

The Semantics of Nested Protocol Machines

During recent discussions with colleagues about the potential for using Protocol Modelling ideas to model Aspects, it occurred to us that two features of Protocol Modelling, as it is described in the paper “Protocol Modelling” paper¹ published in SoSyM², seem similar and could possibly be combined. These features are:

- Nesting: The idea that one Protocol Machine (PM) can be nested, or embedded, inside another; and
- Model Extension: The idea that a Protocol Model can be “extended” by adding event handling procedures that can execute logic and generate further events.

The questions of interest are:

- If a model m1 is extended to create a new model m2, by the addition of “event handling” routines for some of the events in m1, is it possible to view m1 as being nested inside m2?
- Is it possible to formalise the idea of nesting, and will this increase the expressive power of Protocol Modelling in a useful way?

We think that the answer to these questions is “Yes”, and this paper proposes an enhancement to the definition of Protocol Modelling in this area.

1 The Idea of “Nesting”

The essential idea of nesting is that one machine can be embedded or nested in another. Nesting is described very briefly in the SoSyM paper (in Sections 2.2.2 and 2.2.4) but there this description is sketchy and incomplete. In particular, while it is stated that: “*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*”, there is no definition of whether or how the behaviours of the inner and outer machines are combined or what relationship they might have. The main objective of this paper is to rectify this, by providing a complete definition of nesting and an exact description of behavioural relationship between the inner and outer machines of a nesting.

2 Proposal for a Semantics of Nesting

This section makes a series of proposals concerning the semantics of nesting.

The proposals in this paper make some substantive changes from the definitions in the SoSyM paper. However, these changes are improvements in that they contribute to a more complete and rationalized formulation of the Protocol Modelling language, and greater expressive power. The key rationalization achieved is in unifying nesting and Model Extension. However, there are some other improvements in expressiveness of the language, as described in Section 3.

¹ A. McNeile, N. Simons. Protocol Modelling. A modelling approach that supports reusable behavioural abstractions. *The Journal of Software and System Modeling*, 5(1):91–107, 2006.

² The Journal of Software and Systems Modeling, published by Springer (www.springerlink.com)

2.1 Explicit Event Submission

Figure 1 shows two machines, $m1$ nested inside $m2$. Associated with both machines is a state transition diagram, representing the definition of event handling of that machine. Each machine has its own local storage. Recall that the local storage, $\sigma(m)$, of a machine, m , is “owned” by m , in the sense that m and only m may alter values stored in $\sigma(m)$.

Recall also that, in the state transition diagram style representation of protocol models, the code for the updates that a machine makes its local storage as the result of accepting an event is shown as a bubble attached to a transition in the diagram.

Figure 1 shows two machines, $m1$ and $m2$, with $m1$ nested inside $m2$.

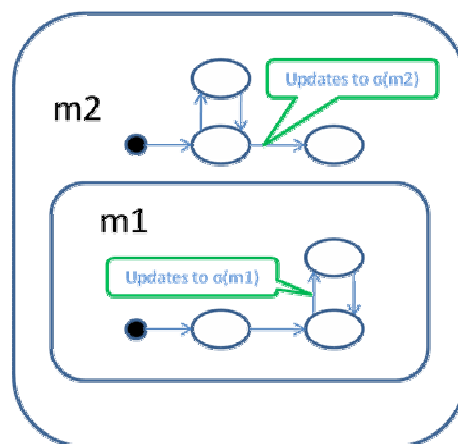


Figure 1 Nested Machines

The nested machine $m1$ forms a wholly owned part of $m2$. As such, the local storage of $m1$, $\sigma(m1)$, is to be regarded as forming an extension to the local storage, $\sigma(m2)$, of $m2$.

Because $m1$ is, itself, a protocol machine, the only way that its state and local storage, $\sigma(m1)$, can be changed is by presenting it with an event – any direct update of $m1$'s local storage by another machine (in particular, by $m2$) is not allowed. If $m2$ wants to change the state and local storage of $m1$ it must present it with an event, and this it does by executing a “submit event” command as part of the code attached to one of its transitions. In reaction to this submission (or presentation) of an event, $m1$ will then “fire” a transition of its own and perform the updates to its own local storage, $\sigma(m1)$, associated with this transition in the usual way.

This is shown in Figure 2. In this figures, $m2$ submits an event instance “ e ” (of type “ E ”) to $m1$. Within the definition of $m1$ there is a transition corresponding to events of type E which will fire, and associated updates will be performed on the local storage, $\sigma(m1)$, of $m1$.

