Reachability Analysis of Feature Interactions: A Progress Report

Keith P. Pomakis Joanne M. Atlee*
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1

Abstract. Features are added to an existing system to add functionality. A new feature *interacts* with an existing feature if the behavior of the existing feature is changed by the presence of the new feature. Our research group has started to investigate how to detect feature interactions during the requirements phase of feature development. We have adopted a layered state-transition machine model that prioritizes features and avoids interactions due to non-determinism. We have a tabular notation for specifying behavioral requirements of services and features. Specifications are composed into a reachability graph, and the graph is searched for feature interactions. This paper demonstrates how reachability analysis has been used to automatically detect known control interactions, data interactions, and resource contentions among telephony features.

1 Introduction

A *feature* in a software system is a sub-program that adds functionality to another feature or to the original system. A new feature *interacts* with an existing feature if the behavior of the existing feature is changed by the presence of the new feature. A key problem in software enhancement is how to add features to a system without disrupting the services and features already provided. Note that features, by definition, interact: at the least, a new feature is expected to interact with those features and/or services whose functionalities are intentionally modified by the new feature. Thus, the problem of detecting feature interactions is twofold: we want to validate specified interactions and to detect unspecified interactions.

*This research has been supported in part by the Natural Sciences and Engineering Research Council of Canada grant FSP137101, with matching funds from Bell-Northern Research Ltd.

It is essential that the analysis of feature interactions be automated. There exist systems that provide hundreds of features: the DMS-100[1] telephone switch offers over 800 telephony features (e.g., *Call Waiting*, *Call Forwarding*, etc.). Given a system that supports N features, there are $N(N-1)/2$ pairs of features that need to be checked for possible interactions.

Previously, we presented a requirements notation for specifying features [3]. This paper introduces a set of tools that incrementally compose feature specifications into a reachability graph and search the graph for certain classes of interactions. The tools' interaction-detection algorithms test *states* as opposed to *paths* in the reachability graph. This means that tested, reachable states can be discarded if the reachability graph generator will not re-generate the state and if the state is not important in future phases of the composition.

The following section describes the architectural model we have adopted and our notation for specifying features. Section 3 describes the execution model, based on the architectural model, for constructing the composition of a set of features. Section 4 defines the classes of feature interactions that can be detected automatically, using examples of interactions among telephony features. The paper concludes with a discussion of open issues.

2 Architectural Model

A large class of feature interactions are due to non-determinism. For example, if two active features f and g operate on the same input from the user, which feature gets the input data first? If feature f operates on the data first and g never sees the input, then the presence of feature f has altered the control-flow of feature g. Similarly, if feature f operates on the data first and g sees a modified version of the input data, then again the presence of f may affect the behavior of feature g.

We adopt a stack architecture (or layered architecture) of services and features, which resolves interactions due to non-determinism by prioritizing features.

[1]DMS is a trademark of Northern Telecom.

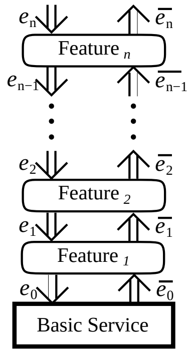

Figure 1: Feature stack architecture

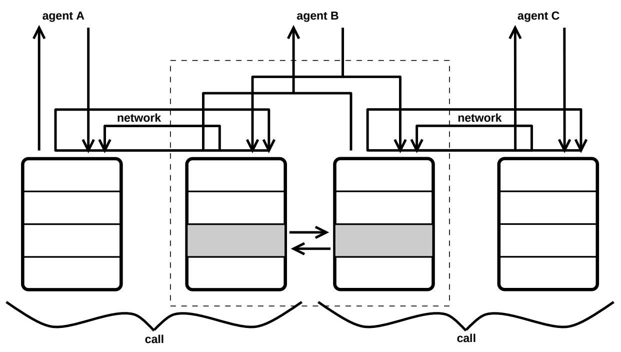

Figure 2: A call system

The basic service resides at the bottom of the stack, and the features are placed in a prioritized order above the basic service (see Figure 1). Events from the *agent* (the user of the service) are input to the top-level feature and flow down through the layers towards the basic service. Responses from the basic-service machine are sent to the bottom-level feature and progress up through the layers towards the user. The feature at the top of the stack has top priority: it is the first feature to operate on data entering the system and is the last feature to modify data that is leaving the system. Newly activated features are given highest priority and are placed on the top of the stack, because it is assumed that users will primarily want to interface with the feature they have most recently invoked.

