Protocol Modelling

A Modelling Approach that Supports Reusable Behavioural Abstractions

Ashley McNeile(✉) and Nicholas Simons

Metamaxim Ltd., 48 Brunswick Gardens, London W8 4AN, UK
{ashley.mcneile,nick.simons}@metamaxim.com

Abstract. We describe a behavioural modelling approach based on the concept of a "Protocol Machine", a machine whose behaviour is governed by rules that determine whether it accepts or refuses events that are presented to it. We show how these machines can be composed in the manner of mixins to model object behaviour and show how the approach provides a basis for defining reusable fine-grained behavioural abstractions. We suggest that this approach provides better encapsulation of object behaviour than traditional object modelling techniques when modelling transactional business systems.

We relate the approach to work going on in model driven approaches, specifically the Model Driven Architecture initiative sponsored by the Object Management Group.

Keywords: Behavioural modelling · Reuse · Protocols · State machines · Mixins · Executable modelling

1 Introduction

1.1 Background and Purpose

The modelling ideas described in this paper have their origins in work done by the authors in the late 1980 s to develop a scheme for generating code from models. Since that early work we have developed a series of tools that implement the approach and have refined the ideas and their formal basis. Our motivation has been to develop suitable abstractions for describing the behaviour of a class of systems normally termed "transactional business systems". This class includes such familiar applications as accounting, order processing, workflow, stock control, etc.

This paper explores a possible formal basis for the ideas. We believe that this formalisation will help others in the modelling community to understand and assess our work, and provide a basis to extend and improve it. We also compare aspects of our approach with more traditional object-oriented modelling (as supported, for instance, by the Unified Modelling Language, UML) and explain how ours differs and why it may be better.

This paper originally appeared in *Software & Systems Modeling*, 5(1):91–107, 2006 and is reproduced here by kind permission of Springer-Verlag.

© Springer International Publishing Switzerland 2015
E. Roubtsova et al. (Eds.): BM-FA 2009-2014, LNCS 6368, pp. 167–196, 2015.
DOI: 10.1007/978-3-319-21912-7_7

1.2 Structure of the Paper

In Sect. 2 we introduce the underlying concepts of Events, Protocol Machines and Protocol Systems. These are the basis for the rest of the paper.

In Sect. 3 we show how these basic concepts can be used to model the behaviour of objects, and in Sect. 4 how the behavioural metadata of objects is defined. This section includes an example illustrating a graphical notation for metadata.

Section 5 explains how the ideas can be used to build semantically complete object models, capable of execution. Section 6 describes how the approach supports reuse of behavioural metadata.

Section 7 discusses the differences between our approach and other forms of object based modelling. We note that our focus on modelling behaviour is aligned to some of the aims of the Object Management Group's "Model Driven Architecture" (MDA) initiative, and we relate our work to other work in the MDA field.

Finally, Sect. 8 gives a short history of implementations of the ideas described in this paper.

2 Underlying Concepts

This section describes the three concepts that form the basis of the approach described in this paper. These concepts are:

- events,
- protocol machines, and
- protocol systems.

2.1 Events

Our focus has been on modelling the behaviour of event driven systems. Our modelling approach uses a core behavioural abstraction that we refer to as a "protocol machine". Before describing the concept of a protocol machine in detail, we start with the notion of an "event".

An "event" (properly an "event instance"[1]) is the data representation of an occurrence of interest in the real world business domain. Examples of such real world occurrences are *Customer Fred places an order for 100 widgets to be delivered on 12th August* or *Policy holder Jim makes a claim for £250 against policy number P1234*. These occurrences are considered to be atomic and instantaneous in the domain.

An event represents such an occurrence as a set of data attributes. Every event is an instance of an event-type, and the type of an event determines its metadata (or attribute schema), this being the set of data attributes that completely define an instance of the event-type. In a banking system, for instance, an

[1] We will use "event" for event instance throughout this paper. Where we need to refer to an event type we will use the explicit term.

event-type might be "Withdraw" with metadata (Account Id, Date and Time, Amount).

This approach to modelling events is identical to that used in other event based modelling approaches such as Jackson System Development [8] and Syntropy [15]. From a more formal standpoint, our treatment of events corresponds to that described by Jackson and Zave [9].

We use meaningful, natural language names for Event-types to aid their identification with the real world occurrences that they represent. For the same reason, we use natural language names for attributes in the metadata of an event-type.

2.2 Protocol Machines

A "protocol machine" is a conceptual machine that has a defined repertoire of events that it understands, and the ability to accept, refuse or ignore any event that is presented to it. Protocol machines have the property that large, complex protocol machines can be assembled from small, simple ones. We are interested in using them to build models of systems that comprise objects and, in this use, the smallest protocol machines are more fine-grained than objects. The largest, however, represent whole systems.

The use of the term "protocol" in this context is borrowed from UML, as used in the concept of a "protocol state machine" [13]. In UML, protocol is used to mean an allowable sequence of operation (method) invocations in the life of an object. Although we are talking about sequencing of events rather than operations, the intent (namely to define allowable sequences) is the same so we use the same term.

Type. A protocol machine has a "machine-type" which is a fixed, immutable property of the machine. The machine type, M, of a machine instance, m, can be determined using a type function:

$$M = \tau(m)$$

(Note: Throughout this paper we use upper case to denote a type and lower case to denote an instance.)

State. A protocol machine has a stored "local state" which only it can alter, and only when moving to a new state in response to an event.

Machines can be nested. If a machine $m1$ is immediately nested in machine $m2$, then:

- The local state of $m1$ is a subset of the local state of $m2$.
- Only $m1$ can alter that part of the local state of $m2$ that is also the local state of $m1$.

In addition, a machine has a "state environment", being stored state that the machine can access but cannot alter. The state environment of $m1$ is defined as the union of:

- the local state of $m2$ that is not local state of $m1$, and
- the state environment of $m2$.

A machine that is not nested inside another has an empty state environment, and is called a "closed machine". This is summarised in Fig. 1.

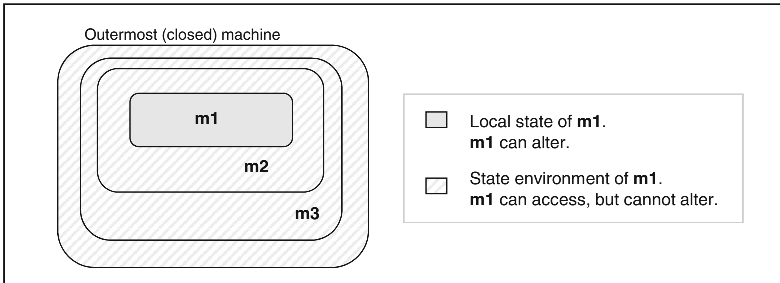


Fig. 1. Machine Nesting

Repertoire. A protocol machine has a set called its "repertoire" that determines the events that it is able to understand[2]. The only requirement of a repertoire is that when an event is presented to a machine, it must be possible to determine whether or not that event is represented in the repertoire. Exactly how this determination is made is defined in Sect. 3.3 in terms of "binding". For the present we shall take it that the repertoire of a machine is a set of event-types. The machine can understand any event with a type that belongs to its repertoire.

As we describe later in Sect. 5.4, the repertoire of a machine can change. However, it must always be possible to determine the repertoire of a machine when the machine is in a stable state.

When machines are nested, the repertoire of an inner machine is always a subset of the repertoire of the machine within which it is embedded. This is required for consistency, as it makes no sense for a machine to ignore an event that another machine nested inside could understand.

Behaviour. The behaviour of a protocol machine is entirely event driven. Events are presented to the machine one at a time by its environment.

