Specifying the Suspend - Resume Behavior in LOTOS

Source: Canada

0. Problem statement

There is a need for specifying the capability of a behavior to temporarily suspend and then resume. Many real-life interrupt mechanisms follow this behavior. Unfortunately, such a behavior is cumbersome to specify with standard LOTOS operators. These enable to specify mostly alternative or interleaved sequences of actions. Alternative operators allow to suspend a behavior but not to resume it. The interleave operator is also not adequate because it is impossible to distinguish which of the two behaviors involved is the suspending behavior, and also there is no easy way to freeze the suspended behavior until it can resume. Usually, the only way to specify such a behavior relatively easily is by means of a state-oriented style.

The suspend - resume behavior is very common in protocols, operating systems, etc. It is particularly useful for the specification of telephone systems features. It would be a convenient way to specify some telephone features and it would be helpful in the study of the feature interaction problem. For the telecommunications industry, this is a **hold** operator.

1. The new Suspend-Resume operator: B |[> SB

This operator specifies that behavior SB (the suspending behavior) can suspend behavior B and that after completion of SB, B resumes execution at the point of invocation of the suspending behavior SB. The corresponding symbol "**|[>**" is made out of the combination of a bar suggesting parallelism and of the disable operator suggesting suspension.

Inference rules:

To start, two rules are necessary:
The first rule says that after any action of B, SB can still be activated.

```
      B --a--> B'
--------------------------------
B |[> SB --a--> B' |[> SB
```

This rule is very similar to rules of the traditional LOTOS alternative or parallel operators.
The second rule consists in reordering the execution of B and SB.

```
      SB --s-->SB'
--------------------------------
B |[> SB --s--> SB' >> B
```

This means that after the suspending behavior SB has taken over, behavior B will be re-enabled as soon as SB has terminated. In other words, we perform a re-sequencing of behaviors when allowing the suspend operator to take control. This implies that SB should normally be terminated by an exit unless some other type of termination occurs that generates inaction.

The second rule however allows only one instance of activation of the suspending behavior. In some cases, there is a need for repeated activation of the suspending behaviors. For example in telephony, the *hold* feature can occur several times during a connection, the same for call waiting. Consequently, we can improve the second inference rule as follows:

$$\frac{SB \xrightarrow{s} SB'}{B \mid[> SB \xrightarrow{s} SB' \gg (B \mid[> SB)}$$

In this rule, a new copy of the suspending behavior SB is available after completion of the first instance of SB.

Note how the operator >> is used in the definition of |[>.

Finally a third inference rule is necessary in order to specify that if behavior B has completed successfully, behavior SB can no longer appear.

$$\frac{B \xrightarrow{\delta} stop}{B \mid[> SB \xrightarrow{\delta} stop}$$

Limitations:

The scoping rules for this new operator are the same as the scoping rules for the usual LOTOS operators. Basically any variable declared in any of the two behaviors has only its own behavior as a scope. Therefore, the disabling behavior SB cannot **exit** any values.

2. A small example

The call-waiting and the three-way-call features of ISDN or AIN can be easily specified with this operator. The call initiator side of a connection can be specified in the following way:

```
user: offhook ;
(
    network ! dial ;
    network ! connect ;
    (
        voice_transfer[user, network]
        |[>
            three_way_call[user,network]
    )
    |[>
      call_waiting[user,network]
    )
```

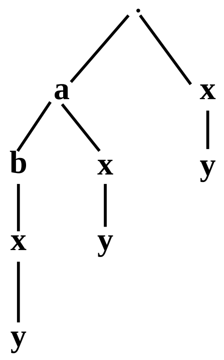
)

The above example means that three_way_call can interrupt the voice_transfer at any time and after establishing the new third party connection, resume voice_transfer with the first party. Further, the call waiting feature can be activated at any time during a call setup after the initial off_-hook, this implies that someone trying to establish a third party connection can accept a call waiting and resume third party connection establishment.

