Reachability Analysis of Feature Interactions in Service-Oriented Software Systems

by

Keith Patrick Pomakis

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1996

Abstract

A *feature* in the context of a service-oriented software system is a program module that is added to a basic service in order to add functionality. An example of this type of system is the telephone system, where customers can subscribe to add-on telephony features such as *Call Waiting*, *Three-Way Calling*, and *Call Number Display*. An inherent problem of such a system is the *feature interaction problem.* A new feature *interacts* with an existing feature if the behavior of the existing feature is changed by the presence of the new feature.

This thesis describes an approach to detecting feature interactions during the requirements phase of feature development. The approach involves specifying features in the context of a layered state-transition machine model that prioritizes features and avoids interactions due to non-determinism. A tabular notation for specifying features has been developed. These specifications are composed incrementally, and a reachability graph of the composite system is generated. This thesis demonstrates how reachability analysis has been used to automatically detect six types of feature interactions, with an emphasis on telephony features.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A *service-oriented software system* is a software system whose purpose is to provide a set of services to a user or set of users. Further definitions employed by this thesis are as follows:[1]

- A *service* is some functionality provided to the user by a software system. Services can be used to enhance or extend other services. For example, the telephony service *Call Waiting* extends the basic POTS (Plain Old Telephone Service) telecommunication service.

- A *feature* is a component which provides part or all of the functionality of a service.

Thus, each service of a service-oriented software system is modeled as one or more interacting features. In cases where a service is modeled as exactly one feature, the two terms may be used interchangably.

Two features *interact* with each other if the behavior of one feature is changed by the presence of the other. A key problem in software enhancement is how to add features

[1]Note that these definitions may differ from those employed by other papers.

to a system without disrupting the services 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 functionality is intentionally modified by the new feature. Thus, the problem of detecting feature interactions is twofold: we want to validate specified interactions and detect unintended interactions.

There are two possible approaches to the feature-interaction problem that can be exercised at the requirements phase of feature development. The first approach is to specify the features with respect to an architectural model that by design avoids feature interactions. The second is to detect feature interactions by analyzing the formal specification of the system. [1]. The approach discussed in this thesis is a mixture of both of the above approaches.

Currently the problem of analyzing formal specifications in search of feature interactions is usually tackled by inspections and walkthroughs of the feature specifications, often coupled with testing of the implementations themselves. Such manual proof techniques are based on the inspection of state transitions. However, manual proofs can be tedious and are inevitably susceptible to human error. This is especially true of larger systems; as the number of services in a system grows, and, more importantly, as the number of service providers for a particular system grows, such manual techniques become unwieldy. For one thing, it becomes unfeasible to cognitively consider the effects of a new feature when combined with arbitrary combinations of existing features. Also, it is unlikely that third-party developers will be intimately familiar with the base system and existing features. Thus, automated analysis of feature interactions becomes essential.

This thesis proposes a specification notation for features that is designed to capture only the functional behavior of features. Thus, features can be described with this language during the requirements phase of feature development.

Several classes of feature interactions are due to non-deterministic execution of concurrent features. This thesis also proposes an execution model that resolves such interactions by prioritizing features and forcing a total ordering on the occurrence of features' actions. Even though many interactions can be resolved by prioritizing features, it is still desirable for feature designers to be aware of the interactions and to document them. As stated above, automated detection of feature interactions is essential: if a system has N features, then there are $N(N-1)/2$ possible pairs of features that need to be checked. This thesis describes a set of tools that can be used to incrementally compose specifications and to search the reachability graph for certain classes of interactions.

It is important to note that the tools' interaction-detection algorithms test *states* as opposed to *paths* in the reachability graph. This means that once a reachable state is tested, it can be discarded by the algorithm if it is not important in future phases of composition. This reduces the size of the reachability graphs at each incremental stage of composition.

Many of the examples presented in this thesis are taken from specifications of telephony systems. However, the principles discussed are general enough that they should be applicable to a wide class of service-oriented software systems.

The following chapter discusses the feature interaction problem by examining other proposed solutions. A general overview of reachability graph generation as it applies to feature interaction detection is also given. Chapter 3 describes a notation for specifying requirements of features and an existing architectural model, which we have adopted, that imposes a priority on features. Chapter 4 describes an execution model, based on the architectural model, for constructing the composition of a set of features. Chapter 5 defines the classes of feature interactions that can be detected automatically, using examples of interactions among telephony features. The thesis concludes with a discussion of open issues.

Chapter 2

Related Work

2.1 Other Approaches

There have been several approaches to the feature-interaction problem. This section describes some of these approaches as they relate to the three major areas of the feature-interaction problem: specification of features, composition of specifications, and detection and resolution of feature interactions.

2.1.1 Specification of Features

It is necessary to specify features in a concise and consistent manner so that the specifications of features can be composed and analyzed. A specification language designed for this purpose must have the ability to model enough information about features so that the detection of a large class of feature interactions is possible. Generally, the number of different types of interactions that can be detected is directly proportional to the expressibility of the specification language. However, a specification language should be simple enough that the feature designer is not overwhelmed. Also, a specification language should be formal enough to lend itself towards automated analysis.

Several standard languages already exist for the purpose of formally specifying the behavior of telecommunications systems. ESTELLE, LOTOS, SDL and Promela are examples of such languages. All of these languages are based on *labeled state-transition systems*, systems whose transitions between states are activated by the occurrence of the labeled events. These languages all have similar expressibility, and can model sequential behavior, choice, concurrency and non-determinism in an unambiguous way. The largest distinguishing factor between the languages is in the way data is represented. For example, SDL is a graphical language, whereas ESTELLE's syntax is similar to Pascal.

An advantage of using a standard specification language is that, in most cases, tools already exist for analyzing and composing such specifications. For example, Faci and Logrippo [1] of the University of Ottawa use LOTOS to specify features in their model, and use a verification tool called the Interactive System for LOTOS Applications (ISLA) in order to verify the features. Similarly, Combes and Picken [5] use the GEODE/FV verification tool to verify their SDL specifications. Lin and Lin [9] of Bellcore use Promela as their specification language, and verify their specifications with the SPIN verification tool. In fact, they have built an automated verification environment called WHEEL which operates on top of SPIN.

Although the use of pre-defined specification languages allows for powerful verification methods using existing tools, it limits the choice of model. Therefore, some approaches to the feature-interaction problem define their own specification language. For example, Ohta and Harada [10] of ATR Communication Systems Research Laboratories have developed a language called STR (State Transition Rule) for the purpose of describing features in a system. In STR, a state primitive can either define the behavior of a terminal (e.g. idle(P)) or a set of behaviors among multiple terminals (e.g. ringing(P,Q)). STR allows each feature to be modeled as a set of state transition rules of the form $S, E \rightarrow N$, where S and N are sets of state primitives and E is an event. The rule specifies that if E occurs and $S \subseteq G$, where G represents the set of state primitives that are currently true in the system, then N is to be applied to the global state G (by removing S from G and adding N). The specification of the system is then merely the union of the state transition rules describing each of the features in the system and a set of state primitives representing the initial global state.

2.1.2 Composition of Specifications

One major obstacle in the analysis of feature interactions is that the number of interactions (i.e., the number of combinations of features that should be examined) increases exponentially with the number of features in the system [2]. Kimbler *et al.* [8] of PIEN, A Eurescom[1] project, attempt to tackle this problem by eliminating all combinations of features that can never lead to any interaction, and only analyzing remaining combinations. This is achieved by categorizing all features in a system, where two features are in the same category if they have similar functionality. If two categories A and B have nothing in common with each other, then a feature in A cannot interact with a feature in B. This drastically reduces the number of combinations to be analyzed. Kimbler *et al.* reduce the problem set even further by analyzing the roles of the services to which the features belong. A given combination of features $F1$ and $F2$ can cause an interaction if and only if there exist two services $S1$ and $S2$ such that $F1$ belongs to $S1$, $F2$ belongs to $S2$, and both $S1$ and $S2$ can be active at the same time. As a telephony example, a *Hold* feature could not possibly interact with a *911* feature, since these two services are defined to be mutually exlusive.

[1]Eurescom is a joint research initiative of several European public network operators.

Lin and Lin [9] attempt to reduce the complexity of composed telephony specifications by performing incremental composition. This not only reduces the complexity of each stage of composition, but allows packages of precomposed subsystems to be used repeatedly without the need for recomposition. Lin and Lin incrementally compose their Promela specifications using a building block approach. This involves three levels of composition: the composition of Basic Feature Contexts (BFCs) from Basic Call Models (BCMs), the composition of Feature Contexts (FCs) from BFCs, and the composition of a system of features from BFCs and feature specifications. Five BCMs have been identified: originating user's behavior (ORIG), terminating user's behavior (TERM), originating basic call model (OBCM), terminating basic call model (TBCM) and system environment (SYS). These BCMs have been composed into three common BFCs: originating, terminating and two-party[2].

To illustrate this incremental approach, the composition of the originating and terminating BFCs are shown in Figure 2.1, the FCs of features *Call Waiting (incoming call)* and *Call Forwarding on Busy* are shown in Figure 2.2, and the compositions of these two features with their FCs are shown in Figure 2.3. The arrows in these figures represent interaction between the corresponding entities. The dashed rectangle encapsulating the two TERMs in the *Call Waiting* FC (Figures 2.2 and 2.3) indicates that both TERMS are modeling the same user and are therefore merged. Similarly, the double rectangle encapsulating the SYS BCM in the *Call Forwarding on Busy* FC (Figures 2.2 and 2.3) indicates the merging of two separate SYS BCMs (from the two previously unrelated FCs).

The BFCs in Lin and Lin's approach are specified in terms of *options*. Through options, the state space of a BFC is decomposed into a number of disjoint subspaces, providing finer control over the complexity of the model. For example, the terminating

[2]The two-party BFC is merely a combination of the originating and terminating BFCs.

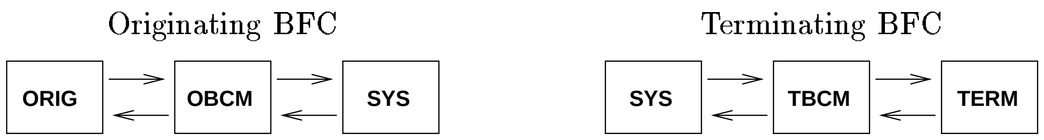

Figure 2.1: Originating and Terminating Basic Feature Contexts (Lin and Lin)

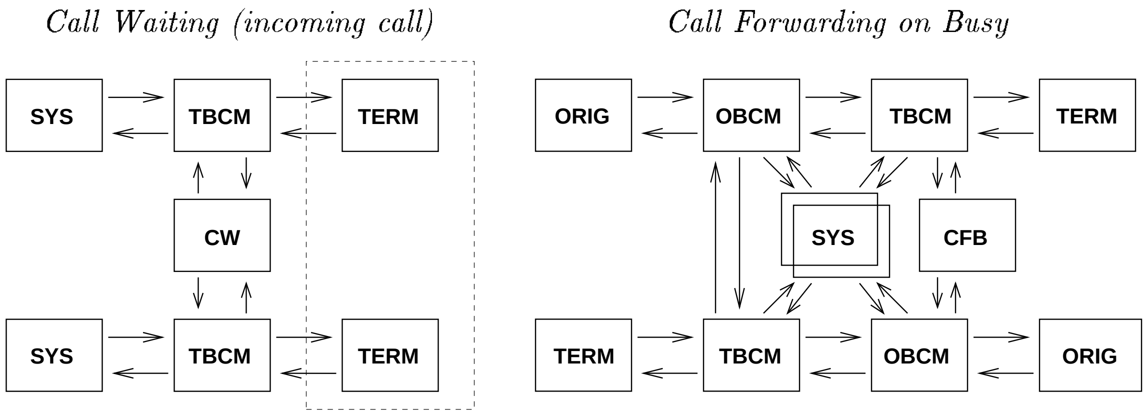

Figure 2.2: Feature Contexts for features *Call Waiting* and *Call Forwarding on Busy*, composed from the Basic Feature Contexts of Figure 2.1 (Lin and Lin)

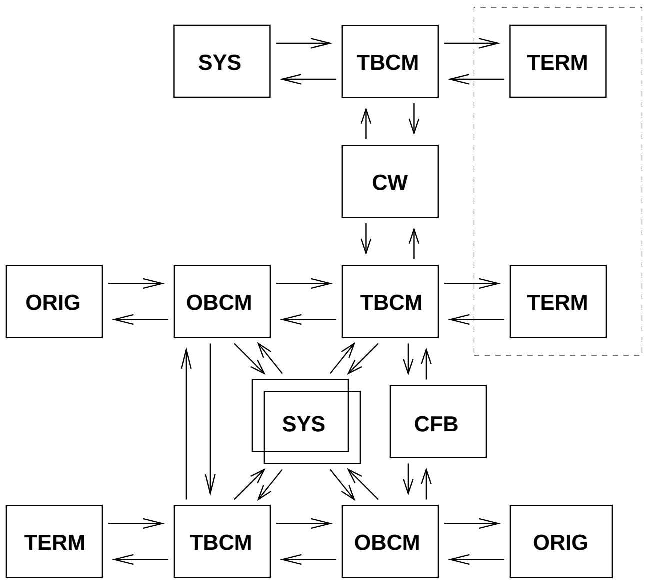

Figure 2.3: Composition of *Call Waiting* and *Call Forwarding on Busy* (Lin and Lin)

BFC has nine separate options which can be turned on or off, controlling which portions of the state space should be included in the composition. If the user does not wish to model cases where the line is busy, for example, he/she can disable the `LINE_BUSY` option, reducing the size of the composed specification.

2.1.3 Detection and Resolution of Feature Interactions

Once feature specifications are composed into one large, composite specification, the composite specification can be analyzed, and interactions between the features within can be detected. For each specification language, it is necessary to define what constitutes a feature interaction in specifications written in that language. Combes and Picken [5] define the term *feature interaction* in the following way: Let F_1, F_2, $\ldots$, F_n be feature specifications (specified in SDL), let S be a software system (also specified in SDL), and let $S \oplus F_i$ be the network obtained by adding feature F_i to S. Also, let P_1, P_2, $\ldots$, P_n be formulae expressing feature requirements in a suitable property language. A feature interaction occurs when $S \oplus F_i$ satisfies P_i, $1 \leq i \leq n$, but $S \oplus F_1 \oplus \ldots \oplus F_n$ does not satisfy $P_i \wedge \ldots \wedge P_n$. Although this method can be used to detect interactions between a number of interacting features, Combes and Picken usually only consider features pairwise under the assumption that most interactions involve only two features.