[2] More formally: an event-type is included in the repertoire of a protocol machine if there is some state of the machine in which it would be capable of accepting or refusing it.

The machine has no capability to process more than one event at a time, and is stable between events. Presenting an event to a machine also presents it to every embedded machine.

The behaviour of a machine is defined as follows:

a. When presented with an event that is not represented in its repertoire, the machine ignores it.
b. When presented with an event that is represented in its repertoire, it will either accept it or refuse it.
c. Acceptance of an event is not possible unless the quiescent values of the machine's local state and of its state environment both before and after the event meet constraints specified by the machine.
d. By accepting an event, the machine is allowing (but not requiring) its new quiescent state to be designated as its new stable state.

Quiescence and Stability. A machine is quiescent when, if starved of further events (no further events presented to it), it cannot undergo any further change of local state. Informally, this just means that the processing that the machine performs to update its local state in response to an event is complete.

When a machine reaches quiescence after being presented with an event, one of the following must take place:

- Its new quiescent local state is designated as its new stable state; or
- The new quiescent local state is abandoned, and the machine remains at the stable state that pertained before the last event.

Whichever of these takes place, it applies to the machine itself and to all embedded machines. The decision as to which happens is, in general, not made by the machine itself. This is discussed later in Sect. 5.2.

Only after a new stable state has been established can a new event be presented to the machine.

Determinism. Protocol Machines are deterministic in the following sense. The new quiescent state that a machine reaches as a result of being presented with an event is completely determined by:

- the last stable value of its own local state,
- the last stable value of its state environment, and
- the event-type and the values of the attributes of the event presented to it.

This also means that whether or not a machine accepts an event is similarly deterministic, and therefore that executions of a protocol machine based model are repeatable.

To ensure that this degree of determinism is achieved requires that:

a. The algorithm by which a machine updates its local state is deterministic[3].

[3] This rules out, for instance, using multiple threads of processing in the algorithm if this can cause indeterminism by introducing race conditions.

b. When a machine accesses its state environment in the course of update, it will always obtain the last stable value, not yet reflecting any update for the current event.
c. A machine does not begin to update its own local state until all embedded machines have reached their new quiescent states, and access to the state of an embedded machine yields its new quiescent state.

The last of these means that, using the terminology defined above, if machine $m2$ is performing updates to its local state for an event e:

- $m2$ cannot start to alter its own local state before e has been presented to $m1$ and $m1$ has reached quiescence.
- $m2$ has access to this new quiescent state of $m1$.
- $m2$ also has access to its own state environment in its last stable state, i.e., not reflecting any updates for e.

The scheme is shown pictorially in Fig. 2.

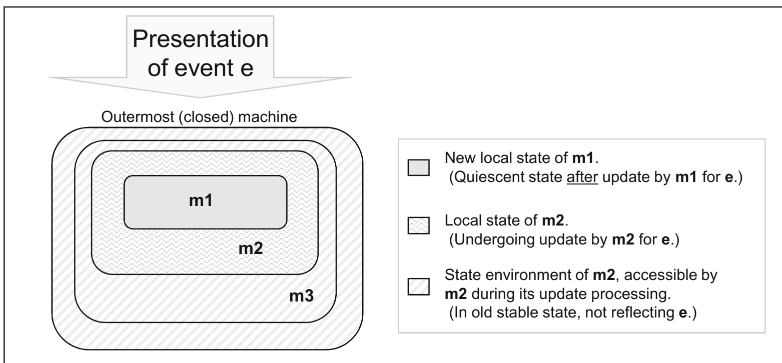


Fig. 2. State Update Discipline

2.3 Protocol Systems

We are interested in building executable models by putting together protocol machines.

In this section we describe, in general, how a set of protocol machines can be composed in parallel to form another protocol machine. We call a machine formed in this way a "protocol system". A protocol system has no stored state or behavioural capacity beyond that provided by the machines that constitute it.

In the following sections we describe how a protocol system behaves and show how it conforms to the definition of a protocol machine. For ease of reading across, the headings mirror those used in Sect. 2.2 to describe a protocol machine.

Type. A system has a system-type which is a unique, immutable property of the system. The type of a system has associated metadata which determines a fixed set of machine types to which any constituent machine must belong.

The system-type, S, of a system, s, can be determined using a type function:

$$S = \tau(s)$$

As a system is itself a protocol machine, its system-type is also its machine-type.

Repertoire. The repertoire of a system is the union of the repertoires of its constituent machines.

State. The local state of a system is the union of the local states of the constituent machines.

The state environment of each constituent machine is the union of:

– the local states of all the other constituent machines, and
– the state environment of the system.

Behaviour. When an event is presented to a system it is presented, in some order, to all of the constituent machines of the system.

The disposition (ignored, refused or accepted) of an event presented to a system is determined by, and only by, the responses of the constituent machines in the system. This determination is made as follows:

a. If the event is not represented in the repertoire of the system, the system ignores it.
b. If any constituent machine refuses the event, the system refuses it.
c. Otherwise the system accepts the event.

Note that this definition of the semantics of composition bears a close resemblance to the parallel composition operator, $P \parallel Q$, in Hoare's Communicating Sequential Processes [5]. We explore some aspects of this resemblance in an earlier paper, State Machines as Mixins [2].

Quiescence and Stability. It follows from the definition of quiescence for a protocol machine that the system is quiescent when:

– all constituent machines that could update their local state as a result of the last event presented to the system have done so, and
– all of its constituent machines are quiescent.

Once a system is quiescent a determination of its new stable state is made, as described in Sect. 5.2.

Determinism. A protocol system is deterministic, shown inductively as follows:

a. By assumption, the constituent machines are deterministic. So the only source of non-determinism in the system is in different choices of the order in which the constituent machines are presented with an event and perform their update.
b. As any constituent machine, while in progress of updating its local state, cannot see updates made for the current event by any other constituent machine, the ordering of updates does not influence the result of the updates.

This induction requires that elementary machines (those not defined in terms of other machines) are deterministic. This is addressed in Sect. 4.3.

3 Modelling Objects

This section describes how the concepts of events, protocol machines and protocol systems can be used to construct models that have a notion of object.

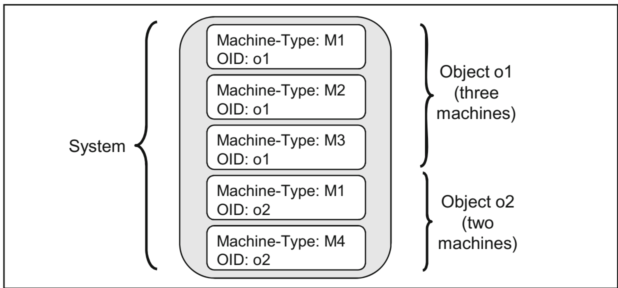


Fig. 3. Object Composition

3.1 Object Identifiers

Our aim is to be able to model a population of objects as a protocol system. We do not want each object instance to require its own machine type, so we have to be able to accommodate multiple instances of a given machine type in a system and identify them as different individuals.

To provide for unique identification of machines in a system, and to tie every machine to the object whose partial description it represents, we require that every constituent machine of a system has a property called its "object identifier", or OID. The OID is a fixed, immutable property of the machine. The combination of the OID and machine-type properties of a machine must be unique in a system.

The collection of all machines sharing a given OID represents an object, so the relationship between objects and machines is one to many[4], as shown in Fig. 3.

Figure 3 shows a system containing two objects, one comprising three machines and the other two machines. The combination of machine-type and OID yields a unique identifier for each machine. Note that machine type $M1$ is used by both objects.

3.2 Repertoire Specification

In Sect. 2.2 we introduced the idea of a repertoire, and suggested that the repertoire of a machine type can be thought of as a set of event-types.

