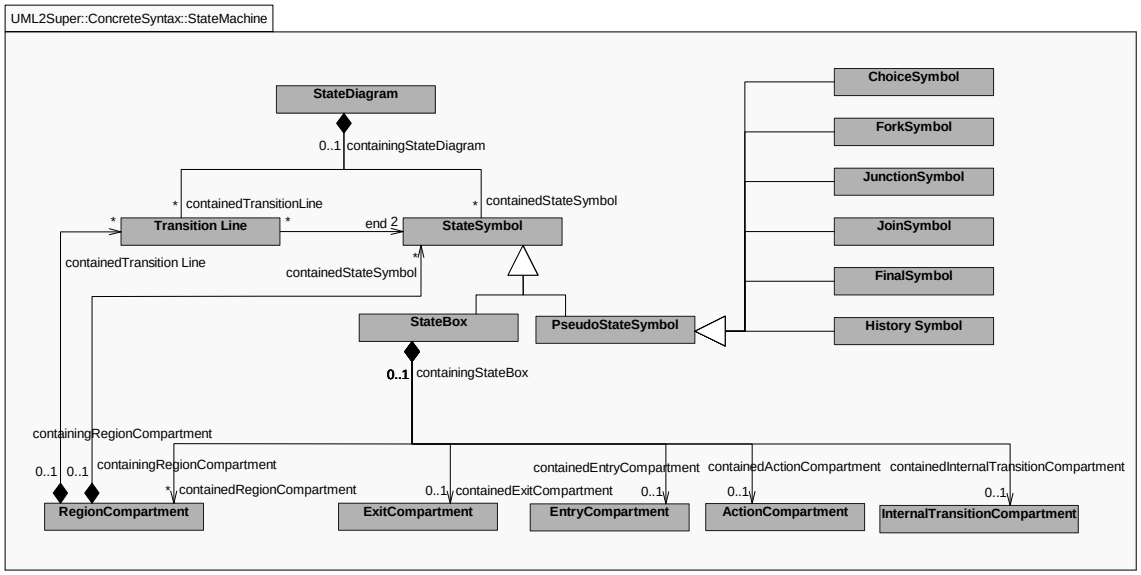

Figure 12: State machine concrete syntax

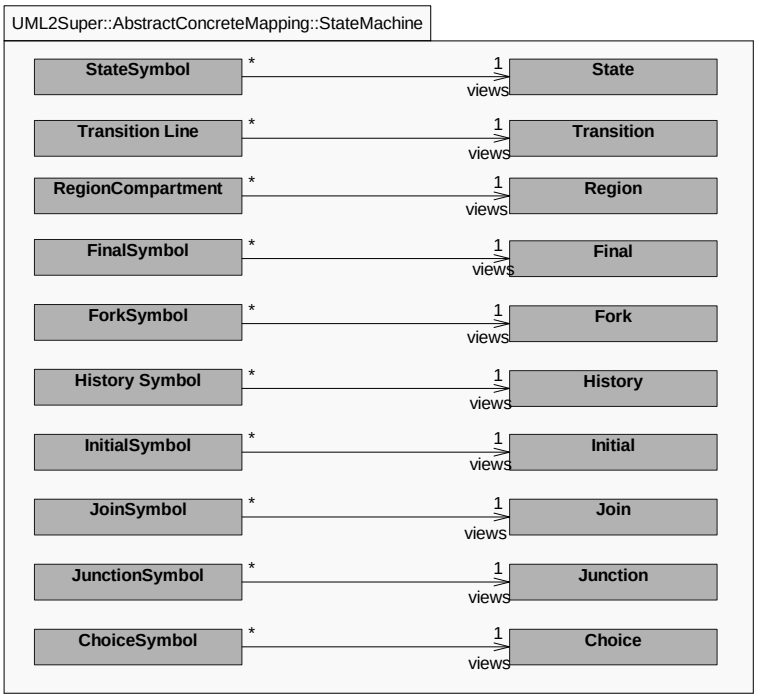

Figure 13: State machine abstract syntax mapping

In the current model there is an ingredient of non-determinism. A *Transition* can occur when the *Transition*'s guard is enabled, but there is no semantic which *ensures* a *Transition* will definitely occur when enabled. This is consistent with UML 1.x.

