A proposal for a process description language for SHAKEN

Jim Blythe*

February 1, 2001

The SHAKEN system needs to be able to reason about processes that contain loops, conditional steps, partially ordered steps and disjunctive steps. I propose minimal necessary extensions to the process description language already in place that will allow this. Because the extensions are small, implementing them in each of the components of SHAKEN should be straightforward, and further extensions can be made in the future as needed. Because a process description will be shared between several components, we need a very clear understanding of the meaning of each construct in the language and also of how each component can use the language. This document summarizes the existing language and describes the extensions.

The existing process language

The current language for describing processes makes use of steps that can have several sub-steps. When a step in a process is to be executed, all of its sub-steps should be executed. A step in a process may also include a `next-step` slot that determines the next step to be executed. The following example is taken from Peter's slides and modified slightly:

```
(every RNA-Transcription-Plan has
  (sub-steps
    ((a Collide with
            (next-step (((the Recognize sub-steps of Self)))))
     (a Recognize with
           (next-step (((the RNA-copy sub-steps of Self)))))
     (a RNA-Copy with
          (next-step ((the Release sub-steps of Self))))
     (a Release))))
```

The meaning of the `sub-steps` and `next-step` values on this example is that to execute an RNA-Transcription-Plan, the steps "Collide", "Recognize", "RNA-Copy" and "Release" are executed in that order.

*With ideas from Peter, John, Paul, Gary, Pat and others.

No sub-step is explicitly designated the first step in this process. Any step that is not the next-step for some other step could potentially take place first. In the current version of the language, I presume this must not be true of more than one step.

Events, steps and event instances

An *event* like "Move" is represented in KM independently of any particular process description. When a process description refers to a Move, it refers to a *step* in the process that implements the Move event. A process description may have several, distinct Move steps. Likewise a step in a process description has a `next-step`, but the disembodied event desription does not. Finally when a process is executed, the actual execution of the step is an *event instance*. In the next section I only refer to steps.

Loops, disjunctive, conditional and partially ordered steps

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

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

A *conditional step* is one that is executed 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 step.

Two or more steps 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".

For ease of implementation, I suggest implementing both loops and disjunctive steps in terms of conditional steps, as I describe below.

Extending the SHAKEN representation

I believe that the following four categories, partially ordered steps, conditional steps, loops and disjunctive steps, are all that are needed to represent the processes we need to for the year 1 end-to-end experiment.

Partially ordered steps

A concise way to represent partial orders is to allow more than one step in the `next-step` field, with the intended meaning that each step is executed eventually, but their order is undefined. However, steps do not take place simultaneously. If two or more

steps designate the same step e as `next-step`, and the designating steps have a common ancestor, then e is considered a *join* step, meaning that it cannot be executed until all the designating steps have been executed.

For example, suppose a process to move two trucks through a gate requires first opening the gate, then moving both trucks (in any order), then closing the gate. This can be represented as follows:

```
(every move-trucks-process has
  (sub-steps
    ((a open-gate with
       (next-step ((the sub-steps of self called "move1")
                   (the sub-steps of self called "move2"))))
     (a move called move1 with
       (next-step ((the close-gate sub-steps of self))))
     (a move called move2 with
       (next-step ((the close-gate sub-steps of self))))
     (a close-gate))))
```

Since `open-gate` is the only step in the process that is not the value of `next-step` for some other step, it is the first step to be executed. After `open-gate`, either `move1` or `move2` are executable. Suppose `move1` is executed. Although the `next-step` field is `close-gate`, this step cannot be executed because `move2`, which also has `close-gate` as its `next-step` field, has not yet been executed. So after executing `move1`, the order of the remaining steps is uniquely determined.

If there is more than one step in a process that is not designated as the `next-step` of some other step, they should be interpreted as a set of potential first steps that are unordered with respect to each other.

Conditional steps

The following syntax can be used to represent conditional steps:

```
(a move with
  (next-step ((if (equal (the location of self) (the promoter re-
gion ..))
                then ((the recognize substep of self))
                else ((the move substep of self))))))
```

The format of a conditional step statement is
`if <test> then <statement1> else <statement2>`.
Both `<statement1>` and `<statement2>` can be either steps or conditional step statements, allowing for nested branches. Either statement can also be empty, indicating that there is no next step in that case.

Loops

The above example implements a loop where the `move` step is repeated until the test in the `next-step` condition is met. The condition plays the role of the termination

condition for the loop. A loop involving several steps can also be made simply by having the `next-step` of a step point to an "earlier" step.

For example, the RNA-Copy step from Peter's slides could be defined like this:

```
(every RNA-Copy has
  (sub-steps
    ((a Create with
      (next-step ((the Attach sub-step of self))))
     (a Attach with
      (next-step ((the Move sub-step of self))))
     (a Move with
      (next-step
((if (equal (the location of self) (the promoter-region..))
          then ()
          else ((the create sub-step of self)))))))))
```

This approach to defining loops is flexible since it allows multiple entry points and multiple exit points under different conditions. However a 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 (an idea that Peter referred to as a "macro"). For example the construct
`repeat <step1> <step2> .. <stepn> until <condition>`
could be translated into a loop in which ¡stepn¿ has a conditional `next-step`.

Disjunctive steps

Disjunctive steps can be represented in terms of a conditional step using an explicit non-deterministic test. Again an explicit macro construct may help an SME, for example the SME might 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`.