3. Comparison of behavior trees between the disable, interleave and suspend-resume operators

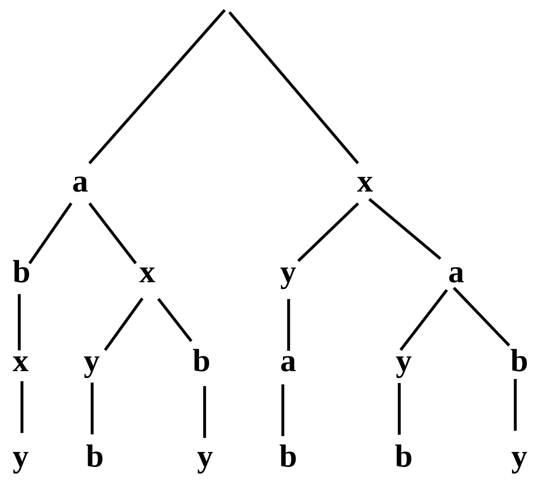
3.1 Behavior tree of the disable operator

a ; b ; stop **[**> x ; y ; stop



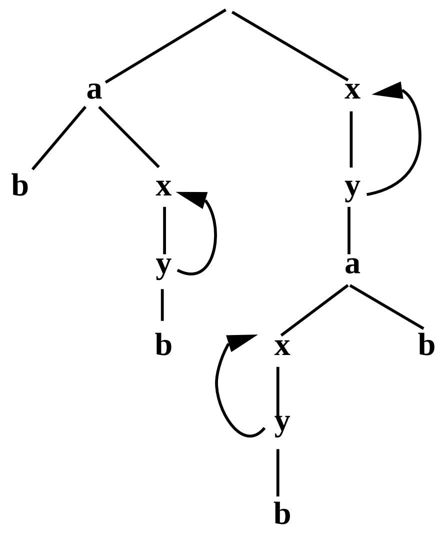
3.2 Behavior tree of the interleave operator

a ; b ; stop ||| x ; y ; stop



3.3 Behavior tree of the suspend-resume operator

a ; b ; stop **|[>** x ; y ; exit



4. Algebraic properties of the suspend-resume operator

To be provided

5. Difficulties specifying the suspend-resume in standard LOTOS.

To see the usefulness of this proposed new operator, it is interesting to see how an equivalent behavior can be specified in standard LOTOS. Following is a review of possible solutions and comments on their success or failure.

The basic problem, expressed informally, consists in allowing a behavior B2 to suspend the execution of behavior B1 in such a way that B2 once activated has to complete execution and then resume B1. Also B2 should be able to suspend B1 repeatedly at various points of the execution of B1.

The use of a disable operator:

This behavior cannot be specified using the disable construct B1 [> B2 because the disable operator does not allow B1 to resume.

The use of the interleave operator:

The use of the interleave construct **B1 ||| B2** is also not appropriate because while B2 can interleave with actions of B1, the reverse is also true thus violating the condition that B2 has to be executed entirely before B1 can resume.

The use of a State-oriented style:

This consists in offering the alternative suspending behavior B2 for each action of B1. B2 must terminate with an exit.

$$\mathbf{(a_1 \ ; exit \ [] \ B_2 \ ) >> (a_2 \ ; exit \ [] \ B_2 \ ) >> ... >> a_n \ ; exit}$$

This solution is very cumbersome, especially when behavior B1 is not a simple sequence but is the result of more complex behavior expressions involving parallelism etc.

The use of a status-oriented style:

The interleave construct is however a good start, which needs some logic to enforce the desired behavior. While B1 and B2 behave independently, a constraint in parallel should solve the problem.

(B1 ||| B2) || C.

This method requires adding a behavior tag to each action of a given behavior. The Constraint logic will record the behavior tag of a behavior through synchronization with the trigger action of the behavior and then will impose this behavior tag to the remaining actions of the behavior until encountering a release action that enables resumption of the suspended behavior.

