

METHODS OF BEHAVIOUR MODELLING

A Commentary on Behaviour Modelling Techniques for MDA

Ashley McNeile
Metamaxim Ltd.
Email: ashley.mcneile@metamaxim.com

Nicholas Simons
Metamaxim Ltd.
Email: nick.simons@metamaxim.com

Keywords: MDA, Behavioural Modelling, Code Generation, Design By Contract, Protocols, State Machines

Abstract: The vision of MDA is to decouple the way that application systems are defined from their deployment platform, thereby ensuring that the investments made in building systems can be preserved even when the underlying technology platforms change. Achieving this vision requires that models are rich enough to capture and represent the behaviour of the application. However, the UML has many ways of representing behaviour and there is little consensus on which technique or techniques should be used in the context of MDA. To address this, we present a classification of techniques and a commentary on the strengths and weaknesses of each.

1. INTRODUCTION

We take as our starting point the idea that MDA requires that the models at the PIM level need to capture and represent the behaviour of the modelled application. The OMG characterise MDA as follows:

“Fully-specified platform-independent models (including behaviour) can enable intellectual property to move away from technology-specific code, helping to insulate business applications from technology evolution

and further enable interoperability” [1]. (The underlining is ours.)

Moreover, key architects of the MDA vision talk of the need to be able to execute and test an MDA model. Richard Soley, CEO of OMG, says that one of the aims of MDA is that *“Models are testable and simulatable”* [2]. Oliver Sims, a member of various OMG Task Forces who served for several years on the OMG Architecture Board, says that *“The aim*

[of MDA] is to build computationally complete PIMs” [3]. As Oliver Sims points out, the term “computationally complete” means capable of execution.

Executability of a model, whether by interpretation or code generation, is only possible if behaviour is represented but when it comes to how behaviour should be represented the vision fragments. While there is general agreement in the MDA community that behaviour modelling is essential, or at least that it will be hard to meet the goals of MDA without it, there are widely differing views on how behaviour should be represented in a model. These different views have their origins in different methods of behaviour modelling, which pre-date MDA, and sometimes even pre-date the UML. Proponents of these various methods claim the ability to meet the aims of MDA, have therefore adopted MDA branding.

MDA has benefited from the infusion of expertise and experience that these imports embody, as they provide a wealth of intellectual raw material. However, they represent a wide range of ideas and techniques for modelling behaviour so have not served to forge a unified view of how behaviour modelling for MDA should be done.

While a considerable amount has been published on the individual approaches by their proponents, for instance in [5] and [9], there has

not been much work to try and provide an overall framework for examining the different approaches side-by-side with a view to comparing and contrasting them.

2. THE PURPOSE OF THIS PAPER

This paper is an attempt to step back and examine the different approaches to behaviour modelling, with a view to gaining a better understanding of the power and limitations of each. Ultimately, the purpose is to initiate a debate that will arrive at a better understanding of the type or types of behaviour modelling that are best suited to delivering the vision of MDA.

The authors are not experts in all the techniques presented. This paper is intended to initiate and contribute to the debate rather than propose or present any kind of closing position.

Finally, the authors do not claim to be disinterested in the debate, as we are engaged in tool development work for one the approaches discussed – the one we term Event Based. Our intention is that this interest has not compromised the quality of the ideas put forward.

3. THE PURPOSE OF A MODEL

Traditionally, the principal uses of models are:

- Exploration – to explore different solutions to an application development problem and determine the best
- Communication – to make it easy to communicate the requirements and design to others
- Validation - to walk through required scenarios and demonstrate that the proposed solution works
- Implementation – to provide a blueprint from which the implementation (code) can be constructed

Implicit in this is the idea that a “model” is a description that is readable, understandable and easy to change. This requires that models observe separation of concerns and are constructed at high levels of abstraction.

In the context of MDA the following additional aims may be added:

- Investment Protection – to make the intellectual and financial investment embodied in an application solution immune to changes in deployment platform
- Simulation – to support deeper early validation of models by direct execution or simulation of their behaviour
- Automatic Implementation – to improve development productivity by generating final code from the model

However, models used in the context of MDA should still have the key qualities of being

readable, understandable and easy to change. Part of the challenge of the MDA venture is devising modelling languages that have the expressive power needed for MDA without compromising these qualities. In other words the MDA focus on executability and code generation must not cause us to lose sight of the need for modelling languages to support and encourage separation of concerns and description at a high level of abstraction.

