Programming Research Group

A PATTERN FOR CONCURRENCY IN UML

Charles Crichton, Jim Davies,
Alessandra Cavarra

PRG-RR-01-22

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

A Pattern for Concurrency in UML

Charles Crichton, Jim Davies, and Alessandra Cavarra

Oxford University Computing Laboratory,
Wolfson Building, Parks Road,
Oxford, OX1 3QD UK
{crc,ale,jdavies}@comlab.ox.ac.uk

Abstract. This paper presents a pattern of usage for the Unified Modeling Language (UML), intended for the description of systems in which two or more operations may be acting concurrently upon the same object. The pattern addresses two common problems—inadequate models, and complicated state diagrams—with a simple separation of concerns. Changes in attribute state, and changes in operation state, are described separately, using two different types of diagram.

Simple examples are used to demonstrate the application of the pattern: at an implementation level, and at a more abstract, design level. The semantics for concurrency within UML is examined, not only to clarify the interpretation of the pattern, but also to explain why the existing provision—concurrent composite states and concurrency attributes—is not applicable.

1 Introduction

The Unified Modeling Language (UML) [1] is widely used in both industry and academia, providing a common syntax, and a common foundation for reasoning about object-oriented designs. It is supported by as many as 80 different tools; a significant number of textbooks and manuals have been written.

To date, most UML descriptions have focussed upon architectural, or static, aspects of design. Where dynamic or concurrent behaviour is described, it is usually in terms of representative *instances* of interaction; no attempt is made to present models that characterise every possible sequence of interaction (in terms of the set of actions and events defined in the model). And yet such a complete, behavioural description is exactly what is required if we wish to: establish properties of a system in advance of construction; determine how a component might behave in combination with other, specified components; use the model to generate a comprehensive selection of tests.

One reason for the absence of such descriptions is the difficulty of creating them using the state diagram notation of UML. There is often too much information for a single state diagram; and the pattern of communication, and the distribution of attributes, between multiple state diagrams are poorly understood. These problems are exacerbated in the presence of concurrency.

In this paper, we propose a possible solution: a pattern for the description of concurrent behaviour in UML. In an application of the pattern, we use a

single state diagram to describe changes in attribute state—the 'local' state of the object or component—and an activity diagram to explain the possible states of an operation: the sequences of actions associated with an invocation.

The paper begins with a description of the pattern: an abbreviated meta-model; an explanation of the diagrams used; and an account of the features of UML that have a bearing on its application. In Section 3, we show how the pattern could be used in the description of a single object, with an operation defined purely in terms of assignment actions. This example demonstrates the use of the pattern in describing concurrent invocation.

In Section 4, we present a more abstract example, in which the operations on an object are defined in terms of call and signal actions. This demonstrates the use of the pattern as a foundation for subsequent analysis: a formal version of the model is presented, and various properties are established. The paper ends with a discussion of the existing provision for concurrency within UML, and a review of related work.

2 A Pattern for Concurrency

The essence of the pattern is simple: use a state diagram for the possible states of attributes, and activity diagrams for the possible states of operations. A class diagram will also be required, to introduce the attributes, operations, and references that appear in the state and activity diagrams.

2.1 Pattern meta-models

The essence of the pattern can be represented as a fragment of the UML meta-model: a class diagram linking the diagrams and the entities that they represent. This diagram—see Fig. 1—abbreviates the associations: each of the lines here corresponds to a sequence of associations in the actual meta-model.

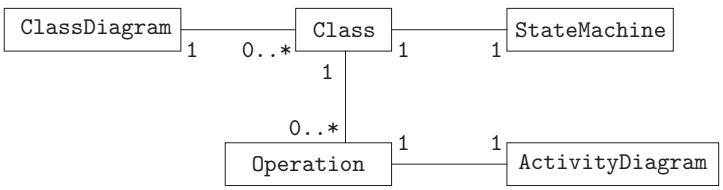

Fig. 1. A meta-model for the pattern

The meta-model for snapshots of (applications of) the pattern in execution cannot be defined within the current UML framework. An essential requirement of the pattern is the ability to refer to instances, or invocations, of operations. Although invocations are mentioned in the current UML documentation, there is no corresponding meta-model element.

2.2 Class diagram

A class diagram may be used to introduce names for classes of objects, together with collections of attributes. Operations may also be named, together with parameter types and return values. Finally, there may be *associations* between classes. For the purposes of the pattern, these are important only in that they introduce names—formal parameters—for referenced objects.