Because $m1$ is a protocol machine there nominally the possibility that $m1$ is not in a state in which it can accept the event e that $m2$ submits to it, and refuses it: this is a “protocol error”. Normally (i.e., in the context of an event submitted by the User at the User Interface rather than an event submitted by one machine to another) a protocol error would cause the current user event (i.e., the event that was submitted at the User Interface) to be abandoned and the state of the model to being restored to that which pertained before this user event began. However, that is not

appropriate here. The process m_2 which submits the event to m_1 has full knowledge of the state of m_1 and therefore of whether or not it is capable of accepting an event submitted to it; and, if properly designed, m_2 should never submit an event to m_1 that m_1 is unable to accept. This leads to the following rule about event submission in the context of machine nesting:

Rule: If a machine m_2 submits an event e to another machine m_1 that is directly nested within it, and m_1 refuses that event, then the model is badly formed and a run-time error is generated.

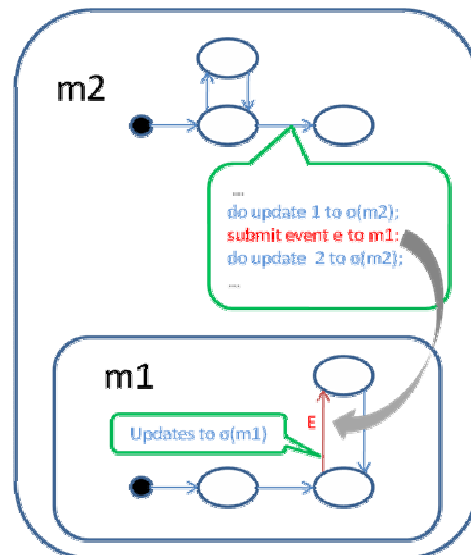


Figure 2 Event Submission to a Nested Machine

In other words, this is a bug in the model and the model must be changed to put it right. Nesting of machines is a form of strong ownership, and refusal of a submitted event is manifesting the fact that that owning machine does not understand how to achieve a successful update of an owned machine. This situation is to be viewed as similar to, for instance, a process trying to perform a numerical operation on a string variable in its local storage.

2.2 Event Submission

When a machine submits an event to another machine, the effect is similar to a “call” (i.e., invocation of a method or a subroutine) in conventional programming languages. In particular, it is synchronous in that the submitting machine is blocked (stuck at the “submit” statement) until the machine receiving the event has finished handling it and returned control.

The event that is submitted is a populated data structure, and conceptually similar to a structure of parameter values in a conventional call.

The difference between a “submit” and a conventional “call” is that, in the case of the submit, the receiving machine will determine whether it will *ignore*, *accept* or *refuse* the presented event, and react as follows:

- If it *ignores* the event, do nothing and return control to the submitting machine;
- If it *refuses* the event, generate an exception (this is a “protocol error”);

- If it *accepts* the event, handle the event according to the rules of the receiving machine.

2.3 Behavioural Semantics of Nesting

This section defines the behavioural semantics of nested machines, in terms of how a machine that uses nesting reacts to events presented to it. Consider the scenario shown in Figure 3, where an event *e* is presented to *m2*, which has *m1* nested inside it.

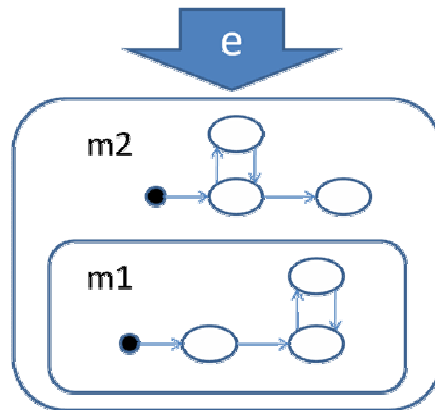


Figure 3 Presentation of an Event to Nested Machines

We refer to machine shown in Figure 3 (comprising *m2* with *m1* nested within it) as “*mc*” (the “combined machine”) to distinguish it from the individual machines (*m1* and *m2*) that contribute to its definition and behaviour.

The definition of the semantics of the *mc* must cover:

- The **protocol** that *mc* presents to its environment – i.e., how *mc* determines whether to ignore, accept, or refuse events presented to it
- The **processing** that *mc* performs when it accepts an event.