4. STRUCTURAL AND BEHAVIOURAL MODELS

UML divides the universe of modelling artefacts into Structural and Behavioural Models as follows:

- Structural: Class Diagram, Implementation Diagram
- Behavioural: Use Case Diagram, Interaction Diagram (Sequence and Collaboration Diagrams), State Machine Diagram

We will use this distinction between Structural and Behavioural models in this paper.

5. AN ILLUSION OF PROGRESS?

It is generally agreed at an informal level that the code of an application, whether produced using a model driven approach or not, can be divided into:

- Infrastructure code
- Business logic

Infrastructure code is generally concerned with implementing the chosen architecture of the system: handling object distribution, object relational mapping, transaction concurrency control, recovery after failure, etc.

Business logic, on the other hand, is concerned with the implementation of the users' functional requirements for the system: updating an account balance when a deposit takes place, determining whether or not a withdraw can take place based on the account's balance, etc.

A large amount of the infrastructure code for an application can be generated from the Structural (Class) Model, even if no Behavioural Model has been created. Well before the articulation of MDA, a number of tool vendors had exploited this fact by providing code generators that create "code skeletons" of infrastructure code from Class Models. The programmer would then add the business logic to these skeletons.

There is no doubt that this kind of code generation is successful and valuable. But the success of this approach has created an illusion of progress towards MDA. There is a temptation to argue as follows:

- Code generation from models is one of the results that can be expected from MDA
- Some current tools generate code from Structural Models

- Therefore these tools implement MDA

If we take the vision of MDA articulated in the Introduction seriously this argument is specious. Generating code from a Structural Model does not represent progress towards capturing the business rules, logic or behaviour of an application.

6. STRUCTURE CENTRIC

Those who have been working on the generation of infrastructure code from Structural Models have been able to take their work a step further.

As well as generating the architectural infrastructure code it is not too hard to generate a complete working application that supports CRUD (Create, Read, Update, Delete) behaviour. This yields a usable application, including user interface and persistency, that allows instances of the object and relationship types represented in the model to be instantiated, amended and deleted. The generated application can also include code to check that structural constraints, such as the cardinality on relationships, are not broken.

This is impressive, as a complete application is generated from a model, and begins to look like real progress towards the MDA dream of applications created directly from models.

But arguably it is still not MDA. The Structural (Class) Model that is the source of the code generation has no behavioural semantics, not even CRUD, so the behaviour implemented by such tools is, in the absence of any other information, only a default. In any but the very simplest application this default behaviour has to be supplemented or replaced in the PSM, or in the generated code, by adding hand-written code that implements the required behaviour. This will normally require changes to the user interface and perhaps the persistency layer as well as the business objects. The source PIM is only structural, and it is hard to see how a PIM without any behaviour can act as the repository of intellectual and financial investment made in an application.

The underlying issue is that there is a gap between the expressive power of structural models and the potential complexity of the behavioural requirements that need to be expressed. One line of research is investigating whether this gap can be closed, or at least sufficiently reduced, by using a formal language, the Object Constraint Language (OCL), to add behavioural requirements to a structural model. We characterise this line of research as “Contract Based” and it is discussed further in Section 8.2. For reasons discussed there we believe that this approach is limited as a behaviour modelling paradigm for MDA.

Modelling approaches based on Class Diagrams, possibly augmented by contracts specified in OCL, we term “Structure Centric”.

7. STATE-MACHINE CENTRIC

A completely separate line of work stems from work using state-machine based models to describe behaviour. We characterise this line of research as “State-Machine Centric”. The common theme of these approaches is to use state-transition diagrams to describe the lifecycles of the objects in the model.

The use of state-machines to specify behaviour means that there is no gap between expressive power and behavioural complexity. An imperative language, referred to as an Action Language, is used to specify the processing to be performed. The state-machines and adorning action language statements are enough to support generation of a complete application. It is therefore possible to build tools that generate complete and behaviourally complex applications from suitably constructed models. Such tools, some of which embody complex configurable code generation and optimisation, are sometimes called “model compilers”.

Within the State-Machine Centric school there are two distinct approaches.

