

# Composition Semantics for Executable and Evolvable Behavioral Modeling in MDA

Ashley McNeile  
Metamaxim Ltd  
48 Brunswick Gardens  
London W8 4AN, UK  
ashley.mcneile@metamaxim.com

Ella Roubtsova  
Open University of the Netherlands  
Postbus 2960, 6401DL Heerlen  
The Netherlands  
ella.roubtsova@ieee.org

## ABSTRACT

The vision of MDA is to decouple the way that application systems are defined from the specification of their deployment platform. Achieving this vision requires that Platform Independent models are rich enough to capture the behavior of the application, and to support reasoning and execution of functional behavior.

We focus on state transition modeling as being the best able to support MDA and appraise the two types of state machine (Behavior State Machines and Protocol State Machines) defined in UML. We conclude that, for different reasons, neither has semantics that are well placed to serve as a basis for PIM level behavior modeling.

We propose that state transition modeling can be both simplified and strengthened by providing semantics that support process algebraic composition. We claim a number of important advantages for this. Firstly, it provides a common language for defining a range of behavioral abstractions, including software components, behavioral contracts and cross-cutting aspects. Secondly that it better supports analysis of models, by exploiting the formal analysis techniques of process algebra. Thirdly, the semantics enable model execution and testing at the platform independent level across a wider domain than is possible with current UML formalisms.

## Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications—*Languages, Methodologies*; D.2.2 [Software Engineering]: Design Tools and Techniques—*State diagrams*; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*Process models*

## Keywords

MDA, behavior modeling, Platform Independent Model, CSP parallel composition, CCS composition, reasoning

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

BM-MDA '09, June 23 2009, Enschede, The Netherlands

Copyright © 2009 ACM 978-1-60558-503-1/09/06 ...\$10.00.

## 1. INTRODUCTION

The Model Driven Architecture (MDA), an initiative launched in 2001 by the OMG, aims to promote modeling to a central role in the development and management of application systems. In particular, it suggests that “fully-specified platform-independent models (including behavior) can enable intellectual property to move away from technology-specific code, helping to insulate business applications from technology evolution and further enable interoperability” [14] and source code for a specific platform would be largely or completely generated from the model, thus removing the current expensive coupling between applications and the technologies required to run them.

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” [17]. 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” [13]. 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 behavior is fully represented, and the capabilities and properties of the techniques available for modeling behavior are therefore key to achieving the MDA vision.

While there is a general agreement in the MDA community that behavior modeling is essential to the MDA mission and the models should be executable, the behavior modeling notations of UML (Figure 1) [15] do not lend themselves well to executable modeling or model level reasoning.

- *Use Cases* are used to describe scenarios, or episodes, of use of a system by specifying the set of interactions between the system and domain in which it is embedded. Use Cases are described using a combination of natural language and informal diagrams. While “animation” (like playing a movie) of a Use Case may be possible, “execution” in the sense of interactive behavior is not. Their informal and partial nature makes formal reasoning with Use Cases hard.
- *Interaction Diagrams* (Sequence Diagrams and Communication Diagrams) can be used to express interaction of lifelines communication classifiers. Normally an Interaction Diagram presents one scenario of interaction or, with the use of decision and loop constructs,

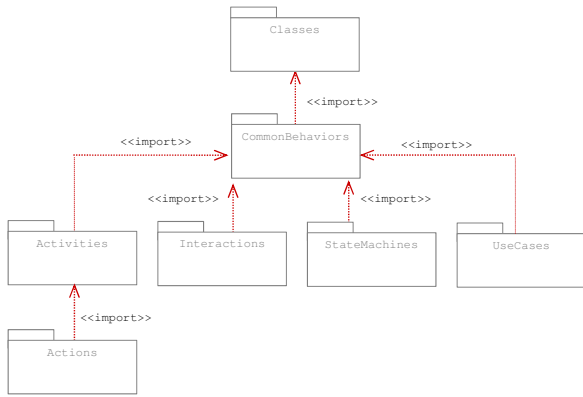


Figure 1: UML Behavior Diagrams

a limited set of related scenarios. As they are scenario based, Interaction Diagrams can be animated to play out a scenario but, because they are not exhaustive of all behavior, they cannot form a basis for model execution. Some researchers have suggested approaches to composition in order to combine scenarios to enable reasoning about the total behavior represented, for example [7, 11], but no composition of Interaction Diagrams is defined in UML.