Each of the machine *m1* and *m2* in Figure 3 has an alphabet, $\lambda(m1)$ and $\lambda(m2)$ respectively. The alphabet of *mc* is the union of these, so: $\lambda(mc) = \lambda(m1) \cup \lambda(m2)$. The full set of possibilities is represented in the Table 1 below.

	$e \notin \lambda(m2)$	$e \in \lambda(m2)$
$e \notin \lambda(m1)$	mc ignores <i>e</i> .	Protocol: As specified by <i>m2</i> . Processing: As specified by <i>m2</i> .
$e \in \lambda(m1)$	Protocol: As specified by <i>m1</i> . Processing: As specified by <i>m1</i> .	Protocol: As specified by <i>m2</i> . Processing: As specified by <i>m2</i> .

Table 1 Behavioural Rules for Nesting

This table specifies the **protocol** that is applied to event, in other words what governs whether the event is accepted or refused by *mc*; and the **processing** that is applied, in other words what is done with the event by *mc* if it is accepted. Note the following about this table:

1. There is a general principle followed in this table that the machine that specifies the protocol also specifies the processing.
2. According to the lower left cell of the table, if the event is not in the alphabet of *m2* but is in the alphabet of *m1*, the protocol presented by *mc* is as specified by *m1*. The event “passes

through” m2 and is handled by m1 as though m2 were not there. This property of nesting we call *transparency*.

3. According to the lower right cell, if the event is in the alphabet of both m1 and m2, only the protocol of m2 is visible externally and governs acceptance or refusal of the event by mc. The protocol specified by m1 is ignored and, in particular, there is no CSP style composition of the protocols of m1 and m2.

It is important to understand that, if an event is in the alphabet of m2, it does **not** automatically pass through to m1. If m2 accepts it, m2 may submit the event explicitly to m1 in the code associated with the transition for the event (as shown in Figure 2), but it is not obliged to do so. If m2 **does** submit the event to m1, then there are two cases to consider:

1. If the event is also in the alphabet of m1, it must be accepted by m1, otherwise there will be a run-time exception (as described above in Section 2.1). In other words, this is a bug in the model.
2. If the event is not in the alphabet of m1, m1 ignores it (but this is not an error, and no exception is generated).

If m2 does **not** explicitly pass the event to m1, even though it is in m1’s alphabet, arguably the model is incoherent – but this is a matter for the modeller and the language does not impose this coherency.

2.4 Comparison with the SoSyM paper

In the SoSyM paper, there is a discussion of how nested machines handle an event, and the values of storage that is obtained when one machine accesses the local storage of another during processing. This is discussed, in particular, in the SoSyM paper (Section 2.2.6). This discussion needs to be revisited in the context of the new definitions being proposed in this paper.

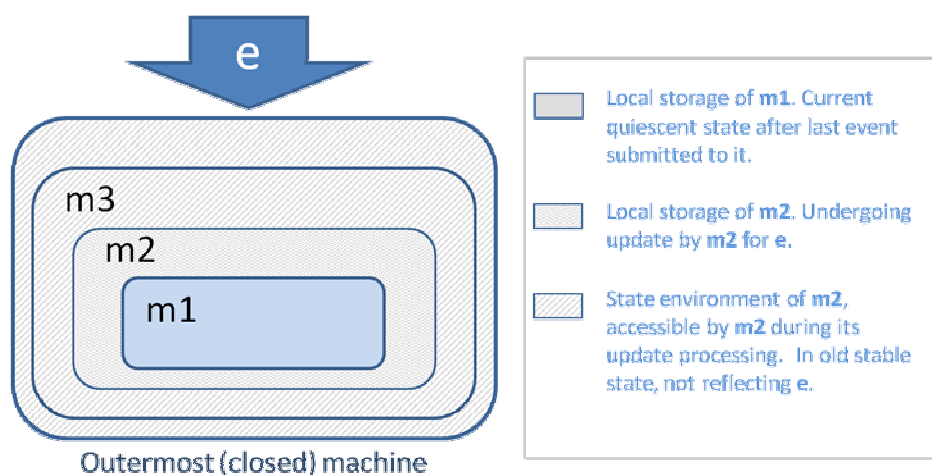


Figure 4 Access to Local Storage

Figure 4 here is a revised version of the figure shown in Section 2.2.6 of the SoSyM paper. The differences are as follows:

- In the SoSyM paper, it was assumed that the event e is presented “automatically” to each machine in turn, starting with the most deeply nested (in this case, $m1$) and working outwards. This does not happen with the new proposal: the processing works from outer level to inner, and events are only submitted from an outer to an inner machine explicitly; so there is no “automatic” submission. During its processing, resulting from the presentation of e to the outermost machine, $m2$ may submit any number of events to $m1$.
- If an outer machine ($m2$) accesses the local storage of a machine nested within it ($m1$), it obtains the quiescent value of local storage of the accessed machine reflecting the last event submitted to it.