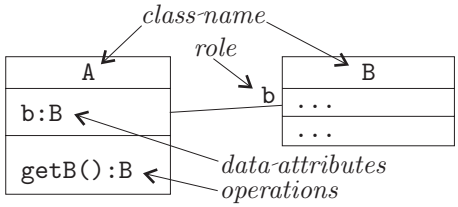

Fig. 2. the components of a class diagram

Figure 2 shows the components that may appear in a class diagram. The line linking the two class boxes is an association, and the role `b` introduces a name used for an object of Class `B` in the state and activity diagrams for Class `A`. The class diagram notation includes many other features—most notably inheritance associations—that have no direct bearing on the current pattern.

2.3 Actions

The UML documentation allows for several types of action. Two of these particular relevance for the pattern:

- A `call` action corresponds to the invocation of an operation; this is at an abstract level, so no particular mechanism for invocation is specified; the calling state machine will wait until the invocation is complete before proceeding.
- A `send` action represents the sending of a signal; again, no particular mechanism for transmission is specified; the sending machine does not wait.

We may wish to introduce two further types of action:

- A `call*` action corresponds to the 'non-blocking' invocation of an operation; the calling state machine will not wait for the operation to complete.
- A local action—typically, an assignment—has effects only upon the current object; these can be described without the introduction of explicit events.

The use of `call*` is not necessary; `send` actions could be used instead. However, it seems desirable—for reasons of clarity and consistency—to retain the present separation of signals and operations. (The use of the `call*` syntax corresponds to the setting of the *isAsynchronous* attribute for actions in the UML meta-model).

2.4 State diagram

A state diagram can be used to describe those aspects of the behaviour that are expressible in terms of the values of data attributes, and the effects of atomic transitions. Transitions and states may be associated with effects, expressed as sequences of actions; these actions are performed when a transition is triggered, or on entering or leaving a particular state.

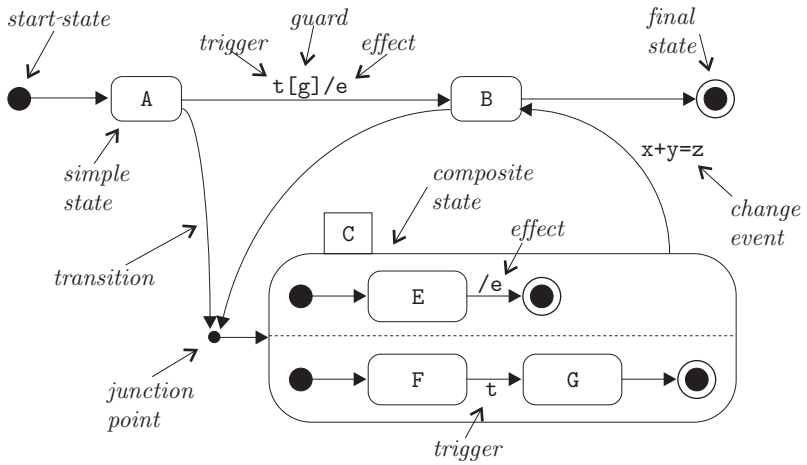

Fig. 3. the components of a state diagram

Unlabelled transitions are triggered by the completion of actions within the source state. Other transitions are triggered by events: these may correspond to an actions, or a change in the values of data attributes—in which case we say that a *change event* has occurred: see the labelled predicate in Fig. 3.

Different kinds of state (or pseudo-state) may appear in these diagrams: composite states may introduce one or more regions in which sequences of actions may be arbitrarily interleaved; junction points simplify the presentation of multiple transitions to or from a single state; start and final states have an obvious interpretation.

In applications of the pattern, (attribute) state diagrams should not make use of `call` or `call*` actions. The primary reason for this is the run-to-completion semantics:

- if a state machine calls a synchronous operation upon itself then it will deadlock—the corresponding call event cannot be processed until the current action has been completed, but the current action cannot complete until the corresponding call event has been processed;
- if a state machine calls an asynchronous operation upon itself, then that operation will be delayed until the current sequence of actions has been

completed; this is problematic, as it excludes the (expected) possibility of the operation starting earlier.

A treatment of invocation in which calls on the current object are a special case would seem undesirable, particularly as we might not know whether a particular reference is to this object, or to another object of the same class. Furthermore, threaded behaviour is easier to explain if call actions are restricted to the operation diagrams.

Whether call *events* appear on a state diagram depends upon our view of the operation concerned. If the operation is considered atomic—nothing *else* can happen to the state of the current object while it is executing—then we can use the corresponding call event in the state diagram. Of course, having added a call event to the attribute state diagram, it would not make sense to introduce a separate activity diagram for the same operation.