Lin and Lin [9] use the reachability-analysis tool SPIN to verify temporal logic assertions of Promela specifications in a manner similar to Combes and Picken's method. When undesired feature interactions are detected, a feature manager, through which all features must interact, is added to the system. The feature manager acts as an arbitrator between BCMs and features, and enforces a manually derived resolution scheme over all the features. Separation of concerns is poorly addressed with this method, as every combination of features requires its own version of the feature manager.

Ohta and Harada [10] use a different approach in their definition of a feature interaction. They identify five types of interactions that can occur at the feature specification design stage: deadlock, loop (a state for which there is no path back to the initial state), non-determinacy, transition to an abnormal state, and duplicated terminology. The first two types of interactions, deadlock and loop, are easily detected via standard finite-state-machine techniques. If these types of interactions exist in the system, there are logical problems with the finite-state machine which must be analyzed by hand.

STR, the language developed by Ohta and Harada, avoids certain types of nondeterminism by prioritizing transitions: if two rules $S_i, E_i \rightarrow N_i$ and $S_j, E_j \rightarrow N_j$ exist, where $E_i = E_j$ and both S_i and S_j are subsets of G with $S_j \subset S_i$, then N_i is applied prior to N_j. Non-determinacy exists when an applicable rule cannot be uniquely selected. Ohta and Harada present an algorithm which detects non-determinacy by examining the global set of rules in the system and searching for such situations. When such an interaction is detected, it can be resolved by asking the user to specify the priorities of competing rules. The rule that is given higher priority is then modified, such that the set of current state primitives S_i is replaced by the union of the sets of primitives S_i and S_j of both competing rules.

Transition to an abnormal state occurs when two features both have applicable rules, but there exists no relation between the rules' sets of current primitives. Only one rule will be applied, and the next state of the feature with the non-applied rule will not be reached. This can be detected by analyzing the changes to the global state resulting from the application of such rules and comparing them to the states expected by the individual features. Resolution of this interaction involves adding rules to the system whose current state is equivalent to the disjunction of the current states of the two competing rules.

This model provides a straight-forward framework for detecting and resolving feature interactions. One advantage of the model is that the individual features are described from the user's viewpoint and thus are easy to conceptualize. However, the model fails to address issues such as resource contention and data flow, both of which can be the source of feature interactions during the specification design stage. Also, since resolution of feature interactions involves additional rules that only exist when certain features are combined, separation of concerns is poorly addressed.

The availability of flexible tools for detecting feature interactions is as at least as important as the model itself. Boumezbeur and Logrippo [1] have developed three separate tools for analyzing their LOTOS-based specifications. The first tool performs step-by-step execution of the specifications which allows the user to choose the next action to take at point of execution. The second tool generates a symbolic execution tree of the system which explores all the possible paths of the system. The third composes the features in a system with respect to a user-supplied sequence of events and produces the execution path taken.

2.2 Reachability Graph Generation

Most automated validation methods are based on exhaustive reachability analysis, which focuses directly on reachable system states rather than indirectly on the transitions that connect them. A reachability analysis algorithm generates and inspects all of the states of a distributed system that are reachable from a given initial state. Due to synchronization between the system's components, the set of reachable states is often much smaller than the cross product of the states of the individual features in the system. Therefore, it is feasible to perform a full state space search on most systems of a reasonable size.[3] [7]

[3]For larger systems, partial search techniques become necessary, the simplest and most effective of which is based on selecting a random subset of successor states to follow at each state. [7]

Holzmann [7] describes a simple algorithm which performs reachability analysis based on a full state space search. It is given in Figure 2.4. This algorithm recursively steps through every reachable state in the system, checking each state for validity. An error is reported for every invalid state encountered.

```
start() {
        W = { initial_state }        // work set: to be analyzed
        A = { }                      // previously analyzed states
        analyze()
}

analyze() {
        if W is empty
                return
        else
                q = element from W
                add q to A
                if q is an error state
                        report_error()
                else
                        for each successor state s of q
                                if s is not in A or W
                                        add s to W
                                        analyze()
                delete q from W
}
```

Figure 2.4: Holzmann's reachability analysis algorithm

Let **W** be the working set of states to be analyzed. Holzmann [7] suggests retrieving the elements from **W** in a *first-in last-out* (i.e. stack) order, since the size of **W** would then be a function of the depth of the tree rather than the width which is likely to be much larger. Also, such a depth-first search would allow easy construction of an execution sequence of state transitions from the initial state at any point during the analysis.

Chapter 3

Model

3.1 Background

A large class of feature interactions is due to non-determinism: if two active features f and g react to 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 alters the control-flow of feature g. Alternatively, 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.

Before my studies at the University of Waterloo commenced, Ken Braithwaite and Joanne Atlee began investigating the problem of detecting such interactions in an automated fashion [3]. They adopted a *stack* architecture (or layered architecture) of services and features where each service is represented by one or more features in a stack. The basic service (modeled as a stand-alone feature) resides at the bottom of the stack, and the features are placed in a prioritized order above the basic service (see Figure 3.1). Features within a stack communicate with each other by passing around messages called

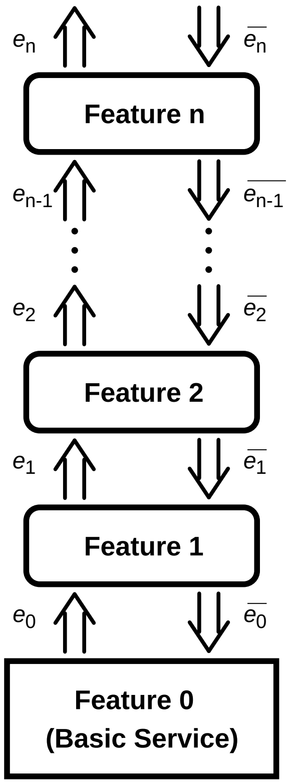

Figure 3.1: Feature stack architecture

tokens. Tokens representing 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 similarly progress upwards through the layers towards the agent. 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. This prioritization of features effectively resolves interactions due to non-determinism by explicitly specifying when a feature can operate on input and output data. Databases, operating systems, and control software [4, 6] can be modeled using stack architectures.

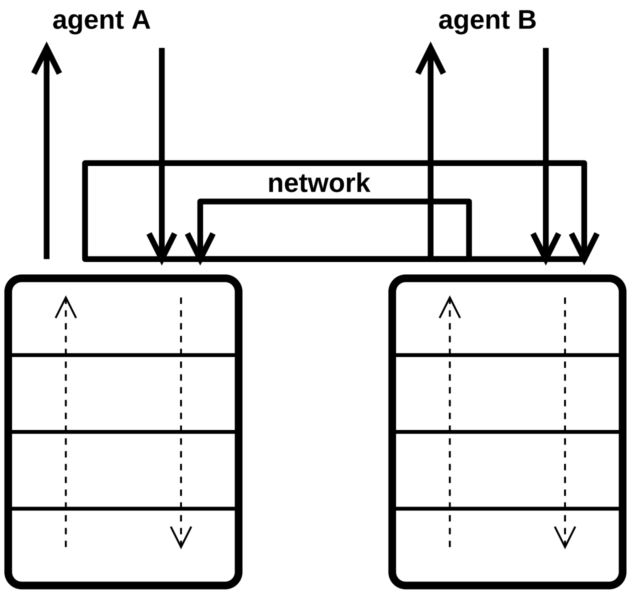

Figure 3.2: A call

Communication protocols and telephony systems [12] can also be modeled using stack architectures. However, due to the distributed nature of such systems, it is not sufficient to model them with a single feature stack; hence, the model also includes *calls*: two feature stacks (one for each agent) interacting via a communication link called a *network* (see Figure 3.2). Tokens entering a stack originate from either the stack's agent or the remote stack (via the network). Similarly, tokens leaving a stack are destined for either the agent or the remote stack, but not both. An agent does not directly receive events from the network. An event received from a network is passed down the feature stack towards the basic service, which in turn may attempt to communicate the event to the agent by passing it back up through the feature stack.

In order to address the issue of separation of concerns, a service which conceptually spans a call must be modeled as two separate features, one for each feature stack. Each feature operates in the context of a single side of a call, and communicates with the other feature of the service through the network. This greatly reduces the complexity of the specifications, and allows for incremental composition.

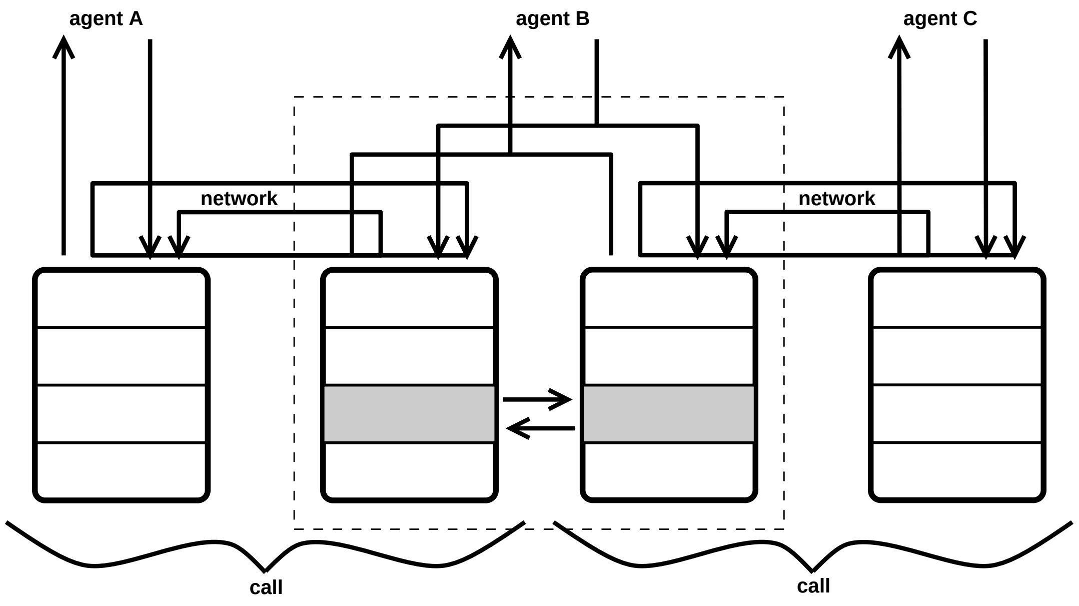

Figure 3.3: A call system

Certain telephony services (for example, *Three-Way Calling*) allow an agent to engage in two or more different telephone calls at the same time. Since a feature stack is only capable of representing one end of a single call, such services require the agent to communicate with two or more feature stacks at a time (one for each call the agent is engaged in). Services that involve more than one call in this manner are modeled as separate features, one operating in each of the agent's feature stacks affected by the service. Tokens from the agent are passed to each of the agent's stacks simultaneously (see agent B in Figure 3.3). The individual features comprising the service communicate via message passing, as illustrated by the shaded features in Figure 3.3. For example, an activated *Three-Way Calling* service consists of two features, one for the feature stack of the original call, and one for the feature stack of the new call to the third party. A set of two or more calls that have a common agent is referred to as a *call system.* When a single agent controls two or more feature stacks in this manner, those feature stacks are considered to be *parallel.*

3.2 Tabular Specification Notation

Ken Braithwaite and Joanne Atlee developed a tabular notation for specifying features in this model [3]. Although many of the details of this notation have changed from their original form due to my research, the basic structure remains the same. I will present the refined notation here.

A feature is specified in tabular form as one or more state-transition machines. Each state-transition machine table consists of a number of rows, each of which specifies a state transition, to be activated by the reception of some input event, from a current state to a new state. A state transition may produce output events and may request and/or release resources. Therefore, the basic structure of each row of such a table is as depicted in Figure 3.4. Note that the only difference in structure between tables of this form and standard state transition tables is the addition of a **Resources** field.

Tabular specifications for two basic telephony services exist: the *Originating Call*

State	**Input**	**Output**	**NewState**	**Resources**
current state	*input event*	*output events*	*new state*	*resource requests/releases*

Figure 3.4: The structure of a row in a tabular specification

Model (*OCM*) models the originating end of a call, and the *Terminating Call Model* (*TCM*) models the receiving end of a call. Many telephony features are designed to operate on top of either an *OCM* or a *TCM*. Because the two call models accept and output different data, a telephony feature that is designed to operate on top of either basic call model may need to specified by multiple state-transition machines, where each machine operates on top of one call model. Note, then, the following relationship between services, features and state-transition machines: A service is modeled as one or more features, while a feature is specified by one or more state-transition machines. Specifically, a service must be modeled as more than one feature when it either spans a network connecting two or more agents in a single call or when it spans two or more calls controlled by a single agent, whereas a feature may need to be specified by more than one state-transition machine if it is designed to operate on top of either basic call model.

Perhaps the best way to introduce the tabular notation is to provide an example. Table 3.1 contains a specification of the behavior of the *Call Waiting* feature for the incoming call that activates the feature. The single machine specifying this particular feature modifies a *TCM* since the incoming call is necessarily originated by another agent. Separate tables specifying the behavior of the feature for the original call (which modifies either an *OCM* or a *TCM*, depending on whether the original call was initiated or received by the agent) are not shown here (see the Appendix for these specifications).