With the introduction of objects the use of a simple event-type as a repertoire entry is not sufficient, and needs to be qualified in two ways as described below.

The first need for qualification results from having multiple machines of the same type in a system. When an event is submitted to a system it will be relevant to some objects and not others. Thus a "Withdraw" occurrence in a banking domain will be for a particular account. Other accounts are not affected by it and should ignore it.

From the objects' point of view the event instance corresponding to the Withdraw is in the repertoire of one particular account, and should be ignored by other accounts. To allow the repertoire to be used to determine that an event is to be ignored because it is for a different object, the entries in the repertoire of the machine are made to include the OID of the machine. Concretely, we express this by specifying an entry in the repertoire of a machine using both the event-type and the OID:

$$(E, o, \dots)$$

Here E is an event-type and o is the OID of the machine. The ellipsis indicates that we are now going to add more to the entry, because of the second need for qualification.

The second need for qualification is more subtle. It is required because, in some cases, event-type is not a unique determinant of the meaning of an event to a machine. Ambiguity can occur when an event instance can be presented to two machines of the same type.

Consider the case of a "Transfer" event that moves money from one bank account to another. Clearly an instance of Transfer has a different effect in the two accounts: in one (the source) it causes a reduction in the balance but in the other (the target) it causes an increase. It may also be that the Transfer is subject to different protocol rules in the two accounts: for instance, if an account is in a "frozen state", this may mean that it cannot serve as the source of a Transfer but may not affect its ability to act as a target.

[4] An alternative formation, having one machine per OID, is also possible but adds complexity with no apparent advantage.

If both accounts are represented by a machine of the same type, it is necessary for the machine to know both the event-type (Transfer) and its role in the event (Source or Target) to determine its behaviour. To ensure lack of ambiguity, the role (e.g., Source or Target) that an object can play when engaging in an event is added to the event-type and OID to create an unambiguous repertoire entry thus:

$$(E, o, R)$$

Here R is a role name. To keep the formalisation uniform, and without loss of generality, we assume that all repertoire entries use this triple form, even if the role name is not required for disambiguation.

As would be expected, the roles that an event-type can play are specified in its metadata. The metadata for the Transfer event-type might be as shown in Fig. 4. This shows the attributes of the Transfer event and an indication, in parentheses, of the type of value that each attribute is allowed to take.

```
Transfer:
  Source (OID),
  Target (OID),
  Amount (Currency).
```

Fig. 4. Event Metadata

We shall use the names of the OID valued attributes, "Source" and "Target" in Fig. 4, as the names of the roles associated with the event-type. For example, the entry `(Transfer, 12345, Source)` in a machine's repertoire signifies that the machine will understand, and either accept or refuse but will not ignore, an event that wants to transfer funds, using the account corresponding to OID 12345 as the source of the transfer.

3.3 Binding

When an event is presented to a machine its treatment by the machine depends on whether the event is represented in the repertoire of the machine. We use the term binding to describe whether or not an event is represented in the repertoire of a machine.

When an event is created, some of its attributes (as defined by the event-type's metadata, see Fig. 4) take OIDs as their values. Suppose an event instance e of event-type E containing an attribute R with OID value o is presented to machine m. We say that the attribute is "bound" to m if m's repertoire contains the entry (E, o, R).

Based on this, we can define the possible levels of binding between an event e and a machine m as follows:

a. If e has no OID valued attributes, the binding between e and m is undefined.

b. If any OID valued attribute of e is bound to m, e is bound to m.
c. If all OID valued attributes in e are bound to m, e is fully bound to m.
d. If no OID attribute of e is bound to m, e is not bound.

The phrase "an event e is represented in the repertoire of a machine m" introduced in Sect. 2.2 we now formally define to mean "e is bound to m". Thus a machine m will ignore an event e that is presented to it unless e is bound to m.

3.4 Single Binding Rule

Consider an event of type E that has two OID valued attributes ($R1$ and $R2$) with the same OID value, o. There is the risk here that the model contains a machine that has both $(E, o, R1)$ and $(E, o, R2)$ in its repertoire, and that this machine will accept and react in two different ways to the event. This could give rise to non-deterministic behaviour.

To prevent this, we require that a given OID may appear as the value of at most one attribute in a given event instance, so that it is not possible for two attributes of the same event instance to bind to the same machine. Having an event instance that binds twice with a given machine is avoided by using another event-type that only binds once. Thus a "Suicide" event would be used in place of "Murder" if the murderer and victim are the same person.

3.5 Object Types

The scheme described so far does not place any restriction on the combinations of machine types that may be used to form an object. If a system has n machine types, any of the $2^n - 1$ combinations could be used. We now describe a more restricted, but much more useful, formulation in which there is a fixed number of predefined object types.

To do this we ensure that every system only allows a predetermined combination of machine types to share the same OID. Each combination of machine types allowed is specified as a set called an "object type". The set of object types supported by a system of type S is denoted by $\Omega(S)$ and is part of the metadata associated with the system-type.

4 Machine Metadata

Ultimately any protocol machine is defined in terms of "elementary machines": machines that are not defined (as systems) in terms of other machines.

The properties and behaviour of an elementary machine are determined by the metadata associated with its machine-type. This metadata determines:

a. The repertoire of the machine.
b. The initialized value of the local state of a newly instantiated machine.

c. The tests that the machine applies to the quiescent states before and after an event to determine whether or not it accepts the event.
d. The updates that it makes to its local state as the result of an event.

It is not the purpose of this paper to describe the details of a concrete modelling language, but some aspects of the metadata definition are described below.

4.1 Machine Meta-Repertoire

The metadata of an elementary machine type M defines the set of event-type/role combinations (E, R) that any machine of type M can understand. This set is called the "meta-repertoire" of M and is denoted by $\Lambda(M)$.

When an elementary machine is instantiated from metadata and given an OID, o, the machine's repertoire entries in the form (E, o, R) are created from the OID and the meta-repertoire in the obvious way.

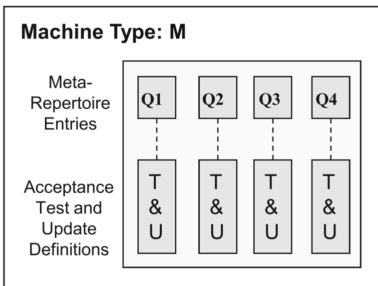


Fig. 5. Metadata Structure

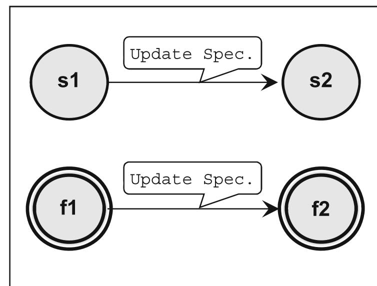


Fig. 6. Transition Notations

4.2 Machine Behaviour

For each entry in the meta-repertoire of a machine, the machine's metadata must provide:

- A definition of the tests that determine whether or not an event instance that matches the repertoire entry is accepted, and
- The updates to the machine's local state that are performed if it is.

The metadata for a machine can be thought of as having the logical structure shown in Fig. 5.

This picture depicts a structure for the metadata for a machine type, M. The upper boxes represent the entries of the meta-repertoire of M, so: $\Lambda(M) = \{Q1, Q2, Q3, Q4\}$. Each member of $\Lambda(M)$ has the form (E, R) described in Sect. 4.1.

The lower box for a given meta-repertoire entry defines how a machine of type M handles an event that matches the meta-repertoire entry. The handling is defined by:

a. Tests to be applied to the quiescent states before and after an event that determine whether or not the event can be accepted by the machine. These tests have as their domain the local state and the state environment of the machine.
b. The algorithm that defines the update to be applied to the local state of the machine as a result of the event.