Figure 14 describes the behaviour of the *Dispenser* component of a vending machine as an instance of state machine concrete syntax definition. This serves to demonstrate the conventional consideration of state machines in the presented definition.

Activity diagram

Activity diagrams share many common features with state machine diagram. Activity state and state machine state are semantically equivalent, transitions are a kind of flow with some added detail (see figure 11) and both *fork* and *join* actions are common to both types of diagrams. The homogenous nature of state machine diagrams and activity diagrams is reflected in the mapping of activity diagram concrete syntax (figure 15) as shown in figure 16 with concepts shared by the state machine concrete syntax (figure 12).

Figure 17 illustrates the use of the activity diagram in describing the behaviour of the vending machine. Unlike component diagrams and class diagrams it does not make sense to show the same model viewed by these alternative notations, during the design of systems they are usually used mutually exclusively. What figures 14

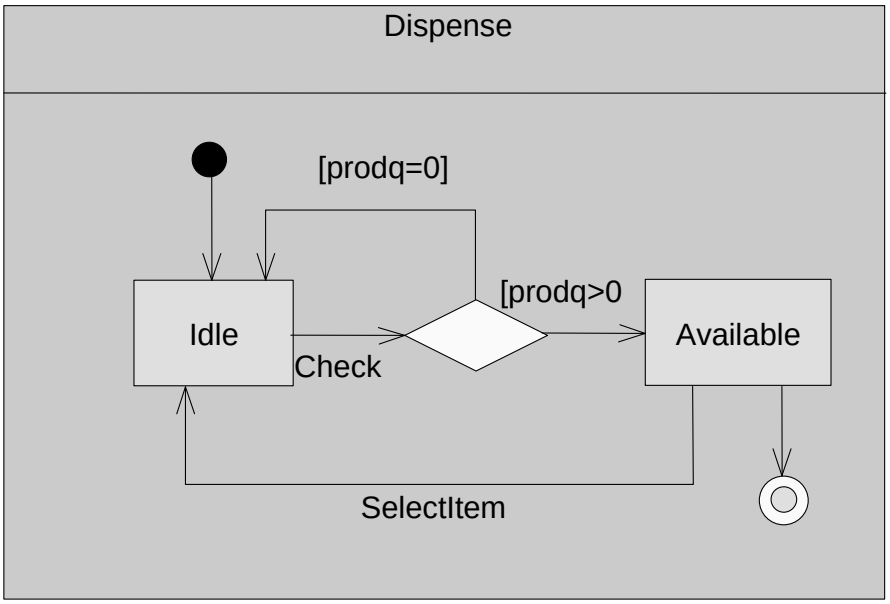

Figure 14: State machine of the vending machine

and 17 do demonstrate is that the sharing of semantics between state machines and activity diagrams. For example, the *Junction* symbol in figure 17 has precisely the same underlying abstract syntax and semantics as that in the state machine of figure 14.