There are databases, operating systems, and control software [4, 6] that have stack architectures. Communication protocols and telephone calls [11] can also be modeled using stack architectures: a call can be represented by two communicating feature stacks, where each stack models the behavior of one end of the call. If an agent is involved in multiple calls, then the agent will interface with multiple feature stacks (see agentB in Figure 2).

If a feature affects more than one call, then a component of the feature will reside in each appropriate feature stack (see Figure 2). For example, telephony feature *Call Waiting* allows one agent to engage in two different telephone calls at the same time. Two or more calls are referred to as a *call system*.

The remainder of this paper will concentrate on detecting and resolving feature interactions in telephony system. We anticipate that our methods can be adapted for use in other service-oriented systems.

2.1 Tabular Specification Notation

Basic telephone service is specified as two communicating state-transition machines. One machine (called the the Originating Call Model, or simply OCM) models the initiation of a call: it specifies the interaction between the calling agent and the telephone service (e.g., receiving a dial-tone, dialing digits, etc.). The other machine (called the Terminating Call Model, or simply TCM) models the receiving end of a call: it specifies the interaction between the called agent and the service (e.g., ringing phone). In addition, both machines send messages to each other, to synchronize the creation and dismantlement of a telephone connection.

Telephone features are specified as *sets* of state-transition machines. Each machine in the set describes behavior of a feature with respect to one end of a call. Thus, if a feature can modify either the originating or receiving end of a call, or if a feature affects more than one call, then a feature's specification may consist of more than one state-transition machine. The decision to decompose feature specifications into multiple sub-feature specifications was made to ease the reasoning and writing of specifications.

Sub-feature specifications have a tabular format. Each row in the table specifies a state transition from the state on the left to the new state on the right. A state transition is activated by an input event, and its occurrence may produce output events and request/release resources.

Table 1 formally specifies the behavior of the *Three-Way Calling (3WC)* feature for the initial call with respect to the OCM (i.e., the agent initiated the call)[2]. The 3WC sub-feature starts in state NULL. The agent invokes the feature by quickly pressing the receiver hook (which is normally used to hang-up the phone) and causing a $\Downarrow_A$ FlashHook event. This puts the current call in a HOLDING state and starts a second feature stack (**NewCall**(*OCM*)) to model the call to the third party. Eventually the second call is established, and the two calls are joined into a single CONFERENCE call involving all three parties. The feature is deactivated if either of the two calls is terminated. If the agent

[2] Separate tables would specify the behavior of the initial call with respect to the TCM (i.e., the agent received the initial call) and the behavior of the new call to the third party.

State	**Input**	**Output**	**NewState**	**Resources**
Null	$\Downarrow_A$FlashHook	**NewCall**(OCM)	Holding	**+bridge**
Holding	$\Rightarrow_{3WC}$Conference		Conference	
	$\Rightarrow_{3WC}$Null	Alert$\Uparrow_A$	Ringback	
	$\Uparrow_R$ReleaseTimeout	$\ll$*forward*$\gg$	Null	–**bridge**
	$\Downarrow_A*$		Holding	
	$\Uparrow_A*$		Holding	
Conference	$\Downarrow_A$Disconnect	$\ll$*forward*$\gg$	Null	–**bridge**
	$\Uparrow_R$ReleaseTimeout	$\ll$*forward*$\gg$	Null	–**bridge**
Ringback	$\Downarrow_A$Answered		Null	–**bridge**
	$\triangleright$**RingingTimeout**	Disconnect$\Downarrow_A$	Null	–**bridge**
	$\Uparrow_R$ReleaseTimeout	$\ll$*forward*$\gg$	Null	–**bridge**
	$\Downarrow_A*$		Ringback	
	$\Uparrow_A*$		Ringback	

Table 1: Three-Way Calling, for original call with respect to OCM.

hangs up the phone while trying to call the third party, leaving the second party on hold, the 3WC feature will attempt to re-establish the call by ringing the agent's telephone (in the RingBack state). Other transitions in the table specify the behavior of the feature under abnormal conditions (e.g., the feature consumes events from the environment if the call is on hold).