4.3 A Graphical Notation

In principle, the metadata for the test and update definitions (the lower boxes in Fig. 5) can take any convenient form: for instance a programming language. However, the following graphical form is attractive and works well in practice.

The graphical notation uses state transitions. There are two variants of the notation, as shown in Fig. 6.

In both variants, the circles represent states of the machine, and the transition represents the effect of an event. In both cases, the diagram represents a successful event acceptance scenario.

In the upper variant, the states "s1" and "s2" are values of a distinguished stored variable (the "state variable") in the local state of the machine. The semantics of the upper diagram is:

a. The scenario is applicable if the value of the state variable in the quiescent local state of the machine before the event is "s1".
b. The scenario results in a set of updates, specified by `Update Spec`, being applied to the local state of the machine.
c. In addition to the updates specified by `Update Spec`, the scenario results in the machine's state variable being set to "s2" after the event, this being the only mechanism whereby the state variable can be changed.

In the lower variant, the circles (this time with a double outline) represent values that are computed by the machine, using a distinguished function called the machine's "state function". This is a function on the local state and state environment of the machine that returns an enumerated type, of which "f1" and "f2" are two possible values. Again, the diagram represents a successful event acceptance scenario, but with the following semantics:

a. The scenario applies if the value returned by the state function in the quiescent state of the machine before the event is "f1" and the value returned by the state function in the quiescent state after the event is "f2".
b. The scenario results in a set of updates, specified by `Update Spec`, being applied to the local state of the machine.

One or more scenarios can be used to specify the lower box metadata in Fig. 5 for a given entry in the meta-repertoire of a machine. An event presented to the machine is accepted if there is a corresponding scenario specified against the repertoire entry for the event that is successful according to the semantics defined above.

The rule that a machine must be deterministic (as specified in Sect. 2.2) requires that the scenarios are designed so that a given machine cannot have more than one successful scenario for a given event. This is most easily arranged by requiring that, where more than one scenario is specified for a given repertoire entry, the starting states of the scenarios are mutually exclusive.

Also note the following:

- The two variants are not mixed within a given machine-type. A machine type is either "stored state" in which case it has a single, distinguished, state variable as part of its local state and only uses upper variant scenarios; or it is "derived state" in which case it has a single, distinguished, state function that returns its state value and only uses lower variant scenarios.
- The success scenario diagrams for a single machine type can be "stitched together" to form a single graphical state transition diagram that represents the behaviour of the machine.

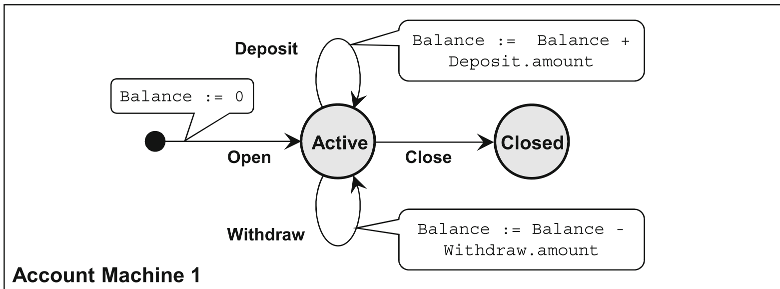


Fig. 7. Simple Bank Account

4.4 Example

Figure 7 shows a state transition diagram for a machine that represents a simple bank account.

Each arrow represents a scenario for a different meta-repertoire entry, and each is labelled with the meta-repertoire to which it belongs: **Open**, **Deposit**, **Withdraw** or **Close**[5]. This machine uses a stored state and thus uses the upper variant described in Fig. 6. The current state (**Active** or **Closed**) is stored as part of the local state of the machine and changes to its value are driven by the transitions as shown in the diagram.

Now suppose that an account can be frozen and that, while frozen, funds cannot be withdrawn. Figure 8 shows a machine that specifies this behaviour.

[5] Strictly speaking these meta-repertoire entries should be specified in the form (E, R). In these examples the role, R, is not required for disambiguation and we have omitted it for ease of reading.

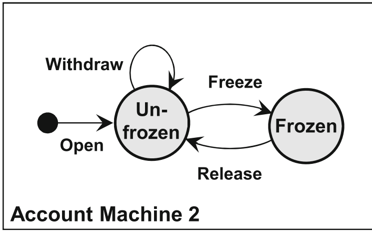


Fig. 8. Account Freezing

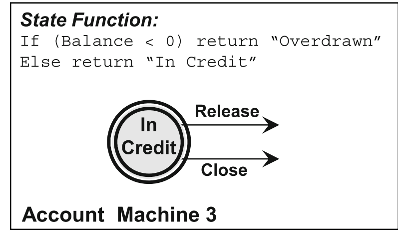


Fig. 9. Release\Close Control

This machine type, then, is added to the set that defines the Account object type.

This machine also uses a stored state and specifies four scenarios. The machine performs no updates to its local state, apart from the state variable update implicit in the transitions.

As the **Withdraw** event is subject to scenarios in both **Account Machine 1** and **Account Machine 2**, it can only be accepted if allowed by both: in other words, if the Account is both **Active** (in **Account Machine 1**) and **Unfrozen** (in **Account Machine 2**).

Suppose that additionally we want to specify that an account cannot be released (from a frozen state) or closed if it is overdrawn. A third machine with the metadata shown in Fig. 9 is added to the Account object type.

This machine specifies two scenarios, corresponding to the two arrows, one for **Release** and one for **Close**. The scenarios state that these events cannot take place unless the Account state as returned by the machine's state function before the event is **In Credit**. This machine uses the lower variant from Fig. 6 but, as we are not interested in the state after the events, no ending state is needed on the scenarios. Note that the state function refers to the Balance maintained by **Account Machine 1**, which forms part of **Account Machine 3**'s state environment.

Finally, suppose that the account is subject to an overdraft limit of £50. We add a fourth machine, shown in Fig. 10.

In this machine, the overdraft limit rule is expressed as a constraint on the state that results from a **Withdraw** event. If the state after the event is not as specified by this machine, the event is refused. So no withdrawal can be made that results in a violation of the £50 overdraft limit.

As a final note, a state function may be more complex than shown here. In a previous paper, Mixin Based Behaviour Modelling [3], we describe a model of marriages in which the marital status of a **Person** (**Single** or **Married**) is computed by detecting the presence of a valid **Marriage Contract**, another object type in the model.

4.5 State Spaces

Generally, the number of machines required to model the behaviour of an object is the number of separate "state spaces" that the object possesses. For instance,

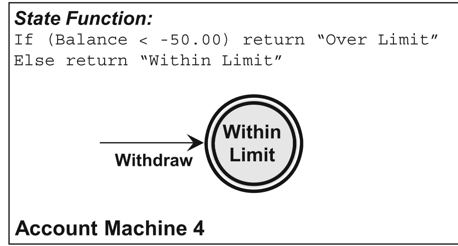


Fig. 10. Withdraw Control

a model of **Person** might have two state spaces, (**Single** or **Married**) and (**Working** or **Unemployed**), and would therefore require two machines to model its event protocol.

This is only a guideline. Sometimes it is appropriate to use more than one machine for a state space to improve model readability or to render machines more re-usable across different object types.

4.6 Proto Machines

To support the instantiation of new machines when a new object is created, it is assumed that a system has an inexhaustible supply of "proto-machines", being machines that have a type but no OID.

Proto-machines are in an initial state. The initial state of an elementary machine is specified by the machine's metadata. In particular, stored state variables are initialised to a value that corresponds to the starting pseudo-state (the black dots in Figs. 7 and 8). The initial state of a system is a system with no constituent machines. Proto-machines have no repertoire, so ignore all events.

