Representing and Verifying Intentions in Telephony Features Using Abstract Data Types

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

Abstract. Feature intentions describe the intended behavior of telephony features. A method for formally specifying feature intentions by abstract data types is given. Further, a method for detecting violation of certain types of feature intentions at the design stage is provided. Specification languages considered are SDL, Prolog, and LOTOS. If the specification is in LOTOS, detection can be helped by the use of goal-oriented execution. Examples of specification and detection are provided, namely involving Originating Call Screening and Call Forwarding.

1. Introduction

[CGL94] provide a survey of the many types of feature interactions possible in telecommunications systems, and propose some categorization schemes. In one of these they distinguish among three main causes of feature interactions: violation of feature assumptions, limitations on network support, and intrinsic problems in distributed systems. [BW94] categorize the different techniques to deal with the problem in the three following categories: avoidance, detection, and resolution. In this paper, we propose a new detection technique, which can be used at the design and specification stage, to find interactions caused by violations of feature assumptions and by intrinsic problems in distributed systems.

Feature interactions are not necessarily a problem. Some are intended and some are undesirable. In this context, it is difficult sometimes to base detection on service specifications alone, that cover only the procedural or technological aspects of features. In many instances, features are meant to implement some subscriber’s wishes or intentions. An undesired feature interaction can thus be defined as a violation of intentions. This concept is reflected by many authors in their work at various levels of their analysis. For example [LL94] distinguish between procedural and behavioral specifications of features and [GV94] distinguish between technological and policy features. In the first case, the authors use procedural and behavioral specifications concurrently to detect feature interactions. In the second case, the authors separate feature interactions into two groups and decide to address the policy features that reveal mainly intention violations. Technological feature interactions are detectable by inspection of the procedural specification. Feature interactions resulting from violations of intentions however cannot be always detected from such inspection alone because they depend on the intentions of the user (behaviors or policies). Consequently, we need to be able to describe explicitly and separately such intentions and verify them against the specification of the various features present in a system. Some feature interactions resulting from scattering in a distributed system can also be detected using this method. The two papers mentioned so far have clearly separated the actual specification of intentions from the feature interaction detection mechanism. While the basic principles are similar, their implementations were very different. [LL94] use temporal logic while [GV94] use proposals or goals in a Prolog form to represent user intentions.

Intentions have also been described as *invariant* properties [Zav93], which must hold throughout the system's lifetime.

In this work, we propose a technique for feature intention violation detection at the design stage, based on Formal Description Techniques (FDT) that use Abstract Data Types (ADT) to represent intentions. We show how the technique can be used to detect feature interactions in association with various standard procedural formal description techniques. LOTOS appears to be the most suitable one, but SDL is also suitable to a certain extent. A more generic technique based on Prolog programming is also considered. We stress some benefits of the systematic specification and detection approach that is made possible by FDTs.

1.1 Definition of a generic intention representation

We start by constructing an executable formal model (or prototype) of the system with features. This formal model can be built in any of the languages mentioned above. Intentions (i.e. invariants) are represented in such a way that when a feature is activated, they can be verified for every subsequent action. When verification fails, the invariant has become false, so a flag is raised. Consequently, we need a generic mechanism to verify simultaneously many different features that might have been activated during the execution of a specification that represents a connection attempt between two subscribers within their own respective environments. The principle of a generic data type representation is useful, since the specification of the features and of their interaction detection mechanism is isolated in the data part of the specification. Our technique is related to the one of [GV94]. Although their paper deals with run-time detection, while we deal with detection at the design stage, we do detection by executing the specification. Furthermore, we use a flat, or tabular, representation, as opposed to the hierarchical system of [GV94]. The latter was motivated by a resolution technique, which is not among our goals.

2. Representing intentions using Abstract Data Types

In order to define a generic representation and verification mechanism to support the specification of a variety of features, we need first to determine the categories of data and the rules associated with features. Telephone systems operate with a limited set of data or rules: phone numbers, signals, feature names, and databases of various types of names and phone numbers. Rules usually are restriction sets such as in the *Originate Call Screening* feature. In its ADT representation, a feature is an operation that has the above mentioned data as its domain, and uses either a substitution or a restriction rule to determine its range.