2.5 Operation diagrams

Each operation will be described using a separate activity diagram: a restricted form of state diagram in which the emphasis is upon activity, or flow of control, rather than state. The principal component of an activity diagram—see Fig. 4—is the *action state*: a box that contains a sequence of actions to be performed.

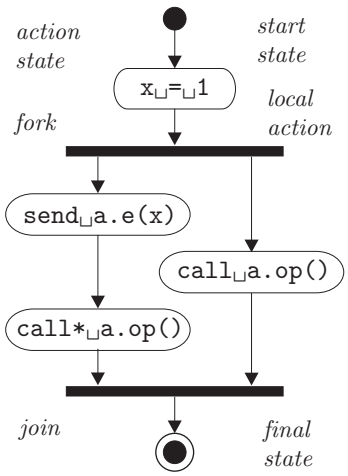

Fig. 4. the components of an activity diagram

In applications of the pattern, activity diagrams may perform any of the four types of action described above: `send`, `call`, `call*`, and local actions. As well as action states, these diagrams can use forks and joins, for interleaving sequences of actions, as well as the familiar symbols for start and final states.

The transitions in an activity diagram have neither triggers nor effects; however, they may be guarded by predicates upon attributes of the current class, formal parameters, or received events: see the next section for suggestions of how these may be represented.

2.6 Related issues

The definition and use of the pattern raises a number of issues regarding the syntax and meta-model of UML. Some of these need to be resolved before the pattern can be applied more widely.

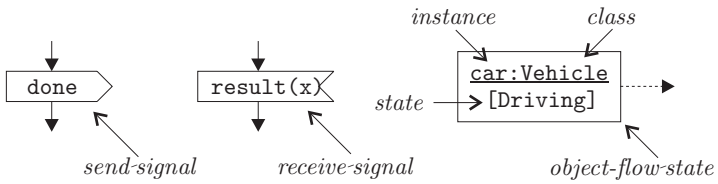

Fig. 5. Additional notation

Local actions The operation diagrams can refer to any attribute of the corresponding object: reading current values, and assigning new values. Each of these *local actions* is considered atomic; their effects could be represented explicitly, using the standard action–event communication mechanism. However, a more attractive solution might be to regard local actions as communications with the underlying object state, and include their effects as *change events* in the attribute state diagram: this is the approach taken in the first example.

Signal notation The language of activity diagrams includes control icons to represent the sending and receiving of signals: see Fig. 5. The *send signal* icon may be seen as an abbreviated state: it is equivalent to an action state containing a single send action. The *receive signal* action is an abbreviated transition. These icons are used in the second example, although they would not be required if state diagrams were used instead—see the final issue listed below.

Passing parameters The UML documentation suggests the following method of indicating the formal parameter list for an operation: annotate the transition from the initial state with a call event with the parameters as attributes, stereotyped `<<create>>`. Perhaps a better approach would be to use the *object-flow* notation. An object-flow state for each incoming parameter is linked to the first proper state (the initial state is considered as *pseudo state* in UML) of the activity diagram, using a dashed version of the transition arrow. This form of state—see Fig. 5—contains not only the name and class of the formal parameter, but also an optional constraint upon its current state. This offers an opportunity to specify preconditions for operations.

Return values The existing UML meta-model includes a `return` action, but its use would appear to be limited to identifying the value to be returned at the completion of a synchronous operation call. The target of the action is left implicit. This is certainly sufficient for the return of references and other expression. Where an object is being returned, and we wish to specify that it is in a particular state, we may augment the `return` action state with an outgoing object flow. The optional constraint upon the outgoing object-flow state could be used to specify a postcondition upon the operation.

Invocation references We may wish to refer to a particular invocation of an operation: to make explicit the target of a return action, or to describe the effect of an exception. The existing language syntax (and the meta-model) offers no means of doing this. One approach would be to associate a value with a call action, allowing an assignment such as `i = call a.op()`. This need not be confused with the assignment of return values, as the syntax `call x = a.op()` could be used for this purpose. An alternative, and perhaps more attractive, approach would be to allow a special reference similar to `self`, or `this`. This would identify the caller of an operation, and could be passed as a reference.