In order to interpret a table such as the one in Table 3.1, one must understand the meanings of the different types of input and output events specified within. Figures 3.5

State	**Input**	**Output**	**NewState**	**Resources**
Null	$\Rightarrow_{TCM}$AuthTermination($\Downarrow_R$TerminationAttempt)		Waiting	
Waiting	HuntingFacility$\Rightarrow_{TCM}$PresentingCall ($\triangleright$**FacilityFound**)	$\ll$*forward*$\gg$	Null	
	HuntingFacility$\Rightarrow_{TCM}$Exception($\triangleright$**Busy**)	$\ll$*forward*$\gg$	HuntingFacility	+**bridge**
	HuntingFacility$\Rightarrow_{TCM}$Null($\Downarrow_R$CallCleared)	$\ll$*forward*$\gg$	Null	
	HuntingFacility$\Rightarrow_{TCM}$Null($\Downarrow_A$Disconnect)	$\ll$*forward*$\gg$	Null	
Hunting Facility	$\triangleright$**FacilityFound**	HuntingFacility$\Rightarrow_{TCM}$PresentingCall ($\triangleright$**FacilityFound**)	PresentingCall	
	$\triangleright$**Busy**	HuntingFacility$\Rightarrow_{TCM}$Exception($\triangleright$**Busy**)	Null	–**bridge**
	$\Rightarrow_{CWT}$Null	HuntingFacility$\Rightarrow_{TCM}$Null($\Downarrow_R$CallCleared)	Null	–**bridge**
	$\Downarrow_R$CallCleared	$\ll$*forward*$\gg$	Null	–**bridge**
	$\Downarrow_A$Disconnect	$\ll$*forward*$\gg$	Null	–**bridge**
Presenting Call	$\Uparrow_A$Alert	Beep$\Uparrow_A$	Alert	
	PresentingCall$\Rightarrow_{TCM}$Exception($\Uparrow_R$CallCleared)	$\ll$*forward*$\gg$	Null	–**bridge**
	$\Rightarrow_{CWT}$Null		Null	
	$\Downarrow_R$CallCleared	$\ll$*forward*$\gg$	Null	–**bridge**
Alert	$\Downarrow_A$SwitchCalls	Alerting$\Rightarrow_{TCM}$Active($\Downarrow_A$Connected)	Active	
	$\Downarrow_A$Connected	$\ll$*forward*$\gg$	Active	
	$\Rightarrow_{CWT}$Null		Null	
	$\Downarrow_R$CallCleared	$\ll$*forward*$\gg$	Null	–**bridge**
	Alerting$\Rightarrow_{TCM}$Exception($\triangleright$**CallRejected**)	$\ll$*forward*$\gg$	Null	–**bridge**
	Alerting$\Rightarrow_{TCM}$Exception($\triangleright$**RingingTimeout**)	$\ll$*forward*$\gg$	Null	–**bridge**
Active	$\Downarrow_A$SwitchCalls		HeldCall	
	ReleasePending$\Rightarrow_{TCM}$Null($\Downarrow_R$ReleaseTimeout)	$\ll$*forward*$\gg$	Null	–**bridge**
	$\Rightarrow_{CWT}$Null		Null	
	$\Downarrow_R$CallCleared	$\ll$*forward*$\gg$	Null	–**bridge**
HeldCall	$\Downarrow_A$SwitchCalls		Active	
	$\Downarrow_A$Disconnect		RingBack	–**bridge**
	$\Rightarrow_{CWT}$Null		Null	
	$\Downarrow_R$CallCleared	$\ll$*forward*$\gg$	Null	–**bridge**
	$\Downarrow_A*$		HeldCall	
	$\Uparrow_A*$		HeldCall	
Ringback	$\Downarrow_A$CallAnswered		Null	
	$\triangleright$**RingingTimeout**	$\Downarrow_A$Disconnect	Null	
	$\Downarrow_A*$		Ringback	
	$\Uparrow_A*$		Ringback	

Table 3.1: *Call Waiting* specification for new incoming (received) call

Input events:

$\Downarrow_A$token $\Downarrow_R$token	– A Token event – a token from the environment, either from the agent ($\Downarrow_A$) or from the connection's remote agent ($\Downarrow_R$) via the network, that is being sent towards the basic service machine.
$\Uparrow_A$token $\Uparrow_R$token	– A Token event – a token from the basic service machine that is being sent towards the environment, either to the agent ($\Uparrow_A$) or the connection's remote agent ($\Uparrow_R$) via the network.
$\Downarrow_A *$ $\Downarrow_R *$	– A Token event – matches an arbitrary token from the environment, either from the agent ($\Downarrow_A$) or from the connection's remote agent ($\Downarrow_R$). All other state transitions from the current state have priority over a state transition activated by receipt of an arbitrary token.
$\mathrm{S1}\Rightarrow_f\mathrm{S2}(e)$	– A StateChangeRequest event – notification that a lower-priority feature f (possibly the underlying call model) is requesting a state transition from S1 to S2, triggered by input event e. A feature can modify the behavior of a lower-priority feature by intercepting the feature's state-transition notification and imposing a different state transition on the feature.
$\Rightarrow_f\mathrm{S}(e)$	– An Activation event – notification that feature f (possibly the underlying call model) has just activated by transitioning from NULL to S due to the reception of event e.
$\Rightarrow_f\mathrm{S}$	– A Parallel event – a signal from another feature of the same service (operating on a parallel feature stack) indicating that the other feature has transitioned into state S. Such signals are only visible to features modeling the same service.
$\triangleright$event	– An Internal event – a signal indicating the termination of internal computation. This signal is only visible to the machine performing the internal computation.

Figure 3.5: Notation for input events used in tabular specifications

Output events:

$\mathsf{token}\Downarrow_A$ $\mathsf{token}\Downarrow_R$	– A Token event – a token sent from the feature towards the basic-service machine. The output token will be interpreted as being from the environment, either from the agent ($\Downarrow_A$) or the connection's remote agent ($\Downarrow_R$). If the output token is a modification of a previously input token, then the output event will be primed (e.g., $\mathsf{token'}\Downarrow_A$).
$\mathsf{token}\Uparrow_A$ $\mathsf{token}\Uparrow_R$	– A Token event – a token sent from the feature towards the environment, either towards the agent ($\Uparrow_A$) or the connection's remote agent ($\Uparrow_R$). The output token will be interpreted as being from the basic-service machine. If the output token is a modification of a previously input token, then the output event will be primed (e.g., $\mathsf{token'}\Uparrow_A$).
$\ll$*forward*$\gg$	– The input token or state-transition notification is forwarded to the next level machine without alteration.
$S1 \Rightarrow_f S2(e)$	– A $\mathsf{StateChangeRequest}$ event – the feature imposes a new state transition from S1 to S2, triggered by input event e, in a lower-priority feature f (possibly the underlying call model).
$\Rightarrow_f S$	– A $\mathsf{Parallel}$ event – a signal sent to other features of the same service (operating on parallel feature stacks) indicating that the feature has transitioned into state S. This may be implicit.
NewCall(*OCM***)**	– A $\mathsf{NewCall}$ event – instantiation of a new call stack, based on an *Originating Call Model* (*OCM*). There is no **NewCall(***TCM***)** event, as a feature cannot instantiate a *TCM* call stack if there is no reciprocal originating end of the call.

Figure 3.6: Notation for output events used in tabular specifications

and 3.6 list the types of events that may appear in a tabular specification. They are each placed into one of six categories:

Token events are events which signal either the transmission or reception of a data item (called a *token*). Some tokens carry no information other than their name, some contain a single datum, and others are compound, having other tokens as components. Tokens may be passed between neighboring features on the same feature stack, between a feature stack and its associated agent, or between two feature stacks that are linked together in a telephone call. Tokens always have a name, a direction (up or down), a source and a destination. Tokens moving down the stack are destined for the basic service; their source is designated as either the stack's agent ($\Downarrow_A$) or the feature stack at the remote end of the call ($\Downarrow_R$). For upward-moving tokens, the annotation ($\Uparrow_A$ or $\Uparrow_R$) specifies the destination of the token, while the source is assumed to be the basic service.

Internal events are events which indicate an internal decision made by a feature. For example, when *Call Waiting* (*CWT*) is in state HUNTINGFACILITY, it must determine whether or not hardware facilities exist to establish a connection (See Figure 3.1). An Internal event can not be specified as an output event in the feature's transition relation; features are not allowed to know about and react to each other's Internal events.

Parallel events are signals passed between multiple features modeling a single-agent service which spans multiple calls. It is through this type of event that a such features can synchronize with one another.

StateChangeRequest events are events which indicate the desire of a lower-level feature to change states. When a feature intercepts this event, it can either pass it on unchanged; pass on a modified StateChangeRequest event (which must specify a valid state transition rule in the target feature); or pass on nothing, thereby disallowing the state transition. These events are discussed in more detail in Section 3.3.3.

Activation events are events which indicate that a feature has transitioned from the NULL state and therefore has activated. Events of this type exist so that one feature can activate based on the activation of another, and thus are only legal as the input of activation rules (rules which specify a transition from NULL). An appropriate Activation event is implicitly generated every time a feature activates.

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. NewCall events are only valid as output events in a feature's transition relation.

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

A state transition may affect resource allocation. Notation +**resource** specifies a request for an instance of **resource**, while −**resource** specifies the release of an instance of **resource**. As we will see in Section 5.3.4, modelling such information provides a powerful mechanism for detecting resource contention interactions.

Although this thesis introduces the proposed tabular specification mechanism in a more-or-less informal manner, we will for a moment consider the mechanism in a more formal light. 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-transition requests made by the call models and features; we define inR and $outR$ to 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 currently allocated resources. Then formally, a tabular specification $\mathcal{T}$ represents the feature's transition relation:[1]

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

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.

3.3 Mechanisms of Operation

A feature is not a stand-alone entity; it must always be examined in the context of the system in which it executes. For this reason, it is important to understand the mechanisms through which features communicate with one another. Before I began my studies, Ken Braithwaite and Joanne Atlee had only informally described such mechanisms [3]. A large part of my thesis work involved refining and formally specifying these mechanisms. In particular, it was necessary to define the mechanisms so that the composition of features behaves deterministically.

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

3.3.1 Activation of Features

A feature can only affect the basic service if it is *active*. Features to which the agent has subscribed but which have not been activated are not on the feature stack; inactive features are 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. A feature is considered inactive if it is in the NULL state. When the feature activator detects an event in the system that will cause one or more of the inactive features to transition from its NULL state, it activates those features by passing the event to them and placing them in the feature stack. 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 some high-priority features (such as telephony feature *911*) 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 control of the feature activator, which may activate it again at a later time. It is necessary to place features in the stack only after they become active and to remove them once they deactivate in order to allow the ordering of features in a stack to change due to the deactivation and reactivation of features. The basic service (i.e., basic call model) 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 (i.e., reaches the basic call model which then passes it on to the feature activator) 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 attempts to transition to a new state and feature F has a transition from its NULL state on the corresponding StateTransitionRequest event (See Section 3.3.3).
- if another feature activates and feature F has a transition from its NULL state on the corresponding Activation event.
- if another feature of the same service in another feature stack controlled by the same agent sends a Parallel event feature F has a transition from its NULL state on that event.

Before a feature can activate, its activation must be approved by all currently-active features. Section 3.3.3 describes this in more detail.

3.3.2 Token Queues

Each feature is provided with two input queues: one which accepts downward-moving tokens from higher-level features, and one which accepts upward-moving tokens from lower-level features. A feature may output zero, one or multiple tokens during each state transition. As a consequence, there may be many tokens traveling through the feature stack at any one time. In order to impose a total ordering on the resolution of tokens, a token is only accepted from a queue if all queues of lower-level features in the feature stack are empty. Also, a token is only accepted from the upper queue of a feature (i.e. the queue which accepts tokens passed down from higher-level features) if the the lower queue of that feature is empty. One important implication of these rules is that input tokens from the environment (which enter the top queue of the highest-level feature) are not accepted until all other queues in the feature stack are empty.

When a feature receives a token from one of its queues but has no rule specifying a state transition from its current state on the reception of that token, the token is passed to the next feature in the stack, or to the environment (the agent in the case of upward-moving tokens and the feature activator in the case of downward-moving tokens) if there is no next feature. In effect, this implies that all feature specifications implicitly contain the following state transition rules for each state x, unless overridden explicitly:

State	**Input**	**Output**	**NewState**
x	$\Downarrow_A *$	$\ll$*forward*$\gg$	x
	$\Downarrow_R *$	$\ll$*forward*$\gg$	x
	$\Uparrow_A *$	$\ll$*forward*$\gg$	x
	$\Uparrow_R *$	$\ll$*forward*$\gg$	x

Table 3.2: Implicit rules in feature specifications

3.3.3 State Transition Verification

Because higher-level features have priority over a feature at level x, the feature at level x must 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 all other events (see Section 3.3.5) and are resolved immediately. When a feature receives a StateChangeRequest event, it may either forward the event unchanged to the next lower-level feature, it may output a modified StateChangeRequest, or output nothing, in effect disallowing the state transition[2]. As with tokens, if a feature has no rule for specifying a state transition from its current state on receipt of a particular StateChangeRequest event, the event is passed to the next feature in the stack.

StateChangeRequest events are annotated with a boolean flag specifying whether or not they have been modified. When a StateChangeRequest event is generated due to a feature wishing to make a state transition, this flag is initially false. If a feature intercepts this event and passes it on without modifying it, the flag remains false. However, if a feature modifies the event to impose a different state transition on the requesting feature, the modified event is marked as such. If a StateChangeRequest event for a lower-priority feature is generated by a higher-priority feature, then the event is automatically marked as modified[3].

When a StateChangeRequest reaches the destination feature (usually the one that originally generated the request), the received StateChangeRequest event is checked to see if it has been modified. If it has, the modified StateChangeRequest event must be approved (or reapproved) by all higher-level features[4]. This mechanism insures that a

[2]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, and the recursion will eventually terminate.

[3]In practice, most imposed StateChangeRequests are modifications to outstanding StateChangeRequests.

[4]It is possible in the current model for an infinite loop to occur if two features try to modify a StateChangeRequest of a common lower-priority feature at the same time. Each feature may reverse the modifications of the other. At present the algorithms I have developed don't detect such infinite loops. However, they could be expanded to maintain enough information to detect these types of interactions. For example, if a history of pending StateChangeRequests were kept, duplicate entries would indicate such loops.

feature will never make a state transition that a higher-priority feature is specified to disallow.

When an unmodified StateChangeRequest event reaches the destination feature, the feature activator is queried to see if an inactive feature wishes 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 where the newly activated feature is to be inserted into the feature stack (but its location must be above the feature whose StateChangeRequest activated the new feature). Once a feature is activated by a StateChangeRequest event, it becomes part of the state transition verification cycle and must forward the StateChangeRequest event onwards if it wishes to activate unobtrusively. 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. Note that this mechanism allows more than one feature to activate on a single StateChangeRequest event. If a feature unobtrusively activates on a StateChangeRequest event, the event will once more percolate down through the feature stack and reach the destination feature, at which point the feature activator will be queried again.

3.3.4 Triggering of Internal Events

When an Internal event is triggered within a feature, it typically results in a transition to another state. However, since higher-priority features have the ability to deny state transitions, the triggering of an Internal event may not result in a transition to another state. Once an Internal event has been triggered, no other Internal events can be triggered within that feature until a state transition has been made. For example, if the *TCM* in state HUNTINGFACILITY determines that the necessary hardware facilities do not exist by triggering the internal $\triangleright$**Busy** event, but a higher-priority feature (e.g., *CWT*) disallows the state transition into EXCEPTION, then *TCM* stays in state HUNTINGFACILITY. However, internal events $\triangleright$**FacilityFound** and $\triangleright$**Busy** effectively become disabled because the internal processing that results in either of those events is done. Transition out of state HUNTINGFACILITY is now limited to receiving a Token event, receiving a Parellel event, or receiving a StateChangeRequest event (e.g. (HUNTINGFACILITY$\Rightarrow_{TCM}$PRESENTINGCALL($\triangleright$**FacilityFound**)).