The first approach takes states as its modelling focus and views the transitions that

move the state-machine from state to state as secondary. This approach is most widely used in the real-time/embedded systems domain and the best known example is probably the Shlaer-Mellor Method, developed in the 1970s [4]. The Shlaer-Mellor approach has recently been repositioned as an MDA approach under the name “Executable UML” [5]. We characterise this approach as “State Based” and it is discussed further in Section 8.3.

The second approach takes events as its modelling focus, and views the states as secondary. The Event Based approach has its roots in the work of Jackson et al. in the JSD method developed in the 1970s and 1980s [12]. The authors have developed these ideas as an MDA approach and built supporting tools. We characterise this approach as “Event Based” and it is discussed further in Section 8.4.

Because of their ability to capture arbitrarily complex behaviour at a high level of abstraction our view is that only these State-Machine Centric approaches qualify as capable of fully supporting the MDA vision of Platform Independent Models that capture application behaviour.

8. A TAXONOMY OF METHODS

For the purpose of further analysis we propose the following taxonomy of behaviour modelling methods:

- Interaction Based
- Contract Based (Structure Centric)
- State Based (State-Machine Centric)
- Event Based (State-Machine Centric)

The last three relate to the discussion above (Structure or State-Machine Centric) as indicated. The first is included for completeness but has little support as a serious contender in the MDA arena.

Each of the four is described and considered below. For each, we focus on the extent to which they can support the MDA vision of generating a behaviourally complete application from a model and the extent to which they support the aim of modelling at a high level of abstraction.

8.1 Interaction Based

The traditionally favoured approach to modelling behaviour in UML has been the use of Interaction Diagrams (Sequence and Collaboration Diagrams). In the context of MDA it makes sense to ask whether these provide a suitable basis for behaviour definition and can therefore continue to be used.

A key theoretical question is whether Interaction Diagrams are “extensional” or “intentional”. The former term denotes specification of behaviour by enumerating the set of possible traces, whereas the latter denotes the specification of an abstract machine that

induces a set of traces. This difference is important in this context as code-generation only makes sense from an intentional model. Bernhard Rumpe [6] also makes this point.

In early versions of UML, Interaction Diagrams were clearly extensional as they did not have the expressive power to be otherwise. However, as features have been added allowing the specification of iterative and selective logic to increase their expressive power, the position has become less clear. To quote Alan Moore (VP, Artisan Software) “[*There is*]confusion, even among UML2 experts, about whether interactions are intentional or extensional.” [7]. Our view is that it is premature to consider the use of Interaction Diagrams for behavioural code generation while this confusion persists and fruitless to consider it unless the resolution of this confusion sees Interaction Diagrams repositioned as intentional behaviour specification mechanisms.

Nevertheless, some work has been done towards generating code from Interaction Diagrams. Engels et al. [8] describe an approach based on Collaboration Diagrams. Their assumed starting point is multiple diagrams, built assuming an extensional use of the notation, that first have to be combined manually into a single diagram, in order to yield an intentional behaviour specification. This has to be done by hand. Once this is done some

degree of code generation, albeit incomplete, can then be achieved.

The manual merging described by Engels et al. is a consequence of the extensional nature of the starting point. We doubt whether an approach that requires this degree of manual work could be considered a satisfactory basis for MDA.

8.2 Contract Based

The Contract Based approach to behaviour specification has its origins in the ideas of “Design by Contract” as espoused by Bertrand Meyer et al. Rather than being based on the use of a specific behavioural model, such as Interaction or State-Machine Diagrams, this approach relies on adorning the Class (Structural) Model with pre- and post-conditions on the operations in each class. These specify, respectively, the conditions under which a legitimate execution of the operation can be made and the results that are guaranteed to pertain after a legitimate invocation has completed. These represent a “contract” in the sense that, if the pre-condition is satisfied, the implementation must ensure that the post-condition is satisfied.

Within the MDA context the approach is characterised by the notion that “*The dynamics [behaviour] of a system can be expressed by pre- and post- conditions on operations.*” [9, page 36]. The operations are identified in the

PIM as part of the UML Class Model, and the pre- and post-conditions are specified using the OCL [10].

Because behaviour is specified by adorning the structural (Class) model, rather than using a separate behavioural modelling formalism, such as state-machines, we classify this approach as Structure Centric. This approach represents a possible way for those who favour Structure Centric MDA, as described in Section 6, to move beyond simple CRUD behaviour.