Activity diagrams The use of two different diagram types—activity diagrams for operations, and state diagrams for attribute state—makes the intended separation more obvious. However, it makes it more difficult to represent the effect of exceptions upon operations. Ideally, we would be able to use the composite state mechanism to achieve the effect of a `try`–`catch` block in a language such as Java: that is, the execution of a particular sequence of actions, in a series of sub-states, can be interrupted at any point, explicitly, by a transition leading to a new state. The language of activity diagrams has only subactivity states, which are not interruptable.

3 An implementation-level example

To demonstrate the application of the pattern, we present three descriptions of a class with two data attributes and a single operation: a sequential description without the pattern; a concurrent description that uses the pattern; finally, a complex, concurrent description without the pattern.

3.1 A simple class

Consider a class `A`, with integer-valued data attributes `x` and `y`, and a single operation `swap()`. The intended effect of the operation is to swap the values of the attributes, simultaneously setting `x` to the current value of `y`, and `y` to the current value of `x`. This effect is to be achieved using a transient variable `z`, in the obvious fashion: $\texttt{z} = \texttt{x}$; $\texttt{x} = \texttt{y}$; $\texttt{y} = \texttt{z}$, where $=$ denotes assignment.

We will divide the state space for `A` into three regions: `LT`, corresponding to situations in which `x` is strictly less than `y`; `EQ`, corresponding to situations in

which `x` is equal to `y`; and `GT`, corresponding to situations in which `x` strictly greater than `y`. In a more sophisticated system, these states, and the transitions between them, could be associated with actions; here, however, no actions are necessary.

3.2 A `synchronized` model

We will begin with a simple model that uses a single state diagram; this describes the behaviour of an object of class `A` under the assumption that no more than one invocation of `swap()` may be in progress at any one time: that is, each instance of `swap()` must complete before the operation can be invoked again.

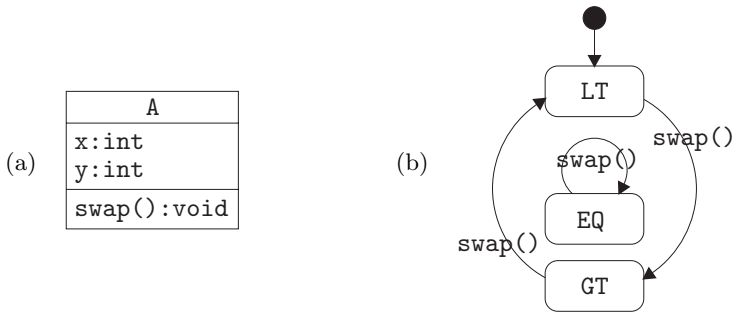

Fig. 6. A `synchronized` model

Figure 6(a) shows the class diagram for `A`, alongside—Fig. 6(b)—the single state diagram. Each transition in the state diagram is labelled with `swap()`; the run-to-completion semantics of state diagrams means that each transition is regarded as atomic and uninterruptable.

This model tells us nothing about the effect of multiple, concurrent invocations of `swap()`. Indeed, the run-to-completion semantics suggests that a faithful implementation of this design should employ a concurrency control mechanism similar to that indicated by the `synchronized` keyword in Java.

3.3 Using the pattern

Each of the actions associated with the operation `swap()` is a local action: an assignment to one of the data attributes of the current object. In using the pattern, therefore, we may construct a state diagram—see Fig. 7(b)—in which each transition is labelled by a change event.

The activity diagram representing the operation `swap()`—see Fig. 7(c)—has three action states: one for each assignment. As no other actions are performed, this separation is simple for clarity: we could employ a single action state containing the same sequence of assignments.

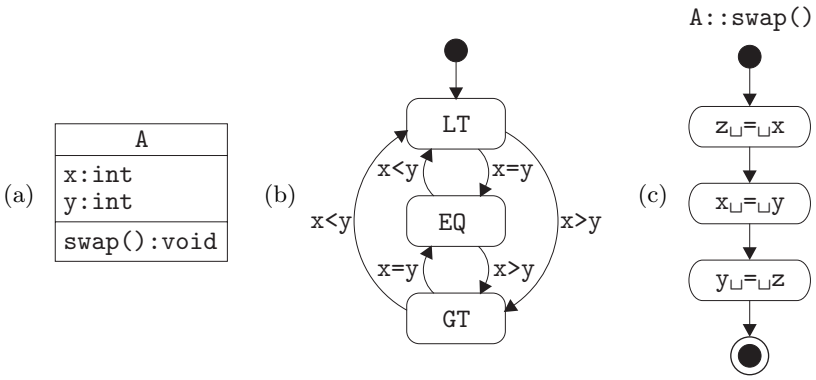

Fig. 7. A model using the pattern

Following two simultaneous calls of `swap()`, this model admits any interleaving of action sequences: for example,