The intention of a feature can be abstracted to rules with specific combinations of the four basic classes of data delimited above. So we can represent an intention as an operation of boolean result that indicates whether a given combination of the basic data involved in an operation is allowed.

intention: Fid, partyRole, operation, Restriction_set -> Bool

For example the originate call screening (OCS) prohibits any connection with a number that is in the screening list. This can be formulated thus:

The number that has the called role shall not be connected if it is in the screening list.

The above example can thus be expressed with the following equation:

$$\textbf{intention}(\mathbf{F_{ocs}}, \textbf{called}(\mathbf{N}), \textbf{connect}, \mathbf{L}) = \mathbf{N}\ \textbf{NotIn}\ \mathbf{L};$$

where $\mathbf{F_{ocs}}$ identifies the feature Originating Call Screening, N the phone number involved in a called role in operation *connect* and L is the restriction set that in this case is a screening list, *N NotIn L* is a boolean expression that verifies if the number N is not found in the restriction set L, which would mean that the intention is verified for this particular operation. Note this set will be used in different ways.

It is clear from this definition that equations have to be provided for each type of operation, including the operations that are specific only to certain features. Similarly, the interaction detection methods used in [BA94] uses a tabular notation, where analysis of these tables against actual features can lead to interaction detection. Our equations play the same role as the tables used in their method. However, the representation of feature intentions is only a first step toward feature interaction detection. In our method, only the actual simulation of two simultaneous features can verify if an interaction occurs. However with intentions clearly defined, the detection of an interaction can be automated (at least in part) and is no longer subject to the designer's judgement. Also, it is now clear that the specification of an intention consists in specifying the constraints on a set of operations. For the application of our method, two guidelines must be respected:

• Intentions of a feature are described independently of other features. This means that the specification of an intention on a specific operation is done solely in consideration of the given feature. There is no consideration of potential interactions with another feature at this stage. This is important especially in the context of a multi-vendor environment where characteristics of a feature of a vendor may not be known to another vendor. Also, from a design point of view, this characteristic allows the feature provider to add new features without having to understand their behavior in combination with other features, as already mentioned in [GV94].

• Intentions of a feature must be described for every operation that exists in the system regardless of which feature is actually used. This means that operations must be considered as belonging to some sort of generic pool of operations that can be used by various features. This, even if some operations were created specifically for a feature such as for example Originating Number Display. While originally an operation may have been created for a feature, chances are that it will be re-used by other features and thus become more generic. This convention is a result of the formalism we are using, because with ADTs the equation set must be complete and convergent.

For implementation purposes we have chosen to represent intentions as ADT operations that actually detect the intention's violation. We thus define operation *violates* as having the same domain as the intention operation defined above. The range of this operator is also a boolean value that indicates whether the intention has been violated or not. However, the boolean expression is reversed. For example, here we are saying that, in the case of F_{ocs}, a violation occurs if the number involved in operation *connect* is found in the restriction list:

violates(Focs, called(N), connect,L) = N IsIn L;

2.1 Examples

The use of our representation of intentions can be illustrated in three examples that cover the two main categories of violation of assumptions and problems in distributed systems defined in [CGL94]. The remaining category of limitations in network support is not addressed here because it is mainly the result of problems linked to ambiguities that have been covered with the backward reasoning methods in [SL94].