3.3.5 Priorities of Events

To ensure a deterministic execution of features in a feature stack, call or call system, the following priorities among events are imposed:

- All StateChangeRequest and Activation events are resolved first.
- Tokens waiting in queues are then processed. A feature will only accept a token from a queue when no lower-level features have tokens in their queues. If a feature has tokens in both of its input queues, it will accept upward-moving tokens from the bottom queue first.
- Next, network tokens (from remote feature stacks) are resolved.

- Signals between multiple features modeling a single-agent service which spans multiple calls (represented by Parallel events) are then resolved.
- Finally, internal decisions (represented by Internal events) and tokens from the agent are accepted.

Section 4.2 discusses the relevance of these priorities to the different stages of composition during reachability analysis.

Chapter 4

Composition of Specifications

In order to detect interactions between features, it is necessary to compose their specifications together. Since feature specifications have finite structure and accept finite sets of input, it is possible to generate finite reachability graphs of feature stacks, calls and call systems (See Section 2.2). Analysis of information flow during the generation of a reachability graph can reveal a large class of feature interactions in the system being composed.

4.1 ASCII Representation of Specifications

It is necessary to transform the tabular specifications of features into computer-readable input so that they may be used as input to the tools described in Section 4.3. Transformation in the other direction is also desirable, since the composite specifications generated are expressed in the same language as the component features. Therefore, a one-to-one mapping between tabular specifications and ASCII representations of the specifications has been defined.

4.1.1 Structure

Each state transition machine of which a feature is composed is specified in a separate ASCII file. In order to provide coherent feedback to the user of the tool, each specification is given a distinct, descriptive name. This name must appear as the first non-empty line in the file. The rest of the file is used for specifying the state transition rules of the feature, one rule per non-empty line. An ASCII-specified state-transition rule is of the form:

state *input* *output-list* *new-state* [*resource-allocation-list*]

As the brackets indicate, the *resource-allocation-list* field is optional. If it is omitted, an empy list is assumed. Any amount of whitespace can be used to separate the elements. The '`#`' character is used to denote a comment; any text appearing after this character to the end of the line is ignored by the tool performing the analysis.

Figure 4.1 shows the ASCII representation of the feature *CWT* for an incoming call (shown in tabular form in Table 3.1). The specifications of feature stacks, calls and call systems that are generated by the tools are also expressed in this manner, albeit not formatted and indented for human readers.

The following is a description of the syntax of such specifications.

4.1.2 Hierarchical Names for Specifications and States

The name of a feature specification can consist of any number of printable non-whitespace characters except for '`[`', '`]`' and '`/`'. For example, we use names '`3WC`' and '`CWT`' for features *Three-Way Calling* and *Call Waiting*, respectively. Names of higher-order specifications such as feature stacks, calls and call systems retain the hierarchical structure of

```
CWT

# State         Input                                                    Output                             New State       Resources
# -----         -----                                                    ------                             ---------       ---------

Null            a:TCM:AuthTermination:t:TerminationAttempt:d:r           ()                                 Waiting         ()

Waiting         s:HuntingFacility:TCM:PresentingCall:i:TCM:FacilityFound forward                            Null            ()
Waiting         s:HuntingFacility:TCM:Exception:i:TCM:Busy               ()                                 HuntingFacility +bridge
Waiting         s:HuntingFacility:TCM:Null:t:CallCleared:d:r             forward                            Null            ()
Waiting         s:HuntingFacility:TCM:Null:t:Disconnect:d:a              forward                            Null            ()

HuntingFacility i:CWT:FacilityFound                s:HuntingFacility:TCM:PresentingCall:i:TCM:FacilityFound PresentingCall  ()
HuntingFacility i:CWT:Busy                         s:HuntingFacility:TCM:Exception:i:TCM:Busy               Null            -bridge
HuntingFacility p:CWT:Null                         s:HuntingFacility:TCM:Null:t:CallCleared:d:r             Null            -bridge
HuntingFacility t:CallCleared:d:r                                        forward                            Null            -bridge
HuntingFacility t:Disconnect:d:a                                         forward                            Null            -bridge

PresentingCall  token:Alert:u:a                                          token:Beep:u:a                     Alert           ()
PresentingCall  s:PresentingCall:TCM:Exception:t:CallCleared:u:r         forward                            Null            -bridge
PresentingCall  parallel:CWT:Null                                        ()                                 Null            ()
PresentingCall  token:CallCleared:d:r                                    forward                            Null            -bridge

Alert           token:SwitchCalls:d:a                                 s:Alerting:TCM:Active:t:Connected:d:a Active          ()
Alert           token:Connected:d:a                                      forward                            Active          ()
Alert           parallel:CWT:Null                                        ()                                 Null            ()
Alert           token:CallCleared:d:r                                    forward                            Null            -bridge
Alert           s:Alerting:TCM:Exception:i:TCM:CallRejected              forward                            Null            -bridge
Alert           s:Alerting:TCM:Exception:i:TCM:RingingTimeout            forward                            Null            -bridge

Active          token:SwitchCalls:d:a                                    ()                                 HeldCall        ()
Active          s:ReleasePending:TCM:Null:t:ReleaseTimeout:d:r           forward                            Null            -bridge
Active          parallel:CWT:Null                                        ()                                 Null            ()
Active          token:CallCleared:d:r                                    forward                            Null            -bridge

HeldCall        token:SwitchCalls:d:a                                    ()                                 Active          ()
HeldCall        token:Disconnect:d:a                                     ()                                 RingBack        -bridge
HeldCall        parallel:CWT:Null                                        ()                                 Null            ()
HeldCall        token:CallCleared:d:r                                    forward                            Null            -bridge
HeldCall        token:*:d:a                                              ()                                 HeldCall        ()
HeldCall        token:*:u:a                                              ()                                 HeldCall        ()

RingBack        token:CallAnswered:d:a                                   ()                                 Null            ()
RingBack        internal:CWT:RingingTimeout                              token:Disconnect:d:a               Null            ()
RingBack        token:*:d:a                                              ()                                 RingBack        ()
RingBack        token:*:u:a                                              ()                                 RingBack        ()
```

Figure 4.1: ASCII representation of *Call Waiting* specification for new incoming (received) call

the specifications: characters '`[`' and '`]`' are used to indicate levels of composition, and character '`/`' is used to separate the names of component specifications at the same level. For example, the name '`[[A/B/C]/[D/E]]`' represents a call specification containing the feature stacks '`[A/B/C]`' and '`[D/E]`'. At the feature stack level, the names of the features within are ordered with the name of the top (highest priority) feature first, followed by the name of the feature with the next-to-highest priority, and so on.

The names given to the states must also have the same hierarchical structure, where each component state name represents the state of the corresponding entity in the name of the specification. For example, if A $\in A$, B $\in B$, C $\in C$, D $\in D$, and E $\in E$, then [[A/B/C]/[D/E]] is a possible state in specification *[[A/B/C]/[D/E]]*.

4.1.3 Events

For each type of event in the tabular specification language, there exists a corresponding ASCII representation. Table 4.1 shows this correspondence.

The *output* field of a state transition rule must be specified as a list of zero or more events separated by commas and delimited by parentheses. No whitespace is permitted in this list. If exactly one event exists in a list, the parentheses may be omitted.

4.1.4 Resource Allocation

The *resource-allocation-list* field of an ASCII specification, if present, must contain a comma-separated list of resource-allocation directives delimited by parentheses. A resource-allocation directive is of the form '`+resource`' or '`-resource`', representing the request or release of a resource, respectively. No whitespace is permitted in the resource-allocation list. If zero or one directives exist in the list, then the parentheses may be

Tabular Representation	**ASCII Representation**	**Abbreviated Form**
$\Downarrow_A$x, x$\Downarrow_A$	`token:x:down:agent`	`t:x:d:a`
$\Downarrow_R$x, x$\Downarrow_R$	`token:x:down:remote_stack`	`t:x:d:r`
$\Uparrow_A$x, x$\Uparrow_A$	`token:x:up:agent`	`t:x:u:a`
$\Uparrow_R$x, x$\Uparrow_R$	`token:x:up:remote_stack`	`t:x:u:r`
$\Downarrow_A *$	`token:*:down:agent`	`t:*:d:a`
$\Downarrow_R *$	`token:*:down:remote_stack`	`t:*:d:r`
$\Uparrow_A *$	`token:*:up:agent`	`t:*:u:a`
$\Uparrow_R *$	`token:*:up:remote_stack`	`t:*:u:r`
S1$\Rightarrow_f$S2(e)	`state_transition_request:S1:f:S2:`e	`s:S1:f:S2:`e
$\Rightarrow_f$S(e)	`activation:f:S:`e	`a:f:S:`e
$\Rightarrow_f$S	`parallel:f:S`	`p:f:S`
$\triangleright$**event**	`internal_signal:event`	`i:event`
NewCall(OCM)	`newcall:OCM`	`n:OCM`
$\ll$*forward*$\gg$	`forward`	`f`

Table 4.1: ASCII representations of events

omitted.

4.2 Composite Specifications

When feature specifications are composed to form the specification of a feature stack, all StateChangeRequest events, Activation events and queued Token events are resolved, and therefore the resulting specification should be void of such events. In the same manner, composing the specifications of two feature stacks to form a call specification will resolve all tokens that are passed between the two feature stacks. At this stage, an event trace exists for each of the two agents. In order to distinguish between the two traces, Token events in call specifications are annotated with a number uniquely identifying one of the two agents.

During the final stage of composition of telephony features, when two or more call specifications are composed to form the specification of a call system, all Parallel events are resolved, leaving only Token events (to and from the agents) and Internal events in the specification of the system.

In order to prevent the reachability graph generator from cycling, a special state called DONE is used in place of NULL once a feature deactivates.

4.3 Reachability Graph Generator

In the course of my thesis work, I have developed several tools for detecting feature interactions through reachability analysis of composed specifications. The composition technique employed by these tools is incremental in nature. Section 4.3.1 describes the framework upon which the tools were built, Sections 4.3.2 through 4.3.4 discuss the algorithms that perform the reachability analysis within this framework, and Section 4.3.5 introduces the tools themselves.

4.3.1 Object-Oriented Framework

In order to ensure a consistent framework upon which all of the reachability analysis tools can be based, as well as a backbone of code which can be easily expanded in the future to provide extended reachability-analysis services, I spent considerable time designing and coding a set of C++ classes to model the entities involved. Among the twenty-nine classes defined, the classes depicted in Figure 4.2 are the most significant.

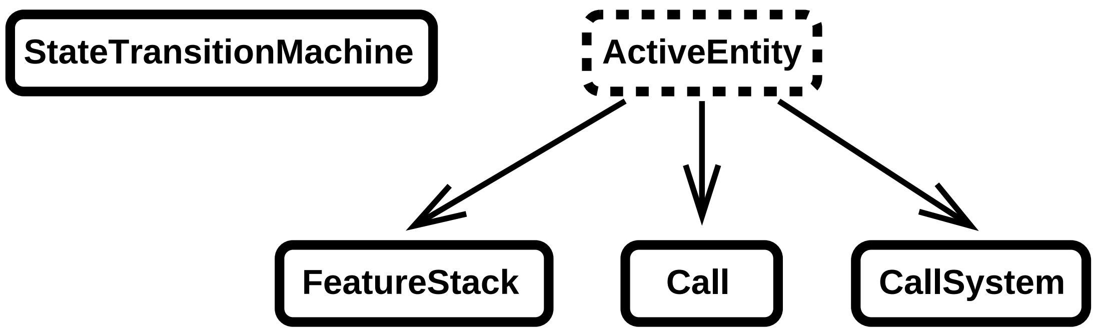

Figure 4.2: C++ Classes

Objects of the StateTransitionMachine class represent tabular specifications of features, feature stacks, calls, and call systems. A StateTransitionMachine object contains a set of state transition rules, and can respond to queries regarding these rules. It also has the ability to read and write itself in ASCII form (see Section 4.1), and to perform simple self-integrity checks. For the sake of brevity, StateTransitionMachines will be referred to as STMs for the remainder of this thesis.

ActiveEntity is an abstract class which is used to model the reachability graphs of composite entities at the various levels of composition, i.e., feature stacks, calls and call systems. It defines the interface to the mechanics necessary to simulate the behavior of such entities. Mechanisms which are common to all such entities are defined within this class, while entity-specific mechanisms are left undefined and must be defined by derived classes used to model specific types of entities.

When an ActiveEntity-derived object is instantiated, it is given the state transition machines of the entities of which it is composed. For example, a FeatureStack is given the list of STMs for features in the stack, while a Call is given the pair of STMs representing the two ends of the call (i.e., two feature stacks). The ActiveEntity then behaves as the composition of the entities defined by these machines. In order to do this, it must keep track of its own state, which consists of the current state of each of its component entities, the set of currently used resources, and all pending events. When an ActiveEntity is first instantiated, the state of each component entity is NULL, the set of currently used resources is empty, and there are no pending events in the system.

The most significant method of an ActiveEntity is process_event(), which accepts an event as an argument and causes the entity to behave as if the event occurred. process_event() will not return until another *stable state* is reached. A stable state is a state in which there exists no pending events in the system, i.e. when there are no unresolved tokens, parallel events or state transition requests.

Since the mechanics of feature stacks, calls and call systems differ significantly, process_event() is defined separately for each of these entity types. Section 4.3.2 describes the different incarnations of this method as they apply to each type of entity.

Two other ActiveEntity methods of importance are compose() and simulate(). The former generates a composite state transition machine representing the entity, while the latter provides the user with an interactive interface, allowing him/her to interactively explore the behavior of the entity. The algorithms for these methods are described in Sections 4.3.3 and 4.3.4.

4.3.2 Algorithm: process_event()

FeatureStack::process_event() is the most complex of all the algorithms in the system, since at this stage of composition the priorities of the features must be taken into account in order to avoid non-determinism in the system. This algorithm implements priority rules for resolving multiple outstanding tokens waiting in queues. It also implements a recursive proceesure for keeping track of pending StateChangeRequest events, as discussed in Section 3.3.3.

Call::process_event(), on the other hand, is relatively simple. At this stage of composition all intermediate states involving pending StateChangeRequests and queued tokens have already been resolved and are therefore not considered. The only events that Call::process_event() accepts are Parallel events, Internal events and Token events (from the remote agent). The event is passed to the appropriate feature stack, which will presumably change state upon receipt of the event. All output generated by this state transition is considered output of the call with the exception of Token events destined for the other feature stack via the network. These events are passed to the other stack, one at a time, and the process continues until there are no pending tokens flowing through the network.

The necessary mechanics of CallSystem::process_event() are not fully understood yet, and so this method has not been implemented.

