Elsevier Copyright Notice

Published in: ***Computer Networks***, Vol. 32, No. 4, April 2000

"A Hybrid Model for Specifying Features and Detecting Interactions"

Cite as:

Saheem Siddiqi and Joanne M. Atlee. 2000. A hybrid model for specifying features and detecting interactions. Computer Networks, 32, 4 (April 2000), 471-485.

BibTex:

@article{Siddiqi:2000:HMS:343667.343684,
 author = {Siddiqi, Saheem and Atlee, Joanne M.},
 title = {A Hybrid Model for Specifying Features and Detecting Interactions},
 journal = {Computer Networks},
 issue_date = {April 2000},
 volume = {32},
 number = {4},
 month = Apr,
 year = {2000},
 pages = {471--485}
}

DOI: http://dx.doi.org/10.1016/S1389-1286(00)00011-6

A Hybrid Model for Specifying Features and Detecting Interactions

Saheem Siddiqi Joanne M. Atlee [1]

Department of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1 Canada
{*s4siddiq,jmatlee*}*@se.uwaterloo.ca*

Abstract

This paper presents a formal modeling and analysis technique for automatically detecting interactions in feature specifications. The model is based on state-transition machines annotated with logic assertions that express constraints on the behavior of the system. Specifications are composed into a reachability graph, and the graph is searched for feature interactions. The new model and analysis techniques are evaluated with respect to the Feature Interaction Detection Contest benchmark.

Key words: feature interactions; composite specifications; reachability analysis.

1 Introduction

Modern software engineering techniques facilitate the development of feature-rich applications. For example in *incremental programming*, one first implements the application's core functionality – its basic *service* – and adds new functionality in small increments – as *features* of the service. In *separation of concerns*, one decomposes a problem into several weakly-coupled sub-problems, each of which is smaller and simpler than the original and can be considered in isolation. Using these two techniques, the designer of a feature-rich application can consider each feature as an independent increment of the basic service.

The problem is that seemingly unrelated features can subtly interfere with each other because they enhance the same basic service and manipulate the

[1] This research is supported by MITEL, with matching funds from Communications and Information Technology Ontario.

same state variables. For example, Plain Old Telephone Service (POTS) establishes voice connections between pairs of users; Three-Way Calling (3WC) is a feature that enhances POTS by enabling conference calls among 3 users; Terminating Call Screening (TCS) is a feature that prohibits calls from numbers on a user-defined list. If a user A invokes 3WC to establish a conference call with user B and someone on B's screening list, then a connection is made that violates the intention of B's call screening feature. In general, a *feature interaction* occurs when one feature affects the behaviour of another [8].

The benefits of incremental programming and separation of concerns outweigh the problems of feature interactions. Applying incremental programming and separation of concerns simplifies the application's design and helps prevent architectural decay as the application is maintained and extended. It also decreases the time-to-market for new features, because independent features can be developed in parallel or contracted out to third-party programmers.

Our goal is to automatically detect feature interactions during the requirements phase of feature development. In this paper, we introduce a hybrid model for specifying and analyzing features that augments our previous state-machine model [1, 3, 12] with logic formulae. We model services and features as state-transition machines, whose transitions are annotated with logic assertions that specify constraints on service behaviour. Our analysis tools compose feature specifications into a reachability graph and test each reachable state for violations of logic assertions and other interactions.

This paper describes our feature model, our model of composition, and our analysis tools. It also describes and evaluates the performance of our models and tools at the first Feature Interaction Detection Contest.

2 Features, Combining Features, and Feature Interactions

We use the terminology presented in the Bellcore Memorandum for Discussion titled *Definitions of Services, Features, and Feature Interactions* [6].

Definition 1 *A* service *provides stand-alone functionality.*

The basic service in call processing is Plain Old Telephone Service (POTS), which establishes and maintains voice connections between pairs of users. The service is specified as two cooperating machines (see Figure 1), one that models service provided to the caller (the Originating Call Model, or OCM) and one that models service provided to the callee (the Terminating Call Model, or TCM). The environment for each machine is the human user (i.e., the caller or callee) and the connection to the other machine.