This means that, if $m2$ submits multiple events to $m1$ during the processing of the single event e presented at the outmost level, $m2$ can see the quiescent state of $m1$ between these events. This brings the way $m2$ “sees” $m1$ into line with the way $m2$ “sees” its own local storage, where $m2$ can see the coherent state of its own local storage after execution of each update statement it performs on it.

2.5 Objects and Instantiation

The SoSyM paper describes the ideas of “Protocol Machine” and “Protocol System” (in Sections 2.2 and 2.3). Here we describe a slight elaboration on this idea that has advantages in unification of the formulation of protocol modelling. In the context of this proposal, we make a slight change to the way that these are defined.

1. A “Protocol Machine” is a machine which supports a fixed population of nested machines. The nested machines are instantiated with the parent machine, and instantiated once each, and are composed using CSP. The set of machine types that are instantiated with the parent are specified as part of the metadata of the parent machine.
2. A “Protocol System” is a machine which supports a variable population of nested machines. The set of machine types is fixed but, within a machine type, any number of instances of that type can be supported. All the instances nested within the Protocol System are composed using CSP. When a Protocol System is instantiated it is empty (has no nested instances), and instances are created during execution as described in the SoSyM paper, Section 5.4.

Figure 5 illustrates the distinction.

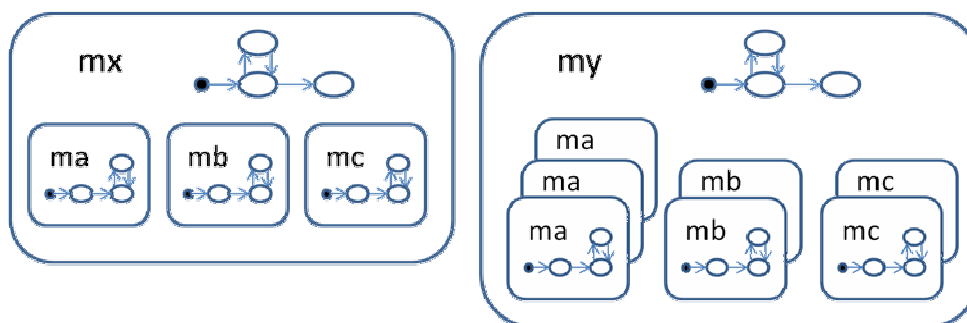


Figure 5 A Protocol Machine and a Protocol System

On the left is a **Protocol Machine** mx of type MX. The metadata for MX states that, when instantiated, it has one each of an instance of MA, MB and MC nested within it.

On the right is a **Protocol System** my of type MY. The metadata for MY states that it is capable of supporting instances of MA, MB and MC nested within it. When my is instantiated, it has no nested machines within it. The nested instances are created as the model executes, according to the technique described in the SoSyM paper (Section 5.4).

2.6 Object Identifiers (OIDs)

In the context of object modelling, and as described in the SoSyM paper (Section 3), every machine in a protocol model has an OID³ which ties the machine to the object whose partial definition it represents. As nesting represents a form of strong ownership, the only sensible rule is that a nested machine belongs to the same object as the parent machine.

In the case of a Protocol Machine (left hand picture of Figure 5) this means that the nested machines (ma, mb, and mc) have the same OID as their parent, mx.

In the case of a Protocol System (right hand picture of Figure 5), the nested machines have their own unique OIDs, but these are tied to the OID of the parent in the sense that, given the OID of one of the nested machines, the OID of the parent can be derived from it.

On a final (and speculative) note, it appears that the distinction made here is the same as the distinction between “Aggregation” and “Composition” in UML, so that the two scenarios shown in Figure 5 can be depicted in UML as shown in Figure 6.