Since the process_event() methods are responsible for following and controlling the flow of events in a system, the mechanisms for detecting feature interactions lie within these methods. Due to the incremental nature of compositions in the model, each incarnation of process_event() is only capable of detecting a subset of the class of feature interactions described in Section 5.3. However, together all such interactions are detected.

4.3.3 Algorithm: compose()

The purpose of ActiveEntity::compose() is twofold: it performs a reachability analysis of the composite specification by exploring all reachable composite states while at the same time constructing a state transition machine describing the behavior of the composed entity. It differs from the reachability analysis algorithm suggested by Holzmann [7] (see Section 2.2) in that it is iterative. This desision was made mainly for efficiency reasons; the reachability coverage is just as complete. Also, compose() performs a breadth-first search rather than a depth-first by using a queue rather than a stack to store the working set of states. This has the advantage that it finds the shortest error sequences first.

The algorithm is listed in Figure 4.3. **Q** represents the working set of states to be analyzed. In the initial state, all of the composite entities are in the NULL state and the set of currently used resources is empty. The initial state is placed in **Q**. Since there are only a finite number of states in each of the composite entities, there are a finite number of states in the composed system and this algorithm is guaranteed to halt (assuming a finite supply of resources).

```
$\mathbf{Q}$ = { initial_state }      // queue of states to be analyzed
$\mathbf{V}$ = { }                    // set of visited states
$\mathbf{C}$ = { }                    // composed state transition machine
while $\mathbf{Q}$ nonempty
    $\mathbf{s}$ = $\mathbf{Q}$.dequeue()
    for each accepted input $\mathbf{i}$ in state $\mathbf{s}$
        place system in state $\mathbf{s}$
        $\mathbf{s}'$ = process_event($\mathbf{i}$)
        if $\mathbf{s}'$ is not in $\mathbf{V}$
            add $\mathbf{s}'$ to $\mathbf{V}$
            $\mathbf{Q}$.enqueue($\mathbf{s}'$)
    if rule $name(\mathbf{s}) \xrightarrow{\mathbf{i}} name(\mathbf{s}')$ is not in $\mathbf{C}$
        add rule $name(\mathbf{s}) \xrightarrow{\mathbf{i}} name(\mathbf{s}')$ to $\mathbf{C}$
return $\mathbf{C}$
```

Figure 4.3: compose() algorithm

The function *name(*s*)* generates a state name consisting of the names of the states of each of the composite entities in *s*. This state name does not contain any information about the current set of resources. Therefore there is a many-to-one mapping between the states of the system and the states of the resulting specification[1]. Let **C** be the set of all state transition rules describing the behavior of the composed state transition machine. If a similar transition (having the same source state, desitination state and input) has not been encountered, the transition is added to C.

This algorithm is common for each of the stages of composition, and therefore is only implemented once in the base class. As mentioned in Section 4.3.2, the detection of feature interactions and the actual mechanics of operation at each stage are the responsibility of the individual process_event() methods, and not of the compose() algorithm itself.

Note that since compositions in our model are performed incrementally, and since intermediate (non-stable) system states are resolved at each stage of composition, the resulting state space search at each stage is reduced. For example, consider the following features (partially specified):

Feature F

State	**Input**	**Output**	**NewState**
A	$\Downarrow_A t_1$	$t_2\Uparrow_A$, $t_3\Downarrow_A$	B
B	$\Uparrow_A t_4$	$t_6\Uparrow_A$	C

Feature G

State	**Input**	**Output**	**NewState**
X	$\Downarrow_A t_3$	$t_4\Uparrow_A$, $t_5\Uparrow_A$	Y

Table 4.2: Example: partially-specified features F and G

If these two features are composed into feature stack *[F/G]*, then the following series of events will occur when event $\Downarrow_A t_1$ is applied to state [A/X]:

- F will accept $\Downarrow_A t_1$ and transition into state B, outputting token events $t_2\Uparrow_A$ and $t_3\Downarrow_A$.

[1]However, the resulting specification can be used to compute the complete state space of the system by making use of the *resource-allocation-list* field which provides delta information about the usage of global resources.

- G will accept $t_3{\Downarrow}_A$ and transition into state Y, outputting token events $t_4{\Uparrow}_A$ and $t_5{\Uparrow}_A$.
- F will accept ${\Uparrow}_A t_4$ and transition into state C, outputting token event $t_6{\Uparrow}_A$.

Barring interference from other features, tokens $t_2{\Uparrow}_A$, $t_5{\Uparrow}_A$ and $t_6{\Uparrow}_A$ will escape from the stack, and a stable state ([C/Y]) will be reached. The generated state transition machine for the feature stack will simply record this series of state transitions as

State	**Input**	**Output**	**NewState**
A/X	${\Downarrow}_A t_1$	$t_2{\Uparrow}_A$, $t_5{\Uparrow}_A$, $t_6{\Uparrow}_A$	C/Y

Table 4.3: Example: resulting state transition rule in *[F/G]*

When *[F/G]* is later used during the composition of a call, tokens $t_3{\Downarrow}_A$ and $t_4{\Uparrow}_A$ will not come into play at all.

4.3.4 Algorithm: simulate()

It is useful to simulate the composition of a set of features when a feature interaction has been detected; it helps the user detect the sequences of events leading up to the interaction. The algorithm employed by the simulate() method is shown in Figure 4.4. The list of commands specified within is not complete; other commands exist for requesting different levels of verboseness from the system, querying for help, etc. See the user's manual in Appendix A for a complete list of commands.

```
do forever
    I = prompt_user()
    switch I
        case "state":
            report current system state
        case "state s":
            place system directly into state s
        case "events":
            report list of events currently accepted from environment
        case "event e":
            process_event(e)
        case "quit":
            return
```

Figure 4.4: simulate() algorithm

4.3.5 Feature Interaction Detection Tools

The following tools have been developed, and can be invoked from the UNIX command line:

`stmcheck` – Performs syntax checking and structural analysis on an ASCII specification. See Sections 5.1 and 5.2 for a list of checks performed.

`stackcompose` – Uses FeatureStack::compose() to compose one or more feature specifications together into the specification of a feature stack. Also performs syntax checking and structural analysis on both the original and resulting specifications.

`callcompose` – Uses Call::compose() to compose two feature stack specifications together into the specification of a call. Also performs syntax checking and structural analysis on both the original and resulting specifications.

`stacksim` – Uses FeatureStack::simulate() to simulate the behavior of one or more features in a feature stack.

`callsim` – Uses Call::simulate() to simulate the behavior of two feature stacks connected via a network in a call.

These tools provide functionality similar to the tools developed by Boumezbeur and Logrippo [1]. Specifically, `stackcompose` and `callcompose` are similar to their symbolic execution tool, while `stacksim` and `callsim` are similar to their step-by-step processing tool.

Please see the user's manual in Appendix A for a complete description of these tools.

Chapter 5

Reachability Analysis

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

5.1 Syntax Checking

It is important to verify certain properties about feature specifications before composing them together. One of the first things that is checked by the algorithm is the syntax of the specifications. Since feature specifications are constructed manually and are entered into the computer by humans, syntax checking is essential. All of the reachability analysis tools perform the following syntax checks:

- All state transition rules must be minimally specified. That is, each transition must have a *state*, an *input*, and a *newstate* field.
- State names must have the proper format: feature states must not have hierarchical names, whereas the names of states of higher-order entities (such as calls or call

systems) must have the hierarchical structure described in Section 4.1.2.

- All tokens in the specification must be elements of the set of valid tokens, which is specified separately in a file called `token_types`. See Appendix A for details.
- All resource names must be elements of the set of valid resources, which is specified in a file called `resources`. See Appendix A for details.

If any of these checks fail, an descriptive error message is displayed by the tool before exiting.

5.2 Structural Analysis

In addition to syntactic checks, the overall structures of feature specifications are checked for correctness and completeness. It is possible for this level of verification to detect deficiencies in the original specification of features. The following properties of feature specifications are verified:

- State and input must be unique for every state transition rule. This restriction ensures that the feature specifications are deterministic.
- The initial state of a feature must be NULL, indicating that the feature is initially inactive.
- There must be a path from the NULL state to every other state. If this were not the case, the feature specification would describe states that could never be reached.
- There must be a path from every state to the NULL state. If this were not the case, then there would be reachable states from which the feature could never be deactivated.

- All specified input and output events must be possible. For example, it is impossible for an internal signal event to activate a feature, since a feature must already be active in order for such internal signals to occur.

If any of these checks fail, a descriptive error message is displayed by the tool before exiting.

5.3 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 only on the current state of the system, which consists of the current states of the individual features, the heads of the input queues in the feature stack(s), and the set of currently used resources.

In many cases, detected interactions are desired interactions and the feature designer wants to make sure that the composition of the features results in an appropriate interaction. Alternately, feature interactions due to nondeterminism are resolved by prioritizing the features. The interaction warnings generated by the composition tools then serve to warn the feature designer of an interaction that has been resolved, in case the designer wishes to resolve the interaction differently.

The composition algorithm can detect six types of feature interactions: call control, information invalidation, data loss, resource contention, state control, and state loss. This is a superset of the class of feature interactions discussed in [3]. Each type of feature interaction that can be detected is described below.

5.3.1 Call Control

A *call-control* interaction occurs when one feature intercepts input data that another feature is waiting for. Such interactions are detected when multiple features are ready to operate on the same input event. The higher-priority feature may either consume the input event, thereby preventing the lower-priority feature from ever seeing the input, or it may forward the event to the lower-priority feature after taking some action. Call-control interactions are resolved by the prioritization of features on the feature stack; thus, warnings about call-control interactions are used by feature designers to verify desired data interceptions and to reveal resolved, undesired data interceptions which need to be documented.

As mentioned above, such interactions may be desired. For example, in the feature *Call Waiting* (*CWT*) for an originating call, the rule

State	**Input**	**Output**	**NewState**
HELDCALL	$\Downarrow_A *$		HELDCALL

places the originating call on hold so that it doesn't interfere with the agent's communication with the new call. It does this by intercepting and consuming all tokens destined for the *Originating Call Model* (*OCM*). While *CWT* is in state HELDCALL, the *OCM* will never see any tokens originating from the agent. Conversely, the agent will never see any tokens sent by the *OCM*.

The `stackcompose` tool generates feature interaction warning messages when such a call-control interaction interaction is detected. For the above example, the following warning message is one of several similar messages generated:

```
WARNING: Call Control Interaction detected
         at State "[HeldCall/AuthOrigAttempt]" -
         Features (CWT,OCM) accept "t:Disconnect:d:a1".
```

As well, there are several well-known, undesired interactions between services *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[1], the agent uses the $\Downarrow_A$FlashHook event to alternate between the two *CWT* calls. In *3WC*, the agent uses a $\Downarrow_A$FlashHook event to initially activate the feature; when the call to the third party has been established, the agent uses the $\Downarrow_A$FlashHook event again to join the three parties in a conference call. A call control interaction occurs if features of both services are ready to receive a $\Downarrow_A$FlashHook event, and one feature inputs the token without forwarding it to the other feature.

In one configuration, in which *3WC* is used to establish a second call and *CWT* is activated on the second call, the composition algorithm detects 12 call-control interactions, 5 of which are unique[2]:

```
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]
```

Three distinct call control interactions involving the $\Downarrow_A$FlashHook event are detected between *CWT* and *3WC*. In addition, Two distinct call control interactions are detected between *CWT* and *OCM*. One such interaction is depicted by the following scenario.

[1]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*.

[2]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 distinct warning messages.

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.

5.3.2 Information Invalidation

An *information-invalidation* interaction occurs at the feature stack composition level 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.

The same type of interaction is detected at the call composition level when a feature in one feature stack sends a modified token to a feature in another feature stack. For example, there is a known, desired, information-invalidation 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 get the initiator's telephone number and to display it when the call is presented to the receiving agent. Feature CNDB operates on top of an OCM; its purpose is to modify the CallRequest token so that feature CND will not display the initiator's number. This interaction is depicted in Figure 5.1.

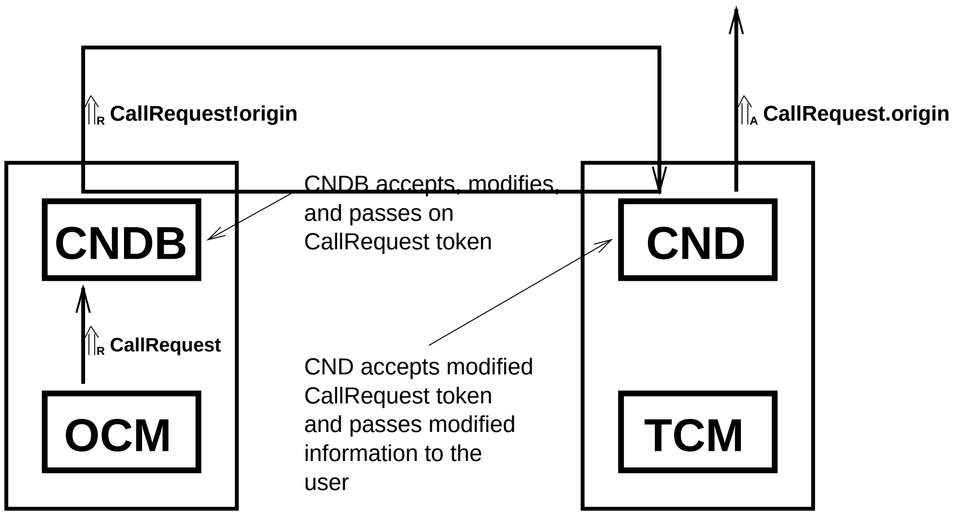
CallRequest!origin
CallRequest.origin
CNDB accepts, modifies, and passes on CallRequest token
CNDB
CND
CallRequest
CND accepts modified CallRequest token and passes modified information to the user
OCM
TCM

Figure 5.1: Desired Information Invalidation

When the feature stack *[CNDB/OCM]* is composed with the feature stack *[CND/TCM]* using the tool `callcompose`, the following feature interaction warning message is generated:

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

This message correctly indicates the fact that the feature stack *[CNDB/OCM]* is sending the feature stack *[CND/TCM]* the CallRequest token with the origin field modified. This is desired behavior, as the purpose of *CNDB* is to prevent *CND* from seeing the correct origin of the caller. Note that at this level of composition it is impossible for the algorithm to determine exactly which features are involved in the interaction, so the algorithm only reports the feature stacks involved (as well as the token involved).

As with call-control interactions, the prioritization of features resolves information-invalidation interactions; warnings simply notify the feature designer of the resolved interaction.