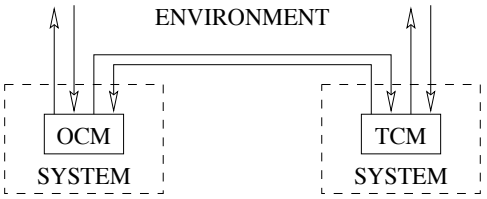

Fig. 1. Basic Service

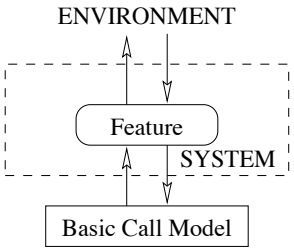

Fig. 2. Feature

Definition 2 *A* feature *provides added functionality to an existing feature or service; a feature cannot operate stand-alone.*

A feature (see Figure 2) is modelled as a machine that intercedes between some basic service and the service's environment. It interposes its behavior by intercepting and possibly modifying communication between the service and the environment and by controlling the service's reactions to events. Features are not independent components. Rather, the composition of a feature with its basic service creates a new virtual service to the environment.

Definition 3 *A* feature interaction *is a discrepancy between a feature's specified behaviour and its actual behaviour when combined with other features.*

Some software developers limit their notion of feature interactions to unexpected interactions between supposedly unrelated features. Such interactions are *harmless* if the feature combination deviates from the features' planned behaviours but is still acceptable. A *conflict* interaction occurs when a feature combination violates the intentions of the individual features (e.g., the interaction between Three-Way Calling and Terminating Call Screening described in the Introduction). However, feature designers would also like to ensure that *planned interactions* continue to occur in the presence of additional features. We want our tools to report all interactions, so that the human analyst can verify planned interactions and identify conflict interactions.

2.1 Combining Features

Each feature is specified as an isolated extension of the basic service.[2] To understand how features behave in concert, we combine them in different call configurations. Each *call configuration* defines which features execute on behalf of each user and in what order. Different call configurations reveal different behaviour, so we analyze several configurations when studying a combination of features. Call configurations are discussed in Section 4.

[2] A feature may extend another feature. However, this paper will focus on features that extend services and only mention features that extend features when necessary.

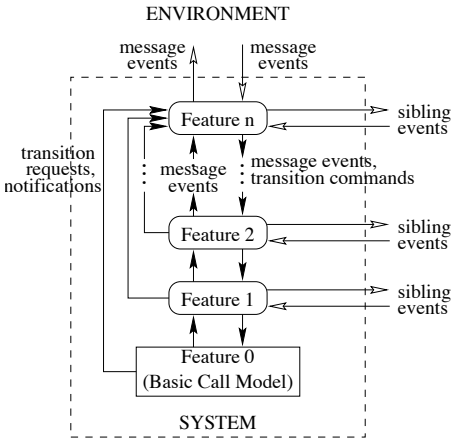

Fig. 3. Half-Call Configuration

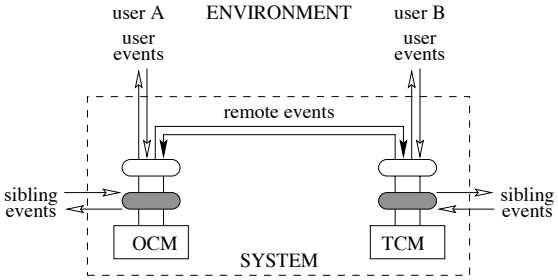

Fig. 4. Call Configuration

Feature compositions are generated incrementally: the features executing on one end of a call are composed into a *half-call*, the two halves of a call are composed into a *call*, and related calls are composed into a *call-group*. The latter composition is needed to study the behaviour of features that affect more than one call.

A *half-call* (Figure 3) represents the combined behaviour of all features executing on one half of a call. The features are arranged in a pipe-and-filter-like architecture, where the feature order realizes a priority scheme. The features closest to the environment have highest priority; they are the first to intercept inputs and are the last to modify outputs to the environment. The basic service is furthest from the environment and has lowest priority. Messages (e.g., offhook events, dialed digits, audible tones, etc.) are passed from the environment to the basic call model, or from the basic call model to the environment, visiting the intervening features along the way. Each feature can react to a message or allow it to pass. If a feature reacts to a message, it can re-emit the same message or emit zero or more different messages.

Intelligent Network (IN) features (not shown) are implemented outside of the telephone switch and have different composition rules from the other half-call features. They are activated by state transitions in the call model: IN features activated by the same transition execute concurrently, suspending normal call processing until they all terminate. IN features cannot access network services (e.g., to play an message or forward a call); instead, they send requests and actions to the call model, which accesses services on the features' behalf. When an IN feature terminates, it can resume the call model in an arbitrary state.