The way in which proto-machines are used to create new objects is described in the following section.

5 Protocol Models

A "protocol model" is a protocol machine built out of other protocol machines by combining them recursively as systems[6]. A protocol model is privileged in having the following two capabilities, not shared by its constituent machines:

- The capability to determine whether or not a new quiescent state becomes a new stable state, and
- The capability to bring new objects into existence.

To qualify for this privilege, a protocol model must be closed (in the sense defined in Sect. 2.2) and requires that every event that is presented to it is fully bound to it, otherwise the result of presentation is undefined.

[6] Subject, of course, to the constraint that a system is not defined directly or indirectly in terms of itself.

5.1 Full Binding

The implication of full binding is that every OID valued attribute of an event is represented in the repertoire of at least one elementary machine in the model (see Sect. 3.3).

Consider, for example, the Transfer event in Fig. 4. Full binding means that when an instance of the Transfer event type is created, Accounts exist for both OIDs specified in the event. This does not mean that the event is necessarily accepted, as one or both Accounts involved in the transfer might refuse the event.

However it does mean that no OID in the event can be ignored and in this sense, the full binding requirement ensures that the behavioural semantics of a model is complete. This is a prerequisite for meaningful execution.

5.2 Determination of Stability

After an event has been presented to a model and the model has reached a new quiescent state, it is able to determine a new stable state for itself. It does this as follows:

a. If the event was accepted by the model, the new quiescent state becomes the new stable state.
b. If the event was refused, the new stable state is the same as the one that pertained before presentation of the event.

This determination defines the new stable state of the model itself and of all nested machines.

5.3 OIDs in Models

A model does not itself have an OID but every other machine in the model does have one. This is because every machine in a model, with the exception of the model itself which is at the top of the nesting, belongs to a system and so requires an OID as described in Sect. 3.1. In particular, every elementary machine in a model has an OID[7].

Because a constituent machine of a system can itself be a system, there can be many systems in a model, each having its own set of OIDs. This recursion can be used, for instance, to model an object that structurally owns a homogeneous population of further objects, as an Order owns Order Lines[8].

The OIDs of a model must satisfy the following rule: Given an OID, even one not currently used by any machine in the model, it must be possible to determine the system to which the OID belongs. In other words, there must be an "object ownership" function, ω, which gives the system s to which an OID o belongs: $s = \omega(o)$.

[7] We ignore the case of a model that consists of a single elementary machine.
[8] UML refers to this kind of ownership relationship as "aggregation".

5.4 Object Creation

A new object is created when an event containing an OID that does not currently exist (i.e. there are no machines in the model with that OID) is presented to a model. This is done before the level of binding between the event and the model has been determined.

Suppose an event contains an OID valued attribute with value o, and that no object exits in the model with this OID. The event creation mechanism must ensure that $\omega(o)$ exists at the time the event is presented. The model then creates a new object in the system $\omega(o)$ with OID o.

Object creation is done by assigning the OID o to a set of proto-machines in $\omega(o)$. The object creation mechanism must ensure that the set of types of the proto-machines selected to instantiate the object is a member of $\Omega(\tau(\omega(o)))$, the set of object types allowed in $\omega(o)$.

When an elementary proto-machine is given an OID, it also acquires a repertoire as described in Sect. 4.1. Because the repertoire of a system is the union of the repertoires of its constituent machines (see Sect. 2.3) the new repertoire entries percolate up the nesting hierarchy and contribute new entries to the repertoire of the model. Only after the model repertoire has been re-established is the level of binding of the event with the model determined.

5.5 Attribute Typing

The metadata for an event-type includes a "type" for each attribute of the event. Giving an event attribute a type is an instruction to the mechanism that handles event instance creation concerning the allowable values that may be loaded into the attribute. This mechanism is normally a user interface, although it could also be software.

For a non-OID valued attribute the type is a primitive value type such as String, Integer, Real, Date, Boolean, etc. with the obvious meaning for what may be loaded at event creation time.

For OID valued attributes we have so far specified the type as "OID" (for instance in Fig. 4) indicating that the attribute references an object. However, this gives no indication of what type of object is an appropriate addressee of the event, and therefore no guarantee that the event will be bound. A better scheme is to type event attributes that reference objects with a machine-type. In the next section, we show how it is then possible to ensure that events are properly (fully) bound to a model.

First, we define what is meant by giving an OID valued attribute a type. Suppose an attribute R of an event type E is given type M (a machine-type). Suppose that an instance e of type E is created, and that the attribute R is given the OID value o. To honour the type, M, of the attribute, the event creation mechanism must ensure that there is a machine of type M with OID o in the model. Moreover, this must be the case whether o is an existing object or a new object created according to the description in Sect. 5.4.

If a new object is to be created, and there is more than one object-type that meets the type match criterion given above, which one is chosen is undefined by the model and must be determined externally. This determination is normally by user choice at event creation time.

5.6 Design Time Binding

With OID valued attributes typed in this way, it is possible to ensure at design time (i.e., based on metadata of a model) that events are always fully bound to a model. We now describe a recipe for doing this.

First we define the meta-repertoire, $\Lambda(O)$, of an object type O as follows:

$$\Lambda(O) \equiv \{Q \mid (Q \in \Lambda(M)) \ \wedge \ (M \in O) \ \wedge \ M \text{ is elementary}\}$$

To guarantee that an attribute R in event-type E will be bound to a model X, we choose the type M for the attribute as follows:

a. Let Σ be the set of all system-types used in the metadata of X. Define $\Omega(X)$, the set of object types in X, as: $\Omega(X) \equiv \{O \mid (O \in \Omega(S)) \wedge (S \in \Sigma)\}$
b. Define the set $O(E, R, X)$ of all object types in X that have (E, R) in their meta-repertoire: $O(E, R, X) \equiv \{O \mid (O \in \Omega(X)) \wedge ((E, R) \in \Lambda(O))\}$
c. For the type of R in E, find a machine-type M such that any object using a machine of type M has (E, R) in its meta-repertoire: $\forall\ O \in \Lambda(X), M \in O \Rightarrow O \in O(E, R, X)$

This construction of the attribute type for R, combined with the assurance provided by the event creation mechanism that any value loaded into R must conform to type, guarantees that the attribute R will be bound to the model. So if this construction is carried out for all the OID valued attributes of an event type, event instances of that type will be fully bound.

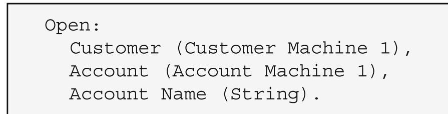
In step c of the construction it is possible that no machine-type M with the property required exists. In this case it is possible to create one as follows[9]:

i. Create a new elementary machine type M' with $\Lambda(M') = \{\}$. Any machine of type M' will have an empty repertoire so will ignore all events presented to it. Hence adding M' to an object-type has no effect on the behaviour of objects of that type.
ii. Add the machine-type M' to all object-types in $O(E, R, X)$ (from step b above). Use M' as the type for the attribute.

5.7 Example

We now expand the banking example introduced earlier to illustrate the effect of having an event bound to multiple objects.

[9] In practice, the need to create a machine type that has no behaviour is an indication that a model contains poorly chosen behavioural abstractions.



```
Open:
  Customer (Customer Machine 1),
  Account (Account Machine 1),
  Account Name (String).
```

Fig. 11. Metadata for Open

Consider the **Open** event, that opens a bank account for a customer with the metadata shown in Fig. 11.

This event has two OID valued attributes. The Account attribute is typed by the machine **Account Machine 1** (Fig. 7). The machine for Customer is shown in Fig. 12.

