Feature Interaction Detection using Backward Reasoning with LOTOS

Bernard Stepien and Luigi Logrippo

Telecommunications Software Engineering Research Group
Department of Computer Science, University of Ottawa
Ottawa, Ont. Canada, K1M 6N5
(bernard | luigi)@csi.uottawa.ca

The problem of detecting feature interactions in telephone systems design is addressed. The method proposed involves specification of the features in LOTOS, and uses an analysis technique called backward reasoning. This is is implemented in LOTOS by a combination of backward and forward execution. A tool to help carry out backward execution is presented. A detailed example of the use of the technique is given, involving the three-way-calling and call-waiting features.

0. OVERVIEW

Feature interactions have been categorized according to the kinds of features, the number of users and the number of network elements [CGLN93]. The causes of feature interactions are equally varied. They mainly revolve around problems of ambiguity and errors in logic intrinsic to distributed systems. It is reasonable to think that different causes require different detection methods. This paper concentrates on the problem of detecting ambiguous actions, which are seen as symptoms of feature interactions. Features are specified in LOTOS, and the specification is analyzed by using backward reasoning [DB78] [Hol85] [Lin 90].

1. SPECIFICATION OF FEATURES USING THE STATUS ORIENTED STYLE

There are several published LOTOS specifications of telephones with features [BL93][Najm93]. They often use the constraint-oriented specification style where abstract data types play an important control role. The status-oriented style [SL93], a variation of the resource-oriented style, makes greater use of the principle of synchronization on discrete values, which usually represent signals and phone numbers. When complex control structures need to be specified, the logic is

encoded into separate processes that work as constraints on a local basis, corresponding to what one would use as implementation structure. In the constraint-oriented style, all processes collaborate together to form the global behavior. Thus the behavior expressions representing features are mixed together and it may become difficult to follow the life cycle of a feature. The status-oriented style is closer to the concept of components communicating via interfaces, a concept that is central in ODP.

The specification we have used describes three cases of feature interactions that are illustrations of the classification of feature interactions found in [CGLN93].

The Single User Multiple Element (SUME) class is represented by the case of the interaction caused when a user presses the pound key to change her personal options in voice mail when accessing it through a calling card call (action which also involves the use of the pound key).

The Single User Single Element (SUSE) class is represented by the case of the interaction caused when a user attempting to establish a three way call presses the *flash_hook* key while being called by another user and after having subscribed to call waiting. The flash-hook key becomes ambiguous because it can also be used to answer the waiting call.

The Multiple User Single Element (MUSE) class is represented by the case of the interaction caused when a user has a number in her originate screen list that is the number to which another number this user is calling is forwarded to.

The two first cases are essentially cases of detections of ambiguities while the third case is a violation of intentions.

While developing the specification we have been faced with one interesting reality: for specification of features in LOTOS one can observe the same phenomenon as for implementation of features in software: features are scattered throughout the code. However while code can be tested only by running various test scenarios, LOTOS (due to its algebraic nature) can be manipulated to verify properties. Also the very activity of specifying features using LOTOS forces the designer to think more and thus detect feature interactions just by trying to specify them. One difference is that the cost of experimenting with specifications is considerably less than the cost of experimenting with real software or hardware.

The general structure of the specification is as follows:

```
(
    phone[u, n](num1, { three_way_calling, call_waiting,originate_screening }, no_active_service , num1, {
num4 } )
    |||
    phone[u, n](num2, { call_forward }, no_active_service ,num4, no_screen_list)
    |||
    voice_mail[u, n](num3)
    |||
    phone[u, n](num4,no_services, no_active_service,num4, no_screen_list)
    |||
    phone[u, n](num5,no_services, no_active_service,num5, no_screen_list)
)
```

|[n]|

network[n,a]

The formal parameter list of each instance of *phone* determines the activation of the various services presented here. Each *phone* is defined as a choice of *call initiator* and *call responder*. The process *network* allows to manage simultaneous connections and is of the usual following form:

```
process network[n,a]:noexit:=
    n ? CALLER:phone_number ! tone  ;  ( connection_handler[n,a](CALLER)  ||| network[n,a] )
endproc
```

The features are inserted wherever they can be invoked within the basic POTS behaviors. For example, in the *call_initiator_phone* behavior, the call-waiting and three-way-calling features are inserted after the *connect* event in process *complete_connection* as an interleaving between instances of processes *user_events* and *features*.

```
process call_initiator_phone[...](...) := off_hook ; tone ; dial ; complete_connection[...](...)
where
 process complete_connection[u,n](NUMBER:phone_number,SUBS_SERV:subscrib_services):noexit:=
    n ! NUMBER ! connect
    ;  ( user_events[u,n](NUMBER,SUBS_SERV)  ||| features[u,n](NUMBER,SUBS_SERV)   )
 endproc
```

The process *features* is itself an interleaving of the two features.