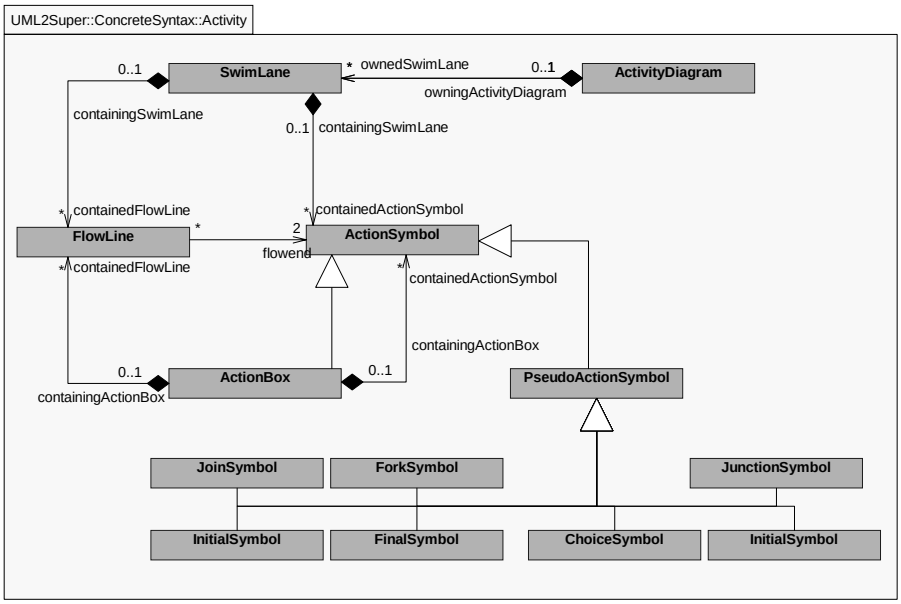

Figure 15: Activity diagram concrete syntax

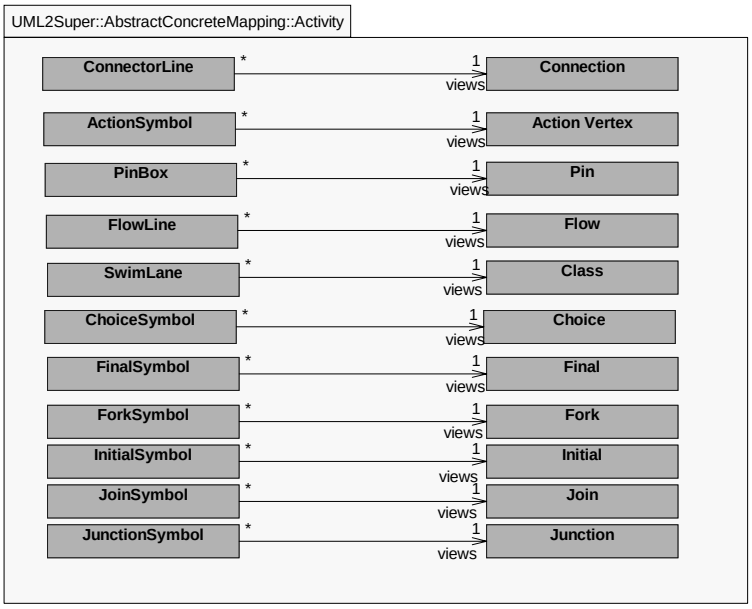

Figure 16: Activity diagram abstract syntax mapping

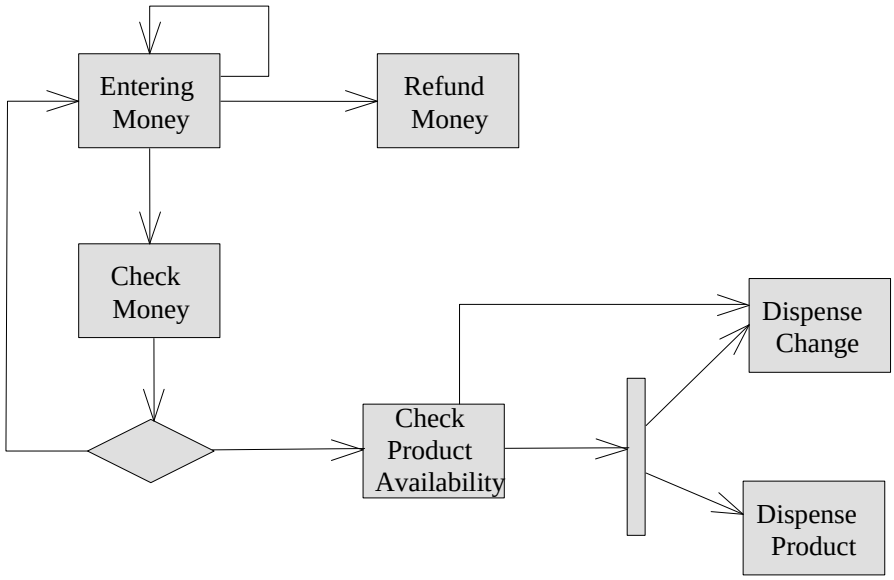

Figure 17: Activity diagram of the vending machine

4 CONCLUSION

The next generation of UML is an important step in enabling model oriented software engineering. In order for it to cater for diverse requirements UML 2 must contain new notations not currently found in UML 1.x. We have argued that this does not necessarily mean introducing fundamentally new abstractions (new semantics). Instead the new abstractions can be considered as new syntax which share a semantic underpinning with existing notations. The abstract syntax presented here is substantially smaller than UML 1.x even though it supports the new notations required by UML 2.0. The models described in this paper have been implemented by Tata Consultancy Services (TCS) in their Mastercraft tool. The vending machine model shown throughout the paper have been modelled using Mastercraft, adding weight to the presented approach.

Because of the unification of abstract syntax, and interesting aspect of the presented approach is that new concrete syntaxes can be formed consisting of a hybrid of the distinct concrete syntaxes. For instance, it is possible to have a behavioural notation that combines aspects of state machines, activity diagrams and sequence diagrams. We intend to explore this avenue as future work.

Acknowledgments

We are grateful to Sreedhar Reedy and R. Venkatesh of TCS India who have over seen the tool implementation of the described models. We are also grateful to TCS and BAe Systems who have generously funded the research described here.

REFERENCES

[1] 2U. 2nd revised submission to OMG RFP unified modeling language superstructure version 2.0. http://www.2uworks.org/uml2submission/. ad/02-12-23).

[2] 2U. 3rd revised submission to OMG RFP unified modeling language infrastructure version 2.0. http://www.2uworks.org/uml2submission/. ad/03-01-08).

[3] Object Management Group. Unified modeling language specification version 1.4. http://www.omg.org. ad/01-09-67.

[4] U2 Partners. 2nd revised submission to OMG RFP unified modeling language superstructure version 2.0. http://www.u2-partners.org/artifacts.htm. ad/03-01-01).