Suppose that an instance, e, of the **Open** event is created with value $o1$ for the Customer attribute and $o2$ for the Account attribute.

Assuming that e is fully bound and successful (not refused) then e must have been accepted by a machine of type **Customer Machine 1** with OID $o1$ and by a machine of type **Account Machine 1** with OID $o2$. Moreover, the $o1$ machine must have been in the state **Registered** and the $o2$ machine in the initial pseudo-state (the black dot on Fig. 7), otherwise the event would not have been accepted.

In the case of the Customer machine, there may not have been any change to the local state of the machine as a result of the event. The point here is that acceptance of an event by a machine signifies consent to the event being possible, and may or may not involve an update to the local state of the accepting machine. Designing the metadata of a protocol model is therefore about designing the "event consent schema" of each event type, across all the object types it involves. We emphasise this to dispel the possible misconception that an event-type only needs to appear in the metadata of a machine if an occurrence of the event actually alters the state of the machine.

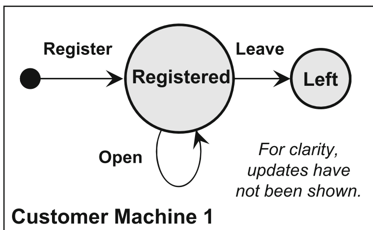


Fig. 12. Customer Machine

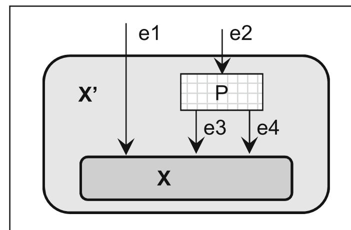


Fig. 13. Model Extension

5.8 Extended Models

It is possible to extend a model so that the effect[10] of one event is defined in terms of a set of generated events.

The general scheme is shown in Fig. 13. Here the model X' is created by extending the model X with:

- A new event type, EP.
- A process P that handles instances of EP.
- Protocol metadata within X that determines when events of type EP can be accepted.

Events presented to X' that are not of type EP are presented directly to X (as in the case of $e1$ in Fig. 13).

Events of type EP are presented to a process P that is in X' but outside X, as shown in the case of $e2$. P creates new events and presents them to X, as is the case with $e3$ and $e4$. The generated events are handled by X exactly as though X were a stand-alone model: P is simulating an environment for X by presenting it with events. This means that, while P is active, X determines acceptance and state stability for the generated events.

Note the following points about P:

- P is specific to an event type and is used for every instance of that event type presented to X'.
- P is not, itself, a protocol machine. When an event is presented to it, it must accept it.
- While P is active, X behaves like a model for all the events generated by P. This includes object creation in X and determining acceptance or refusal of the generated events.
- P has access to the attributes of $e2$ and to the interim stable states that X reaches between the presentation of successive generated events.
- Once P completes and X reaches a quiescent state, the quiescent state of X' is the same as that of X. In other words, P does not contribute to this state.

Although presented to X, $e2$ must not itself cause any change to the state of X. Instead, the required updates are performed by the events created by P. This rule ensures that there is no possible indeterminacy in the processing of $e2$ resulting from different interleavings of the updates required for $e2$ itself with those required for the generated events.

A necessary condition for the acceptance of $e2$ by X' is that every generated event is accepted by X. This is not a sufficient condition, as the metadata of X may impose protocol constraints on the acceptance of $e2$ that are not guaranteed by the acceptance of the generated events.

X' is responsible for determining state stability for events presented to X'. So the new quiescent state of X' after presentation of $e2$ becomes the new stable

[10] Here "effect" of an event means the change to the total stored state of the model resulting from its acceptance.

state if, and only if, all generated events are accepted by X and $e2$ itself is accepted by X'. Otherwise the new stable state of X' is the stable state prior to presentation of $e2$.

As an example of the use of model extension, suppose that in a banking system separate events have been defined to:

- Register the new customer, and
- Open a current account, and
- Open a savings account.

Suppose also that, for most new customers, it is usual to do all three of these and do them together. It would be possible to use model extension to define a new "standard customer set-up" event that registers a new customer and opens the two accounts as a single event.

Model extension can also be used to create and send events to a number of objects.

6 Behaviour Reuse

One of the motivations for the approach to behaviour modelling described in this paper is to support the reuse of behavioural abstractions. The basis for this is that the same elementary machine type may be included in the definition of many object types. As the behaviour of a given machine type is specified by its metadata, this translates into a mechanism for reuse of behavioural metadata across many object types.

This style of object behaviour definition by combining the metadata of smaller elements that can be reused conforms to the pattern described by Bracha et al. as a pure mixin approach [6].

While this provides the foundation for behaviour reuse, its utility is limited without a means to define behaviour that abstracts from particular event-types, as the next section illustrates.

6.1 Approach to Re-Use

Suppose that the Account example described in Sect. 4.4 is to amended to support two different types of account: a Current Account and Savings Account. A Current Account allows overdrafts, and each Current Account has an overdraft limit agreed with the customer at the time the account is opened. Savings Accounts, on the other hand, do not allow overdrafts.

When a Current Account is opened its overdraft limit is specified as part of the open event, so the open event must include this as an attribute in its metadata. The open event for a Savings Account does not have this attribute and is therefore a different event-type.

It makes sense to want to use **Account Machine 1** (Fig. 7), which describes the basic mechanism for account balance maintenance, for both types of account. However, there is an apparent problem here: how do you represent the "open"

event in this machine when there are two different kinds of open event, one for each of the two different kinds of account?

There are a number of possible mechanisms that might be considered, none of which is satisfactory:

- Include both event types in the definition of **Account Machine 1**. However, this pollutes the repertoire of each account type with an **Open** event that does not belong to it.
- Require that the set of events defined for the application be re-factored, for instance so that opening a current account is separated into two events: a generic open and another event that sets the overdraft limit. However, it is not proper that the vocabulary of event types should be driven or constrained by limitations of the modelling language.
- Take a similar line to that described above, but hide the internal configuration of events by using the Model Extension mechanism (see Sect. 5.8) to generate them. However, the consequent separation between the vocabularies of external and internal, generated, events dilutes the clarity of the protocols as statements about rules of the domain.

Instead of any of these, our route has been to build mechanisms into the modelling language that allow abstract events to be defined. Two mechanisms are involved:

- Conditional Repertoire Entries
- Repertoire Macros

These are described in the following sections.

6.2 Conditional Repertoire Entries

This is a mechanism that allows the behaviour of a machine to be influenced by its context, as defined by the other machines which belong to the same object.

Suppose that a machine type M has an entry $Q = (E, R)$ in its meta-repertoire. We can define a new machine type, $M/\{Q\}$, which is the same as M except that the entry Q is absent from its meta-repertoire. This means that a machine of type $M/\{Q\}$ will ignore requests to participate as player of role R in an event of type E. Otherwise the machine behaves exactly like one of type M.

Now we define $M//\{Q\}$, a machine in which Q has been made a "conditional" entry in the meta-repertoire of M. The behaviour of a machine of this type depends on its context in a protocol system. Suppose that a machine m of type $M//\{Q\}$ and with OID o is present in a system s. Then:

a. If any other constituent machine of s has the entry (E, o, R) in its repertoire, resulting from a non-conditional entry (E, R) in its meta-repertoire, m behaves as though it had type M.
b. Otherwise m behaves as though it had type $M/\{Q\}$.

The effect of this is that the meta-repertoire entry Q in a machine of type $M//\{Q\}$ is "active" (i.e., causes the machine to accept or refuse events) if and only if the machine is part of an object that has at least one other machine whose meta-repertoire contains Q as a non-conditional entry.