The structuring of feature interactions proposed in [CGLN93] has been useful to explore the problem using LOTOS. It enabled us to isolate problems and to concentrate on them. This idea was further developed in the concept of backward execution of a specification. By using this method, we can produce rapidly a scenario where an interaction can be found under the guidance of the designer. The method consists in decomposing problems into sub-problems to isolate scenarios that are likely to generate interactions.

2. DETECTING AMBIGUITIES BY BACKWARD REASONING

2.1 Ambiguous actions

We say that an observable action in a LOTOS specification is *ambiguous* if in the behavior tree of the specification there is a branching point where the action is the first observable one in at least two branches. Ambiguity represents nondeterministic behavior of the 'black box' being specified: it represents the situation where the user presses a button, and one of two different effects can follow. Ambiguous actions can be offered in several cases: most obvious are the cases where two behaviors are specified using the choice [] or disable [> operator. These are easy to detect, thus they are of limited interest. The most important case is when two behaviors are interleaved and both contain the same action, leading to the possibility that one of the many combinations of interleavings produces a sit-

uation of ambiguity.
    As a trivial example, in the following behavior

        a ; **b** ; c ; **stop**
    | | |
        e ; **b** ; f ; **stop**

after trace *<ae>* we end up with

        b ; c ; **stop**
    | | |
        b ; f ; **stop**

    We can think that **b** here represents an ambiguous key, leading to two behaviors that represent two different features. Because of the way the specification is written, a disorderly mix of the two features results if the user presses **b**. One can easily construct more complex examples involving process instantiations, etc. where the nondeterminism is far from obvious. In [BL93] such an example is given. We give a similar example in this paper, however much simpler and specified in a different style.
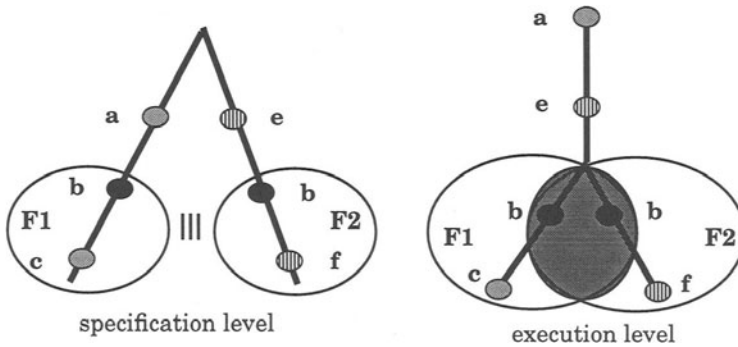

Fig 1. specification and resulting execution paths

    We concentrate on the problem of detecting cases of ambiguity, seen as symptoms of feature interaction. Note that in general ambiguity and nondeterminism can involve internal actions, however these will be ignored here for simplicity of discussion.

2.2 Case studies
    The case of call-waiting and three-way calling is relatively simple. Both involve the action *flash_hook*. Since the two processes are interleaved, it is clear that an interaction could exist.

process *features*[u, n](NUMBER:phone_number,SUBS_SERV:subscrib_services):**noexit**:=
    call_waiting[u,n](NUMBER,SUBS_SERV)
    |||
    three_way_calling[u,n](NUMBER,SUBS_SERV)
  endproc

where

process *call_waiting*[u,n](NUMBER:phone_number, SUBS_SERV:subscrib_services):**noexit**:=
    [call_waiting IsIn SUBS_SERV] ->
        n ! NUMBER ! ring
        ; **u ! NUMBER ! flash_hook**
        ; n ! NUMBER ! connect
      ;
       (user_events[u,n](NUMBER,SUBS_SERV)
        [>
        (n ! discon_req
         ; **stop**
)
)
endproc

process *three_way_calling*[u,n](NUMBER:phone_number,
                   SUBS_SERV:subscrib_services):**noexit**:=
    [three_way_calling IsIn SUBS_SERV] ->
        u ! NUMBER ! flash_hook
        ; u ! NUMBER ! dial ? C:phone_number
        ; n ! NUMBER ! conreq ! C
        ; n ! NUMBER ! connect
      ;
       (user_events[u,n](NUMBER,SUBS_SERV)
        [>
        (n ! discon_req
         ; **stop**
)
)
endproc

Note that the simplest case of call-waiting sequence has been proposed. In reality one should be able to answer call waiting immediately after receiving a dial tone. We found that it is difficult to specify this more general case by using existing LOTOS operators. This is because when a subscriber is involved in a feature, the other features can sometimes be disabled, something that cannot be expressed easily with the interleave operator. In order to deal with such cases, we have proposed a new *suspend and resume* operator for LOTOS, which is described in [CDN93].