5.3.3 Data Loss

A *data loss* interaction occurs at the feature stack composition level when a token is passed down through a feature stack and eventually falls off the bottom of the stack. It can also occur at the call composition level when a token is sent across a network to another feature stack which does not accept the token. When this type of interaction is detected, it typically indicates that one feature has made an invalid assumption about the readiness of another feature to accept a particular token. This type of interaction may also be detected in cases where a reachable state at the feature stack level is unreachable at the call system level due to desired interactions between multiple features of a single service which operate on separate feature stacks. In this case the states are *nonsense* states since they do not exist in the fully-composed system. Therefore their behavior is unimportant and any feature interaction warning messages generated by such states should be ignored.

```
WARNING: token "t:Disconnect:d:a1" dropped off bottom of feature stack.
```

5.3.4 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 after each state transition with the specified capacity of each resource. Resource contention is also reported if the set of composed features releases more instances of a resource than it acquires.

For example, services *Three-Way Calling* (*3WC*) and *Call Waiting* (*CWT*) both require the use of a piece of hardware known as a bridge, but there is only one bridge available to an agent. If an agent attempts to use both services at the same time, there will be a resource contention interaction. A bridge is requested by the feature of *CWT* which operates on the incoming call that the service activates. In *3WC*, a bridge is requested by the feature that initiates a call to the third party. If these two 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 features reside on parallel feature stacks, then the resource contention would be detected when the calls were composed into call systems.

Some feature pairs interacting through a call system, such as the two *Call Waiting* (*CWT*) features, will falsely generate resource contention interaction messages during the feature-stack and call levels of composition. This is because the *CWT* feature at the active origining end of the call is responsible for releasing resources that the *CWT* feature at the incoming end of the call acquires. If the call-system level of composition were implemented, no such warning messages would be generated for the *CWT* features at the call level of composition, indicating that no resource contention exists between the *CWT* features.

5.3.5 State Control

Sometimes one feature will attempt to control the behavior of another feature in the same feature stack by forcing the second feature to undergo a particular state transition. A *state 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. The following is an example of the type of message that is generated when this type of feature interaction is detected:

```
Feature Interaction - Feature f received state change event "s:a:f:b:e"
but was in state c.  Cannot continue.
```

Since the composition algorithm cannot generate the next state, the algorithm must abort. A state control interaction is the only feature interaction that causes the composition algorithm to abort.

5.3.6 State Loss

A *state loss* interaction occurs when states that were reachable in a feature specification become unreachable when the feature is composed with other feature specifications into a feature stack. When this type of interaction is detected at the feature stack composition stage, it typically indicates an undesired feature interaction.

This type of interaction can also occur at the call composition stage, when states that were reachable in a feature stack specification become unreachable when the feature stack is composed with another feature stack. At this level of composition, however, such a feature interaction may be a normal side effect of two interacting feature stacks.

Chapter 6

Conclusion

During my thesis work I extended the work began by my supervisor in [3]. This involved helping to define a language for the specification of service-oriented software systems, with special extensions for the specification of telephony systems. I have also helped define the execution model for the features in such a system. Specifically, I have defined the rules which cause systems defined under the model to behave deterministically.

A large portion of my thesis work involved implementing a set of tools for incrementally composing features into feature stacks and feature stacks into calls. These tools are built around an efficient reachability-graph generator which examines all reachable states of a system in order to detect feature interactions. Algorithms have been developed and implemented for detecting six types of feature interactions: call control, information invalidation, data loss, resource contention, state control, and state loss.

Continuation of this work would involve the implementation of the third stage of composition: the composition of calls into call systems. This will complete the set of tools necessary to analyze complete call system configurations.

A longer term goal is to allow features to raise and lower *assertions* during their execution so that logical properties about the system can be verified. 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 the 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 that 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.

The composition algorithm already makes provisions for allowing features to raise and lower assertions by allowing an optional sixth field in each state transition rule of a feature specification. This field may contain a comma-separated list of assertion directives delimited by parentheses. An assertion directive is of the form '`a:assertion`' or '`u:assertion`' representing the raising (asserting) or lowering (unasserting) of an assertion. The algorithm also keeps track of the list of active assertions while performing reachability analysis. However, the algorithm currently contains no logic for the interpretation of the assertions. Algorithms must be developed which can analyze the active assertion list at every reachable state in order to detect logical inconsistencies in the specification of the system.

Bibliography

[1] R. Boumezbeur and L. Logrippo. "Specifying Telephone Systems in LOTOS". In *ICCC Communications Magazine*, pages 38–45, August 1993.

[2] T. F. Bowen, F.S. Dworack, C. H. Chow, N. Griffeth, G. E. German, and Y-J Lin. "The Feature Interaction Problem in Telecommunications Systems". In *Proceedings of the Seventh International Conference on Software Engineering for Telecommunication Switching Systems*, pages 59–62, July 1989.

[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.

[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] P. Combes and S. Pickin. "Formalisation of a User View of Network and Services for Feature Interaction Detection". In *Feature Interactions in Telecommunications Systems*, pages 120–135. IOS Press, 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] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[8] K. Kimbler, E. Kuisch, and J. Muller. "Feature Interactions among Pan-European Services". In *Feature Interactions in Telecommunications Systems*, pages 73–85. IOS Press, May 1994.

[9] 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. IOS Press, May 1994.

[10] T. Ohta and Y. Harada. "Classification, Detection and Resolution of Service Interactions in Telecommunication Services". In *Feature Interactions in Telecommunications Systems*, pages 60–72. IOS Press, May 1994.

[11] G. Titus and M. Stavely. "Finding Reachable States of Finite-State Concurrent Systems". *The Journal of Systems and Software*, 9:253–272, 1989.

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

[13] P. Zave. "Feature Interactions and Formal Specifications in Telecommunications". *Computer*, 26(8):20–29, August 1993.

Appendix A

Reachability Analysis Tools User's Manual

The following is a description of the tools that have been developed for the purpose of performing reachability analysis on the composition of feature specifications. These tools can be invoked from the Unix command-line. Each of the tools accept command-line options which specify the filename(s) of the ASCII specification(s) to work on, as well as optional arguments (indicated by enclosing brackets ([ ]) in the syntax descriptions below). In addition to the specification files, each of the tools also use two supplementary files which must reside in the current working directory as the tools are invoked. They are:

`resources` – Contains a list of resources available to the system being composed, one resource per line, with each resource followed by a number indicating how many of that particular resource are available. For example:

```
bridge 1
```

If the system being composed attempts to use a non-existent resource or attempts to use more of a particular resource than are available, the user will be informed.

`token_types` – Contains a list of the valid token types available to the system being composed. In this list it is possible to assign more than one synonym per token type. This is done by placing all of the synonyms for a single token type on a single line, with each line representing a different token type. For example:

```
CallRequest TerminationAttempt
Disconnect
NetworkBusy
ReleaseTimeout
FlashHook JoinConf SwitchCalls InitiateNewCall
```

Here, `FlashHook` and `SwitchCalls` are synonyms for each other, so if one feature emits a SwitchCalls token, another feature may intercept it as a FlashHook token. If a specification contains a token type that is not listed in this file, it will be flagged as an error. This mechanism is helpful in capturing typos in manually-typed specifications.

A.1 stmcheck

stmcheck performs syntax checking and structural analysis on an ASCII specification. See Sections 5.1 and 5.2 for a list of checks performed. The syntax for invoking **stmcheck** is as follows:

`stmcheck` [-d] *specification1* [*specification2* ...]

Option "`-d`" dictates that each ASCII specification should be dumped to `stdout` after being checked. Any number of specifications can be provided on the command line; each of them will be checked in order.

A.2 `stackcompose`

`stackcompose` composes one or more feature specifications together into the specification of a feature stack. It also performs syntax checking and structural analysis on both the original and resulting specifications. The syntax for invoking `stackcompose` is as follows:

`stackcompose` [`-v`[`#`]] [`-w`[`#`]] *feature activation_order ...*

The features must be supplied in order of priority. That is, the feature highest on the feature stack should be listed first on the command line, followed by the next-to-highest feature, and so on. The *activation_order* parameters (one for each feature) specify the order in which features become active and are placed into the feature stack. For example, if the user wishes to compose the behavior of a feature stack containing *Call Number Display Blocking (CNDB)*, *Originating Call Screening (OCS)*, and the *Originating Call Model (OCM)*, where *CNDB* has priority over *OCS* and is also activated before *OCS*, the following command can be used:

```
stackcompose CNDB 2 OCS 3 OCM 1
```

The last (i.e. lowest-priority) feature specified must be a basic call model such as *OCM* and *TCM*, and it must be the first feature to activate.

The "`-v`" option specifies the level of verboseness during the composition:

level 0 – outputs no trace information.

level 1 – outputs every system state and input combination tried.

level 2 – outputs level 1 information plus the resulting stable system state after each input is processed.

level 3 – outputs level 2 information plus individual state transitions, including transitions to non-stable states (i.e., transitions taken to resolve signals and tokens that the feature stack produces in the processing of a single external event).

level 4 – outputs level 3 information plus trace information from a user-supplied C++ `check_function()`, which must be compiled in with the rest of the code defining the tool set. This function is called for each new system state reached, and may be used for verifying specific state properties.

The following is a sample of level-3 verbose output from the `stackcompose` command while composing the feature *CNDB* on top of the *OCM*:

```
******************************************************************************
Trying state: [Null/SelectingRoute]
   resources: ()
       input: i:OCM:RouteSelected

Actions:
    CNDB: Null --> WaitForSend
                     (s:SelectingRoute:OCM:AuthCallSetup:i:OCM:RouteSelected)
    OCM: SelectingRoute --> AuthCallSetup (i:OCM:RouteSelected)

 New state: [WaitForSend/AuthCallSetup]
 resources: ()
******************************************************************************
```

If no "`-v`" option is present, the default verboseness is level 0. If the "`-v`" option is present without a number, the default verboseness is level 1.

The "`-w`" option specifies how many types of feature interactions to warn about during composition:

level 0 – outputs no warnings.

level 1 – outputs call control, information invalidation, data loss, state control, and state loss feature interaction warnings.

level 2 – outputs level 1 warnings plus resource contention warnings.

If no "`-w`" option is present, the default warn level is level 0. If the "`-w`" option is present without a number, the default warn level is level 2.

A.3 `callcompose`

`callcompose` composes two feature stack specifications together into the specification of a call. It also performs syntax checking and structural analysis on both the original and resulting specifications. The syntax for invoking `callcompose` is as follows:

`callcompose` [`-v`[#]] [`-w`[#]] *stack0 stack1*

The order of the feature stacks is irrelevant. The "`-v`" and "`-w`" options are the same as with the `stackcompose` tool.

A.4 `stacksim`

`stacksim` simulates the behavior of one or more features in a feature stack by providing the user with an interactive interface, allowing him/her to interactively explore the behavior of the feature stack. The command-line arguments and their meanings are the same as those for the `stackcompose` command.

When `stacksim` is invoked, the feature stack is started in the NULL state. The user is then repeatedly supplied with a prompt, at which the user can enter one of the following commands:

`state` – Queries the current state of the feature stack.

`state` s – Changes the state name of the feature stack. That is, it changes the states of the individual features in the stack without changing the list of active resources.

`state` s r – Changes the state of the feature stack, where s is the state name representing the new states of the individual features in the stack and r is the new list of active resources.

`event` – Queries the feature stack as to what events will cause it to transition into a new state.

`event` e – Instructs the feature stack to act as if event e has occurred. The prompt will not reappear until a stable state has been reached.

`verboseness` – Queries the current level of verboseness.

`verboseness` n – Changes the level of verboseness to n.

`warnlevel` – Queries the current warn level.

`warnlevel` n – Changes the warn level to n.

`help` – Provides a brief help menu containing this list of commands.

`quit` – Exits the simulation.

A.5 `callsim`

`callsim` simulates the behavior of two features stacks in a call by providing the user with an interactive interface, allowing him/her to interactively explore the behavior of the call. The command-line arguments and their meanings are the same as those for the `callcompose` command, and the commands that can be entered at the prompt are the same as those for the `stacksim` command.

Appendix B

Tabular Feature Specifications

The following are the tabular specifications of all the telephony features used in the case studies in Section 5.3.

B.1 Hold

State	**Input**	**Output**	**NewState**
NULL	$\Downarrow_A$ ActivateHold		FILTERING
FILTERING	$\Downarrow_A$ DeactivateHold		NULL
	$\Downarrow_A *$		FILTERING
	$\Uparrow_A *$		FILTERING

B.2 Originating Call Model

State	**Input**	**Output**	**NewState**
Null	$\Downarrow_A$OriginationAttempt		AuthOrigAttempt
AuthOrig Attempt	$\triangleright$**Originated**	CollectInfo$\Uparrow_A$	CollectingInfo
	$\triangleright$**OriginationDenied**	OriginationDenied$\Uparrow_A$	Exception
	$\Downarrow_A$Disconnect		Null
Collecting Info	$\Downarrow_A$Info		CollectingInfo
	$\triangleright$**InfoCollected**		AnalyzingInfo
	$\triangleright$**CollectionTimeout**	CollectionTimeout$\Uparrow_A$	Exception
	$\Downarrow_A$Disconnect		Null
Analyzing Info	$\triangleright$**ValidInfo**		SelectingRoute
	$\triangleright$**InvalidInfo**	InvalidInfo$\Uparrow_A$	Exception
	$\Downarrow_A$Disconnect		Null
Selecting Route	$\triangleright$**RouteSelected**		AuthCallSetup
	$\triangleright$**NetworkBusy**	NetworkBusy$\Uparrow_A$	Exception
	$\Downarrow_A$Disconnect		Null
Auth CallSetup	$\triangleright$**CallSetupAuthorized**	CallRequest$\Uparrow_R$	SendCall
	$\triangleright$**CallSetupDenied**	CallSetDenied$\Uparrow_A$	Exception
	$\Downarrow_A$Disconnect		Null
SendCall	$\Downarrow_R$CallDelivered	CallDelivered$\Uparrow_A$	Alerting
	$\Downarrow_R$Answered	Answered$\Uparrow_A$	Active
	$\Downarrow_R$CalledPartyBusy	CalledPartyBusy$\Uparrow_A$	Exception
	$\Downarrow_R$CallCleared		Exception
	$\Downarrow_A$Disconnect	CallCleared$\Uparrow_R$	Null
Alerting	$\Downarrow_R$Answered		Active
	$\Downarrow_R$CalledPartyBusy	CalledPartyBusy$\Uparrow_A$	Exception
	$\Downarrow_R$CallCleared		Exception
	$\Downarrow_A$Disconnect	CallCleared$\Uparrow_R$	Null
Active	$\Downarrow_R$CallCleared		ReleasePending
	$\Downarrow_R$Disconnect	CallCleared$\Uparrow_R$	Null
Release Pending	$\Downarrow_R$CalledPartyReconnect		Active
	$\triangleright$**ReleaseTimeout**	ReleaseTimeout$\Uparrow_R$	Null
	$\Downarrow_A$Disconnect	CallCleared$\Uparrow_R$	Null
Exception	$\Downarrow_A$Disconnect		Null