- *State Machines* exist in two variants: *Behavioral State Machines* (BSM) and *Protocol State Machines* (PSM) [15]. While there is some lack of clarity in their definitions (see, for example, Fecher et al. [9]) state machines do provide complete behavior descriptions and can be used for model based execution. We discuss the UML state machine constructs in some detail in this paper.
- *Activity Diagrams* provide a flow based modeling medium, similar to traditional Petri Nets. They are used to show “the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviors” [15]. This means that Activity Diagrams are not suitable for representing the behavior of an object model, which we believe is essential to MDA. While Activity Diagrams can be executed (see, for example, Engels et al. [8]), execution is at the level of a single overall flow and does not encompass the behavior of objects.

This summary suggests that, if we are to be able to capture and represent full and formal descriptions of systems at the model level, State Machines offer the most promising basis. In this paper we look at the forms of State Machine (BSM and PSM) defined in UML and argue that their semantics are not well geared to the aims of MDA. In this paper we suggest that state-transition based behavior modeling can be both simplified and strengthened by introducing support for the parallel composition constructs of process algebras. We explain how this might be done, and argue that this has advantages in terms of improved ability to:

- specify composable behavioral abstractions that support both behavior specification and expression of behavioral contracts,
- abstract on states and actions in a way that supports re-use and the description of cross-cutting behaviors,

- execute models of behavior at the Platform Independent Level.

The remainder of the paper is organized as follows: Section 2 analyzes the semantics of the UML state machine formalisms for MDA purposes, with a focus on their composition semantics. Section 3 contains our proposal for using process algebraic composition semantics in MDA behavior modeling. Section 4 illustrates the ideas with a small example. Section 5 discusses benefits of the proposed ideas and presents some conclusions.

## 2. STATE MACHINE SEMANTICS IN UML

The *State Machine Package* in the UML Superstructure document v.2.1 and v2.2 [15] describes a set of concepts that can be used for modeling discrete behavior through finite state transition systems. The State Machine package defines two behavioral semantics for finite state transition systems: Behavioral State Machines (BSM) and Protocol State Machines (PSM). Our view is that both of these variants are flawed as a basis for MDA development, and in this section we explain this view.

### 2.1 Behavioral State Machines (BSM)

#### BSM Semantics.

A Behavior State Machine usually presents behavior of one classifier. “Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences. During this traversal, the state machine executes a series of activities associated with various elements of the state machine” [15].

A transition label is of the form:

$$event [guard] / action$$

where *event* specifies the event that triggers the transition, *guard* defines a guard condition that can restrict firing of the transition, and *action* is the action that happens when the transition fires.

The composition semantics for BSMs defines state machines as executing asynchronously and communicating using events, created as a result of actions within the system or in the environment. An event instance is queued until *dispatched*, at which point it is conveyed to one or more BSMs. An event dispatcher mechanism selects and de-queues event instances and an event processor handles the firing of state machine transitions and execution of consequent activity defined by the machines [15].

The consumption of events depends on the active state of a state machine. If an event triggers a transition in the current state of the machine for which it is queued it is dispatched and consumed, and this involves the firing of one or more transitions. If an event can cause two (or more) transitions to fire, which transition is chosen is not defined. If no transition is triggered then either the event is discarded, or it may be held (deferred) for later processing. “A state may specify a set of event types that may be deferred in the state. An event instance that does not trigger any transition in the current state will not be dispatched if its type matches with the type of one of the deferred events. Instead it remains in the event queue while another not deferred message is dispatched instead” [15]. In other words:

- if an event enables a transition in the current state of the machine then it can be dispatched and consumed;
- if an event does not enable a transition in the current state of the machine, but is listed as a deferrable event for this state, it is kept in a queue for later processing;
- otherwise the event is discarded.

The resulting behavior of a population of state machines is, in general, asynchronous and non-deterministic.

### BSM Commentary.

The semantic model used for Behavior State Machine Execution in UML2 (which was first included in UML at version 1.5) is based on the “Recursive Design” method of Shlaer and Mellor [19] whose work has been mainly in the real-time/embedded systems domain. Following the adoption of Shlaer/Mellor semantics into UML the MDA approach based on their ideas has been rebranded as “Executable UML” [18].