Six types of events have been identified:

Token events ($\Downarrow_A$t, $\Downarrow_R$t, $\Uparrow_A$t, $\Uparrow_R$t) indicate the receipt or sending of a message **t** (called a *token*) nominally passed between the agent and the basic service. If there are active features on the feature stack, then these features have access to tokens that are traveling to their nominal destination. Tokens always have a direction ($\Downarrow$ or $\Uparrow$) and a destination (Agent or Remote feature stack). An input event designating the receipt of an asterik token (e.g., $\Downarrow_R*$and $\Downarrow_A*$) matches any token event.

Internal events ($\triangleright$e) are events which indicate an internal decision made by a feature. For example when *Three-Way Calling (3WC)* is in state RingBack, the feature rings the agent's phone in an attempt to re-establish a held call that the agent hung up on. After a specified amount of time, the feature makes an internal decision to dismantle the call.

Parallel events ($\Rightarrow_f$S) are the communication mechanism among a feature's sub-features. If a feature *f* is represented by several sub-features residing on different feature stacks, each sub-feature can use a parallel event to notify their sibling sub-features that it has made a transition to a new state S.

StateChangeRequest events ($S_1 \Rightarrow_f S_2$(e)) are events which indicate the desire of a feature to change state. When a feature intercepts a StateChangeRequest from a lower-level feature, it can either pass it on unchanged, pass on a modified StateChangeRequest (which must specify a valid state transition rule in the target feature), or pass on nothing (thereby disallowing the requested state transition). These events are discussed in more detail in Section 2.3.3.

NewCall events are events specifying the creation of a new feature stack (i.e., a new call based on an Originating Call Model) on behalf of the agent.

Note that there are two methods by which a feature can affect the operation of the underlying call model. One method is to generate tokens on behalf of the local or remote agent; these tokens are passed down the feature stack towards the call model and are expected to cause the call model to change state. The second method is to circumvent certain state transitions in the call model or in lower-level features and replace them with new state transitions.

The tables specify the behavior of a feature in terms of functions. Let T be the set of all tokens that can be passed among features on the same feature stack; we define inT to be the set of input events denoting the receipt of a token $t \in T$ and $outT$ to be the set of events denoting the output of a token $t \in T$. Similarly, let R be the set of state-change-request events made by the call models and features; we define inR and $outR$ be the sets of events denoting the receipt or the output of a state-change-request, respectively. For a given feature f, let S be the feature's set of states, let I be the feature's set of internal signals, and let A be the set of available resources. Then formally, a tabular specification $\mathcal{T}$ represents the feature's transition relation:[3]

$$\mathcal{T} : (S \times A \times (inT \cup inR \cup I)) \mapsto (S \times A \times \mathcal{P}(outT \cup outR))$$

[3] In the above relation definition, $\mathcal{P}(s)$ represents the powerset of set s.

That is, $\mathcal{T}$ is a partial function from (states, allocated resources and input events) to (states, allocated resources and sets of output events). Note that sets T, R, and A will grow as new tokens, state-transition-requests, and resources are needed to implement new features.

2.2 Composition of Specifications

A feature is not a stand-alone-entity; it must always be examined in the context of the system or systems in which it is to exist. For this reason, it is important to understand the mechanisms through which features communicate when their specifications are composed.

2.2.1 Activation of Features

A feature can only affect the basic service if it is *active*. Features that the agent has subscribed to but that are not yet active are not on the feature stack but are instead managed by an entity called the *feature activator*. Every feature stack has a feature activator which monitors the stack for events which may trigger the activation of any of the features it manages.

In the initial configuration of a feature stack, all features, including the basic service, are inactive (i.e., in state NULL). A feature is activated and placed in the feature stack when the feature activator detects an event that causes the feature to transition from the NULL state. Normally when a feature is activated it is placed at the top of the feature stack. However, this may not always be the case, as there may exist certain high-priority features (such as *911*) which must remain at the top of the feature stack despite not being the most recently activated feature. When a feature deactivates (returns to the NULL state) it is removed from the feature stack and returned to the feature activator, which may activate it again at a later time. The basic service should always be the first feature to activate, as well as the last feature to deactivate.