A *call* (Figure 4) is the composition of two half-calls, one based on the Originating Call Model (OCM) and the other on the Terminating Call Model (TCM). It represents the combined behaviour of all the features and services that execute during the call. Analysis at the call level explores the call's reactions to messages passed between the two half-calls.

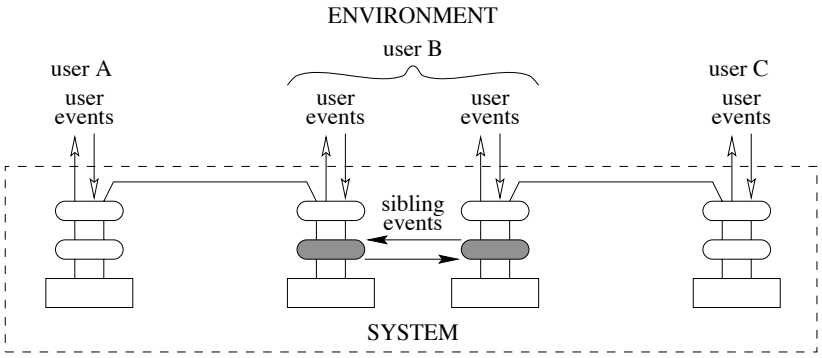

Fig. 5. Call-Group Configuration

A *call-group* (Figure 5) represents the combined behaviour of features executing on two or more related calls. Calls are related by features that apply to more than one call (e.g., Call Waiting or Three-Way Calling). Such features are specified as a set of *sibling features*, one sibling for each affected call, that coordinate and share data by sending each other *sibling events*. Analysis at the call-group level explores the calls' reactions to sibling events.

3 The Specification Model

Services and features are specified as state-transition machines using a tabular notation. The State in the first row of a table is the initial state. Every feature starts in either state Ready (meaning the feature is enabled) or state Null (meaning the feature is not ready to execute). Each row in the table represents a transition from the current state to the next state, activated by an input event. As side effects, a transition can output events, acquire or release resources, and assert or retract properties about the call.

Table 1 shows a partial specification for the Originating Call Model. The call starts when the caller takes the phone off-hook (row 1). This event activates the transition to state AuthOrigAttempt, where the call model decides whether to authorize the new call; as a side effect, the transition asserts that the caller

State	Input Event	Next State	Output Event	Resources	Assertions
Ready	$\Downarrow_U$OffHook	AuthOrigAttempt			$\oplus$busyin(s), $\oplus$busyout(s)
AuthOrigAttempt	$\triangleright_{OCM}$Originated	CollectInfo	$\Uparrow_U$DialTone		
	$\triangleright_{OCM}$OrigDenied	Exception	$\Uparrow_U$CircuitsBusy		
	$\Downarrow_U$OnHook	Ready			$\ominus$busyin(s), $\ominus$busyout(s)
⋮	⋮	⋮	⋮	⋮	⋮

Table 1
Partial specification of the Originating Call Model.

Event Class	Notation	Description
Message Events	$\Downarrow_U$msg $\Uparrow_U$msg	*User Message Event:* message is passed to or from the user
	$\Downarrow_R$msg $\Uparrow_R$msg	*Remote Message Event:* message is passed to or from the remote end-of-call
	$\Downarrow_{IN}$msg $\Uparrow_{IN}$msg	*IN Feature Message Event:* message is passed to or from an IN feature
State Transition Events	$\overset{O}{\rightarrow}_f$	*EnabledEvent:* service/feature f is ready to execute
	$\overset{O}{\rightarrow}_f$S($e$)	*ActivatedEvent:* service/feature f is activated by event e and is in state S
	S1$\overset{R}{\rightarrow}_f$S2($e$)	*StateTransitionRequest Event:* service/feature f requests a transition from S1 to S2, triggered by event e
	${}_g$S1$\overset{M}{\rightarrow}_f$S2($e$)	*ModifyTransition Event:* feature g forces service/feature f to transition from S1 to S2 due to event e
	S1$\overset{O}{\rightarrow}_f$S2($e$)	*OccurredTransition Event:* service/feature f has made a transition from S1 to S2 due to event e
Forwarded Events	$\ll$forward$\gg$	*Forwarded Event:* abbreviation for input events passed unchanged to the next feature in a half-call
Sibling Events	$\Rightarrow_{P(f)}$msg $\Leftarrow_{P(f)}$msg	*Parallel Message Event:* message is passed to or from sibling f
	S1$\overset{O}{\rightarrow}_{P(f)}$S2	*ParallelTransition Event:* sibling f has made a transition to state S2
	NewCall$_f$(BCM)	*NewCall Event:* a new sibling half-call is activated, with basic call model BCM and feature f
Internal Events	$\triangleright_f$event	*Internal Event:* event produced by service/feature f's internal computation
Resources	$+$*resource* $-$*resource*	service/feature needs access to *resource* service/feature releases *resource*
Assertions	$\oplus$v(x), $\oplus$v(x,y) $\ominus$v(x), $\ominus$v(x,y) $\models f$ $\not\models f$	inserts x or pair (x,y) into variable v removes x or pair (x,y) from variable v assert constraint expressed as formula f retract constraint expressed as formula f