The approach is based on using BSMs to model so-called “active objects”: objects whose instances execute autonomously and asynchronously (i.e., as if executing on independent threads) resulting in system behavior that is inherently non-deterministic [20]. It is very hard to reconcile this semantic basis with the characteristics of the business information systems domain, where behavioral issues are related to transactional integrity and business rules, and strictly deterministic behavior of business logic is important to ensure repeatability, auditability and testability. We note that the commercial tools that support Executable UML (such as those from Telelogic, Kennedy Carter and Mentor Graphics) are not well adapted for use in the business information systems domain and are positioned by their vendors to target the real time/embedded market.

The complex composition semantics makes reasoning about behavior difficult. Complete analysis of the behavior of the model must allow, in general, for arbitrary queuing of events between objects and for the accumulation of deferred events. If a model comprises a number of communicating objects this results in a large number of possible execution states for the system as a whole, and reasoning on models is impossible without model checking algorithms. This does not make sense when models are being developed, as they are in most projects, in an iterative manner and subject to frequent change.

While there is some native support in Shlaer/Mellor for behavior abstraction through the use of “polymorphic events”, this has not been included in the UML BSM standard; nor is there any method to compose multiple machines to form the behavior of a single classifier. This places severe limits on the ability of BSMs to describe generalization/specialization of behaviors or to support behavior re-use. As described in [18], a single object class is modeled with a single state machine, and only concrete classes are modeled. This also means that crosscutting behaviors (aspects) have to be addressed by other means, potentially further complicating model analysis.

## 2.2 Protocol State Machines

### PSM Semantics.

Protocol State Machine (PSM) are not related to Shlaer/Mellor, and have semantics that are more general and closer

to the UML state machine semantics that pertained before the import of Shlaer/Mellor semantics in version 1.5<sup>1</sup>.

PSMs are used to express the legal transitions that a classifier can trigger. A PSM is a way to define a lifecycle for objects, or an order of the invocation of its operations. PSMs can express usage scenarios of classifiers, interfaces, and ports. The effect actions of transitions are not specified in a PSM transition as the trigger itself is the operation. However, pre- and post- conditions are specified, so that the label of a transition is of the form

[pre-condition] event/ [post-condition].

The occurrence of an event that a PSM cannot handle is viewed as a precondition violation, but the consequent behavior is left open. “The interpretation of the reception of an event in an unexpected situation (current state, state invariant, and pre-condition) is a *semantic variation point*: the event can be ignored, rejected, or deferred; an exception can be raised; or the application can stop on an error. It corresponds semantically to a pre-condition violation, for which no predefined behavior is defined in UML” [15].

Unlike BSMs, PSMs can (to a limited extent) be composed. “A classifier may have several protocol state machines. This happens frequently, for example, when a class inherits several parent classes having protocol state machine, when the protocols are orthogonal” [15]. In this context, “orthogonal” means that they have a disjoint set of events.

### PSM Commentary.

PSM semantics are simpler and more abstract than the BSM semantics, and this makes them more widely usable and easier to analyze. However, as evidenced by the language used to describe them, PSMs are clearly positioned in UML as *contracts of legal usage*; and this gives it a different meaning and role from that of BSMs. While a contract must specify what is legal, it is **not** concerned with the mechanism by which non-legal behavior is avoided, nor is it required to specify the effect of violation. In other words: A contract cannot be used as the instrument that guarantees its own satisfaction. It would therefore be a logical error to execute PSMs directly or to generate code from them; and other devices must be used in order to ensure that the software that is built complies with the contract PSMs that have been defined for it. To use PSMs as executable models or the basis for code generation in the context of MDA would be inconsistent with this semantic positioning.

## 3. OUR PROPOSAL

Our view is that the state machine formalisms can be both simplified and strengthened by:

- Using a common notational form for both the specification of both contracts and behavior (so eliminating divergence of notation that has emerged in UML between BSMs and PSMs)
- Defining semantics that support process algebraic composition.

<sup>1</sup>It is probable that fracturing of the state machine formalism in UML into two forms was a result of the impossibility of reconciling the semantics of Shlaer/Mellor (reflecting the needs of real-time systems) with the need for a more general capability to specify legal orderings.

The first of these is based on the observation that a machine with behavioral semantics can serve either as a specification or as a contract, depending on the intentions of the author. We contend that there is no penalty, and a good deal to gain, in harmonizing the notations and concepts used across the two.

There is a synergy between these two proposals. In the context of contract definitions it is important to be able to make descriptions that abstract from the full behavior of a classifier, as a contract is normally a **partial** requirement on its behavior. This requirement is met by the second part of the proposal which allows the creation and composition of partial behavioral descriptions.