A feature F can be activated in one of several ways:

- if a token falls off the bottom of the feature stack and feature F has a transition from its NULL state on the corresponding Token event. (This composition rule implies that active features have higher-priority access to input data than inactive features.)
- if an active feature requests a state transition and feature F has a transition from its NULL state on the corresponding StateTransitionRequest event (See Section 2.3.3).
- if a sibling feature in another feature stack controlled by the same agent makes a state transition and feature F has a transition from its NULL state on the corresponding Parallel event.

Before a feature can activate, its activation must be approved by all currently-active features. Section 2.3.3 describes this in more detail.

2.2.2 Token Queues

Since it is possible for a feature to output more than one token during a state transition, many tokens may be traveling through the feature stack at any one time. In order to impose a deterministic priority on the reception of tokens in a feature stack, each feature is provided with two input queues: one which accepts downward-moving from above, and one which accepts upward-moving tokens from below. Tokens may not pass each other in the feature stack, and must maintain their order in the queues. A token is only accepted from a queue if all lower-level queues in the feature stack are empty. When a feature accepts a token from one of its neighboring features (i.e., from one of its input queue) but has no rule specifying a state transition from its current state on the reception of that token, the token is passed to its other neighbor.

2.2.3 State Transition Verification

Because higher-level features have priority over a feature at level x, the feature at level x much ask permission of all higher-level features before making a state transition. A higher-level feature has the ability to permit, modify or disallow a state transition of a lower-level feature.

When a feature wants to make a state transition, a corresponding StateChangeRequest event is generated and passed from the top of the feature stack down to the feature making the request. StateChangeRequest events have priority over Token events (see Section 2.3.5); they bypass all tokens in token queues and are accepted immediately by intercepting features. An intercepting feature may either forward the StateChangeRequest unchanged, pass on a modified StateChangeRequest, or pass on nothing, in effect disallowing the state transition[4]. As with tokens, StateChangeRequest events from one neighboring feature are passed to the other neighbor if they do not activate a transition from the current state.

[4]Note that when a state transition in feature f is triggered by the reception of a StateChangeRequest from a feature g, it too must have its state transition approved by features of higher-priority than itself. This happens in a recursive manner until all StateChangeRequest events have been resolved. Since f is above g in the feature stack, fewer features have priority over f; thus, the recursion will eventually terminate.

When a StateChangeRequest reaches the feature that originally generated the request, the received StateChangeRequest event is compared against the event that was originally generated. If the StateChangeRequest has been modified, a new StateChangeRequest is generated so that the above features may approve the modified state transition[5].

When an unmodified StateChangeRequest event is returned to the requesting feature, the feature activator is queried to see if any new features wish to activate on such an event. If so, an attempt is made to activate the feature. Like all state transition attempts, the activation of a feature must be approved by all higher-priority features, which in this case is the set of all active features. The newly activated feature is usually pushed onto the top of the feature stack by default; if not, the configuration being analyzed must specify that the newly activated feature be inserted into the feature stack somewhere above the feature whose StateChangeRequest activated the new feature. This way, the newly activated feature has the ability to permit, modify, or disallow the StateChangeRequest that activated the feature. For example, if the *Terminating Call Model* (*TCM*) attempts a state transition from HUNTINGFACILITY to EXCEPTION because the agent's line is busy, *Call Waiting* (*CWT*) will intercept the state transition and try to establish the call on a second line; if CWT can establish the call, then it will have effectively prevented the very state transition in the call model that activated the feature.

2.2.4 Priorities of Events

In order for a feature stack to be deterministic in nature, the following priorities are imposed:

- All StateChangeRequest events are resolved first, recursively.
- Tokens waiting in queues are then accepted.
- Signals from sibling features (represented by Parallel events) are then resolved.
- Finally, internal decisions (represented by Internal events) and tokens from the environment are accepted.