B.3 Terminating Call Model

State	**Input**	**Output**	**NewState**
Null	$\Downarrow_R$Termination Attempt		AuthTermination
Auth Termination	$\triangleright$**CallPresented**		HuntingFacility
	$\triangleright$**TerminationDenied**	CallCleared$\Uparrow_R$	Exception
	$\Downarrow_R$CallCleared		Null
	$\Downarrow_A$Disconnect	Call Cleared$\Uparrow_R$	Null
Hunting Facility	$\triangleright$**FacilityFound**		PresentingCall
	$\triangleright$**Busy**	CalledPartyBusy$\Uparrow_R$	Exception
	$\Downarrow_R$CallCleared		Null
	$\Downarrow_A$Disconnect	CallCleared$\Uparrow_R$	Null
Presenting Call	$\triangleright$**CallAccepted**	CallDelivered$\Uparrow_R$ Alert$\Uparrow_R$	Alerting
	$\triangleright$**CallFailure**	Call Cleared$\Uparrow_R$	Exception
	$\Downarrow_A$Connected	Answered$\Uparrow_R$	Active
	$\Downarrow_R$CallCleared		Null
	$\Downarrow_A$Disconnect	CallCleared$\Uparrow_R$	Null
Alerting	$\Downarrow_A$Connected	Answered$\Uparrow_R$	Active
	$\triangleright$**CallRejected**	CallCleared$\Uparrow_R$	Exception
	$\triangleright$**RingingTimeout**	CallCleared$\Uparrow_R$	Exception
	$\Downarrow_R$CallCleared		Null
	$\Downarrow_A$Disconnect	CallCleared$\Uparrow_R$	Null
Active	$\Downarrow_A$Disconnect	CallCleared$\Uparrow_R$	ReleasePending
	$\Downarrow_R$CallCleared		Null
Release Pending	$\Downarrow_A$CalledPartyReconnect	CalledPartyReconnect$\Uparrow_R$	Active
	$\Downarrow_R$ReleaseTimeout		Null
	$\Downarrow_R$CallCleared		Null
Exception	$\Downarrow_R$CallCleared		Null
	$\Downarrow_A$Disconnect		Null

B.4 Three-Way Calling (originating call)

State	**Input**	**Output**	**NewState**	**Resources**
Null	$\Downarrow_A$ InitiateNewCall	**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	

B.5 Three-Way Calling (new call)

State	**Input**	**Output**	**NewState**
Null	$\Rightarrow_{OCM}$ AuthOrigAttempt ($\Downarrow_A$ OriginationAttempt)		WaitAnswer
WaitAnswer	$\Downarrow_A$ Answered	$\ll$*forward*$\gg$	Private
	$\Rightarrow_{3WC}$ Null		Null
	$\Downarrow_A$ Disconnect	$\ll$*forward*$\gg$	Null
	$\Uparrow_R$ ReleaseTimeout	$\ll$*forward*$\gg$	Null
Private	$\Downarrow_A$ JoinConf		Conference
	$\Rightarrow_{3WC}$ Null		Null
	$\Downarrow_A$ Disconnect	$\ll$*forward*$\gg$	Null
	$\Uparrow_R$ ReleaseTimeout	$\ll$*forward*$\gg$	Null
Conference	$\Downarrow_A$ Disconnect	$\ll$*forward*$\gg$	Null
	$\Uparrow_R$ ReleaseTimeout	$\ll$*forward*$\gg$	Null

B.6 Originating Call Screening

State	Input	Output	NewState
Null	AnalyzingInfo$\Rightarrow_{OCM}$ SelectingRoute ($\triangleright$**ValidInfo**)		Test
Test	$\triangleright$**Match**	AnalyzingInfo$\Rightarrow_{OCM}$ Exception ($\triangleright$**InvalidInfo**)	Null
	$\triangleright$**NoMatch**	AnalyzingInfo$\Rightarrow_{OCM}$ SelectingRoute ($\triangleright$**ValidInfo**)	Null
	$\Downarrow_A$Disconnect	$\ll$*forward*$\gg$	Null
	$\Uparrow_R$ReleaseTimeout	$\ll$*forward*$\gg$	Null

B.7 Call Waiting (originating call)

State	Input	Output	NewState	Resources
Null	$\Rightarrow_{CWT}$HuntingFacility		Decision	
Decision	$\Downarrow_A$SwitchCalls		HeldCall	
	$\Rightarrow_{CWT}$Null		Null	
	Active$\Rightarrow_{OCM}$Null($\Downarrow_A$Disconnect)	$\ll$*forward*$\gg$	Null	–**bridge**
	ReleasePending$\Rightarrow_{OCM}$Null($\Downarrow_R$CallCleared)	$\ll$*forward*$\gg$	Null	–**bridge**
HeldCall	$\Downarrow_A$SwitchCalls		Active	
	$\Rightarrow_{CWT}$Null		Null	
	$\Downarrow_A$Disconnect		RingBack	–**bridge**
	ReleasePending$\Rightarrow_{OCM}$Null($\Downarrow_R$CallCleared)	$\ll$*forward*$\gg$	Null	–**bridge**
	$\Downarrow_A *$		HeldCall	
	$\Uparrow_A *$		HeldCall	
Active	$\Downarrow_A$SwitchCalls		HeldCall	
	$\Rightarrow_{CWT}$Null		Null	
	Active$\Rightarrow_{OCM}$Null($\Downarrow_A$Disconnect)	$\ll$*forward*$\gg$	Null	–**bridge**
	ReleasePending$\Rightarrow_{OCM}$Null($\Downarrow_R$CallCleared)	$\ll$*forward*$\gg$	Null	–**bridge**
RingBack	$\Downarrow_A$CallAnswered		Null	
	$\triangleright$**RingBackTimeout**	Disconnect$\Downarrow_A$	Null	
	$\Downarrow_A *$		RingBack	
	$\Uparrow_A *$		RingBack	

B.8 Call Waiting (incoming call)

See Table 3.1.

B.9 Call Number Display

State	**Input**	**Output**	**NewState**
NULL	$\Rightarrow_{TCM}$ AUTHTERMINATION ($\Downarrow_R$ TerminationAttempt)		WAITFORRING
WAITFORRING	$\Uparrow_A$ Alert	≪*forward*≫ TerminationAttempt.origin$\Uparrow_A$	NULL
	$\Downarrow_A$ Disconnect	≪*forward*≫	NULL
	$\Downarrow_R$ CallCleared	≪*forward*≫	NULL

B.10 Call Number Display Blocking

State	**Input**	**Output**	**NewState**
NULL	SELECTINGROUTE$\Rightarrow_{OCM}$ AUTHCALLSETUP ($\triangleright$**RouteSelected**)	≪*forward*≫	WAITFORSEND
WAITFOR SEND	$\Uparrow_R$ CallRequest	CallRequest!origin$\Uparrow_R$	NULL
	$\Downarrow_A$ Disconnect	≪*forward*≫	NULL
	$\Uparrow_R$ ReleaseTimeout	≪*forward*≫	NULL

Appendix C

Composed Specifications

The following are examples of the composition of some of the feature specifications in Appendix B. Each of these compositions were generated with the tools described in Appendix A. See Section 4.1 for a description of how to interepret these composite specifications. Where appropriate, excerpts from the verbose output produced by the tools are included, illustrating the detection of feature interactions.

C.1 Call *[[OCM]/[TCM]]*

The most straightforward call specification is one in which no features are activated. In this case, one feature stack consists solely of the *Originating Call Model* (*OCM*) while the other consists solely of the *Terminating Call Model* (*TCM*).

```
[[OCM]/[TCM]]

# Automatically generated STM specification.

[[Null]/[Null]] t:OriginationAttempt:d:a1 () [[AuthOrigAttempt]/[Null]]
[[AuthOrigAttempt]/[Null]] i:OCM:Originated (t:CollectInfo:u:a1) [[CollectingInfo]/[Null]]
[[AuthOrigAttempt]/[Null]] i:OCM:OriginationDenied (t:OriginationDenied:u:a1) [[Exception]/[Null]]
[[AuthOrigAttempt]/[Null]] t:Disconnect:d:a1 () [[Done]/[Null]]
[[CollectingInfo]/[Null]] t:Info:d:a1 () [[CollectingInfo]/[Null]]
[[CollectingInfo]/[Null]] t:InfoCollected:d:a1 () [[AnalyzingInfo]/[Null]]
[[CollectingInfo]/[Null]] i:OCM:CollectionTimeout (t:CollectionTimeout:u:a1) [[Exception]/[Null]]
[[CollectingInfo]/[Null]] t:Disconnect:d:a1 () [[Done]/[Null]]
[[Exception]/[Null]] t:Disconnect:d:a1 () [[Done]/[Null]]
[[AnalyzingInfo]/[Null]] i:OCM:ValidInfo () [[SelectingRoute]/[Null]]
[[AnalyzingInfo]/[Null]] i:OCM:InvalidInfo (t:InvalidInfo:u:a1) [[Exception]/[Null]]
[[AnalyzingInfo]/[Null]] t:Disconnect:d:a1 () [[Done]/[Null]]
[[SelectingRoute]/[Null]] i:OCM:RouteSelected () [[AuthCallSetup]/[Null]]
[[SelectingRoute]/[Null]] i:OCM:NetworkBusy (t:NetworkBusy:u:a1) [[Exception]/[Null]]
[[SelectingRoute]/[Null]] t:Disconnect:d:a1 () [[Done]/[Null]]
[[AuthCallSetup]/[Null]] i:OCM:CallSetupAuthorized () [[SendCall]/[AuthTermination]]
[[AuthCallSetup]/[Null]] i:OCM:CallSetupDenied (t:CallSetDenied:u:a1) [[Exception]/[Null]]
[[AuthCallSetup]/[Null]] t:Disconnect:d:a1 () [[Done]/[Null]]
[[SendCall]/[AuthTermination]] t:Disconnect:d:a1 () [[Done]/[Done]]
[[SendCall]/[AuthTermination]] i:TCM:CallPresented () [[SendCall]/[HuntingFacility]]
[[SendCall]/[AuthTermination]] i:TCM:TerminationDenied (t:CallCleared:u:a1) [[Exception]/[Exception]]
[[SendCall]/[AuthTermination]] t:Disconnect:d:a2 (t:CallCleared:u:a1) [[Exception]/[Done]]
[[SendCall]/[HuntingFacility]] t:Disconnect:d:a1 () [[Done]/[Done]]
[[SendCall]/[HuntingFacility]] i:TCM:FacilityFound () [[SendCall]/[PresentingCall]]
[[SendCall]/[HuntingFacility]] i:TCM:Busy (t:CalledPartyBusy:u:a1) [[Exception]/[Exception]]
[[SendCall]/[HuntingFacility]] t:Disconnect:d:a2 (t:CallCleared:u:a1) [[Exception]/[Done]]
[[Exception]/[Exception]] t:Disconnect:d:a1 () [[Done]/[Exception]]
[[Exception]/[Exception]] t:Disconnect:d:a2 () [[Exception]/[Done]]
[[Exception]/[Done]] t:Disconnect:d:a1 () [[Done]/[Done]]
[[SendCall]/[PresentingCall]] t:Disconnect:d:a1 () [[Done]/[Done]]
[[SendCall]/[PresentingCall]] i:TCM:CallAccepted (t:Alert:u:a2,t:CallDelivered:u:a1) [[Alerting]/[Alerting]]
[[SendCall]/[PresentingCall]] i:TCM:CallFailure () [[SendCall]/[HuntingFacility]]
[[SendCall]/[PresentingCall]] t:Connected:d:a2 (t:Answered:u:a1) [[Active]/[Active]]
[[SendCall]/[PresentingCall]] t:Disconnect:d:a2 (t:CallCleared:u:a1) [[Exception]/[Done]]
[[Done]/[Exception]] t:Disconnect:d:a2 () [[Done]/[Done]]
[[Alerting]/[Alerting]] t:Disconnect:d:a1 () [[Done]/[Done]]
[[Alerting]/[Alerting]] t:Connected:d:a2 () [[Active]/[Active]]
[[Alerting]/[Alerting]] i:TCM:CallRejected (t:CallCleared:u:a1) [[Exception]/[Exception]]
[[Alerting]/[Alerting]] i:TCM:RingingTimeout (t:CallCleared:u:a1) [[Exception]/[Exception]]
[[Alerting]/[Alerting]] t:Disconnect:d:a2 (t:CallCleared:u:a1) [[Exception]/[Done]]
[[Active]/[Active]] t:Disconnect:d:a1 () [[Done]/[Done]]
[[Active]/[Active]] t:Disconnect:d:a2 () [[ReleasePending]/[ReleasePending]]
[[ReleasePending]/[ReleasePending]] i:OCM:ReleaseTimeout () [[Done]/[Done]]
[[ReleasePending]/[ReleasePending]] t:Disconnect:d:a1 () [[Done]/[Done]]
[[ReleasePending]/[ReleasePending]] t:CalledPartyReconnect:d:a2 () [[Active]/[Active]]

# Total number of rules: 45
```