The composition techniques developed in Process Algebras such as Hoare’s CSP [5] and Milner’s CCS [16] have so far not made their way into UML, perhaps because the domain of *algebraic processes* (CSP and CCS) and *software models* (UML) have been viewed as too different for the techniques of the former to be used in the latter. However, this is a mistake. Research work into behavior specification techniques, such as those by McNeile et al. [3] and Grieskamp et al. [21], have shown that CSP  $\parallel$  composition transplants successfully into software modeling. Other recent work in the context of collaborative service behavior and service choreography specification is making use of CCS and  $\pi$ -calculus, such as the work of Carbone et al. [12]. The proposals we make here exploits and extends the foundations built in this work.

The cornerstone of our proposal is to use a single form of abstract state transition machine, which we call a *protocol machine*, as basis for behavioral modeling. The key property of protocol machines is that their semantics enable composition.

### 3.1 Definition of a Protocol Machine

A *protocol machine* is, like the state machine constructs of UML, a conceptual behavioral machine. However, unlike the state machine constructs of UML, protocol machines can be composed so that large, complex behaviors can be built by combining smaller, simpler ones.

Protocol machines have the ability to *allow*, *refuse* or *ignore* any action in its alphabet. More specifically, the behavior of a protocol machine is defined as follows:

- It has a defined *alphabet*: a set of actions that it understands.
- In a given state it will:
  - *Ignore* any action that is not in its alphabet;
  - Depending on its state, either *allow* or *refuse* an action that is in its alphabet.
- If it engages in an action it moves to a new state.

The nature of the “actions” in the alphabet of a machine depends on the context and purpose for which the machine has been defined. When defining a single software component, an action represents the receipt of a particular message type from the component’s environment. In the context of message based collaboration in a distributed system, an action represents either the sending or receipt of a message of a given type. Informally, by *ignoring* an action a machine is saying “I do not know about this action, and have no opinion on whether it can happen or not”, whereas by *refusing* an action a machine is saying “This is an action that I know

about (it is in my alphabet) and I know that it cannot happen now”. The distinction between these two, which is not made at all in the semantics of UML state machines, is the basis for parallel composition.

Two further properties are key to the definition of protocol machine behavior:

- If it is starved actions a protocol machine is bound to reach *quiescence*, and only at quiescence is its state well defined. A machine with this property is sometimes called *reactive*. This means that a protocol machine cannot engage in an action that results in a computation that does not terminate.
- A protocol machine is *deterministic*, so the new state it moves to when it allows an action is dependent only on the old state and the action in which it engages.

A protocol machine may, like an object, own attributes; and only the machine that owns an attribute may update it. Updates only take place when a machine allows an action, and constitute part of the change of state of the machine.

### 3.2 Composition of Protocol Machines

For the purposes of this paper we define two forms of composition, corresponding to CSP  $\parallel$  composition and CCS  $|$  composition. Other forms of composition are possible but not within the scope of this discussion.

Generally speaking, CSP  $\parallel$  is used to compose machines in the formation of a single software component, so the composed parts are within a computing environment that allows them to share state and data; whereas CCS  $|$  is used for machines that are distributed in such a way that they cannot share state and data and therefore communicate by exchanging messages.

#### CSP Composition.

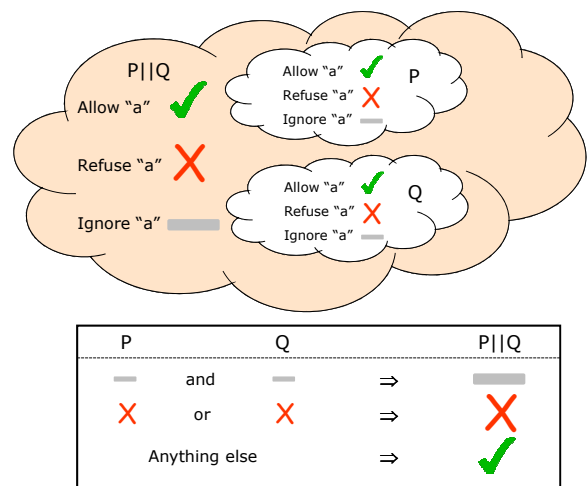


Figure 2: CSP Composition

Suppose two machines  $P$  and  $Q$  are composed to form  $P \parallel Q$ . The ability of the composite to engage in an action,  $a$ , is defined as follows (see Figure 2):

- If both  $P$  and  $Q$  ignore  $a$  then the composite ignores  $a$ .