[5]It is possible in the current model for an infinite loop to occur if two features are trying to modify a StateChangeRequest of a common lower-priority feature at the same time. One feature may reverse the modifications of the other, which then gets intercepted by the first again and re-modified, etc. At present we don't detect such infinite loops. However, we could model our algorithms to maintain enough information to detect these types of interactions.

3 Reachability Analysis

The verification and analysis that is currently performed by the composition algorithm can be divided into three classes – syntax checking, verification of structural properties of specifications, and detection of feature interactions.

3.1 Feature Interaction Detection

Once their syntax and structure have been checked, feature specifications may be composed together to form a feature stack specification. Subsequently, two feature stack specifications may be composed to form a call specification. During composition, each reachable state is tested to determine if a feature interaction can occur at that state. These tests are based on state information only (i.e., the current state, the heads of the input queues, and the contents of the allocated-resources list). At present, the composition algorithm can detect five types of feature interactions.

data consumption A *data-consumption* interaction occurs when one feature consumes input data that another feature is waiting for. Such interactions are detected when multiple features are ready to operate on the same input event and the higher-priority feature consumes the input event, thereby preventing the lower-priority feature from ever seeing the input. Data-consumption interactions are resolved by the prioritization of features on the feature stack. Warnings about data-consumption interactions are used by feature designers to verify desired data consumptions and to reveal resolved, undesired data consumptions which need to be documented.

data modification A *data-modification* interaction occurs when one feature modifies the value of a token that is subsequently used by a second feature. The feature's specification must explicitly indicate when a feature modifies an output token (or a data field in an output token) by annotating the output event with a prime ('). The composition algorithm replaces prime (') annotations with information about which feature is making the modification and, if applicable, which of the token's data fields is being modified. Information about data modifications is cumulative. If the modified token is later intercepted and used by another feature, the interaction is detected and a warning is given, providing the feature designer with the token's modification history. As with data-consumption interactions, the prioritization of features resolves data-modification interactions;

warnings simply notify the feature designer of the resolved interaction.

resource contention Any attempt by a feature to acquire an instance of a resource beyond the specified capacity of that resource is detected as a *resource contention* interaction. Such interactions are detected by comparing the number of each resource already allocated to the agent (as listed in the agent's Resource lists) with the specified capacity of each resource. Resource contention is also reported if the set of composed features release more instances of a resource than they acquired.

data loss A *data loss* interaction occurs when a token is passed down through a feature stack and eventually falls off the bottom of the stack. This type of interaction typically indicates that one feature has made an invalid assumption about the readiness of another feature to accept a particular token. Data loss interactions are detected when the composition algorithm attempts to enqueue the token to an nonexistent output queue below the base service.

control interaction Sometimes one feature will attempt to control the behavior of another feature by forcing the second feature to undergo a particular state transition. A *control* interaction occurs when the second feature is not in the expected state at the time its 'new orders' are received; it also occurs if the requested transition does not exist. Since the composition algorithm cannot generate the next state, the algorithm must abort. A control interaction is the only feature interaction that causes the composition algorithm to abort.

3.2 Experience

A previous paper [3] described four feature interactions that we wanted to eventually be able to detect automatically. This subsection describes our experiences in trying to detect three of these interactions: a data modification, a data consumption, and a resource contention interaction. More complete descriptions of the results of these case studies and the feature specifications used can be found in [9, 10].

3.2.1 Data Modification

There is a known, desired, data-modification interaction between telephony features *Calling Number Display* (*CND*) and *Calling Number Display Blocking* (*CNDB*). When a call is being a initiated, the caller's OCM will send a CallRequest token to the receiver's TCM. Feature CND operates on top of a TCM; its purpose is to extract the caller's telephone number from the CallRequest token when the token is received by the TCM. Feature CNDB operates on top of an OCM; its purpose is to modify the CallRequest token so that feature CND will not extract and display the caller's number. When the caller's and callee's feature stacks are composed, the following data modification warning is reported.

```
Warning:  Feature Stack [CND/TCM] accepted token
      t:CallRequest!origin which was modified
      by [CNDB/OCM]
```

That is, a data modification is detected when the CND feature attempts to extract the caller's number from the CallRequest token and finds that the token as been modified (by the CNDB feature) to prevent the display of the number.