The SUME case of ambiguity, resulting when a user tries to change the personalized options of the voice mail feature while a card connection is being used, is more complex because the ambiguity of the pound key action is due to two processes that are not directly interleaved. The interleaved situation is entirely contained in process *network_complete_connection* via instances of processes

calling_card_intercept and *relay_user_events* where the predicate is *_user_actions* could be evaluated with respect to the value *pound_key*. This case shows that it is not always easy to find feature interactions directly.

```
process network_complete_connection[n,a] (CALLER,CALLED:phone_number):noexit:=
      n ! CALLED ! ring
    ; n ! CALLED ! connect
    ; n ! CALLER ! connect
    ;
      (
        relay_user_events[n](CALLER,CALLED)
        |||
        calling_card_intercept[n,a](CALLER)
      )
endproc


where
process relay_user_events[n] (CALLER,CALLED:phone_number):noexit:=
      n ! CALLER ? EVENT:primitives [ is_user_actions(EVENT) ]
    ; n ! CALLED ! EVENT
    ; relay_user_events[n](CALLER,CALLED)

  []
      ...
endproc


process calling_card_intercept[n,a](CALLER:phone_number):noexit:=
      n ! CALLER ! pound
    ; a ! CALLER ! play_announce_new_number
    ; n ! CALLER ? NEW_NUMBER:phone_number ! conreq
    ; network_complete_connection[n,a](CALLER,NEW_NUMBER)
endproc


 process voice_mail[v, n](NUMBER:phone_number):noexit:=
      n ! NUMBER ! ring
 ;    v ! NUMBER ! voice_mail_answer
 ;    n ! NUMBER ! connect
 ;    v ! NUMBER ! play_announce_pwd
 ;    n ! NUMBER ! pwd
 ;    v ! NUMBER ! deliver_messages
 ;    v ! NUMBER ! play_announce_star_pound
 ;
 (
      n ! NUMBER ! star
    ; v ! Number ! good_bye
    ; stop
  []
      n ! NUMBER ! pound
    ; v ! NUMBER ! play_announce_management
    ; stop
  )
 endproc
```

It is the evaluation of the predicate *[is_user_actions(EVENT)]* that risks producing an ambiguity with respect to the two actions in boldface. If the variable EVENT captures a pound sign, the predicate will be *true* and thus the action will be executable. It is process *voice_mail* that plays the role of the environment and decides this case.

2.3 The method

Most LOTOS tools available today are based on systematic use of inference rules or expansion. In both cases one runs sooner or later into state explosion. Some new techniques such as goal oriented execution [HLS93] [BE93] and interleaved expansion [QLP93] provide partial relief to this problem.

The method presented here consists in exploring a specification by focusing on detection of ambiguities when a new feature is introduced, thus exercising only portions of the specification that are relevant to the problem. This is achieved by using well-known concepts and techniques, but in a particular manner. We decompose the specification, we find local traces, we substitute them into the overall behavior expression, and then conventional stepwise techniques are applied. Therefore, this is an incremental method.

The method consists of three steps:

Step 1: collect the various instances of an action found in different features throughout the specification.

Step 2: verify if the behavior expressions that contain these identical actions are interleaved.

Step 3: apply backward reasoning to verify if there are cases where the behavior expressions found in step 2 that are interleaved can produce non-deterministic choices when these behaviors are executed under the constraints of their environment. This means verifying if other behaviors that are in parallel with them can lead to such a case. In other words the fact that two behaviors containing the same action are interleaved is a necessary but not a sufficient condition.

Note that we have made the hypothesis that features are interleaved, and that their simultaneous occurrence is pathological, which seems to be true in most cases.

The above method is relevant to detect ambiguity of signals. There are other cases of non-determinism that are not caused by ambiguities of signals but by errors in logic. In LOTOS these reveal themselves as incomplete or contradictory abstract values in guards and/or predicates. In these cases, step 1 is no longer necessary, and the method consists in finding features that are interleaved in step 2 and then applying the procedures of step 3.

Step 3 has been implemented in a tool, which is discussed below.