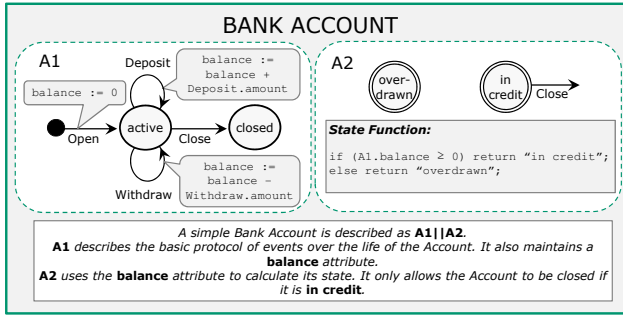


Figure 4: Bank Account with a Derived State

- If either  $P$  or  $Q$  refuses  $a$  then the composite refuses  $a$ .
- Otherwise the composite allows  $a$ .

The CSP  $\parallel$  composition of two protocol machines is another protocol machine. In particular, it is also deterministic (see [3] for further discussion of this).

Note that this form of composition does **not** have the restriction present in UML for composition of PSMs that the composed machines are “orthogonal”. It is the ability to compose non-orthogonal machines (ones whose alphabets have elements in common) that gives the technique its expressive power.

### CCS Composition.

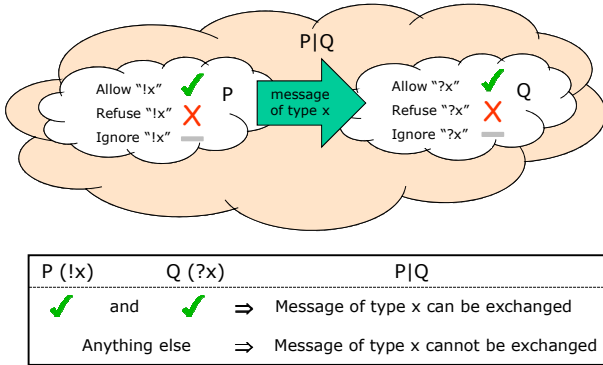


Figure 3: CCS Composition

Suppose that  $P$  is a machine whose alphabet includes a “send action on  $x$ ”, represented as  $!x$ ; and  $Q$  is a machine whose alphabet includes the corresponding receive action, represented as  $?x$ . If these machines are composed to form  $P|Q$  the behavior of the composite is defined by (see Figure 3):

- If  $P$  allows  $!x$  and  $Q$  allows  $?x$  then a reaction on  $x$  between them can take place. If the reaction occurs both machines execute their respective actions on  $x$  and move to new states.
- Otherwise nothing happens.

The result of CCS composition of two protocol machines is **not** another protocol machine. This is because CCS composition does not, in general, give deterministic behavior of the

composite. Suppose  $P$  and  $Q$  are able to engage in a reaction on  $y$  as well as the one on  $x$ , then whether the  $x$  reaction or the  $y$  reaction (or neither) takes place is not determined. We use the term *collaboration* for a set of machines under CCS composition.

### 3.3 Derived States

Because there is no restriction on how the state of a protocol machine may be determined, we allow machines to have *derived states* as well as the more usual *stored states* driven by the transitions of the state machine. Figure 4 shows an example of a Bank Account described as two machines composed using CSP  $\parallel$ . The right hand machine,  $A2$ , uses derived states (*in credit* and *overdrawn*) calculated on the basis of the *balance* attribute owned and maintained by  $A1$ .

A derived state is analogous to the familiar concept of a derived (or calculated) attribute, in that its value is calculated “on the fly” by a function when required. The use of derived states is another departure from standard UML state machine formalism, but increases the expressive power to describe action sequencing protocols that depend on the values of stored data. A machine may use its own attributes and/or those of other, composed, machines to derive its state.

When defining machines, we follow the discipline that a given machine uses either stored states, where updates to the state are defined implicitly by the state-transition topology (as in  $A1$ ); or only uses derived states, where the state values are derived (as in  $A2$ ). This is analogous to the familiar discipline with attributes, where an attribute is either stored or derived. The state icons for derived state machine are given a double outline.