3.2.2 Data Consumption

There are several well-known, undesired interactions between features *Three-Way Calling* (*3WC*) and *Call Waiting* (*CWT*). The agent can create a FlashHook token by pressing the receiver hook quickly, as opposed to pressing the receiver hook long enough to hang up the phone. In CWT[6], the $\Downarrow_A$FlashHook event is used to switch between the two CWT calls. In 3WC, the $\Downarrow_A$FlashHook event is used initially to activate the feature; it is also used to join the three parties in a conference call once the call to the third party has been established. A data consumption interaction occurs if both features are ready to handle a $\Downarrow_A$FlashHook event, and one feature receives the token without forwarding it to the other feature.

If 3WC is used to establish a call to a third party and CWT is activated in the middle of this call, the composition algorithm detects 12 data-consumption interactions, 5 of which are unique[7]:

```
Features (CWT,3WC) accept "FlashHook"
      in state [Decision/Private/Active]
Features (CWT,3WC) accept "FlashHook"
      in state [HeldCall/Private/Active]
Features (CWT,3WC) accept "FlashHook"
      in state [Active/Private/Active]
Features (CWT,OCM) accept "Disconnect"
      in state [HeldCall/Private/Active]
Features (CWT,OCM) accept "Disconnect"
      in state [HeldCall/Private/ReleasePending]
```

[6] The following interactions involving the $\Downarrow_A$FlashHook event only occur when the agent is using a simple telephone that does not have special buttons for CWT and 3WC.

[7] The 7 warning messages not shown are duplicates of the 5 messages presented above. Interactions are detected between pairs of features, and these interactions can occur while the third feature is in various states. Since the composed state (consisting of the three features' current states) is different in every case, the composition algorithm issues a warning message.

As expected, several data consumption interactions involving the $\Downarrow_A$ FlashHook event are detected between CWT and 3WC. In addition, several other data consumption interactions are detected between CWT and the *Originating Call Model (OCM)*. In CWT, one call is always on hold. If the agent forgets about the held CWT call and hangs up (with a Disconnect token), then CWT will ring back the agent. That is, CWT will consume the Disconnect token and try to re-establish the call. Meanwhile, OCM is waiting for a $\Downarrow_A$ Disconnect to terminate the call. Thus, the interactions between CWT and OCM are desired; the consumption of the Disconnect token is specifically designed to delay the termination of the call.

3.2.3 Resource Contention

3WC and CWT features both require the user of a piece of hardware known as a bridge, but there is only one bridge available to each telephone. If an agent attempts to use both features at the same time, there will be a resource contention interaction.

The bridge is requested by the CWT sub-feature that operates on the new incoming call that the feature sets up. In 3WC, the bridge is requested by the sub-feature that initiates a call to the third party. If these two sub-features operate on the same feature stack, the following resource contention warning is reported when the features and call model are composed into a feature stack.

```
Resource Contention: [Null/Active/Active] =>
      [Holding/Active/Active] needs 2 bridges
```

If the sub-features resides on parallel feature stacks, then the resource contention would be detected when the calls are composed into call systems.

4 Related Work

A number of other researchers are investigating the feature interaction problem.

Researchers from PTT Telecom, PTT Research, and Kokusai Denshin Denwa Co., Ltd. (KDD) propose template formats for specifying features. Included in the templates are either extended finite state machine specifications (PTT) or message sequence charts (KDD). In the PTT approach, the extended finite state machines of two features are manually inspected to determine if their lifetimes can overlap (thereby affecting each other's control flow), if one feature uses data modified by the other, or if they use limited resources [7]. In the KDD approach, the message sequence charts of two features are manually composed, and feature interactions are revealed during this composition [12].

The proposed template formats are extremely expressive, and one can argue that all the information needed to detect an interaction can (although in practice may not) be in stated in the features' specifications. However, automated detection of feature interactions is preferred, given the number of features that need to be compared.

SDL [1] and LOTOS [2, 5] are often used to specify communication protocols. Both have an execution model, for which reachability graph generators and simulators can be built. Furthermore, one could use model checking techniques to verify global properties (e.g., safety and liveness properties) with respect to the specification's reachability graph.