$\langle \mathtt{z}_0 = \mathtt{x};\ \mathtt{x} = \mathtt{y};\ \mathtt{y} = \mathtt{z}_0;\ \mathtt{z}_1 = \mathtt{x};\ \mathtt{x} = \mathtt{y};\ \mathtt{y} = \mathtt{z}_1 \rangle$,

$\langle \mathtt{z}_0 = \mathtt{x};\ \mathtt{x} = \mathtt{y};\ \mathtt{z}_1 = \mathtt{x};\ \mathtt{x} = \mathtt{y};\ \mathtt{y} = \mathtt{z}_1;\ \mathtt{y} = \mathtt{z}_0 \rangle$, or

$\langle \mathtt{z}_0 = \mathtt{x};\ \mathtt{z}_1 = \mathtt{x};\ \mathtt{x} = \mathtt{y};\ \mathtt{x} = \mathtt{y};\ \mathtt{y} = \mathtt{z}_0;\ \mathtt{y} = \mathtt{z}_1 \rangle$

where $\mathtt{z}_0$ and $\mathtt{z}_1$ denote the separate, local instances of `z`. Starting from state `LT`, these sequences would lead us to states `LT`, `EQ`, and `GT`, respectively.

3.4 Without the pattern

To represent the interleaving of actions associated with multiple invocations of `swap()` in a single diagram, we would need to introduce additional actions—one for each of the assignments. To explain the effects of these actions, we need to add additional attributes: to the state; and to the actions themselves.

As the above sample of three interleavings would suggest, a state diagram adequate for the description of up to two simultaneous invocations would have sequences of up to five intermediate states between `LT`, `EQ`, and `GT`. There would be transitions between these states, and the final destination might not be decided until the last event matching an action of the form `y = z` had been processed.

Worse still, to be deterministic, the diagram needs to distinguish between the two `y = z` actions: the two copies of temporary variable `z` may have been assigned different values. It is also necessary to distinguish between the two copies of `x = y`, and the two copies of `x = z`.

It is possible, with a degree of inspiration and analysis, to construct a model that uses a single state diagram to describe the possible effects of an arbitrary number of simultaneous invocations. However, to do this would be to miss the point: we want models that reflect our design intentions, and allow us to predict the consequences; a model like this reflects only the consequences—which would need to be calculated *before* drawing the state diagram.

4 A more abstract example

The previous example used only change events in the state diagram, and the operations were described at the level of a programming implementation. To demonstrate the intended use of signals, in communicating between state and activity diagrams, we will consider a more abstract example.

4.1 A printer

The following fragment of specification describes the behaviour of a printer, in terms of a class with five operations:

- `pause()` pause any print job that might be in progress;
- `resume()` resume any print job that is paused;
- `print()` start a print job;
- `service()` replenish the paper tray, pausing the printer first;
- `carelessService()` replenish the paper tray, pausing the printer at the same time.

The first two operations—`pause()` and `resume()`—will be treated as atomic interactions: in our model, there is no notion of a printer having started to pause, but not yet having reached a paused state; the call events for these operations can be used in the state diagram. The other operations are not atomic, and may be called concurrently; these will be described as separate, activity diagrams.

4.2 State diagram

The state diagram—see Fig. 8—for the printer has eight states, the product of three binary conditions: the device is printing or idle; the device is paused, or not; the door is open or closed. (If the diagram were any more complicated, we would use binary variables to represent one or more of these conditions, and place guards upon the transitions.)

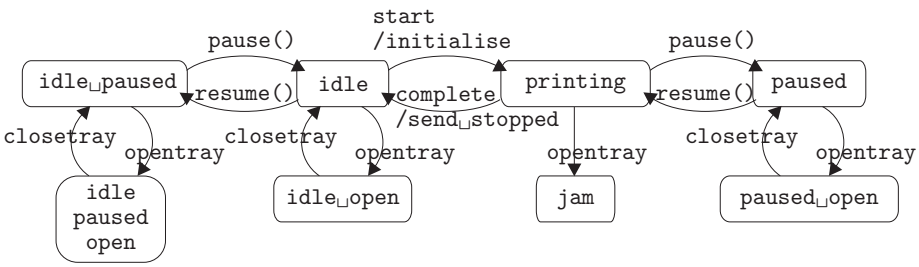

Fig. 8. Printer state

One of the states is clearly problematic: if the event `opentray` is processed in the state `printing`, we reach a state—`jammed`—that is inescapable: subsequent

events will be discarded. The `start` signal will be sent by an invocation of the `print()` operation. The resulting change in state, from `idle` to `printing`, is accompanied by the sending of a signal—`initialise`—to the hardware.