2.4 Backward reasoning in LOTOS

As mentioned, the simple fact that an action is in interleave with itself is not sufficient for two behaviors to lead to ambiguities. It is also necessary that paths leading to the ambiguity be simultaneously possible under a given set of conditions. The fact that two events that are interleaved can create non-determinism

can be proved by executing the specification to reach the first case and then, using the resulting behavior expression, trying to execute the second case. This can be done in several ways, for example by using already known techniques such as forward step-by-step execution in combination with automatic generation of symbolic trees and goal-oriented execution [HLS93][BE93].However these techniques will produce state explosion to various degrees. Most of the state explosion is the result of interleaved behavior expressions. The multiple combinations of solutions all lead to the same desired state where our two ambiguous actions are possible. Backward reasoning allows to find the two paths necessary to reach our ambiguous case without having to worry with the problems created by interleaving.
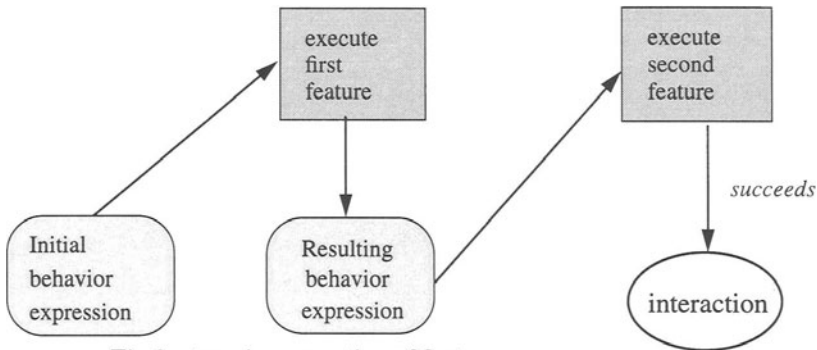

Fig 2. stepwise execution of features

Our method of 'backward reasoning" appears to be effective in many cases. In a step-by-step, interactive way, LOTOS specifications can be executed backward. This is due to the fact that the language has no side effects, with the exception of those that determine the order of actions in traces. One can start from a selected behavior expression and travel in reverse direction through action prefix, choice, disable, and enable operators.Whenever a new variable is encountered, a value for it is provided.This value may fail because of predicates encountered later; however the same thing can also occur in forward execution.When interleaved behavior expressions are encountered, they are attached to the behavior tree developed so far, since they could be needed for synchronization later.When a synchronization operator is encountered, the trace developed so far needs to be validated, so it needs to be executed forward with respect to the synchronized behavior.This may require synchronization with some of the interleaved behavior expressions attached earlier. There can be several such behavior expressions, both on the trace side and on the synchronized behavior side. Note also that this synchronization may depend on appropriate values having been entered earlier. Goal-oriented execution can be used to facilitate the process. When a process declaration is encountered, we must search all the instantiations of that process in the behavior expression, choose one of them, and decide on parameters passed

(this is the only case of choice in backward execution, which otherwise is deterministic). If we can travel all the way to the root of the main behavior declaration, we have found an executable path. Just as for regular forward execution, executed actions are removed from the behavior expression. Obviously, as in forward step-by-step execution, success of the procedure depends on the insight of the user who must select meaningful paths and values to build an appropriate scenario.

The concept of backward execution of protocols and service specifications has already been studied in [DB78] [Hol85], and more recently, by using logic programming concepts related to ours, in [Lin90].
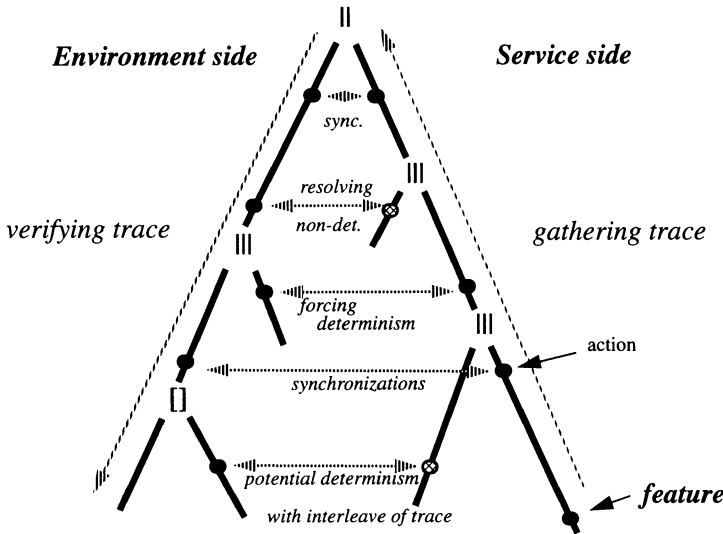

Fig.3 Backward reasoning on a parse tree like representation

From the above, it should be clear that backward reasoning uses both backward execution and conventional forward execution. While backward execution requires specially designed tools, forward execution can be done with existing interpreters and expanders, although in our experience goal-oriented execution is a necessity.