2.1.1 Originate Call Screening

This feature is known to interact with the call forward feature and this is an example of interaction caused by intrinsic problems in distributed systems. Here is the full example of the specification of the intention violation for the feature, where L is the screening list:

```
type intentions is Boolean, features, signals, roles, number_set
        sorts intention
        opns
                violates: feature, role, signals, number_set -> Bool
        eqns
                forall N:number, S: signals, L:number_set
                ofsort Bool
                        violates(Focs,called(N),connect,L) = N IsIn L;
                        violates(Focs,called(N),off_hook,L) = false;
                        violates(Focs,called(N),tone,L) = false;
                        violates(Focs,called(N),dial,L) = false;
                        violates(Focs,called(N),conreq,L) = false;
                        violates(Focs,called(N),detect_forward,L) = false;
                        violates(Focs,called(N),connect_refused,L) = false;
                        violates(Focs,called(N),ring,L) = false;
```

```
                                violates(Focs,called(N),answer,L) = N IsIn L;
                                violates(Focs,called(N),talk,L) = N IsIn L;
                                violates(Focs,caller(N),S,L) = false;
endtype
```

We have here some rules for which the outcome is immediately determined (false) while for others we need to further evaluate if a number belongs to the restriction set. We can also observe that the F_{ocs} rules are complex only for the called role because the restrictions apply for the called party, while there is only one catch-all rule for the caller role that always evaluates to *false*.

2.1.2 Call Forward

Another interesting example related to intrinsic problems in distributed systems is loop or livelock detection for the call forward feature. These loops are the result of personalized instantiation by the subscriber of the feature. Here the restriction set L contains the phone number that has activated its call forward feature, to prevent circular call forwarding. The set of equations is similar to the one we have seen for originate call screening:

```
eqns
      forall N:number, S: signals, L:number_set
      ofsort Bool
            violates(Fcfw,called(N),connect,L) = N IsIn L;
            violates(Fcfw,called(N),off_hook,L) = false;
            violates(Fcfw,called(N),tone,L) = false;
            violates(Fcfw,called(N),dial,L) = false;
            violates(Fcfw,called(N),conreq,L) = false;
            violates(Fcfw,called(N),detect_forward,L) = false;
            violates(Fcfw,called(N),connect_refused,L) = false;
            violates(Fcfw,called(N),ring,L) = N IsIn L;
            violates(Fcfw,called(N),answer,L) = N IsIn L;
            violates(Fcfw,called(N),talk,L) = N IsIn L;
            violates(Fcfw,caller(N),S,L) = false;
```

2.1.3 Unlisted number

Finally, the detection of the interaction between the unlisted number feature and the calling number display feature that is an example of violation of assumptions is feasible when specifying the intention violation rules for the unlisted number only. In this case, the only operation that would cause a violation is a display operation. Consequently there are only two equations required to describe these intentions:

```
eqns
      forall OP:signal, ROLE:role, L:number_set
      ofsort Bool

            violates(Fuln, calling(N), display, L) = N IsIN L;
           OP ne display => violates(Fuln, ROLE, OP, L) = false;
```

Here the restriction list L contains the number that has the unlisted number feature activated. If this number is involved in a display operation in a calling role, this means that there is a violation of intentions.

As a final remark on these examples, we note that one must be careful about how rules that detect violations of intentions are defined, because there could be interactions between the rules themselves, both inside a single feature, and among features. For example, one may decide that ringing a number that is on the originate call screening feature restriction list is a violation. This happens to be true when the OCS feature is considered in isolation from other features. But if that same called number has a call forward feature that is activated, then the caller should not be restricted to hop to another number that is not on the screening list. This however introduces a delicate intention definition problem, well-known in this research area: should we consider numbers or subscribers as a decision criterion. In the second case, if the intention is really to prevent talking to a given subscriber, then ringing and call forwarding should also be disallowed in the originate call screening rules.

2.1.4 Other languages

So far, the ADTs were shown in LOTOS syntax. However SDL syntax is very similar, only keywords and delimiters change, and the semantics is identical.