We will use underlining to denote a conditional meta-repertoire entry, thus: $\underline{Q}$.

6.3 Full Binding Revisited

With the introduction of conditional repertoire entries we need to revisit the discussion of full binding in Sect. 5.6 to check that the ideas set out there still work.

Suppose a machine type M that has a conditional meta-repertoire entry $\underline{Q}$ is used by an object type O, so that $M \in O$. If no other machine type in O contains Q as a non-conditional entry, Q is inactive in O and so should not appear in the meta-repertoire, $\Lambda(O)$, of O.

With this observation it is clear that, provided conditional meta-repertoire entries are ignored when compiling the meta-repertoire of an object, the full binding recipe works as before. The definition of the meta-repertoire of an object type given previously in Sect. 5.6 is therefore amended to:

$$\Lambda(O) \equiv \{Q \mid (Q \in \Lambda(M)) \wedge (M \in O) \wedge M \text{ is elementary } \wedge Q \ \textit{is non-conditional in } M\}$$

6.4 Conditional Entries and Reuse

Returning to the example of the current and savings accounts, we now explain how conditional repertoire entries allow machine metadata reuse.

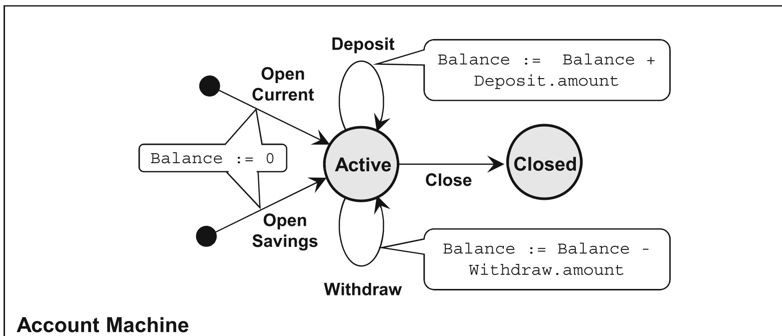


Fig. 14. Modified Bank Account

We start by assuming that we want to use a common machine to describe the basic account mechanism of maintaining a balance, as **Account Machine 1** did in our earlier example, but that we include both types of account opening event. The machine is described in Fig. 14.

This machine type allows either of the two (different) event types **Open Current** or **Open Savings** to be the first event in its lifecycle. Either event has the effect of initialising the balance and enabling deposits and withdrawals to take place until the close event. However, if used as it stands in the context of defining both the current and savings account objects, both objects will end up with both types of open event in their repertoire, and this is not appropriate.

To avoid this, we can create a reusable version of the machine by forming the machine:

$$\textbf{Account Machine}//\{(\textbf{Open Current}, \text{Account}), (\textbf{Open Savings}, \text{Account})\}$$

in which the two entries for the open events are made conditional[11].

Note that the Current Account object must contain at least one (non-reusable) machine that includes **Open Current** non-conditionally in its meta-repertoire, otherwise **Open Current** would not appear in the meta-repertoire of Current Account at all. Assuming however, as is reasonable, that none of the machine types involved in defining the Current Account object contains **Open Savings** non-conditionally in its meta-repertoire, **Open Savings** will not be in the meta-repertoire of Current Account at all. A similar argument applies to Savings Account.

6.5 Repertoire Macros

While the conditional repertoire entry mechanism can support the creation of reusable machines by making it possible to "switch off" the entries in a machine's repertoire not relevant to the context of a particular object, we have not captured an idea of abstraction that would allow the distinction between two event types (such as the two open events) to be hidden where it is not required. The second mechanism, Repertoire Macros, is the basis for doing this.

Suppose a machine type has the metadata shown in Fig. 15.

Consider the relationship between the metadata for $Q1$ and an event instance that it processes. In particular we need to understand what information about an event the metadata needs to obtain from an event instance at run time. Because the metadata, by assumption, is specific to an event type and role (as specified in $Q1$) these are known to the metadata by design, and do not need to be obtained at run time. Therefore, only the values of event attributes need to be obtained at run time. This means that the metadata definitions for two meta-repertoire entries can be identical if:

- the event attributes referred to by the two metadata definitions are present in the metadata of both event types, and
- the processing required (i.e., both testing for event acceptance and performing local state update) defined by the two entries is the same.

[11] For concreteness in showing the meta-repertoire entries it is assumed that the both of the open events have an attribute called Account that takes the OID of the (new) Account being opened.

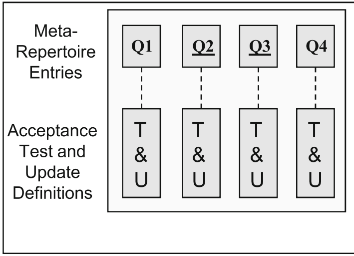


Fig. 15. Base Version

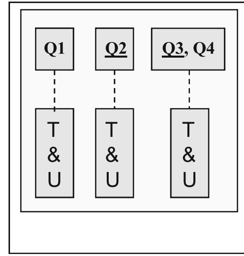


Fig. 16. Shared Metadata

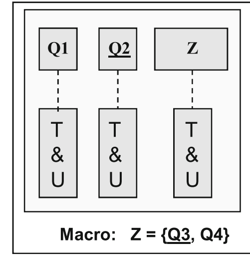


Fig. 17. Repertoire Macro

Under these circumstances, the two entries can share a single lexical copy of the metadata. This is shown in Fig. 16. Figure 16 represents the same machine as Fig. 15 but recognises that the (conditional) meta-repertoire entry $\underline{Q3}$ and the (non-conditional) entry Q4 have identical metadata and can share the same lexical copy. We now provide a simple compile-time lexical replacement (or macro) facility to re-write Fig. 16 as shown in Fig. 17.

Using this technique in the context of the **Account Machine** defined earlier, we can define a macro:

$$\textbf{Open} = \{(\underline{\textbf{Open Current}, \text{Account}}), (\underline{\textbf{Open Savings}, \text{Account}})\}$$

In this definition, **Open** is a macro name now defined in the meta-repertoire of **Account Machine**, but which is replaced by the two conditional meta-repertoire entries for the two types of open event at compile time. This macro definition is possible because the metadata treatment of the two open events in the context of the **Account Machine** is identical, as can be seen in Fig. 14. This is equivalent to noting that, at least in the context of Account Machine, abstraction from the particular types of open to a generic open is appropriate, and the Open macro can be viewed as naming this abstraction.

We can now restore Fig. 14 to the simpler form that it had in Fig. 7, with the single arrow for opening an account, supplemented by the macro definition that makes it re-usable across the different types of account.

7 Discussion

7.1 Behaviour Encapsulation

Most object-oriented modelling techniques, in particular the UML, do not have a primitive notion of an "event" as described here and typically events are modelled as objects. A "Transfer" event that moves funds between accounts would have its own class with responsibility for checking that a transfer can complete, i.e. that both the source and target accounts exist and are in a state to participate

in the transfer. This checking might include a check that the source account is not frozen, as this might prevent the account from being used in this role.

Freezing an account would also prevent cash withdrawal, so the class that models the Withdraw event would need to include a similar check.

In this approach the behavioural states and the event protocol of an object are implicit in the tests that transactions perform to determine whether or not they can proceed. This has two consequences:

- Each transaction can potentially take a different view on how the attributes of an object determine its state. So that there is no guarantee, for instance, that Transfer and Withdraw determine whether or not the account is frozen in the same way. If they do it in different ways, the behaviour of the system becomes incoherent.
- Because event protocols are embedded in the transactions and not the object, sub-classing the object does not sub-class the protocol. This means that sub-classing cannot be used to create behavioural variants of an object in the way that one might hope or expect.