The device stays in the `printing` state until the pause button is pressed, the tray is opened, or a `complete` signal is received from the hardware. A `complete` signal triggers a return to the `idle` state, and sends a `stopped` signal to the `print()` operation.

4.3 Operation diagrams

Figure 9 includes activity diagrams for the three non-atomic operations associated with the printer class. The first, `print()`, begins by sending a `start` signal to the printer state machine—we have used the control icon syntax introduced in Section 2.6—it then waits for a `complete` signal before terminating.

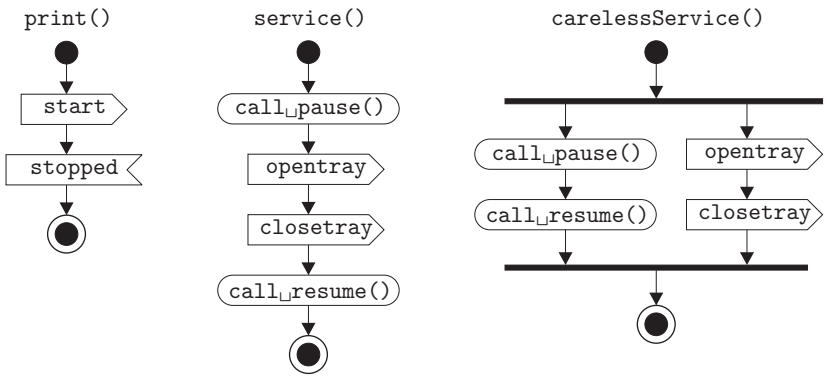

Fig. 9. Printer operations

The `service()` operation pauses, opens the tray, closes the tray, and then resumes. Because we have left `pause` and `resume` as atomic operations on the state diagram, they appear here as call actions, rather than signal sends.

The description of the `carelessService()` operation allows the interleaving of two activities: pausing and resuming, opening and closing. When both activities are complete, the operation terminates.

4.4 Analysis

A careful analysis of the model confirms that, as might be expected, simultaneous execution of the `print()` and `service()` operations is safe: both complete, and the printer does not enter the `jammed` state. Similarly, we can see that the simultaneous execution of `print()` and `carelessService()` can leave us in the `jammed` state. The same is true of the simultaneous execution of `print()` and two or more invocations of `service()`.

Such an analysis is best conducted with the aid of tools. To show how this might be acheived, and to show that an analysis need not involve the construction of a single state diagram, we may translate the above description into the notation of Communicating Sequential Processes (CSP) [2]. The translation is based in part upon the formal semantics for activity diagrams given in [3].

Figure 10 presents a CSP process description, in eight parts, for the attribute state machine, and a process description for each of the operations. UML events and actions become shared, synchronous CSP events (in a multi-object model, intermediate, buffering processes would be necessary). There is one additional event: *error*. This is included to facilitate the analysis of the model: because a UML state machine is always ready to accept any event, we need some way of indicating that a problematic state has been reached.

In these descriptions, $\Box$ denotes external choice, offering a menu of events, with specified consequences. The menu for each state S includes a process of the form $(\Box\, x : A \bullet x \rightarrow S)$, indicating that any event in the set A will be discarded. The prefix notation $\rightarrow$ links an event to a subsequent process. The infix symbol $|||$ indicates that its two arguments—both processes—execute independently, but terminate together. Termination is represented by the symbol *SKIP*.

Having defined such a model, we may use the refinement-checking tool FDR [4] to explore the consequences of our design. To do this, we define a specification process, identifying a range of acceptable behaviours, and a variety of implementation processes, representing possible situations, or combinations of the model components. Figure 11 does exactly this. In this case, the range of acceptable behaviours is quite wide: we allow an internal choice ($\sqcap$) of every event from the set *Interface*; crucially, this set excludes the additional event *error*.

The implementation processes make use of the infix symbol $\|$, representing *communicating* parallel. The two argument processes must agree, or synchronise, upon the occurrence of any event that is mentioned in both descriptions. (The use of $\|$ is facilitated by the definition of process alphabets [2], or by the use of explicit interface parameters [5]).

Figure 11 includes, alongside the definition of each implementation process, the result of the corresponding refinement check. The $\sqsubseteq$ relation holds between two processes precisely when every behaviour—every trace, and every failure [5]—of the second process is also a behaviour of the first.