Advocates of this approach do not claim that complete code generation is possible. *“For relatively simple operations the body of the corresponding operation might be generated from the post-condition, but most of the time the body of the operation must be written in the PSM”* [9, page 36]. Neither do they claim that the behaviour of an application can or should be completely specified: *“...the dynamics of the system still cannot be fully specified in the UML-OCL combination.”* [9, page 36].

Nevertheless the approach does potentially meet the MDA requirement of capturing at least some aspects of behaviour at the PIM level. Two issues, although perhaps not “show stoppers”, may limit the appeal of Contract Based descriptions as a general technique for modelling behaviour. These two issues are described below.

8.2.1 Model Fragmentation

The use of contracts as a means of specifying behaviour entails a degree of fragmentation of the behavioural description making it less easy to understand and maintain than an equivalent diagrammatic behavioural description.

Consider an object with behaviour described in the state diagram shown in Figure 1.

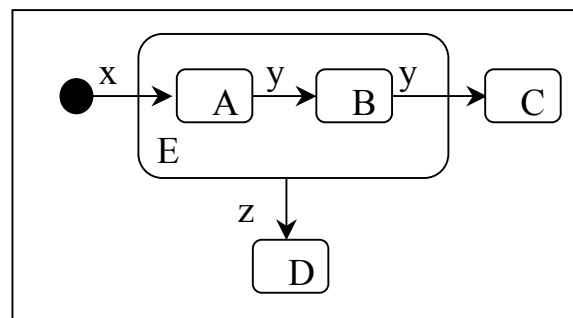


Figure 1. State Diagram Object Description

This behaviour can also be specified in contracts, using pre- and post-conditions on the transitions (events), as shown in Figure 2.

Event x: Pre: - Post: new instance with p(A)	Event z: Pre: p(E) Post: p(D)
Event y: Pre: p(E) Post: if (p(B)@pre) p(C) else p(B)	Derivation: p(E) = = p(A) p(B)
Notes: <ol style="list-style-type: none"> 1. p(A) is true when and only when in state A, etc. 2. @pre denotes evaluation before the event has been processed. 3. The Derivation entry describes how the predicate for the super-state E is derived from those of its sub-states, A and B. 	

Figure 2. Contract Based Description

The set of contracts in Figure 2 is equivalent to the state diagram but it is less readable and therefore less easy to amend safely.

For instance, suppose it is decided that the transition z to state D can take place from state C as well as from A and B. On the diagram this is easily achieved by extending the box representing super-state E to include state C.

In the Contract Based description, the equivalent change would be to alter the definition of the derivation rule defining the super-state E to read:

$$p(E) = p(A) \parallel p(B) \parallel p(C)$$

However this has the, probably unwanted, side-effect of allowing a new transition for event y from C back to B.

This illustrates a potential danger. Because the definition of the object's behaviour is distributed across the contract definitions for each event there is no way in general of ensuring that changes made to the contracts have coherent meaning or representation at a higher level of abstraction. This approach seems to run counter to one of the stated objectives of modelling, namely keeping the level of abstraction as high as possible.

8.2.2 Abstraction Atomicity

The second issue concerns the level of abstraction at which contracts are defined.

Contracts work best where there is a distinct difference between the level of abstraction at which the contract is specified and the level of abstraction required for implementation of the contract. The classic example is a sort algorithm. Here the contract is formed by stating in the post-condition that the output must be ordered. The implementation is achieved by using a procedure that re-arranges the input to achieve ordering. There are a number of different procedural algorithms that may be used, with different characteristics in terms of speed, use of memory, etc. However, they all produce a result that meets the post-condition. Here the implementation (procedural algorithm) is at a lower level of abstraction than the contractual post-condition (non-procedural definition of the required result).

In the case of contracts that are used to specify behaviour this differentiation between levels of abstraction is not always present. Consider the case shown in Figure 3 which represents the post-condition on processing a deposit to an account in a very simple banking system.

<p>Deposit: Pre: Account is open Post: this.balance = = this.balance@pre + input.amount</p>

Figure 3. Contract for a Deposit Event

The business logic required to implement the contract's post-condition could be just a single statement adding the input amount to the balance. Although, of course, the syntax of this programming statement will be different from the post-condition above, arguably the level of abstraction is the same. We call this "abstraction atomicity" as the contract has already reached the atomic level of abstraction and cannot be refined further. This is in contrast with the "sort" example cited earlier where a lower level of abstraction than the contract is required for the implementation.