Here is the SDL specifications of intentions for F_{ocs}:

```
newtype intentions
        operators
                violates: feature, role, signals, number_set -> Bool;
        axioms
                for all N in number, S in signals, L in number_set
                ofsort Bool
                        violates(Focs,called(N),connect,L) == N IsIn L;
                        violates(Focs,called(N),off_hook,L) == false;
                        violates(Focs,called(N),tone,L) == false;
                        violates(Focs,called(N),dial,L) == false;
                        violates(Focs,called(N),conreq,L) == false;
                        violates(Focs,called(N),detect_forward,L) == false;
                        violates(Focs,called(N),connect_refused,L) == false;
                        violates(Focs,called(N),ring,L) == false;
                        violates(Focs,called(N),answer,L) == N IsIn L;
                        violates(Focs,called(N),talk,L) == N IsIn L;
                        violates(Focs,caller(N),S,L) == false;
endnewtype intentions
```

Intentions can be easily represented with PROLOG clauses. The following example illustrates how we can specify the violation condition for the connect operation in a call forward feature.

```
        violates(cfw,called(N),connect,L):-
                                                isin(N,L).
```

where clause isin(_,_) is defined as:

```
isin(X,[X|_]):- !.
isin(X,[_|T]):-
            isin(X,T).
```

3. Detecting violations

3.1 Basic mechanism

So far, we have seen how to specify violations. Mechanisms for detection must now be considered. Our method works by executing a formal specification (which could be called also formal prototype or formal model) of the system with features. The ADTs, included in the specification for this purpose, will check execution sequences to see whether a violation is incurred.

As mentioned earlier, violations of intentions can result from the lack of continuity that is inherent to distributed systems. A network element may forward processing to another network element without passing feature data, or the logic of a feature may be activated too early or too late or not at all when the critical data it needs becomes available. Our technique records the initial intentions of a feature and activates a mechanism that is independent of the distributed system to verify that the original intentions still hold regardless of which component of the system handles a call. The ADTs represent a kind of a central call model monitor that captures every event associated with a call, regardless of the component in which it occurs, and verifies if that event conflicts with the predefined intentions. Such a central mechanism would be impossible to implement in an actual system, but is possible in a specification.

3.2 Implementation

There are two main implementation concerns:

- minimal disruption of feature specifications.
- independence of specification style.

The first concern means that we do not want to modify the specification of the system in detail for the verification of each feature, as mentioned in [DN94]. The second concern means that we do not want to constrain the usefulness of our method to a specific structure, or style, or even specification language.

In LOTOS, when several processes are combined together by means of the parallel composition operator ||, in order for an action to execute, all processes must participate simultaneously (synchronize) in the action. Further, each process can provide its own conditions for the action to execute, and all such conditions must be true simultaneously in order for this to happen. This effect is inspired by CSP [Hoa85] and is further explained in [FLS91]. Thus the monitoring effect we want to obtain can be achieved gracefully by using an independent monitoring process in parallel with the system specification. This process will synchronize with every action occurring in the telephone system, and will check for every action whether a violation is occurring.

```
        telephone_system[u,n] || verify_intentions[v,u,n]
```

In SDL, this type of immediate and continuous monitoring is harder to achieve. One would need to set up some message duplication through a channel connecting a verification process.

In Prolog, one needs to add a verification clause to each execution rule.

```
exec_action(sequence(action(A,F,RL),RS),A,RS):-
                    verify_intentions(A,F,RL).
```

Since LOTOS is the language in which this method appears to be most easily specified, henceforth we limit ourselves to LOTOS and we leave the other languages for further research.

We have studied two different approaches for the procedural part of the verification process. They are based on different principles. In the first one, most of the work is done by appropriate processes, while in the second one, most of the work is done by appropriate ADTs. Thus we call these methods respectively process-oriented and ADT-oriented. In both cases, we need to record the fact that a feature has been activated and we need also to verify for each subsequent action occurring in the system if an intention has been violated.