C.2 Feature Stack *[CNDB/OCM]*

When *Call Number Display Blocking* (*CNDB*) is composed with the *OCM*, the following specification results:

```
[CNDB/OCM]

# Automatically generated STM specification.

[Null/Null] t:OriginationAttempt:d:a1 () [Null/AuthOrigAttempt]
[Null/AuthOrigAttempt] i:OCM:Originated (t:CollectInfo:u:a1) [Null/CollectingInfo]
[Null/AuthOrigAttempt] i:OCM:OriginationDenied (t:OriginationDenied:u:a1) [Null/Exception]
[Null/AuthOrigAttempt] t:Disconnect:d:a1 () [Null/Done]
[Null/CollectingInfo] t:Info:d:a1 () [Null/CollectingInfo]
[Null/CollectingInfo] t:InfoCollected:d:a1 () [Null/AnalyzingInfo]
[Null/CollectingInfo] i:OCM:CollectionTimeout (t:CollectionTimeout:u:a1) [Null/Exception]
[Null/CollectingInfo] t:Disconnect:d:a1 () [Null/Done]
[Null/Exception] t:Disconnect:d:a1 () [Null/Done]
[Null/AnalyzingInfo] i:OCM:ValidInfo () [Null/SelectingRoute]
[Null/AnalyzingInfo] i:OCM:InvalidInfo (t:InvalidInfo:u:a1) [Null/Exception]
[Null/AnalyzingInfo] t:Disconnect:d:a1 () [Null/Done]
[Null/SelectingRoute] i:OCM:RouteSelected () [WaitForSend/AuthCallSetup]
[Null/SelectingRoute] i:OCM:NetworkBusy (t:NetworkBusy:u:a1) [Null/Exception]
[Null/SelectingRoute] t:Disconnect:d:a1 () [Null/Done]
[WaitForSend/AuthCallSetup] t:Disconnect:d:a1 () [Done/Done]
[WaitForSend/AuthCallSetup] i:OCM:CallSetupAuthorized (t:CallRequest!origin:u:r) [Done/SendCall]
[WaitForSend/AuthCallSetup] i:OCM:CallSetupDenied (t:CallSetDenied:u:a1) [WaitForSend/Exception]
[Done/SendCall] t:CallDelivered:d:r (t:CallDelivered:u:a1) [Done/Alerting]
[Done/SendCall] t:Answered:d:r (t:Answered:u:a1) [Done/Active]
[Done/SendCall] t:CalledPartyBusy:d:r (t:CalledPartyBusy:u:a1) [Done/Exception]
[Done/SendCall] t:CallCleared:d:r (t:CallCleared:u:a1) [Done/Exception]
[Done/SendCall] t:Disconnect:d:a1 (t:CallCleared:u:r) [Done/Done]
[WaitForSend/Exception] t:Disconnect:d:a1 () [Done/Done]
[Done/Alerting] t:Answered:d:r () [Done/Active]
[Done/Alerting] t:CalledPartyBusy:d:r () [Done/Exception]
[Done/Alerting] t:CallCleared:d:r (t:CallCleared:u:a1) [Done/Exception]
[Done/Alerting] t:Disconnect:d:a1 (t:CallCleared:u:r) [Done/Done]
[Done/Active] t:CallCleared:d:r () [Done/ReleasePending]
[Done/Active] t:Disconnect:d:a1 (t:CallCleared:u:r) [Done/Done]
[Done/Exception] t:Disconnect:d:a1 () [Done/Done]
[Done/ReleasePending] t:CalledPartyReconnect:d:r () [Done/Active]
[Done/ReleasePending] i:OCM:ReleaseTimeout (t:ReleaseTimeout:u:r) [Done/Done]
[Done/ReleasePending] t:Disconnect:d:a1 (t:CallCleared:u:r) [Done/Done]

# Total number of rules: 34
```

C.3 Feature Stack *[CND/TCM]*

When *Call Number Display* (*CND*) is composed with the *TCM*, the following specification results:

```
[CND/TCM]

# Automatically generated STM specification.

[Null/Null] t:CallRequest:d:r () [WaitForRing/AuthTermination]
[WaitForRing/AuthTermination] t:Disconnect:d:a1 (t:CallCleared:u:r) [Done/Done]
[WaitForRing/AuthTermination] t:CallCleared:d:r () [Done/Done]
[WaitForRing/AuthTermination] i:TCM:CallPresented () [WaitForRing/HuntingFacility]
[WaitForRing/AuthTermination] i:TCM:TerminationDenied (t:CallCleared:u:r) [WaitForRing/Exception]
[WaitForRing/HuntingFacility] t:Disconnect:d:a1 (t:CallCleared:u:r) [Done/Done]
[WaitForRing/HuntingFacility] t:CallCleared:d:r () [Done/Done]
[WaitForRing/HuntingFacility] i:TCM:FacilityFound () [WaitForRing/PresentingCall]
[WaitForRing/HuntingFacility] i:TCM:Busy (t:CalledPartyBusy:u:r) [WaitForRing/Exception]
[WaitForRing/Exception] t:Disconnect:d:a1 () [Done/Done]
[WaitForRing/Exception] t:CallCleared:d:r () [Done/Done]
[WaitForRing/PresentingCall] t:Disconnect:d:a1 (t:CallCleared:u:r) [Done/Done]
[WaitForRing/PresentingCall] t:CallCleared:d:r () [Done/Done]
[WaitForRing/PresentingCall] i:TCM:CallAccepted (t:CallDelivered:u:r,t:Alert:u:a1,
          t:TerminationAttempt.origin:u:a1) [Done/Alerting]
[WaitForRing/PresentingCall] i:TCM:CallFailure () [WaitForRing/HuntingFacility]
[WaitForRing/PresentingCall] t:Connected:d:a1 (t:Answered:u:r) [WaitForRing/Active]
[Done/Alerting] t:Connected:d:a1 (t:Answered:u:r) [Done/Active]
[Done/Alerting] i:TCM:CallRejected (t:CallCleared:u:r) [Done/Exception]
[Done/Alerting] i:TCM:RingingTimeout (t:CallCleared:u:r) [Done/Exception]
[Done/Alerting] t:CallCleared:d:r () [Done/Done]
[Done/Alerting] t:Disconnect:d:a1 (t:CallCleared:u:r) [Done/Done]
[WaitForRing/Active] t:Disconnect:d:a1 (t:CallCleared:u:r) [Done/ReleasePending]
[WaitForRing/Active] t:CallCleared:d:r () [Done/Done]
[Done/Active] t:Disconnect:d:a1 (t:CallCleared:u:r) [Done/ReleasePending]
[Done/Active] t:CallCleared:d:r () [Done/Done]
[Done/Exception] t:CallCleared:d:r () [Done/Done]
[Done/Exception] t:Disconnect:d:a1 () [Done/Done]
[Done/ReleasePending] t:CalledPartyReconnect:d:a1 (t:CalledPartyReconnect:u:r) [Done/Active]
[Done/ReleasePending] t:ReleaseTimeout:d:r () [Done/Done]
[Done/ReleasePending] t:CallCleared:d:r () [Done/Done]

# Total number of rules: 30
```

C.4 Call *[[CNDB/OCM]/[CND/TCM]]*

When the feature stacks *[CNDB/OCM]* and *[CND/TCM]* (see Sections C.2 and C.3 respectively) are composed, the following (desired) feature interaction is detected (see Section 5.3.2 for an explanation):

```
******************************************************************************
Trying state: [[WaitForSend/AuthCallSetup]/[Null/Null]]
   resources: ()
       input: i:OCM:CallSetupAuthorized

Actions:
    [CNDB/OCM]: [WaitForSend/AuthCallSetup] --> [Done/SendCall] (i:OCM:CallSetupAuthorized)
    [CND/TCM]: [Null/Null] --> [WaitForRing/AuthTermination] (t:CallRequest!origin:d:r)

WARNING: Feature Stack [CND/TCM] accepted token t:CallRequest!origin:d:r
         which was modified by [CNDB/OCM].

 New state: [[Done/SendCall]/[WaitForRing/AuthTermination]]
 resources: ()
******************************************************************************
```

The specification generated is as follows:

```
[[CNDB/OCM]/[CND/TCM]] # Automatically generated STM specification.

[[Null/Null]/[Null/Null]] t:OriginationAttempt:d:a1 () [[Null/AuthOrigAttempt]/[Null/Null]]
[[Null/AuthOrigAttempt]/[Null/Null]] i:OCM:Originated (t:CollectInfo:u:a1) [[Null/CollectingInfo]/[Null/Null]]
[[Null/AuthOrigAttempt]/[Null/Null]] i:OCM:OriginationDenied (t:OriginationDenied:u:a1) [[Null/Exception]/[Null/Null]]
[[Null/AuthOrigAttempt]/[Null/Null]] t:Disconnect:d:a1 () [[Null/Done]/[Null/Null]]
[[Null/CollectingInfo]/[Null/Null]] t:Info:d:a1 () [[Null/CollectingInfo]/[Null/Null]]
[[Null/CollectingInfo]/[Null/Null]] t:InfoCollected:d:a1 () [[Null/AnalyzingInfo]/[Null/Null]]
[[Null/CollectingInfo]/[Null/Null]] i:OCM:CollectionTimeout (t:CollectionTimeout:u:a1) [[Null/Exception]/[Null/Null]]
[[Null/CollectingInfo]/[Null/Null]] t:Disconnect:d:a1 () [[Null/Done]/[Null/Null]]
[[Null/Exception]/[Null/Null]] t:Disconnect:d:a1 () [[Null/Done]/[Null/Null]]
[[Null/AnalyzingInfo]/[Null/Null]] i:OCM:ValidInfo () [[Null/SelectingRoute]/[Null/Null]]
[[Null/AnalyzingInfo]/[Null/Null]] i:OCM:InvalidInfo (t:InvalidInfo:u:a1) [[Null/Exception]/[Null/Null]]
[[Null/AnalyzingInfo]/[Null/Null]] t:Disconnect:d:a1 () [[Null/Done]/[Null/Null]]
[[Null/SelectingRoute]/[Null/Null]] i:OCM:RouteSelected () [[WaitForSend/AuthCallSetup]/[Null/Null]]
[[Null/SelectingRoute]/[Null/Null]] i:OCM:NetworkBusy (t:NetworkBusy:u:a1) [[Null/Exception]/[Null/Null]]
[[Null/SelectingRoute]/[Null/Null]] t:Disconnect:d:a1 () [[Null/Done]/[Null/Null]]
[[WaitForSend/AuthCallSetup]/[Null/Null]] t:Disconnect:d:a1 () [[Done/Done]/[Null/Null]]
[[WaitForSend/AuthCallSetup]/[Null/Null]] i:OCM:CallSetupAuthorized () [[Done/SendCall]/[WaitForRing/AuthTermination]]
[[WaitForSend/AuthCallSetup]/[Null/Null]] i:OCM:CallSetupDenied (t:CallSetDenied:u:a1) [[WaitForSend/Exception]/[Null/Null]]
[[Done/SendCall]/[WaitForRing/AuthTermination]] t:Disconnect:d:a1 () [[Done/Done]/[Done/Done]]
[[Done/SendCall]/[WaitForRing/AuthTermination]] t:Disconnect:d:a2 (t:CallCleared:u:a1) [[Done/Exception]/[Done/Done]]
[[Done/SendCall]/[WaitForRing/AuthTermination]] i:TCM:CallPresented () [[Done/SendCall]/[WaitForRing/HuntingFacility]]
[[Done/SendCall]/[WaitForRing/AuthTermination]] i:TCM:TerminationDenied (t:CallCleared:u:a1) [[Done/Exception]/[WaitForRing/Exception]]
[[WaitForSend/Exception]/[Null/Null]] t:Disconnect:d:a1 () [[Done/Done]/[Null/Null]]
[[Done/Exception]/[Done/Done]] t:Disconnect:d:a1 () [[Done/Done]/[Done/Done]]
[[Done/SendCall]/[WaitForRing/HuntingFacility]] t:Disconnect:d:a1 () [[Done/Done]/[Done/Done]]
[[Done/SendCall]/[WaitForRing/HuntingFacility]] t:Disconnect:d:a2 (t:CallCleared:u:a1) [[Done/Exception]/[Done/Done]]
[[Done/SendCall]/[WaitForRing/HuntingFacility]] i:TCM:FacilityFound () [[Done/SendCall]/[WaitForRing/PresentingCall]]
[[Done/SendCall]/[WaitForRing/HuntingFacility]] i:TCM:Busy (t:CalledPartyBusy:u:a1) [[Done/Exception]/[WaitForRing/Exception]]
[[Done/Exception]/[WaitForRing/Exception]] t:Disconnect:d:a1 () [[Done/Done]/[WaitForRing/Exception]]
[[Done/Exception]/[WaitForRing/Exception]] t:Disconnect:d:a2 () [[Done/Exception]/[Done/Done]]
[[Done/SendCall]/[WaitForRing/PresentingCall]] t:Disconnect:d:a1 () [[Done/Done]/[Done/Done]]
[[Done/SendCall]/[WaitForRing/PresentingCall]] t:Disconnect:d:a2 (t:CallCleared:u:a1) [[Done/Exception]/[Done/Done]]
[[Done/SendCall]/[WaitForRing/PresentingCall]] i:TCM:CallAccepted (t:Alert:u:a2,t:TerminationAttempt.origin:u:a2,
        t:CallDelivered:u:a1) [[Done/Alerting]/[Done/Alerting]]
[[Done/SendCall]/[WaitForRing/PresentingCall]] i:TCM:CallFailure () [[Done/SendCall]/[WaitForRing/HuntingFacility]]
[[Done/SendCall]/[WaitForRing/PresentingCall]] t:Connected:d:a2 (t:Answered:u:a1) [[Done/Active]/[WaitForRing/Active]]
[[Done/Done]/[WaitForRing/Exception]] t:Disconnect:d:a2 () [[Done/Done]/[Done/Done]]
[[Done/Alerting]/[Done/Alerting]] t:Disconnect:d:a1 () [[Done/Done]/[Done/Done]]
[[Done/Alerting]/[Done/Alerting]] t:Connected:d:a2 () [[Done/Active]/[Done/Active]]
[[Done/Alerting]/[Done/Alerting]] i:TCM:CallRejected (t:CallCleared:u:a1) [[Done/Exception]/[Done/Exception]]
[[Done/Alerting]/[Done/Alerting]] i:TCM:RingingTimeout (t:CallCleared:u:a1) [[Done/Exception]/[Done/Exception]]
[[Done/Alerting]/[Done/Alerting]] t:Disconnect:d:a2 (t:CallCleared:u:a1) [[Done/Exception]/[Done/Done]]
[[Done/Active]/[WaitForRing/Active]] t:Disconnect:d:a1 () [[Done/Done]/[Done/Done]]
[[Done/Active]/[WaitForRing/Active]] t:Disconnect:d:a2 () [[Done/ReleasePending]/[Done/ReleasePending]]
[[Done/Active]/[Done/Active]] t:Disconnect:d:a1 () [[Done/Done]/[Done/Done]]
[[Done/Active]/[Done/Active]] t:Disconnect:d:a2 () [[Done/ReleasePending]/[Done/ReleasePending]]
[[Done/Exception]/[Done/Exception]] t:Disconnect:d:a1 () [[Done/Done]/[Done/Exception]]
[[Done/Exception]/[Done/Exception]] t:Disconnect:d:a2 () [[Done/Exception]/[Done/Done]]
[[Done/ReleasePending]/[Done/ReleasePending]] i:OCM:ReleaseTimeout () [[Done/Done]/[Done/Done]]
[[Done/ReleasePending]/[Done/ReleasePending]] t:Disconnect:d:a1 () [[Done/Done]/[Done/Done]]
[[Done/ReleasePending]/[Done/ReleasePending]] t:CalledPartyReconnect:d:a2 () [[Done/Active]/[Done/Active]]
[[Done/Done]/[Done/Exception]] t:Disconnect:d:a2 () [[Done/Done]/[Done/Done]]

# Total number of rules: 51
```