If the modeller/developer is required to create the contract and then, separately but at the same level of abstraction, write the implementation then something is wrong. The same work is being done twice.

The duplication can be avoided if the implementation can be generated from the contract (in particular, from the post-condition) and, as noted earlier, Kleppe et al. [9, page 36] mention this as a possibility. The cases where such generation is possible are exactly those cases where the level of abstraction of the contract and of the implementation coincide, and for these, OCL would become a specialist programming language. However not all implementation code be generated from contracts and some would remain to be written by hand using a general purpose programming language. The use of two languages in this way

is likely to be a source of complaint: "Why do I need to learn two programming languages to do MDA, when clearly one would suffice?"

Finally, it is worth asking whether abstraction atomicity is inevitable or whether it is a symptom of poor practice. Perhaps it can be avoided by raising the focus to elevate the abstraction level at which contracts are constructed? This does not seem likely. When dealing with event driven systems the natural focus for contract construction is the event level. If the handling of an event is trivial, involving simple object creation and attribute update, abstraction atomicity seems unavoidable. Creating contracts that span multiple events might allow contracts to be constructed at a higher level of abstraction but there is no reason to suppose that this can be done in general.

8.3 State Based

This and the following (Event Based) approaches are both based on the notion that behaviour can be captured by representing the lifecycles of objects as state-machines. The key difference between the State Based and the Event Based approaches is whether the life cycle analysis focuses on the states of the object or on the events. This has some interesting consequences for the form that state-transition models take.

Both approaches have the ability to support the aims of MDA. But, as far as we know this

paper is the first to articulate the two as separate state-machine modelling paradigms and attempt to compare and contrast them. It may be that the “ultimate” approach is some kind of combination of the two but it is not possible to say today what such a combination would look like. For now, we present them as alternatives.

This section describes the State Based approach, the next section describes the Event Based approach. For ease of comparison both sections follow the same structure. Our aim is to highlight the differences, so the headings used in the two sections have been chosen to focus on aspects where the two approaches differ.

8.3.1 Summary of the State-Based approach

The table below summarises some key aspects of state-machine modelling as used in the State Based approach. Each feature is described in more detail below.

Aspect	Description
Focus	Identification of states of an object.
State-Machine Type	“Behavioural”
Orthogonal state spaces	An object only has a single state machine.
Re-use of behaviour definitions.	Not supported.

Table 1. Summary of State Based Approach

8.3.2 Focus

In the State Based approach the focus of modelling an object’s lifecycle is the identification of the object states, where a state is characterised as “*a situation or condition of the object in which certain physical laws, rules, and policies apply*” [4, page 5].

The modelling of an object lifecycle centres on identification of the relevant states based on an understanding of the domain and the processing requirements. Transitions, fired by events, are added to the state-machine to drive it from one state to another. An event is defined as “*an abstraction of an incident or signal in the real world that tells us that something is moving to a new state*” [4, page 42].

8.3.3 State Machine Type

The UML version 2 standard has introduced a distinction between two types of state-machine: “Behavioural” and “Protocol” [11, page 455]. The purpose of the distinction is to identify two different semantics of state-machines, as set out below.

A “Behavioural” state-machine is intended to show different behavioural states of an object, and provide a home for the processing associated with these different states. Transitions, triggered by events, move the object from state to state. Events that do not cause a change of state do not appear on the state

transition diagram¹ and are ignored by the state-machine. Thus the presence of a transition for an event denotes only whether or not the state-machine reacts to the event, not whether the event is meaningful or allowed.

A “Protocol” state-machine, on the other hand, defines what orderings of events are possible in the lifecycle and what orderings are not. If the object is in some state, lack of a transition for an event from that state means that the event is not possible or meaningful as a next event in the lifecycle of the object and the software owning the object must either prevent the event from happening or handle it as an error.

The above can be summarised as follows: Behavioural state-machines are purely “reactive” – presented with an event, this type of state-machine may or may not transition from its current state to a new one, but will not prevent the occurrence of the event. A Protocol state-machine, on the other hand, is “proactive” – it determines the interactions that may occur with it in the sense that successful delivery of an event to an object is contingent on prior consultation with the object’s state-machine to check that the event is allowed. This type of state-machine must therefore interact with its

environment in some way to ensure that its protocol rules are observed.