3.3 Process oriented feature intentions verification

In this method, the *verify_intentions* process spawns an independent monitoring process every time a new call is created. *verify_intentions* itself is in interleave (operator |||) with the intention monitor, i.e. it runs in parallel with it but without participating in (synchronizing with) its actions. Each intention verification process is independent of the others. Consequently the *verify_intentions* process uses the call initiation action *off_hook* as a trigger to create instances of process *intentions_monitor.*

```
process verify_intentions[f,u,n]:noexit:=

        u ? N:number ! off_hook ? L:location ? ConId: ConIdent
       ;
           (
               intentions_monitor[f,u,n](ConId)
             |||
               verify_intentions[f,u,n]
           )
endproc
```

However, a call may or may not activate a feature. Thus the process *intentions_monitor* has a normal mode of operations in the case no feature is activated, and this mode can be disabled ([> operator) at any time by a mode where a feature is activated and begins to be monitored.

```
 process intentions_monitor[f,u,n](ConId:ConIdent):noexit:=

           normal_operation[u,n](ConId)
           [>
                activate_intention[f,u,n](ConId)

 endproc
```

In order to detect the activation of a feature we need an explicit action in the telephone system specification. This action carries information such as the nature of the feature and its restriction set.

The distinction between many instances of intention monitors corresponding to different connections in the system is achieved using a connection identifier *ConId* that is captured on the *off_hook* action and is propagated throughout the various actions belonging to the same connection.

```
process activate_intention[f,u,n](ConId:ConIdent):noexit:=

            f ! ConId ? Feature: feature ? SL: number_set
           ; monitor_one_feature[u,n](Feature,SL,ConId)
           |||
           activate_intention[f,u,n](ConId)
endproc
```

The process *activate_intention* will spawn a process *monitor_one_feature* that is dedicated to monitoring one activated feature for a given connection. It also uses a recursive interleave construct to allow more than one activated feature to be monitored at the same time. Some connections will activate many different features as they progress in time. For example, using a resource oriented style [VSV91], the originate call screening feature is activated by a call initiator behavior using gate *f* to separate it well from the regular specification of the feature:

```
process phone[u,n,f] (N:number,FWD:feature_data,OCS:feature_data):noexit:=

             u ! N ! off_hook ! origin ? ConId:conIdent
           ; f ! ConId ! extract_feature(OCS) ! extract_ocs(OCS)
           ; n ! N ! tone ! origin ! ConId
           ; u ! N ! dial ? C:number ! origin ! ConId
           ; ...
endproc
```

The process *monitor_one_intention* is where the verification really occurs. It will make use of the generic *violates* operator.

```
 process monitor_one_feature[u,n] (F:feature,SL:number_set,ConId:ConIdent):noexit:=
           hide v in
            n ? N:number ? S:operations ? C:number ? L:location ! ConId
           ; (
            (* a violation has been detected, display a message and stop *)
              [ violates(F,determine_role(L,N),S,SL) ] -> v ! violation ! F ! S ! ConId ; stop
            []
            (* no violation detected, move on to the next action verification *)
              [ not(violates(F,determine_role(L,N),S,SL)) ] -> monitor_one_feature[u,n](F,SL,ConId)
          )
        []
        (* etc...*)
endproc
```

The first action *n ? N:number ? S:operations ? C:number ? L:location ! ConId* captures the data associated with an operation. These are *S*, the name of the operation, and *L* the location indicator (originating or terminating). Then, this data capture operation action is followed by a non deterministic choice expression that offers either an action where the evaluation of the *violates* operation is true (a violation has occurred), or a silent move represented by a recursive invocation of the *monitor_one_intention* process in the case where the evaluation of the *violates* operation is false (no violation).

Normally, in a LOTOS specification one has several action types, where an action type is determined by the gate and by the data elements offered. Since actions can synchronize only if they have the same types, for each type of action we need a verification choice construct. The first guarded expression in the construct handles a violation, while the second guarded expression represents the transition in case there was no violation.

3.4 Abstract Data Type oriented intentions verification

In the second approach, there is only one feature monitor that handles all activated features at once. Feature activations are kept in a set of activated features and every time an action is executed, the verification is performed recursively for each activated feature contained in that set. If a violation is detected, a message indicating the circumstances is displayed and the engine keeps looking for more violations.