Our approach is similar to the building blocks approach proposed by Bellcore [8]. Service and feature specifications are building blocks that are fitted together according to architectural constraints. The architectural constraints define communication flows among the specifications, as opposed to a flat composition of parallel LOTOS specifications. Another advantage is that the configuration of the composed system can change over time (due to the activation and deactivation of features), unlike configurations of SDL specifications.

5 Conclusion

As indicated by the title, this paper reports work in progress. At present, we have implemented an efficient reachability graph generator that composes features into feature stacks and composes feature stacks into calls. We have also implemented interaction-detection algorithms that test the state information of reachable states to determine if an interaction can occur. Finally, we have used the above algorithms to automatically detect several types of interactions.

In the immediate future, we intend to implement the third stage of composition: the composition of calls into call systems. This will complete the reachability graph generator and allow us to analyze complete call configurations.

A longer term goal is to allow features to raise *assertions* during their execution. Assertions are properties about the call that the feature expects to hold, even after the feature is deactivated. For example, telephony feature *Originating Call Screening* (*OCS*) is activated when a call is initiated. Its purpose is to compare a dialed number against a list of invalid numbers. If the dialed number passes the test, then the OCS feature will allow the call to proceed and will terminate normally. However, the feature assumes that the call eventually established will be to the number that passed the OCS test, and this assumption can be invalidated by features than change the destination of the call (e.g., *Call Transfer* and *Call Forward*). If features raise assertions, then conflicts between assertions raised by different features could be detected and reported.

Acknowledgements

We would like to thank Kenneth Braithwaite for his initial work on the architectural model, specification notation, and feature interaction detection algorithms; and for his comments and suggestions regarding changes to the model and notation, which we made to facilitate the generation and analysis of the reachability graph.

We greatly appreciate the time Greg Utas has taken to explain the architecture of the DMS-100 telephone switch and the many ways in which feature interactions can occur (and can be resolved).

We would also like to thank Greg Utas, Andre Vellino and Jack Dyment for their constant encouragement and support of this project.

References

[1] F. Belina and D. Hogrefe. "The CCITT-Specification and Description Language SDL". *Computer Networks and ISDN Systems*, 16:311–341, 1989.

[2] R. Boumezbeur and L. Logrippo. "Specifying Telephone Systems in LOTOS". *IEEE Communications*, 31(8):38–45, August 1993.

[3] K. Braithwaite and J. Atlee. "Towards Automated Detection of Feature Interactions". In *Proceedings of the Second International Workshop on Feature Interactions in Telecommunications Software Systems*, 1994 (to appear).

[4] R. Brooks. "A Robust Layered Control System for a Mobile Robot". *IEEE Journal of Robotics and Automation*, RA-2:14–23, April 1986.

[5] M. Faci and L. Logrippo. "Specifying features and analysing their interactions in a LOTOS environment". In *Feature Interactions in Telecommunications Systems*, pages 120–151, May 1994.

[6] A. Flynn, R. Brooks, and L. Tavrow. Twilight Zones and Cornerstones: A Gnat Robot Double Feature. Technical Report A.I. Memo 1126, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1989.

[7] E. Kuisch, R. Janmaat, H. Mulder, and I. Keesmaat. "A Practical Approach to Service Interactions". *IEEE Communications*, 31(8):24–31, August 1993.

[8] F.J. Lin and Y.-J. Lin. "A building block approach to detecting and resolving feature interactions". In *Feature Interactions in Telecommunications Systems*, pages 86–119, May 1994.

[9] K. Pomakis. *Reachability Analysis of Feature Interactions in Service-Oriented Software Systems.* Master's thesis, Department of Computer Science, University of Waterloo, 1995.

[10] K. Pomakis and J. Atlee. "Reachability Analysis of Feature Interactions: A Progress Report". Technical Report CS-96-21, Department of Computer Science, University of Waterloo, Waterloo, ON Canada., May 1995.

[11] G. Utas. "Feature Processing Environment", December 1992. Presented at the International Workshop on Feature Interactions in Telecommunications Software Systems.

[12] Y. Wakahara, M. Fujioka, H. Kikuta, and H. Yagi. "A Method for Detecting Service Interactions". *IEEE Communications*, 31(8):32–37, August 1993.