Finally, note that a derived state machine does not have to be “topologically connected”. For instance, the *Close* transition in  $A2$  does not lead to another state. This is because the state update is not driven by transitions.

### 3.4 Behavior Re-Use and Aspectual Modeling

The use of composition allows complex behavior to be defined as a composition of smaller, simpler, components. These components can be re-used across the definition of different behavioral entities. Thus, in a banking application that has to support multiple different types of account (Current Account, Savings Account, Student Account, etc.) the basic account behavior described by  $A1$  could be common to them all and could be composed with machines that represent the particular behavioral rules for each account type. Further discussion of this is given in [3].

Moreover, by exploiting the possibility of defining *generalized states and actions*, this form of re-use can be applied to the definition of crosscutting behavioral aspects. Such generalizations are achieved as follows:

- With the ability to define machines with derived states, we can make one machine generalize over the states of another, rather in the way that the state *in credit* in  $A2$  generalizes over all non-negative values of *balance* in  $A1$  (see Figure 4). Such a state, which provides a more abstract view of the states of another machine, is called a *generalized state*.
- If two or more actions are treated identically in the context of a given machine (causing the same transitions and same attribute updates), they can be replaced by a single *generalized action*. This can be

thought of as a macro that expands one transition in the machine into a number of transitions with the same start and end states.

A fuller discussion of the use of these techniques in aspectual modeling has been given in [1].

## 4. EXAMPLE

We illustrate the idea with a small model of a mobile 'phone equipped with a gaming capability. We use this example to show how a model can be described by using a combination of a *Dynamic View* that shows the behavior of the machines of the model, and a *Static View* that shows how the machines are composed.

### 4.1 Dynamic View

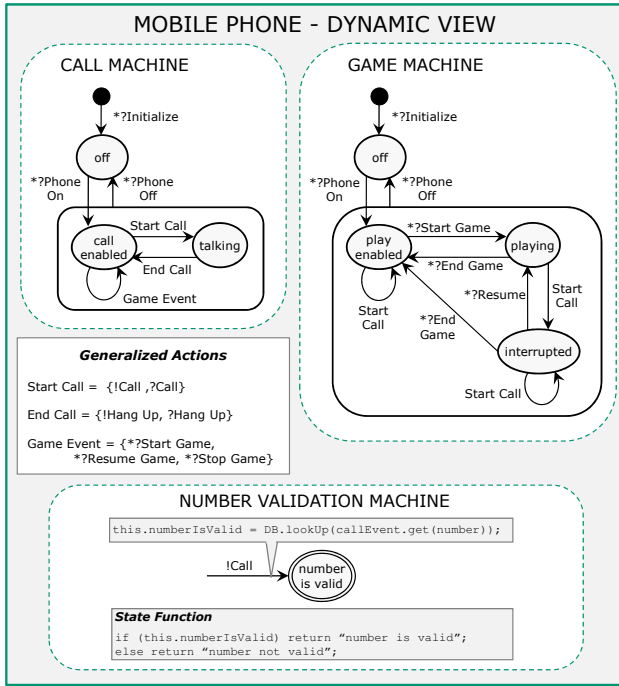


Figure 5: Mobile Phone - Dynamic View

Figure 5 shows the *Dynamic View*. This shows the state transition representation of the machines of the model. Our model consists of three protocol machines:

**Call Machine.** This machine handles calls. Once the 'phone is switched on, a call is initiated by *Start Call*. This is a generalized action, representing either *!Call* (this 'phone makes a call) or *?Call* (this 'phone receives a call). Similarly, a call can be ended by either this 'phone hanging up or the remote 'phone hanging up. The generalized action *Game Event* represents events connected with the game facility, and these may only take place when a call is not in progress.

**Game Machine.** This machine handles game playing. If a call is initiated (by making or receiving a call) and a game is underway, the game is interrupted (the machine moves to the *interrupted* state). It may be resumed later, with the action *\*?Resume*.

**Number Validation Machine.** This machine handles validation of called numbers against a list of valid numbers. A call initiation action *!Call* must end in the derived state *number is valid*, and this means that the call event is only allowed if the number is on the list.

Each action in each machine is prefixed by "!" meaning a message sent by the machine or "?" meaning a message or input received by the machine. The "\*" prefix is used to signify local events from environment; these do not participate in CCS message reactions.