The intentions monitor consist of a non-deterministic choice between a feature activation detection action that will result in the update of the activated features set, and an action verification process that will verify intentions for every action occurring except for a feature activation action.

```
process verify_intentions[f,u,n](AFS: ActFeatSet):noexit:=

        (* feature activation detection mechanism *)
         f ? N:number ? F: feature ? SL: number_set
         ; verify_intentions[f,u,n](Insert(activatedFeature(F,N,SL),AFS))
        []
        (* action verification mechanism *)
         verify_action[f,u,n](AFS)
endproc
```

The action intentions verification process will start the recursive intentions verification for each kind of action.

```
process verify_action[f,u,n](AFS:ActFeatSet):noexit:=

        u ? N:number ! off_hook ? L:location ? ConId:ConIdent
        ; verify_one_intention[f,u,n](N,ConId,off_hook,L,AFS,AFS)
         []
        ... (* one similar construct for each type of action *)
endproc
```

The feature intentions verification process *verify_action* is a recursive process that passes the data associated with an action to process *verify_one_intention*. This process, that is in parallel with the telephone system specification, evaluates this data for each activated feature until all activated features have been verified. It then moves on to the next action by recursively invoking process *verify_one_intention*. Thus, process *verify_one_intention* acts as a kind of list processor.

This alternate method was found to be more implementation-oriented, but also more difficult to use with our simulation strategy (next section), thus it was not investigated further.

4. Simulation strategies: use of goal oriented execution

So far we have presented a technique to represent feature intentions in a specification in such a way that any intention violations encountered when the feature is executed are flagged. We now discuss strategies that will lead to detection of potential violations. To do this, one needs to execute the specification in some way.

As we have seen, the definition of intentions consists in defining equations for each operation. Looking for interactions means looking for the execution of actions which cause the execution of these operations.

Combinatorial explosion of the state space is a well-known problem that is encountered when trying to explore all paths in the specification of a complex system. Some relief strategies are known, in particular two methods were implemented in our tools: goal-oriented and backward execution. These methods help derive the execution paths necessary to reach a goal which is one of the actions or operations we are looking for. By inspecting these paths we can see which features were activated at the time the violation was detected, and conclude that an interaction between them was detected. Goal oriented execution has been introduced in [HLS93] [BE93] and backward execution, which is a variant of goal oriented execution, has been presented in [SL95] for the LOTOS paradigm. As far as we know, no such method currently exists for SDL, but the ideas presented in [DB78][Hol85] apply to finite state machines thus could be adapted to SDL.

Both methods are useful for finding execution paths leading to a specified action, which we call the *goal*. In goal-oriented execution, the execution tree is narrowed by using syntactic information that eliminates paths exploring parts of the specification where obviously the goal action cannot be found. In backward execution, the specification is executed backwards from the goal action. Both techniques are complicated by the need of taking into consideration concurrent paths.

As an example, to find a violation in the case of the Originate Call Screening feature we need to perform a goal oriented execution looking for the action *connect* for a number that is in a called role, and which is in the screening list. This can be found in the specification of a *callResponder_role* process of a phone. Going backward we find that this operation is possible only if a phone has been rung and the high-level specification shows that this can occur only if a switch has synchronized with a phone on the *ring* operation. We now inspect the specification of a switch to find out that there are two different ways a switch can ring a phone: either directly as a result of a connection request or indirectly as a result of a call forward. Going backward on these two paths, we find that a *callInitiator* phone would have had to place a connection request, that such a phone could have had an originate call screening feature activated, and that the phone that has forwarded its calls to the prohibited number could have been the one that was rung by this phone.

In this case, our goal was the action *connect* that belongs to the telephone system. Another way to perform the goal oriented execution would have been by using the violation detection action of the intentions monitoring process that is in parallel with the telephone system specification. This is a powerful and unmistakable method for violation detection, but experience has shown that it is very difficult to implement with the tools we currently have, because of combinatorial explosion. We need more advanced relief techniques to make this method workable.

The feature interaction between F_{ocs} and F_{cfw} has been detected using the University of Ottawa Goal Oriented Execution tool, which is a part of the ELUDO toolkit. The system configuration used was as follows:

```
(
    (
       phone[u,n,f](num1,fwd(none),ocs(Insert(num3,{})) )
        |||
      phone[u,n,f](num2,fwd(num3),ocs({}))
       |||
       phone[u,n,f](num3,fwd(none),ocs({}))
     )

 |[n]|

     network[n]
)

 ||

verify_intentions[f,u,n]
```

In this configuration phone *num1* has F_{ocs} while phone *num2* has F_{cfw} to *num3*.