We think that these are symptoms of poor encapsulation of behaviour. Neither of these issues can arise in protocol based modelling as the protocol of an object is a property of the object itself, by virtue of the protocol machines used to define it.

7.2 MDA

Our focus on model execution is driven by a belief that execution at the model level has value in providing a means of validating models early in the systems development lifecycle. The idea that models embody behavioural semantics that potentially support model execution is also part of the vision of the Model Driven Architecture initiative sponsored by the Object Management Group. In this section we relate our work to some of the stated aims of MDA and compare the protocol machine approach with other approaches that also aspire to address these aims.

MDA and Model Execution. The OMG characterises 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* [12]. (The underlining is ours.)

Moreover, Richard Soley, CEO of OMG, says that one of the aims of MDA is that *Models are testable and simulatable* [14]. 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 [models]* [11]. As Oliver Sims points out, the term "computationally complete" means capable of being executed.

Contract	**Protocol**
A "Contract" is specified in terms of pre- and post-conditions, with the following meaning:	A "Protocol" is specified using pre- and post-constraints, with the following meaning:
If the pre-condition of a function is true before invocation, then the function must ensure that the post-condition is true after invocation. If, on the other hand, the pre-condition is not true, the result of invocation is unspecified.	*If software that exhibits a protocol is presented with an event, then it will refuse to engage (i.e., will not undergo any permanent change of state) if either a pre-constraint is false before the event and/or a post-constraint is false after the event.*
Contracts are a mechanism for formal specification of a function, by placing conditions on what the function returns without constraining the choice of algorithm.	Protocols are a mechanism for specifying the behaviour of software by defining the relationships between the states of the software and its ability to accept events.

Fig. 18. Contracts versus Protocols

Other MDA Approaches. Here we compare our approach to other work currently being promoted under the MDA banner. There are two main camps of MDA, both of which take the UML as their basis, and we consider each in turn below.

The first camp is based on "Design By Contract" [4], and uses the Object Constraint Language (OCL) [7] to specify contracts in terms of pre-and post-conditions on operations invoked in objects. The claim is that adorning a UML structural (class) model with contracts enables behaviour to be captured formally [1].

While the language of Design by Contract (pre- and post-conditions) is very similar to that used in this paper for protocols (pre- and post-constraints), it seems clear to us that Contracts and Protocols are conceptually different (see Fig. 18). In particular, the language of contracts does not bring with it any new primitives with behavioural semantics to raise the level of abstraction at which behaviour is described. This means that a model that expresses behaviour as contracts will either not be executable (and therefore not simulatable or testable) or will not be at a level of abstraction above a programming language.

The second camp is that termed "Executable UML", which is based largely on the work of Shlaer and Mellor [17]. In this approach, state machine descriptions of object life-cycles are adorned with definitions of the processing performed by the object expressed in a high level imperative language, the Action Language [16].

This work bears superficial resemblance to ours in its use of state transition diagrams as a means of describing object behaviour and in its aim of providing model executability. Moreover, the state machines do provide some level of explicit definition for the behavioural states of an object. However, the state machine semantics in Executable UML are different from ours, in that a state machine may ignore an event but cannot refuse one. Without the ability to refuse events, state machines cannot describe event protocols in situations where an event must be accepted by multiple objects, which is usual in transactional business systems. This leads to the need to model transactions (events) as classes in their own right, containing behaviour rules that check event acceptability across multiple contexts. This approach has the encapsulation weaknesses that we outlined in Sect. 7.1.

Two other points where the Executable UML approach differs from that described in this paper are also worth noting:

- The approach advocates a single state machine per class. An object with multiple state spaces, such as the Person example in Sect. 4.5, would need to be modelled using more than one class.
- There is no concept of a calculated state, as described in Sect. 4.3. This means, in general, that state spaces such as (In Credit or Overdrawn) can only be handled by generating internal events that have no domain meaning to drive the machine between these states.

Finally, the Executable UML technique offers no mechanism for behaviour reuse which, in the literature describing the method, is explicitly discouraged.

8 Implementations

The ideas presented in this paper have emerged from 15 years of work building and using tools that support the behavioural design of transactional business systems.

Our focus in this work has been to provide tools that allow behavioural models, described as protocol machines, to be executed and tested early in the development lifecycle. This early testing reduces the risk that severe behavioural problems are found at late stages of development, when rectification can be very expensive. The executable models can be viewed as a form of prototype, and the testing and exploration of model based prototypes provides a vehicle for users and other stakeholders to engage in the modelling process even if they have no understanding of the notations and concepts used to build the model.

All of these tools are aimed at exploring and validating behaviour with users. The user interface for driving model execution is therefore designed to be understandable by people who do not know anything about the modelling concepts used. In particular, the user interface presents object instances without revealing their internal machine level structure or requiring the user to understand this structure.

The first version of the tool, built in 1989, used the concepts and notations of Jackson System Development [8], an early object-based modelling approach developed in the 1970s. This tool only allowed one machine per object and had no support for behaviour reuse. Also, this version had a primitive user interface that required OIDs to be constructed and entered explicitly by the user.

In 1993 we moved to using state transition diagrams as the basic notation for defining protocols. This allowed a more intuitive style of user interface, as it became possible to grey out the event-types incapable of being accepted by an object because they violate a pre-constraint. A new approach to binding events to the model was introduced based on selecting object instances from user interface lists rather than entering OIDs; and OIDs became completely hidden from the user. This tool also supported multiple machines per object and extended the notion of state transition diagrams by allowing states to be

calculated, as described in Sect. 4.3. However, there was only limited support for machine re-use.

The most recent tool, ModelScope [10], developed in 2002, improved the metadata language and added support for the event abstraction ideas described in Sect. 6, greatly increasing the capabilities for behavioural reuse.

References

1. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
2. McNeile, A., Simons, N.: State Machines as Mixins. J. Object Technol. **2**(6), 85–101 (2003)
3. McNeile, A., Simons, N.: Mixin Based Behaviour Modelling. In: 6th International Conference on Enterprise Information Systems, vol. 3, pp. 179–183 (2004)
4. Meyer, B.: Object-Oriented Software Construction. Prentice Hall PTR, Englewood Cliffs (2000)
5. Hoare, C.: Communicating Sequential Processes. Prentice-Hall International, Englewood Cliffs (1985)
6. Bracha, G., Cook, W.: Mixin-based Inheritance. In: ASM Conference on Object-Oriented Programming, Systems, Languages, Applications, pp. 179–183 (1990)
7. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
8. Jackson, M.: System Development. Prentice Hall, Englewood Cliffs (1983)
9. Jackson, M., Zave, P.: Domain Descriptions. In: Proceeding of IEEE International Symposium on Requirements Engineering, pp. 56–64 (1993)
10. Metamaxim Ltd., ModelScope. Metamaxim Website: http://www.metamaxim.com. Accessed Feb 2004
11. Sims, O.: MDA: The Real Value. OMG Website: http://www.omg.org/mda/presentations. Accessed Feb 2004
12. Object Management Group. How Systems Will Be Built. OMG Website: http://www.omg.org/mda. Accessed Jan 2004
13. Object Management Group. UML 2.0 Superstructure Final Adopted Specification. OMG Document reference ptc/03-08-02, August 2003
14. Soley, R.: MDA: An Introduction. OMG Website: http://www.omg.org/mda/presentations. Accessed Feb 2004
15. Cook, S., Daniels, J.: Designing Object Systems: Object-Oriented Modelling with Syntropy. Prentice-Hall, Englewood Cliffs (1994)
16. Mellor, S., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
17. Shlaer, S., Mellor, S.: Object Life Cycles - Modeling the World in States. Yourdon Press/Prentice Hall, Englewood Cliffs (1992)