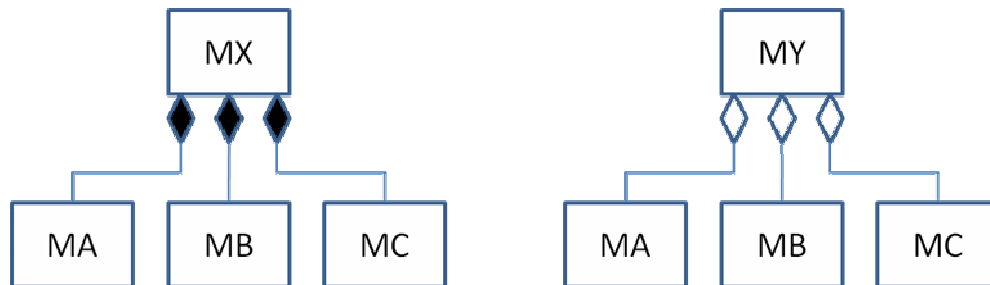


Figure 6 Equivalent UML Notations?

3 Unifications and Uses

This section describes some of the uses of the idea of nesting, and the way in which it unifies some features of the Protocol Modelling language.

3.1 Model Extension

As mentioned in the introduction to this paper, one of the motivations of the ideas is to unify the ideas of nesting and model extension.

The Model Extension technique is described in the SoSyM paper (Section 5.8). It involves creating “event handling” procedures that capture and process user events submitted to a Protocol Model. In

³ Object ID

doing so, the procedure may submit the user event to the model and may create new events and submit them to the model.

The purpose of Model Extension is to allow new events to be defined in terms of existing events. For instance, in a Banking Model we may have defined a *Register* event to establish a new Customer, and an *Open* event to open an Account for a Customer. But suppose it is the case that, almost always when a new customer joins the Bank, he/she will be registered and an Account opened at the same time. In this case, it seems sensible to have a single event, *Set Up*, that does both.

There is a close match between the capabilities of these event handling procedures and the processing that this paper proposes may be attached to a transition, as described in Section 2.1 above. The main difference is that the event handling procedures are not protocol machines: they do not specify any protocol for the events they handle, and all the protocol rules are contained in the model that is being extended.

As an example, consider the *Set Up* event in a banking system described above. Using the Model Extension approach, this would be modeled by:

- Adding the new *Set Up* event to the Customer model, as an alternative first event to *Register* (see Figure 7 left hand side)
- Creating an event handling procedure that captures a *Set Up* event submitted to the Model and converts it into a combination of a *Register* and an *Open* (see Figure 7 right hand side).

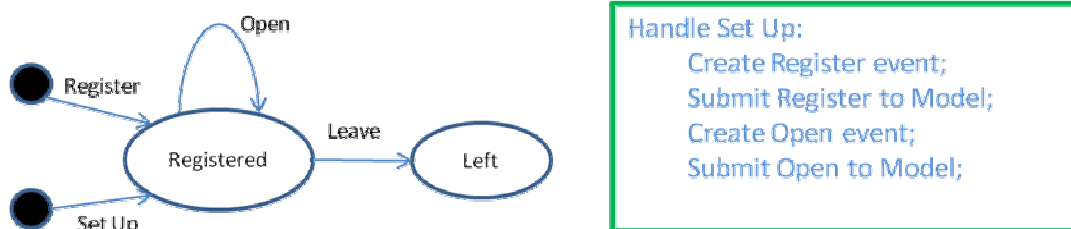


Figure 7 Register Event using Model Extension

Using the ideas proposed in this paper, the event handling procedure becomes the processing of a Register transition in a machine at the model level, and this machine must also specify the protocol for the event. The situation is depicted in Figure 8.

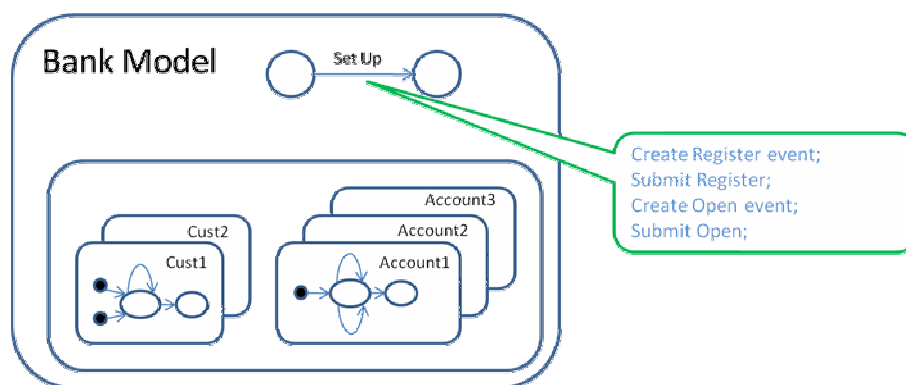


Figure 8 Register Event using Nested Machines

The event handling code is now the update code associated with the machine shown at the model level, with just a single transition for the Set Up event. If this transition is fired, the associated code is executed, having the identical effect to the event handling routine in the Model Extension approach.