Our end goal is *v ! violation ! 'OCS' ! connect ! conn1 ! num3*. However by using only this goal, the tool runs into combinatorial explosion in trying to complete all paths that could possibly lead to this action. The search can be reduced by giving some further directions to the tool. These directions are directly related to the goal. First it is clear that we are placing a call from a phone that has its F_{ocs} activated. In our case this happens to be phone num1. We thus specify the three necessary actions for phone num1 to place a call regardless of the terminating number. This is indicated with the three actions:

```
u !num1 !off_hook !origin !conn1,
u ! num1 ! dial ? C ! origin ! conn1,
n ! num1 ! conreq ? C ! origin ? ocs ! conn1,
```

Also we know that at least two features have to be activated, first the F_{ocs} feature and then any feature. This is indicated with the two generic actions:

```
f~,
...
f~
```

Finally we need to indicate the offending action for which we know that a violation could be detected on the terminating side:

```
n ! num3 ! connect ! dest ! conn1
```

The goal definition, shown in the form of the actual command to the tool, is as follows:

Goal we want to reach:

```
[u !num1 !off_hook !origin !conn1,
 f ~,
 u ! num1 ! dial ? C ! origin ! conn1,
 n ! num1 ! conreq ? C ! origin ? ocs ! conn1,
 f ~,
 n ! num3 ! connect ! dest ! conn1,
 v ! violation ! 'OCS' ! ring ! conn1 ! num3] \ \ [f, u, v].
```

where \ \ [f, u, v] means that no intermediate actions involving these gates should exist in the trace (intermediate actions on gate n only are allowed).

The resulting trace is:

```
--> u !num1:number !off_hook:operations !origin:location !conn1:ConIdent line(s): [303,214]
--> f !conn1:ConIdent !OCS:feature !Insert(num3, {}):number_set line(s): [345,215]
--> n !num1:number !tone:operations !origin:location !conn1:ConIdent line(s): [377,262,216]
--> u !num1:number !dial:operations ?C,C,C=num2:number !origin:location !conn1:ConIdent
        line(s):[389,217]
--> n !num1:number !conreq:operations !num2:number !origin:location !ocs(Insert(num3, {})):feature_-
        data !conn1:ConIdent line(s): [364,269,227]
--> n !num2:number !ring:operations !dest:location !conn1:ConIdent line(s): [377,286,242]
--> f !conn1:ConIdent !call_forward:feature !Insert(num3, {}):number_set line(s): [412,254]
--> n !num2:number !detect_forward:operations !num3:number !dest:location !conn1:ConIdent
        line(s): [353,295,255]
```

```
--> n !num3:number !ring:operations !dest:location !conn1:ConIdent line(s): [377,286,242]
--> n !num3:number ! connect:operations ! dest:location !conn1:ConIdent lines:[377,289,247]
--> v !violation:error_msg !OCS:feature !ring:operations !conn1:ConIdent !num3:number line(s):
        [378]
End of trace.
```

The trace indicates a path that goes through phone *num2,* and a call forward operation to phone *num3*.

This experience shows that with this kind of tool there needs to be some search strategy defined by the analyst, but this strategy can be defined by the characteristics of one feature independently of the others. As a futher detail, one can note that in our search strategy a call forward feature to phone *num3* for phone *num2* was set up.Happily, LOTOS allows more generality than this. A variable can be used instead of a constant by utilizing the *distributed choice* construct. In this way, one can specify that a given phone instance has a call forward feature on, without specifying explicitly to which number. A tool including a *narrower* will try different values chosen from those declared in the ADT and eventually will find a solution. The following specification ca be used, instead of the previous one:

```
(
    (
       phone[u,n,f](num1,fwd(none),ocs(Insert(num3,{})) )
        |||
      (choice X:number [] phone[u,n,f](num2,fwd(X),ocs({})) )
       |||
       phone[u,n,f](num3,fwd(none),ocs({}))
    )

 |[n]|

    network[n]
)

 ||

verify_intentions[f,u,n]
```

The variable X stands for an unspecified forward number in expression:

```
(choice X:number [] phone[u,n,f](num2,fwd(X),ocs({})) )
```

The results of the goal oriented search are identical to the previous case.