2.5 Applying the method
In this section, we present the application of the method to the SUSE case of ambiguity of the *flash_hook* in the three-way calling and call-waiting case. First we assume that we have a fully functioning specification of the three way calling service. We add the call waiting feature and try to see if some of the actions of this new feature risk nondeterminism with respect to existing features.

The method is applied in two steps. First we must prove that a backward path satisfying all conditions of synchronizing processes, starting with the action

flash_hook, does exist within the first feature, in our case the three-way calling. This path is summarized in Fig. 4. Then, given the behavior expression after producing this path, we attempt to execute a second path starting from the call waiting *flash_hook* action. The resulting path is summarized in Fig. 5. Since the two paths are executable within each feature, we can conclude that it is possible for the *flash_hook* action to present itself in a situation of nondeterminism.

We now describe how the first path is derived. Processes involved in this path have been shaded in Fig. 4. Boxes represent behavior expressions or instantiated processes. Let's start with the *flash_hook*. Step 1 tells us to look for more occurrences of a *flash_hook* action. We find out that there is one in the three-way calling feature. Step 2 tells us to verify if the behavior expressions where the two *flash_hook* actions belong are in a situation of interleaving. This is indeed the case, as it can be seen by simple inspection. Having these two behavior expressions, which happen also to be processes, we can start to analyse how they can be executed in interleaving.

We therefore start to execute the three-way-call feature backward. First we observe that our two features are instantiated in a process called *features* that itself is instantiated in a process *completed_connection*:

```
process complete_connection[u,n]  (NUMBER:phone_number,
                                   SUBS_SERV:subscrib_services):noexit:=
       n ! NUMBER ! connect
    ;
     (
        user_events[u,n](NUMBER,SUBS_SERV)
        |||
        features[u,n](NUMBER,SUBS_SERV)
     )
endproc
```

Within the above process we discover that our instance of process *features* is further interleaved with *user_events*. Inspection of this second process leads to the conclusion that there is nothing of interest for our problem in it, consequently we keep going backward following action prefixes to end up with the *n ! NUMBER ! connect* action. This action is important because it tells us that a full connection exists when the *call_waiting* feature is activated.

We are now at the top of the behavior expression of process *complete_connection*. We now must look for an instance of this process. We find one of interest in process *call_initiator_phone*. We observe that we can follow a series of action prefixes all the way back to the top of the behavior expression of this process to the *off_hook* action. Again we look for instances of this process and we find that the only case is in process *phone*. Thus we know the path that leads to a full connection of a phone.
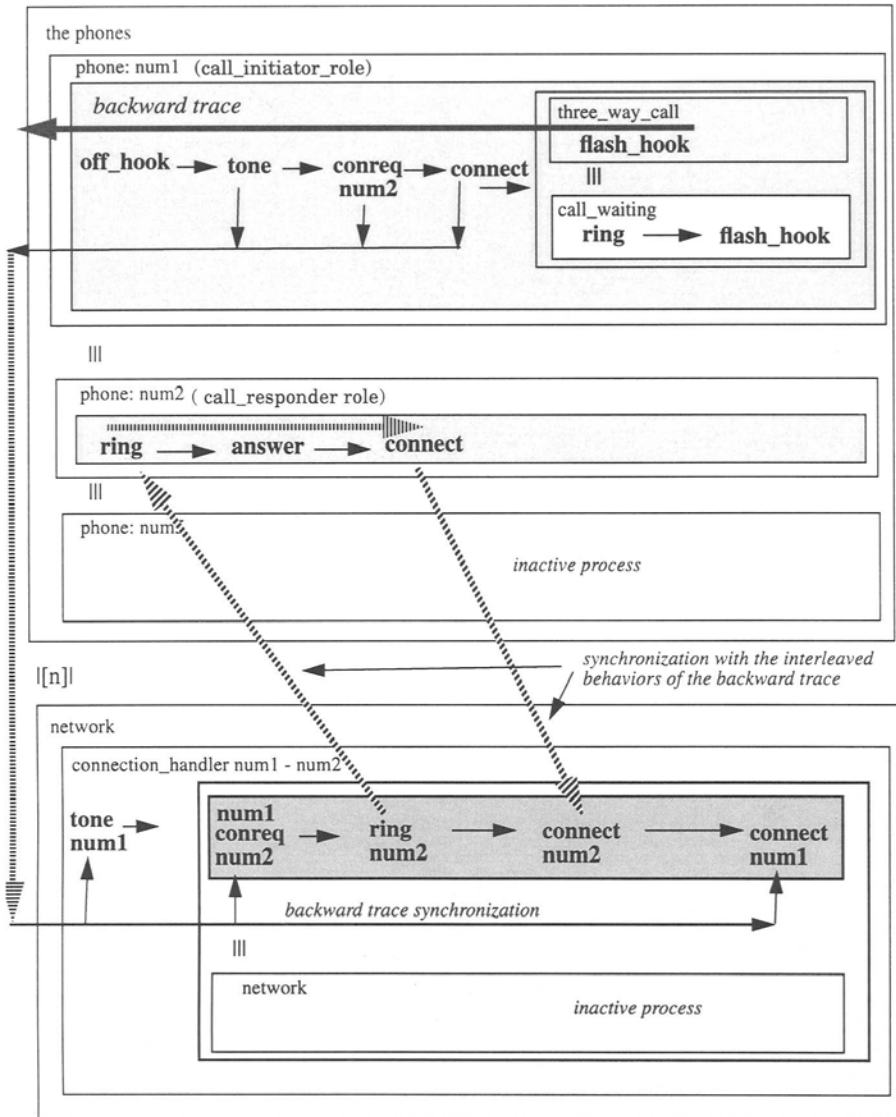
Fig. 4   Call waiting interaction with tree way calling summary