### 4.2 Static View

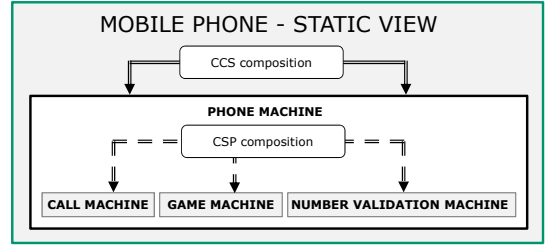


Figure 6: Mobile Phone - Static View

Figure 6 shows the *Static View* and this shows how the machines of the model are composed. This shows the following:

- *Phone Machine* represents the complete behavior of a 'phone.
- The three machines *Call Machine*, *Game Machine* and *Number Validation machine* are composed using CSP || to form the behavior of a 'phone.
- Multiple 'phones are composed using CCS | to form a collaboration whereby different 'phones engage in calling each other.

The Static View also shows the data attributes:

- In general, every machine owns a set of attributes. For instance the *Call Machine* would own the attribute *my number* being the number of this 'phone.
- Also every message type sent or received will in general carry a set of attributes depending on the message type. Thus a *Call* message will contain the *calling number* of the 'phone initiating the call, and the *called number* of the 'phone being called.

For brevity, we have not shown the attributes of the model in the diagrams.

### 4.3 Behavior of the Phone Model

As an example of the behavior of the model as described, suppose the 'phone 1234 attempts to call 'phone 4321. This means that 'phone 1234 must allow a *!Call* action. Consider each of its machines:

- The *Call Machine* allows *!Call* provided that it is in the state *call enabled*. If the machine is switched off or already on a call, the action is not possible.

- The *Game Machine* allows *!Call* provided that it is switched on, as all states have a transition for *Start Call* (which includes *!Call*). If the machine is in the state *playing*, the *!Call* will cause the game to be interrupted.
- The *Number Validation Machine* allows *!Call* provided that the action takes it to the state *number is valid*, and this requires that number is on the list of valid numbers. If this is not the case, the *!Call* action is not possible.

## 5. BENEFITS AND CONCLUSIONS

In this section, we describe some of the motivating factors for the suggested approach, and give brief conclusions.

### 5.1 Local Reasoning and Analysis

The CSP  $\parallel$  composition technique has the property that it gives *Observational Consistency* (as defined by Ebert and Engels [10]) when applied to protocol machines. This property, as discussed in [1], gives the ability to perform *local reasoning* on models, whereby conclusions about system behavior as a whole can be based on examining single machines in isolation. This is essentially because CSP  $\parallel$  composition preserves the *trace behavior* of the composed machines. As has been often noted, for instance by Dantas [6], the ability to perform local reasoning is crucial if intellectual control is to be maintained over a complex model as it grows. As well as this, state-transition models composed using process algebraic techniques can be analyzed using standard model checking algorithms and tools, and this is important in complex cases where “brute force” is needed to check all possibilities (normally where true concurrency is involved).

Together, these provide a basis for ensuring correctness of behavior at the PIM stage of modeling. This, correctly, demotes the role of testing as the means of ensuring that the delivered software works correctly.

### 5.2 Contracts

Our claim is that protocol machines may be used to describe both *behavior* and *contracts* using a single notational platform. This is a subject of on-going research, but the basis of the idea is simple. Suppose that a protocol machine  $C$  is a contract and another machine  $B$  is a behavior specification. We say that  $B$  satisfies  $C$  iff  $B \parallel C = B$ . This requires that:

- The alphabet of  $B$  is a superset of that of  $C$ . This is natural, as you would not expect a design to satisfy a contract if it does not recognize all the actions required by the contract.
- By the rules of  $\parallel$  composition, an action is only allowed in  $B$  if also allowed in  $C$ . This means that the states of  $C$  can be viewed as defining pre-conditions for the actions of  $B$ .

While a full discussion is not possible here, this illustrates the attractive possibility of using a common formalism for both behavior and contractual definitions with a simple formal definition of compliance. This is both more elegant and more powerful than the dual notation approach, with BSMs and PSMs, currently in UML.

## 5.3 Executability

Protocol models are executable. For example, the ModelScope tool [4] interprets the meta-description of PIM level protocol models. Further discussion of execution of protocol machine models is given in [2].

It is our belief that the formalisms suggested here are more domain neutral than those in the current UML. Whereas the UML BSM semantics has a definite “real time systems” flavor, protocol machines have been used to model database and business process centric systems and have proved to be applicable to these domains.