The state-machine semantics used in the State Based approach corresponds to Behavioural. It should be noted that describing the behaviour of an object this way does not mean that the object does not have a protocol and therefore that any possible ordering of its lifecycle events is meaningful. It only means that the protocol (meaningful or allowed orderings of events) is not described by the state-machine and must therefore be described elsewhere using other means. Typically the protocol is described as business rules or validation logic, separate from the state-machine, that checks the state of an object before a transaction (event) is processed and either allows it to go ahead or rejects it with an error message.

8.3.4 Orthogonal State-Spaces

Not all objects have behaviour that can be represented in a single state transition diagram in a simple way. Consider, for instance, modelling a Person who can be working or unemployed and single or married. This object has two state-spaces, one concerned with employment status and one concerned with marital status. The two state-spaces are “orthogonal” meaning that transitions between the states of one state-space happen independently of transitions between the states of the other.

¹ A transition that does not change state may be included because of the need to execute actions (Action Language code) when a firing event occurs. However this has nothing to do with whether the event is allowed or not.

In the context of State Based modelling, the underlying assumption is that an object has a single state space. “*A real-world thing is in exactly one stage of its behaviour pattern at any given time.*” [4, page 34]. Faced with an example such as the one described above, a modeller using the State Based approach has three possibilities:

1. Select one or other of the two state-spaces as the basis of the object lifecycle and ignore the other one (handle it as attribute values rather than states).
2. Use the Cartesian product of the state-spaces (i.e., working and married, working and single, unemployed and married, etc.) as the basis for the lifecycle.
3. Model Person as two separate objects, with two different state-machines.

Which of the three is chosen depends on the modeller’s view of what is necessary to capture and describe the “*situations or conditions of the object in which certain physical laws, rules, and policies apply*”. If the view is that both employment and marital states must be represented the choice is between (2) and (3).

8.3.5 Re-use of behaviour Definitions

A theme of modern OO paradigms is to allow re-use of property definitions across classes that are similar. In particular, the inheritance mechanisms allows a sub-class to

inherit, and thus re-use, data and method (operation) definitions from super-classes. It is legitimate to ask, in the context of state-machine based behaviour modelling, whether state-machine definitions can be similarly re-used.

In the State Based approach there is no mechanism for re-use of state-machine behaviour across different class definitions except in the case that two classes have identical state-machines. There is no concept of being able to refine state-machine definitions as you move down a class hierarchy and state-machines are therefore defined only at the lowest level [5, page 227]². So if two different types of bank account are being defined with slightly different behaviours, each would need to be given its own, entirely separate and complete, state-machine. It is not possible to define the common elements of behaviour in a common abstract class and refine this differently for the two specific types of account.

8.4 Event Based

This section describes the Event Based approach, which also uses state-machines to describe object lifecycles. The same structure is used as the previous section to enable contrast of the two approaches.

² The original Shlaer-Mellor account [4] does describe a technique called “splicing” that allows refinement of state-machines. However, this appears to have been dropped the Mellor-Balcer book [5].

Although somewhat different in focus and its use of the state-machine formalism the aim of the Event Based approach is identical to that of the State Based, namely the formulation of complete description of arbitrarily complex behaviour. As in the State Based approach the state-machine definitions can be adorned with actions and used to yield an executable system.

8.4.1 Summary of the Event Based approach

The table below summarises some key aspects of state-machine modelling as used in the Event Based approach. Each feature is described in more detail below.

Aspect	Description
Focus	Identification of the events and event protocol of an object.
State-Machine Type	“Protocol”
Orthogonal state spaces	An object can have multiple state-machines.
Re-use of behaviour definitions.	State-machines can be re-used across objects.

Table 2. Summary of Event Based Approach

8.4.2 Focus

The focus in the Event Based approach is to define the valid lifecycles of the objects, where a lifecycle is defined as a sequence of events. The approach therefore centres on identifying the complete vocabulary of events that can affect the object, whether they cause state change or

not, and constructing a state-machine that describes the possible orderings of the events over the life of the object.