[5] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language Reference Manual*. Addison Wesley, first edition, 1999.

About the authors

Andy Evans is managing director of Xactium Limited, a company that provides advanced modelling tools and services to the software development industry. He was previously a lecturer at York University, where he was heavily involved in modelling research and industrial consultancy. He was programme chair of UML2000 and is an associate editor of software and systems modelling journal (SOSYM).

Paul Sammut is a senior consultant at Xactium Ltd., a company providing tools, technology and solutions for Model Driven Development. Prior to joining Xactium, he was a Research Associate at the Department of Computer Science, University of York (UK), where his key research interests were model driven development, and tools and languages for metamodelling. Much of this work has been published in journals and conference proceedings, and recently he has co-authored a book on Applied Metamodelling.

James S. Willans holds a PhD from the University of York where he was until recently a Research Associate pursuing interests included metamodelling languages and tool support for language engineering. He now works as senior consultant at Xactium (www.xactium.com), a company providing tools and consultancy for model-driven development, and is co-author of a book on Applied Metamodelling.

Alan Moore is VP for ARTiSAN Software Tools Ltd. Alan has 18 years of experience in the development of real-time and object-oriented methodologies, and their application in a variety of problem domains. He is responsible for the development and evolution of Real-time Perspective, ARTiSAN's process for real-time systems development. He is an active member of the Object Management Group and chaired both the finalisation and revision task forces for the UML Profile for Schedulability and Performance and Time, and is the co-chair of the OMGs Real-time Analysis and Design Working Group. He is ARTiSANs technical lead in the SysML consortium.

Girish Maskeri is a scientist at Tata Resesarch Development and Design Center, Pune, India. His research interests include metamodelling, Model transformation and Component based engineering.