Implementation processes *System*1 and *System*2 describe situations in which a single invocation of `print()`, and the simultaneous invocation of `print()` and `service()` act upon the printer state. In each case, the refinement check succeeds: *error* is impossible; the *Jammed* state is never reached.

*System*3 describes a situation in which `print()` and `carelessService()` may be invoked simultaneously. In this case, the refinement check fails, and the tool returns as evidence the sequence $\langle start, initialise, open, error \rangle$ to show how the *Jammed* state could be reached. Similarly, when we check *System*4, which describes the effect of invoking `print()` concurrently with two invocations of `service()`, we are presented with the sequence $\langle pause, open, close, pause, resume, start, initialise, open, error \rangle$: *Jammed* is reachable here, too.

$$
\begin{aligned}
Idle &= start \rightarrow initialise \rightarrow Printing \\
&\quad \Box\ open \rightarrow IdleOpen \\
&\quad \Box\ pause \rightarrow IdlePaused \\
&\quad \Box\ (\Box\ x : \{resume, close\} \bullet x \rightarrow Idle)
\end{aligned}
$$

$$
\begin{aligned}
IdleOpen &= close \rightarrow Idle \\
&\quad \Box\ (\Box\ x : \{pause, resume, start, open\} \bullet x \rightarrow IdleOpen)
\end{aligned}
$$

$$
\begin{aligned}
IdlePaused &= open \rightarrow IdlePausedOpen \\
&\quad \Box\ resume \rightarrow Idle \\
&\quad \Box\ (\Box\ x : \{pause, close\} \bullet x \rightarrow IdlePaused)
\end{aligned}
$$

$$
\begin{aligned}
IdlePausedOpen &= close \rightarrow IdlePaused \\
&\quad \Box\ (\Box\ x : \{pause, resume, open\} \bullet x \rightarrow IdlePausedOpen)
\end{aligned}
$$

$$
\begin{aligned}
Printing &= pause \rightarrow Paused \\
&\quad \Box\ open \rightarrow Jammed \\
&\quad \Box\ complete \rightarrow stopped \rightarrow Idle \\
&\quad \Box\ (\Box\ x : \{start, resume, close\} \bullet x \rightarrow Printing)
\end{aligned}
$$

$$
\begin{aligned}
Jammed &= (\Box\ x : \{pause, resume, open, close, start\} \bullet x \rightarrow Jammed \\
&\quad \Box\ error \rightarrow Jammed)
\end{aligned}
$$

$$
\begin{aligned}
Paused &= resume \rightarrow Printing \\
&\quad \Box\ open \rightarrow PausedOpen \\
&\quad \Box\ (\Box\ x : \{pause, close, start\} \bullet x \rightarrow Paused)
\end{aligned}
$$

$$
\begin{aligned}
PausedOpen &= close \rightarrow Paused \\
&\quad \Box\ (\Box\ x : \{pause, resume, open, start\} \bullet x \rightarrow PausedOpen)
\end{aligned}
$$

$$Print = start \rightarrow stopped \rightarrow SKIP$$

$$Service = pause \rightarrow open \rightarrow close \rightarrow resume \rightarrow SKIP$$

$$
\begin{aligned}
CarelessService &= (pause \rightarrow resume \rightarrow SKIP) \\
&\quad |||\ (open \rightarrow close \rightarrow SKIP)
\end{aligned}
$$

Fig. 10. A CSP script for the printer model

$$Interface = \{open, close, pause, resume, start, stopped, initialise, complete\}$$

$$Spec = \sqcap\ e : Interface \bullet e \rightarrow Spec$$

$$System1 = Print \parallel Printer \qquad (Spec \sqsubseteq System1)$$

$$System2 = (Print\ |||\ Service) \parallel Printer \qquad (Spec \sqsubseteq System2)$$

$$System3 = (Print\ |||\ CarelessService) \parallel Printer \qquad (Spec \not\sqsubseteq System3)$$

$$System4 = (Print\ |||\ Service\ |||\ Service) \parallel Printer \qquad (Spec \not\sqsubseteq System4)$$

Fig. 11. Analysis of the printer model

5 Discussion

5.1 Existing provision

A casual reader of the UML documentation might be excused for thinking that *concurrent composite states*, allowed in state diagrams, could be used to represent concurrent invocations. However, concurrent invocations of operations are best seen as peers, alongside the attribute state of the object. Any attempt to represent them using concurrent substates, within the object state diagram, is likely to produce a confusing, inadequate model.