An event is defined to be any incident in the real-world whose occurrence is allowed or constrained by the state of the object. This is a wider definition than that used in the State Based approach as it includes events that do not change the state. This wider definition is necessary to define protocols because the circumstances under which such an event can take place are part of the protocol. For instance a “Change Quantity” event on an Order, which does not change its state³, cannot happen after the “Deliver” event.

A by-product of the Event Based approach to constructing the state-machine model is the resultant definition of the states. The states so defined will generally correspond to the states that would be chosen in the State Based approach.

8.4.3 State Machine Type

In the context of the two types of state-machine semantics, described above in section 8.3.3, the Event Based approach uses “Protocol” state-machines.

The use of object protocols for behavioural modelling was pioneered in the JSD method

³ Here we are interpreting “state” in the narrow sense of where it is in its life-cycle, not the wide sense of all the data belonging to the Order.

[12]. JSD used a diagrammatic tree diagram form of simple regular expressions to describe event protocols, which in JSD were called "entity life histories". For a number of reasons, beyond the scope of this paper, we have adopted state-machines as a more appropriate formalism.

A protocol for an object is somewhat analogous to the grammar of a formal language which describes the sequences of tokens that constitute valid sentences of the language. A good protocol model of an object is one whose possible traces are exactly the possible lifecycles of the object.

As a side note, it is important not to confuse this use of the word "protocol" with its somewhat different use in such contexts as "communications protocol". In this latter context, the word describes rules concerning a conversation between two fixed entities. In our current sense protocol defines the life of a single object, over which time it may interact with any number of other entities.

8.4.4 Orthogonal State Spaces

A consequence of the definition of the semantics of protocol state-machines is that multiple protocol state-machines may be composed in parallel. This possibility is noted in the UML2 standard [11, page 465]. This means that the Person example introduced in section 8.3.4 could be modelled as a single object with two, composed, state machines.

We have proposed [13] that the semantics of state machine composition should follow the parallel composition operator of Hoare's CSP [14]. CSP concerns the composition of event protocol processes where the event vocabularies of the composed protocols are not disjoint – i.e., there are events that appear in both.

An example is shown below in Figure 3. The protocol of an Account is described by two composed protocol state machines.

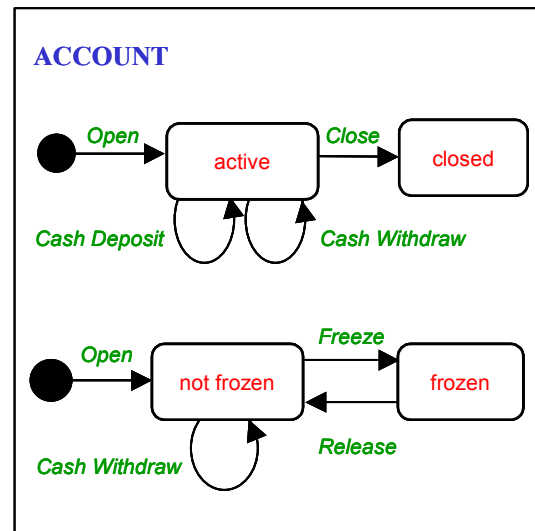


Figure 3. CSP Protocol Composition

The Cash Withdraw event can only take place when the Account is both "active" and "not frozen". The Deposit event, on the other hand, can take place whether the Account is "frozen" or not.

This technique allows the behaviour of a complex object to be expressed using a composition of simple state-machines. (Experience shows that complex objects may be

modelled using a number of simple state machines, as many as five or six on occasion.)

8.4.5 Re-use of behaviour Definitions

A consequence of allowing multiple composed state-machines to be used to define an object is that it also allows a given state-machine to be re-used across the definition of different objects.

For instance if a number of different types of account were being defined, some of which could be frozen and some not, the two state-machines used in Figure 3 could be used selectively as building bricks in the definitions of the different types of account. This allows a pure mixin approach [16] to behaviour definition.

9. OUR PERSPECTIVE

Our interest is in the Event Based behaviour models. This interest is motivated, in particular, by two features of the approach described below.