Backward execution path starting from three way calling *flash_hook*

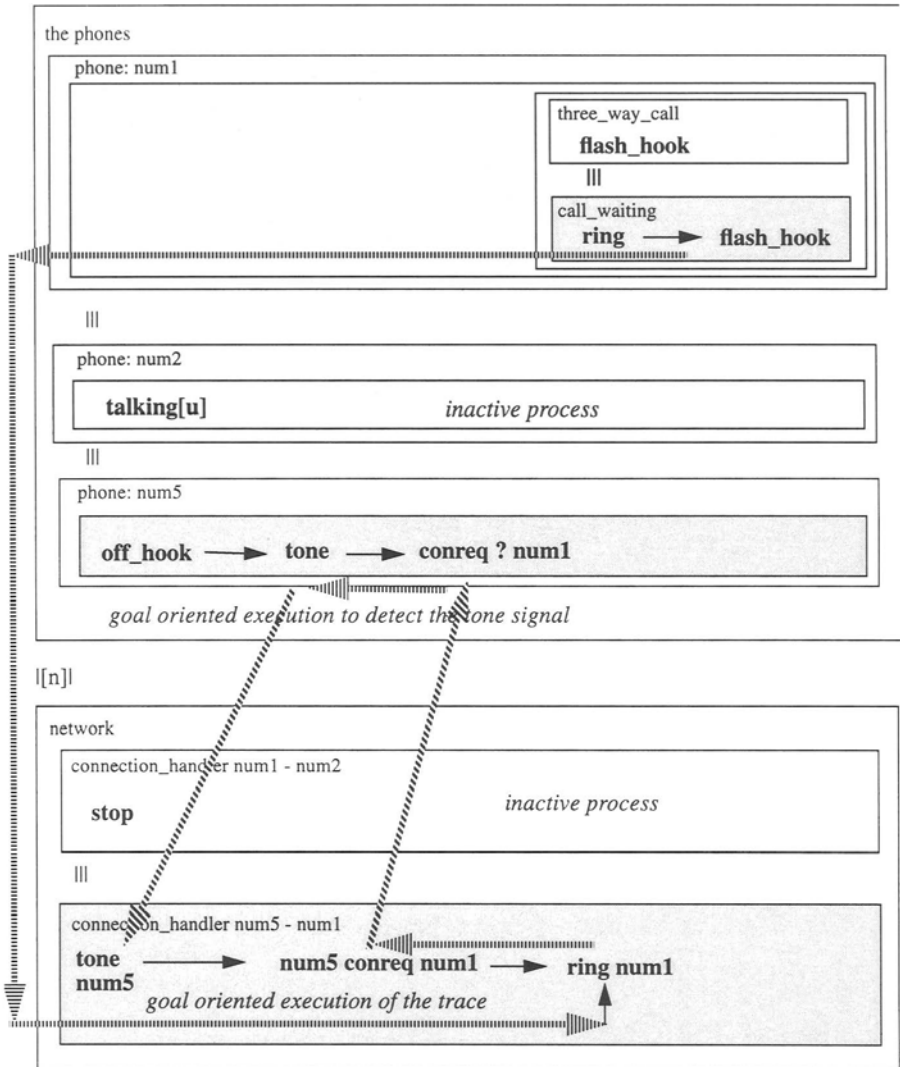This figure depicts the execution of the call waiting path from the state where the three way call is activated.