For example B1 has the following structure, where value “s1” is the behavior tag:

```
g ! trigger1 ! s1 ;
g ! prim12 ! s1 ;
g ! release1 ! s1 ;
stop
```

The same technique is used for the suspending behavior B2.

These two behaviors offer status indicators s1 and s2 which is passed to the constraint behavior C. The latter is in state-oriented style, and will have the following structure (in LOTOS pseudo-code):

```
process memory[g](S:status):noexit:=

        trigger actions to record status
        []
        regular constrained actions
        []
        release action forcing resuming of initial behavior B1
endproc
```

Detail of action type:

• trigger recording actions that capture the status of the triggering behavior:

```
g ! trigger1 ? NS:status ; memory[g](NS)
[]
g ! trigger2 ? NS:status ; memory[g](NS)
```

• regular actions that will be able to synchronize with one of the two behaviors B1 or B2, depending on the current status S passed by the recursive constraint process (memory):

```
g ! prim12 ! S ; memory[g](S)
[]
g ! prim22 ! S ; memory[g](S)
```

• release actions indicating that behavior B1 can resume or restart

```
 g ! release1 ! S ; memory[g](start)
[]
 g ! release2 ! S ; memory[g](s1)
```

Note that this gives six choices.

Finally the interleave construct needs some refinement to prevent behavior B2 from starting before behavior B1. For example in telephony one could not use the call waiting feature if the phone is not at least in an off-hook state.

```
g ! trigger1 ! s1 ;
(
        g ! prim12 ! s1 ;
        g ! release1 ! s1 ;
        stop
|||
        g ! trigger2 ! s2 ;
        g ! prim22 ! s2 ;
        g ! release2 ! s2 ;
        stop
        )
```

The resulting behavior tree is:

```
        g ! trigger1 ! s1
        | g ! prim12 ! s1
        | | g ! release1 ! s1
        | | | g ! trigger2 ! s2 <-- uninterrupted sequence of B2        (1)
        | | | | g ! prim22 ! s2
        | | | | | g ! release2 ! s2
        | | g ! trigger2 ! s2 <-- uninterrupted sequence of B2
        | | | g ! prim22 ! s2
        | | | | g ! release2 ! s2
        | | | | | g ! release1 ! s1 <-- resuming B1
        | g ! trigger2 ! s2 <-- uninterrupted sequence of B2
        | | g ! prim22 ! s2
        | | | g ! release2 ! s2
        | | | | g ! prim12 ! s1 <-- resuming B1
        | | | | | g ! release1 ! s1
```

We see that B2 takes over after the end of B1 (1). It is not clear if this is desirable. For example, in real life, one cannot transfer a call after hanging up the phone.

The use of a resource-oriented solution:

The resource oriented style uses a relay gate (here named “go”) in between each action of the normal sequence (a1,...,a3). The suspending behavior is composed of two alternative behaviors. One is used to keep the normal sequence going, the other one is the actual suspending sequence (s1,...,s3) that reinstantiates the overall suspending behavior.

```
specification suspend_ressource[g, go]:noexit

type events is
  sorts event
  opns  a1, a2, a3, s1, s2, s3:-> event
endtype

behavior

  Pnormal[g, go] |[go]| Psuspend[g, go]

where

  process Pnormal[g, go]:noexit:=

    go ; g ! a1 ;
    go ; g ! a2 ;
    go ; g ! a3 ;
    stop

  endproc

  process Psuspend[g, go]:noexit:=

    go ; Psuspend[g, go]

    []

    g ! s1 ; g ! s2 ; g ! s3 ; exit >> Psuspend[g, go]

  endproc
endspec
```

6. Conclusions

The known methods to specify the suspend-resume behavior are cumbersome to implement, hard to read, and cause loss of structuring, which is contrary to principles of good LOTOS style. It would be very desirable to have a natural notation in LOTOS for such an important notion.