## 5.4 Conclusion

Behavior modeling in MDA should be simple, executable and extensible. Our examination of the state transition formalisms of UML suggests that they do not possess these properties. The main contribution of this paper is a proposal to make process algebraic composition techniques central to state transition behavior modeling in MDA. We argue that the capability to compose behavioral models enables behavior re-use and potentially, because they can be used to describe behavioral contracts as well as behavioral specifications, eliminate the need for two forms of state-transition model (BSM and PSM) used by UML.

The composition semantics presented in this paper are based on the *protocol machine* abstraction. Although relatively simple in concept, protocol machines support the modeling of processes and objects that possess and maintain data attributes and, by allowing states to be derived as well as stored, enable the modeling of cross-cutting concerns using state and action abstractions. In addition their simple compositional semantics make them amenable to analysis, both human reason and machine based.

## 6. REFERENCES

- [1] A. McNeile and E. Roubtsova. CSP parallel composition of aspect models. In *AOM '08: Proceedings of the 2008 AOSD Workshop on Aspect-Oriented Modeling*, pages 13–18, New York, NY, USA, 2008. ACM.
- [2] A. McNeile and E. Roubtsova. Executable Protocol Models as a Requirements Engineering Tool. In *ANSS-41 '08: Proceedings of the 41st Annual Simulation Symposium (anss-41 2008)*, pages 95–102, Washington, DC, USA, 2008. IEEE Computer Society.
- [3] A. McNeile and N. Simons. Protocol Modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. *Software and System Modeling*, 5(1):91–107, 2006.
- [4] A. McNeile, N. Simons. <http://www.metamaxim.com/>.
- [5] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [6] D. Dantas, D. Walker. Harmless Advice. *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. To appear, 2006.
- [7] E. Roubtsova and R. Kuiper. Process Semantics for UML Component Specifications to Assess Inheritance. ENTCS 73(3), Eds. P. Bottoni and M. Minas, 2003.
- [8] G. Engels, A. Kleppe, A. Rensink, M. Semenyak, C. Soltenborn and H. Wehrheim. From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations. In *ECMDA-FA*, pages 94–109, 2008.

- [9] H. Fecher, J. Schönborn, M. Kyas, W. de Roever. 29 New Unclearities in the Semantics of UML 2.0 State Machines. In *ICFEM*, pages 52–65, 2005.
- [10] J. Ebert, G. Engels. Observable or invocable behaviour-You have to choose. *Technical report*. Universität Koblenz, Koblenz, Germany, 1994.
- [11] J. Greenyer, J. Rieke, O. Travkin and E. Kindler. TGGs for Transforming UML to CSP: Contribution to the ACTIVE 2007 Graph Transformation Tools Contest. University of Paderborn, Technical Report tr-ri-08-287, 2008.
- [12] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown and S. Ross-Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming. [www.w3.org/2002/ws/chor/edcopies/theory/note.pdf](http://www.w3.org/2002/ws/chor/edcopies/theory/note.pdf), 2006.
- [13] O. Sims. Presentation: MDA: The Real Value, Object Management Group website: [www.omg.org/mda/presentations.htm](http://www.omg.org/mda/presentations.htm) . 2002.
- [14] OMG. Model Driven Architecture: How Systems Will Be Built. Object Management Group website: [www.omg.org/mda/](http://www.omg.org/mda/).
- [15] OMG. Unified Modeling Language, Superstructure, v2.2. *OMG Document formal/09-02-02 Minor revision to UML, v2.1.2. Supersedes formal 2007-11-02*, 2009.
- [16] R. Milner. *A Calculus of Communicating Systems*, volume 92. 1980.
- [17] R. Soley. Presentation: MDA: An Introduction. Object Management Group website: [www.omg.org/mda/presentations.htm](http://www.omg.org/mda/presentations.htm) . 2002.
- [18] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. 2002.
- [19] S. Shlaer and S. Mellor. *Object Life Cycles - Modeling the World in States*. Yourdon Press/Prentice Hall, 1992.
- [20] T. Santen and D. Seifert. Executing UML State Machines. *Technical Report 2006-04, Fakultät für Elektrotechnik und Informatik, Technische Universität Berlin*, 2006.
- [21] W. Grieskamp, F. Kicillof, N. Tillmann. Action Machines: A Framework for Encoding and Composing Partial Behaviours. *Microsoft Technical Report MSR-TR-2006-11*, 2006.