5. Conclusions

A method for detecting intention violations in the specification of telephony systems with features has been presented. The method can be effectively carried out by using the LOTOS language and associated tools.

Abstract data types are well suited for the verification of intentions because they are apt to specify constraints in an implementation-independent way and they can be used in a variety of language contexts. Specification of intentions is an important step toward feature interaction detection but for actual detection, some type of execution must be performed.We have seen how the goal-oriented execution method can help focussing execution and thus reducing combinatorial explosion. However for this to happen a lot of information must be provided to the tool. More research needs to be done in the field of static inspection of specifications to determine search strategies to reduce as much as possible the scope of execution techniques.

Acknowledgments. We are grateful for support of the Telecommunications Research Institute of Ontario, Bell-Northern Research, and the Natural Science and Engineering Research Council. Yow-Jian Lin of Bellcore asked us the 'right questions' that led to this work. We should thank Antoine Bonavita for having helped us with the goal-oriented tool.

References

[BE93] E. Brinksma and H. Eertink, Goal-Driven LOTOS Execution. In: A. Danthine, G. Leduc, and P. Wolper (eds). *Protocol Specification, Testing and Verification, XIII.* North-Holland, 1993, 45-60.

[BW94] W. Bouma and H. Velthuijsen. Introduction to the book *Feature Interactions in Telecommunications Systems*, IOS Press, 1994, vii-xiv.

[BA94] K.H. Braithwaite and J.M. Atlee. Towards automated detection of feature interactions. In: L.G. Bouma and H. Velthuijsen, *Feature Interactions in Telecommunications Systems*, IOS Press, 1994, 36-59.

[CGL94] E.J.Cameron, N.D.Griffeth, Y.-J. Lin, M.E.Nilson, W.K.Schnure and H. Velthuijsen. A feature interaction benchmark for IN and Beyond.In: L.G. Bouma and H. Velthuijsen, *Feature Interactions in Telecommunications Systems,* IOS Press, 1994,1-23.

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

[DN94] O.C. Dahl and E. Najm. Specification and Detection of IN Service Interference Using LOTOS. In: R.L. Tenney, P.D. Amer, and M.U. Uyar (eds) *Formal Description Techniques, VI.* North-Holland, 1994, 53-69.

[FLS91] Faci, M., Logrippo, L., and Stépien, B. Formal Specification of Telephone Systems in LOTOS: The Constraint-Oriented Approach. *Computer Networks and ISDN Systems 21* (1991) 53-67.

[GV94] N.D.Griffeth and H. Velthuijsen.The negotiating agents approach to runtime feature interaction resolution. In: L.G. Bouma and H. Velthuijsen, *Feature Interactions in Telecommunications Systems,* IOS Press, 1994, 217-235.

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

[Hol85] G.J. Holzmann. 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.

[LL94] F.J.Lin and Y.-J. Lin. A building block approach to detecting and resolving feature interactions. In: L.G. Bouma and H. Velthuijsen, *Feature Interactions in Telecommunications Systems*, IOS Press, 1994,86-119.

[SL95] B. Stepien and L.Logrippo. Feature interaction detection by using backward reasoning with LOTOS. In: S.T. Vuong and S.T. Chanson. *Protocol Specification, Testing and Verification XIV.* Chapman & Hall, 1995, 71-86.

[VSV91]C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma, Specification Styles in Distributed Systems Design and Verification, Theoretical Computer Science 89, 1991, 179-206.

[Zav93] P. Zave. Feature Interactions and Formal Specifications in Telecommunications. IEEE Computer, August 1993, 20-29.