Fig. 5 Call_waiting interaction with three way calling summary
Backward execution path starting from call waiting *flash_hook*

Process *phone* has many instances that are all in parallel with an instance of process *network*. First we must pick one of these instances. This has the effect of instantiating the value of the formal parameter *number*. In our case we have picked the instance of *phone[u,n](num1)*. Concerning this choice, as well as the other choices of parameters that are necessary at this level, note that since the method consists of finding at least one scenario that verifies our goal, any appropriate values can be supplied by the user. In simple phone specifications these values are usually phone numbers that decide which phones are involved in communication. Here we only need to establish a connection between a phone that has both features (call waiting and three way calling) and a sort of neutral phone (without any features) to avoid interferences from features that are not presently of interest. The instance of process *phone[u,n]* that we have selected belongs to a behavior expression that is in parallel with an instance of process network. We must now verify if the collected trace satisfies the conditions of process *network*.

Thus we have obtained, by backward execution, the following instantiated trace, which we call $T_{3waycall}$:

u ! num1 ! off_hook -->
   n ! num1 ! tone -->
      n ! num1 ! conreq ? num2 (instance of variable Called_num) -->
         n ! num1 ! connect -->
            u ! num1 ! flash_hook --> ...

In other words, the trace must be composed with the remaining high-level behavior expression:

     ($T_{3waycall}$ ||| phone[u,n](num2) ||| ... ||| phone[u,n](num5))
  |[n]|
     network[n]

Thus we must try to compose $T_{3waycall}$ with the *network*, this time in forward execution, keeping in mind that any actions of process network that cannot synchronize with the trace could synchronize with the remaining behavior expressions that are interleaved with the trace (here *phone[u,n](num2)* ||| ... ||| *phone[u,n](num5)*).

One available technique to help in this process, is to apply goal oriented execution taking the sequence of actions of trace $T_{3waycall}$ as a goal. However we shall continue describing the process as if it were purely manual one, in order to explain the steps.

Thus, let us examine process *network*.

```
process network[n,a]:noexit:=
     n ? CALLER:phone_number ! tone
   ;
   (
      connection_handler[n,a](CALLER)
      |||
```

```
      network[n,a]
   )
endproc
```

The network's first action is to provide a dial tone. This matches with the second action of T$_{3waycall}$.

The execution of this action instantiates also the variable *CALLER* to the value *num1* by value passing.

The next step is to instantiate process *connection_handler*, noting that we also have the possibility to interleave recursively with process *network*.

```
process connection_handler[n,a](CALLER:phone_number):noexit:=
      n ! CALLER ? CALLED:phone_number ! conreq ?  SCR_LIST:screen_list
   ;
   (
      [ CALLED NotIn SCR_LIST ] ->
            n ! CALLED ! ring
         ;
            (
               n ! CALLED ! connect
               ; n ! CALLER ! connect
               ; relay_user_events[n](CALLER,CALLED)
            []
               ...
endproc
```

Again we see that the next action in T$_{3waycall}$ can match the first action of process *connection_handler*, i.e. *n ! CALLER ? CALLED:phone_number ! conreq* , but then the next action from T$_{3waycall}$ (*n ! NUMBER ! connect*) can no longer match the remaining sequence of process *connection_handler*. Thus at this point we have to go back to the many instances of process *phone* that are interleaved with T$_{3waycall}$ to see if one of these instances can provide a matching action to the ring. We pick instance *phone[u,n](num5)*. Eventually we reach the *connect* action that matches our call initiator. This was the last action before the *flash_hook* action. No other actions from the *network* process are required in order to proceed in T$_{3waycall}$. So far this exercise has produced a full connection between phones 1 and 2 that is necessary in order to reach the *flash_hook* action of feature three-way call.

Now we are back to the original problem: how can process *call_waiting* be activated when a full connection is established ? This is the second path to explore as summarized in Fig. 5. Processes involved in this path have been shaded. As we mentioned at the beginning of this exercise, the second path will be derived with all behavior expressions starting in the state they have reached while deriving the first path. We can use again the same kind of backward reasoning, leaving the details to the reader.

We start with the following behavior expression that results from the backward execution of the first feature (where *Rest of*$_{3way\_call}$ is now reduced to the *flash_hook* action and what follows it).

((Rest of $_{\text{3way\_call}}$ ||| Call_waiting(num1)) ||| talking[u](num2) ||| ... ||| phone[u,n](num5)   )
|[n]|
(Stop of $_{\text{connect\_handler(num1,num2)}}$ ||| network[n])

The call waiting trace will have to find a matching trace in the interleaved process *network* and there we can expect that some synchronization will be required with one of the remaining interleaved *phone* instances.

