Extending KM to represent processes with loops, conditions, disjunctive and partially ordered events: a proposal

Jim Blythe

January 23, 2001

Processes represented in KM need to allow loops, conditional events, disjunctive events and partially ordered events. I suggest ways that KM can be extended to handle them using two different representational styles, one in which a parent event has a `subevent-order` field that specifies the ordering of its child events and one in which each event has a `next-event` field. Implementing this language in KM and its simulator should be straightforward, and the it can be extended in the future as needed.

Loops, disjunctive, conditional and partially ordered events

A process has a *loop* if a sequence of events can be repeated, for example in the description "the polymerase moves forward **until** the promoter region is reached", the `move` event is a single-event loop. Loops usually have a termination condition which is tested each time the loop is completed.

A *conditional event* is one that is applied only if some condition is met, for example "if the gas tank is empty, fill it". The condition is usually tested against the world state immediately before choosing to execute the event.

Two or more events are *disjunctive* if only one of them will take place. For instance: "to get to the zoo, **either** take the 10 freeway **or** take the 5 freeway".

Events are *partially ordered* if the order in which the events take place is incompletely specified. A partial ordering over events does not necessarily mean that the events can be executed simultaneously. For instance if two trucks need to go through a narrow tunnel, they may not be able to both go through at the same time although we may not care which goes first.

For ease of implementation, we suggest implementing both loops and disjunctive events in terms of conditional events in each of the following alternatives.

Using an ordered list of subevents

In this scheme, the order in which the sub-events of an event are to be executed is specified with a list, for example:

```
(every bacterial-RNA-Transcription-Scenario has
  (location ((the location of the the-DNA of self)))
  ...
  (subevent-order ((the make-contact subevent of self)
                   (the move-through subevent of self)
                   (the recognize subevent of self)
                   ...)))
```

I believe this is the current approach used in the component library.

Conditional events

The following syntax can be used to represent conditional event blocks:

```
(subevent-order ((the move subevent of self)
                 (if (equal (the location of self) (the pro-
moter region ..))
                  then ((the recognize subevent of self)
                        (the transcribe subevent of self))
                  else ((the move subevent of self)))
                 (the other subevent of self)))
```

The conditional event block can appear in the list of events in the `subevent-order` field of an event. Events that occur either before or after the block are executed regardless of the outcome of the test. The format of a conditional event block is
`if <test> then <block1> else <block2>`.
Both `<block1>` and `<block2>` can have more than one event and can include conditional events, allowing for nested branches. Either block can also be empty.

Loops

The above example implements a loop where the `move` event is repeated until the test in the conditional event is met. In this way the conditional event effectively implements the termination condition for the loop. The loop itself is implemented by repeating the first event in the loop in the list in `subevent-order`.

This approach is flexible since it allows multiple entry points and multiple exit points with different conditions. However an SME might find it hard to reason about the loop because it is represented implicitly. One possibility is for an explicit loop construct to be translated into this syntax for ease of implementation. For example the construct
`repeat <event1> <event2> ..  <eventn> until <condition>`
could be translated to the following sequence in the `subevent-order`:

`<event1> <event2> .. <eventn> (if <condition> then (<event1>))`

Disjunctive events

Disjunctive events can be represented in terms of a conditional event using an explicit non-deterministic test. Again an explicit construct may help an SME, for example the SME would use the construct
`either <take the 10 freeway> or <take the 5 freeway>`
which could be translated to
`(if (equal (toss-a-coin) heads) then (<take the 10 freeway>) else (<take the 5 freeway>))`.
Here, `toss-a-coin` is a non-deterministic operation that either yields `heads` or `tails`.

Partially ordered events

Loops and disjunctive events can both be implemented in terms of conditional events, making it efficient for KM to reason about them. Unfortunately a new construct is needed to represent partially ordered events within the `subevent-order` list. The `unordered-sequences` construct is followed by a list of blocks of actions, for example:
`(unordered-sequences (<move truck1 through tunnel>) (<move truck2 through tunnel>))`

Each block is totally ordered in the same way as the `subevent-order` list and can also contain conditional-event or partial-order constructs. The intended meaning is that, while the events in each block are executed in the order that they specify, any interleaving of the events from different blocks is possible.

Using a next-event pointer

In this scheme, an event with a number of sub-events designates a `first-event`, and each sub-event designated a `next-event` to define the order in which the events are executed.

Conditional events can be implemented in a similar way to the `subevent-order` case, but each branch contains only a single action instead of a block of actions. Loops and disjunctive events can be defined in terms of conditional events as before.

This representation is slightly less powerful than the `subevent-order` case because the decision about the next event depends only on the current event. *(I can provide an example of this if required)*.

A concise way to represent partial orders is to allow more than one event in the `next-event` field, with the intended meaning that each event is executed eventually, but the order of the two is undefined and in fact the second event may be executed after one or more `next-event` links are followed from the first event to be executed.

Again this representation, although perhaps more elegant, is less powerful because the `next-event` information depends only on the current event, not on the path through which it has been reached.