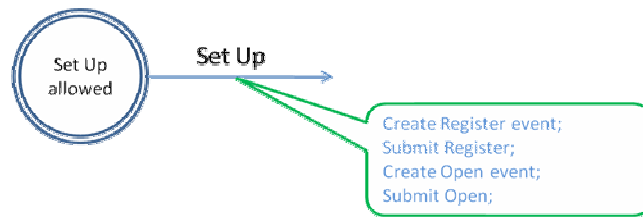


Figure 9 Definition of Machine for Set Up

As stated earlier, the protocol for the Set Up event must be specified in the machine that processes this event (see Section 4) and we must therefore specify appropriate states in this machine. We are only interested in specifying a pre-state (as the event is not constrained by the state that results from its acceptance) and this has to be a derived state, as the actual protocol is specified in the machine for Customer as shown in Figure 7. There is no problem about deriving this state, as the machine has access to the local storage (including state) of all nested machines, and in this case this means all the machines in the model. The appropriate full definition of the machine is shown in Figure 9. Recalling that a pre-state on a transition always represents a test as to whether the event matching the transition is to be accepted or refused, the test that is applied by this pre-state is as follows:

If the Customer referenced (by its OID) in the submitted Set Up event allows the Set Up (according to the protocol of the nested machines), then the event is accepted.

As a final note, using the nesting technique offers a different and, in a sense, superior way of handling this example, as follows:

Omit the *Set Up* transition from the Customer machine (see Figure 7), and code the pre-test in Figure 9 as *Register Allowed* (instead of *Set Up Allowed*). This is specifying that Set Up is to obey the same protocol rules as Register. This approach has the advantage that all definitions pertaining to the new event (Set Up) are contained in the new machine, without polluting the original model and thereby yielding improved separation of definition.

3.2 The ModelScope “INCLUDES” Construct

The language used by ModelScope uses a construct for defining assemblies of machines to be composed in the definition of an object. The idea is to support the creation of “sub-assemblies” of composed machines that can be re-used in different contexts.

It would appear to be possible to reposition this facility as a use of nesting as follows.

In the current ModelScope language, an Object that comprises CSP composed machine types MA, MB, MC, MD, and ME might be defined as follows:

```

OBJECT MA
  INCLUDES MB
  ...
BEHAVIOUR MB
  INCLUDES MC, MD
  ...
BEHAVIOUR MC
  ...
BEHAVIOUR MD
  INCLUDES ME
  
```

This could be represented as a nesting structure, as shown in Figure 10.

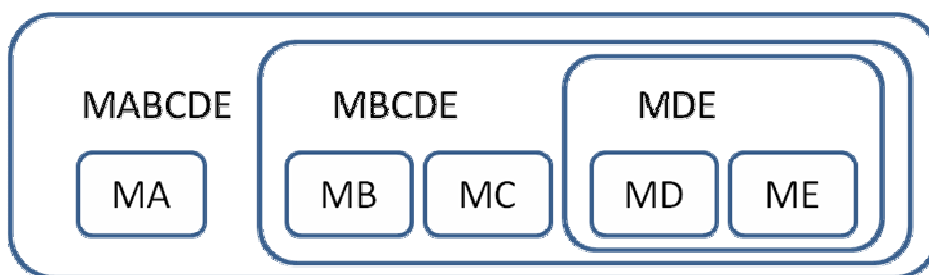


Figure 10 Nesting representation of INCLUDES

All the machines (MA, MB, .. ME) are composed in parallel using CSP. Nesting is used to give names to assemblies of machines, so that an assembly can be re-used easily.

Using the nesting construct in this way ensures that semantics are well defined, which might not be the case if an ad-hoc extension to the language is used.

3.3 Structured Objects

Some objects have a natural structure to them that is best modelled using nesting. An example is an Order (in an Order Processing System) that contains (and owns) a set of Order Lines. This can be represented as shown in Figure 11.