When this is done, we have proof that nondeterminism can exist on the *flash_- hook* trigger for the three-way-calling and the call-waiting features.

The same reasoning can be applied to the call waiting, call forward on busy example. This time it is the *ring* action that is ambiguous.

2.6 A tool

The backward execution method used in step 3 has been implemented in a prototype tool programmed in Prolog. The backtracking feature of Prolog is uniquely suited for this purpose. The internal representation of LOTOS specifications in the tool is quite unlike what is found in usual interpreters or expanders. We have used an inverted data base which makes it possible to walk a behavior both top- down and bottom-up. Actions, process identifiers, etc., are uniquely labelled so that the user can instruct the tool to direct execution to those she wants to use. No new inference rules are needed, the only novelty being in the way the inference rules are applied.Our tool does not try to validate parameter values with relation to expressions contained in parameter lists. For example, if we have an instantiation such as P[...](...,n-1,...) for a process declaration P[...](...,m,...) it would be necessary to see that n has value m+1. Currently, it is responsibility of the user to choose values so that such consistency is assured. In a more sophisticated tool, values would have to be derived automatically whenever possible, or at least consistency checks would have to be included.

To carry out easily the type of analysis advocated in this paper, it would be desirable to include a backward execution facility in conventional LOTOS interpreters, together with features to replace behavior expressions with the obtained traces, and then continue with forward execution (possibly goal-oriented), as we have shown.

3. CONCLUSIONS

Detection of feature interaction is an existence proof. We explore all the potential alternatives until we find a symptom of interaction. Backward and forward execution work together to reduce the number of cases to be considered. The backward trace from the point of interest acts like a partial test case where a number of interleaved actions are missing and can be filled by choices of the user.

We have shown that this technique can be facilitated by the use of appropriate tools. The use of appropriate specification styles may also be important, although we have insufficient experience in this respect.

This technique has other applications beyond the one discussed in this paper.- For example, it allows generating meaningful scenarios or use cases, starting

from the desired goals, in a stepwise fashion and with meaningful values.

REFERENCES

[BE93] Brinksma, E., and Eertink, H. Goal-Driven LOTOS Execution. To appear in: A. Danthine, G. Leduc, and P. Wolper (eds). Protocol Specification, Testing and Verification, XIII. North-Holland.

[BL93] Boumezbeur, R., and Logrippo, L. Specifying telephone systems in LOTOS. IEEE Communications Magazine, Aug. 1993, 38-45.

[CDN93] The suspend and resume operator. Canadian contribution to ISO TC97/ SC21, WG 1, November 1994 (available from authors).

[CGLN93] Cameron, E.J., Griffeth, N., Lin, Y.-J., Nilson, M.E., Schnure, W.K., Velthuijsen, H. A Feature Interaction Benchmark for IN and Beyond. IEEE Communication Magazine, 31, 3, 64-69, 1993.

[DB78] Danthine, A., and Bremer, J. Modeling and Verification of End-to-End Transport Protocols. Computer Networks, 2 (1978), 381-395.

[FL94] Faci, M. and Logrippo, L. Specifying Features and Analyzing their Interactions in a LOTOS Environment. To appear in the Proc. of the 2nd International Workshop on Feature Interaction in Telephone Systems, Amsterdam, 1994.

[HLS93] Haj-Hussein, M., Logrippo, L., and Sincennes, J. Goal-oriented Execution of LOTOS Specifications. In: M. Diaz and R. Groz (Eds.) Formal Description Techniques, V. North-Holland, 1993, 311-327.

[Hol85] Holzmann, G.J. Backward Symbolic Execution of Protocols. In: Y.Yemini, R. Strom, and S. Yemini (eds.) Protocol Specification, Testing, and Verification, IV. North-Holland, 1985, 19-27.

[Lin90] Lin,Y.J. Analyzing Service Specifications Based upon the Logic Programming Paradigm. IEEE GLOBECOMM '90, vol. 1, 611-655.

[Najm93] Dahl, O.C. and Najm E. Specification and Detection of IN Service Interference using LOTOS. To appear in: R. Tenney, P.D. Amer, M. Ü. Uyar (eds) Formal Description Techniques, VI, North-Holland, 1994.

[QLP93] Quemada, J., Larrabeiti, D., and Pavon, S. Compressing the state space representation, To appear in: R. Tenney, P.D. Amer, M. Ü. Uyar (eds) Formal Description Techniques, VI, North-Holland, 1994.

[SL93] Stepien, B. and Logrippo, L. Status-Oriented Telephone Service Specification. To appear in: T. Rus and C. Rattray (eds.) Theories and Experiences for Real-Time System Development, 1994.