9.1 Model Based Prototyping

Given a model defined using protocol state-machines it is possible to create a model driven user interface, as follows:

- By reference to the model, the user interface can assemble a list of the object types in the system, and present this to the user

- Once the user has selected an object type, the user interface can gather and present a list the instances of that object type
- Once the user has selected an object instance, the user interface can use the protocol definition to present a list of the events that the selected instance is prepared to accept (perhaps also “greying out” those that it refuses)

The interface is using “run time discovery” to interact with the model. This is possible because the objects, with their protocol state machines, embody a complete definition of how their environment in general, and the user interface in particular, should interact with them.

Creating model-driven user interfaces in this fashion makes the creation of prototypes that can be used to test and demonstrate models quickly and cheaply. We have found that the ability to validate models by demonstrating and exercising such model driven prototypes provides valuable feedback, early in the development process, on how well an emerging model is aligned to user requirements.

9.2 State-Machine Composition

Secondly, we have found that the ability to compose state-machines opens the way to two interesting modelling mechanisms:

- It is possible to construct state-machines that have a derived rather than a stored state, analogous to the distinction between a

derived or stored attribute. This idea is explored in [15].

- It is possible to define different “flavours” for protocol state-machines that reflect whether the constraints represented by the machine are intrinsic to the domain or represent policies of the business. State-machines with different flavours can be composed within the definition of a single object. This idea is explored briefly in the final section of [13].

Together these features have the possibility to support the incremental development and validation with users of models that embody complex behavioural business rules. We have developed a tool, ModelScope, that implements Event Based modelling for this purpose. This is available at www.metamaxim.com.

10. CONCLUSION

It will be apparent to the reader that we believe the following to be true:

- Realisation of the MDA vision requires that the business logic behaviour of an application be represented explicitly in the PIM.
- State-machines provide the best basis for such representation.

Whether these beliefs are right or wrong the underlying questions, namely the extent to which behaviour needs to be modelled in the

PIM and the best way of doing it, are at the heart MDA. We believe that the evolution of MDA as a coherent framework will benefit from wider debate on these questions and, in particular, on the merits and demerits of particular approaches to the representation of behaviour in models.

REFERENCES

- [1] OMG Model Driven Architecture: *How Systems Will Be Built*. Object Management Group website: www.omg.org/mda/.
- [2] Soley, R., 2002. Presentation: *MDA: An Introduction*. Object Management Group website: www.omg.org/mda/presentations.htm.
- [3] Sims, O., 2002. Presentation: *MDA: The Real Value*. Object Management Group website: www.omg.org/mda/presentations.htm.
- [4] Shlaer, S., and Mellor, S., *Object Life Cycles - Modeling the World in States*. Yourdon Press/Prentice Hall, 1992.
- [5] Mellor, S., and Balcer, M., *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley, 2002.
- [6] Rumpe, B., *Executable Modeling with UML. A Vision or a Nightmare?* In *Issues & Trends of Information Technology Management in Contemporary Associations*, Seattle. Idea Group Publishing, Hershey, London, pp. 697-701. 2002.
- [7] Moore, A., *UML2 - A language for MDA?* Invited presentation to the First International Workshop on Metamodeling for MDA, York, November 2003.
- [8] Engels, G., Hucking, R., Sauer, S., and Wagner, A., *UML collaboration diagrams and their transformation to Java*. In *Proceedings UML'99 - The Unified Modeling Language: Beyond the Standard*. Second International Conference, France R., and Rumpe B., (eds.), vol. 1723 of LNCS, pp. 473-488. Springer, 1999.

- [9] Kleppe, A., Warmer J., and Bast, W., *MDA Explained The Model Driven Architecture: Practice and Promise*. Addison Wesley 2003.
- [10] Warmer J., and Kleppe, A., *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley 2003
- [11] OMG, *UML 2.0 Superstructure Final Adopted specification, Document reference ptc/03-08-02 August 2003*. Available from the Object Management Group website: www.omg.org.
- [12] Jackson, M., *System Development*. Prentice Hall 1983.
- [13] McNeile, A., and Simons, N., *State Machines as Mixins*. In *The Journal of Object Technology*, vol. 2, no. 6, November-December 2003, pp. 85-101.
- [14] Hoare, C., *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [15] McNeile, A., and Simons, N., *Mixin Based Behaviour Modelling*. In *Proceedings of the 6th International Conference on Enterprise Information Systems*, Porto, April 2004, Vol. 3, pp 179-183.
- [16] Bracha, G., and Cook, W., *Mixin-based Inheritance*. *Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1990.