The best results would be obtained by using a concurrent composite state at the outermost level, with a separate region for each operation: rather like the pattern suggested here. However, even this would be unsatisfactory: the run-to-completion assumption means that the enclosing composite state must process one event at a time. As has already been explained, the run-to-completion assumption also prohibits the use of `call` actions when the target is the current state machine.

Another limitation of the existing provision—mentioned in Section 2.1—is that behavioural features such as operations do not have classifiers. Thus, in UML, we cannot speak of an *instance* of an *operation*. This makes it impossible to construct an *explicit* representation of concurrent invocation. The meta-model needs to be re-factored and extended to allow instances of operations, and thus references to invocations.

The way in which events are processed by state diagrams—they are held in a reliable, possibly-reordering medium, until they can be processed; they are then accepted one at a time by the state machine: triggering a transition, being discarded, or being *deferred*—tells us a great deal about communication between state machines; enough to allow a translation of simple state and activity diagrams into CSP processes, as in Section 4.4.

The translation of more complex diagrams, in which transitions cross boundaries between regions, or the same event appears on two conflicting transitions, requires more work. If such diagrams are to be used with the pattern, certain semantic variation points need to be decided: see, for example, [6].

Another source of complexity is the event deferral mechanism. An event `e` is deferred if it is processed when the machine is in a state associated with the attribute `e/defer`. The present semantics tells us that such an event is placed upon a local queue—exclusive to the current region—and then processed (after any `entry` actions) as soon as the machine reaches a state without the `e/defer` attribute.

Although multiple events may be deferred, only one of these will ever be processed: the others will be lost; clearly, in a description of concurrent behaviour, this may not be appropriate. A simple solution is to avoid the use of deferred events, and to include a component within the model whose role is the management and delivery of signal events. Another alternative would be a persistent version of the local queue: one that retains deferred events until they are used to trigger transitions, or are explicitly discarded by the state machine.

5.2 Related work

The work in this paper is based upon the action semantics of UML (Version 1.4). There are proposals [7] to extend this semantics to allow composite action sequences, or *maps*, in which several actions may be executing concurrently. There is also a proposal to introduce the notion of *ports* for objects, allowing the attachment of several state machines to a single object; this would make the implementation of our pattern easier.

There is a considerable amount of research underway regarding concurrency and UML, but—to the best of our knowledge—very little of this is addressing the problem of defining complete, or essential models: models that characterise every possible sequence of interaction. The AGEDIS project [8] requires models, defined in a subset of UML, for the purposes of test generation, but the existing AGEDIS modelling language—a subset of UML—does not allow for concurrent invocation of operations on the same object.

Sendall and Strohmeier [9] have shown how OCL can be used to specify constraints upon state machines: they focus upon timing constraints, but the technique could clearly be applied in combination with the current pattern. Our use of activity diagrams to describe operations has a close parallel in the work of Petriu and Wong [10], who use such diagrams to describe messaging between objects, with the intention of elucidating concurrent behaviour.

References

1. *Unified Modeling Language Specification, Version 1.4*. Object Management Group. `http://www.omg.org/cgi-bin/doc?formal/01-09-67`.
2. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall. 1985.
3. Christie Bolton and Jim Davies and Jim Woodcock. On the Refinement and Simulation of Data Types and Processes. In *Proceedings of the 1st International Conference on Integrated Formal Methods*. Springer 1999.
4. Formal Systems (Europe) Limited. *Failures-Divergences Refinement: FDR2*, 1997. FDR2 User Manual.
5. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, 1998.
6. A. Cavarra. *Applying Abstract State Machines to Formalize and Integrate the UML Lightweight Method*. PhD thesis. University of Catania, Italy. 2000.
7. *Action Semantics for the UML*. OMG ad/2001-03-01. Response to OMG RFP ad/98-11-01. `http://cgi.omg.org/cgi-bin/doc?ad/01-03-01`.
8. *AGEDIS: Automated Generation and Execution of test suites for DIstributed component-based Software*. `www.agedis.de`.
9. S. Sendall and A. Strohmeier. Specifying concurrent system behavior and timing constraints using OCL and UML. In the *Proceedings of UML 2001: The Unified Modeling Language: Modeling Languages, Concepts and Tools* LNCS 2185. Springer 2001.
10. D. C. Petriu and E. Wong. *Using Activity Diagrams for Representing Concurrent Behaviour*. Carleton University, Ottawa, 2001. Submission to concurrency workshop at *UML 2001*: `http://wooddes.intranet.gr/uml2001/SubmittedPapers/`