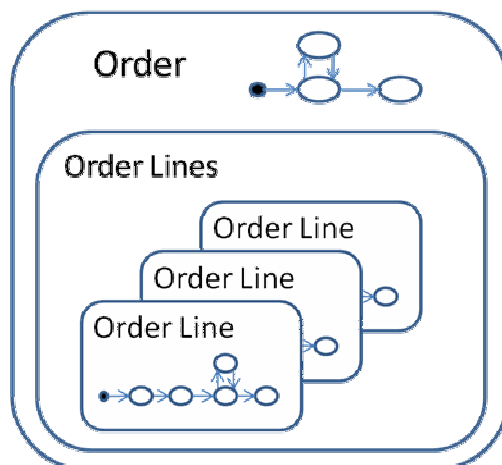


Figure 11 A Structured Object

In Figure 11, the machine *Order* has the machine *Order Lines* nested within it. *Order Lines* is a system, supporting any number of instances of the machine *Order Line*.

3.4 Checking Change of State

The nesting construct provides a way of “monitoring” for a change of state.

Suppose that, in the Banking Model, we wish to send an alert to a customer when he/she goes overdrawn. This could be achieved with the model of Account shown in Figure 12.

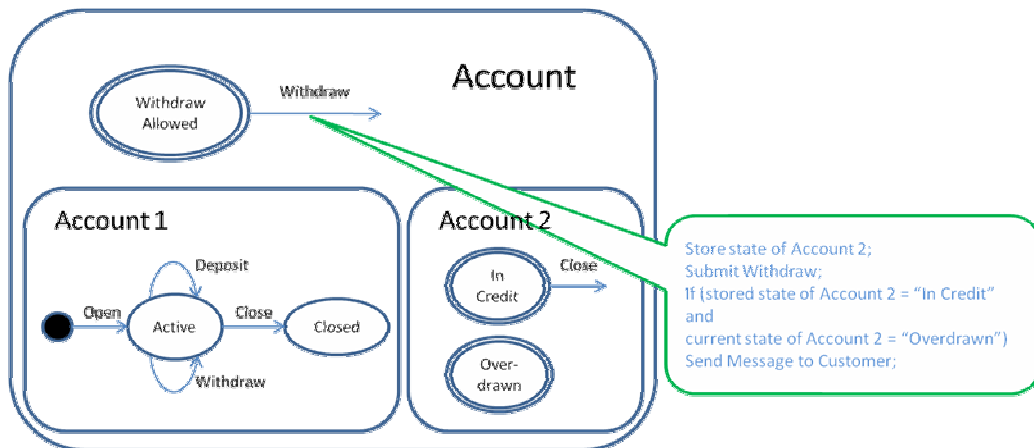


Figure 12 Monitoring for a Change of State

In this model, the machine at the Account level checks for a change of state of Account 2 (from *In Credit* to *Overdrawn*) and sends a message to the (real world) customer if the Account has gone overdrawn.

3.5 Before and After Code

A final use of nesting illustrated is as a means of doing “Before and After” processing of an event, in the context of Aspects. This is shown in Figure 13.

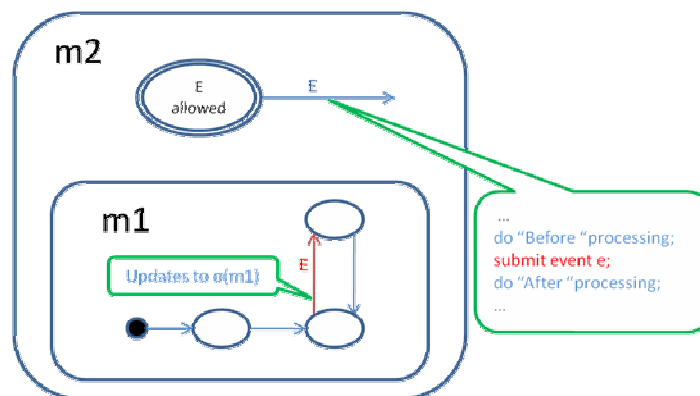


Figure 13 "Before" and "After" Processing

The specification of the event in the outer process can use “event macro” techniques to allow a higher level of abstraction in event definition that used in the inner process. In this way, “Before”

and “After” processing can be defined once and applied to any subset of the events in the inner machine.

Ashley McNeile
24th September 2007