Table 2
Descriptions of Events and Assertions

is now busy. The second and third rows specify what happens if the call is authorized and not authorized, respectively. The fourth row specifies what happens if the user hangs up while the call is being authorized.

3.1 Input and Output Events

Transitions are activated by input events from the environment and from neighbouring machines. A summary of event types is given in Table 2.

Message Events are data passed between the basic service and its environment. The arrow in a message event specifies the message's destination: a downward message is passed from the environment "down" to the service, and an upward message is passed from the service "up" to the environment. The subscript in a message event designates a particular environment: subscript $_U$ refers to the user, $_R$ refers to the remote end of the call, and $_{IN}$ refers an IN feature.

State-transition events announce the call model's occurred and pending state transitions, enabling features on the same half-call to react to or override state-changes in the call model. While such information could be passed via message events, a message might arrive too late or be distorted by the intervening features. Each call model implicitly generates an *Activated Event* when it activates, a *State Transition Request Event* before every transition, and an *Occurred TransitionEvent* after every transition. A feature may delay, deny, or modify a State Transition Request, in the last case issuing a *Modify Transition Event* that instructs the service to make an alternate transition. For example in our model, Call Waiting prevents the Terminating Call Model (TCM) from rejecting a call request when the callee is busy and, instead, instructs the TCM to continue setting up the call. [3]

Sibling features communicate and synchronize with each other via *Sibling Events*. Siblings pass data via *Parallel Message Events* and announce occurred state-transitions via *Parallel Transition Events*. A *New Call Event* initiates a new half-call that includes a newly invoked sibling. For example, our specification of Three-Way-Calling outputs event NewCall$_{3WC}$(OCM), which creates a new OCM and 3WC sibling that initiates the call to the third party.

Internal Events announce results from a feature's internal computations, such as responses to resource requests, timeouts, analyses of dialed numbers, etc.

3.2 Resources

We use *Resources* to specify a feature's usage of shared resources (e.g., the use of a conference bridge). An entry $+res$ specifies when resource *res* is acquired, and an entry $-res$ specifies when it is released. During analysis, we compare the number of each resource with the number acquired by features.

[3] Features also issue state-transition events. In addition to the above-mentioned events, features issue an *Enabled Event* when they are ready to execute. In practice, few features are designed to cooperate with other features; thus, it is rare for features to react to other features' state-transition events.

3.3 Assertions

We use *Assertions* to specify features' long-term effects on service behaviour[5]. The system state is modelled as a collection of relations on users, resources, and other entities. If $\mathbb{U}$ is the set of users, some of our state variables are:

vconn $\subseteq \mathbb{U} \times \mathbb{U}$ - the set of voice connections between users
busyin $\subseteq \mathbb{U}$ - users who cannot receive calls (e.g., line is busy)
billing $\subseteq \mathbb{U} \times \$$ - outstanding charges

Features may introduce new relations to model feature-specific state and data:

hold $\subseteq \mathbb{U} \times \mathbb{U}$ - the set of calls on hold.
cf $\subseteq \mathbb{U} \times \mathbb{U}$ - call-forward destinations
tcs $\subseteq \mathbb{U} \times \mathbb{U}$ - calls screened by terminating call screening

As features execute, they modify variable values as a side effect of their state transitions. An entry $\oplus$*rel(A,B) asserts* that ordered pair (A,B) is an element of relation *rel*; entry $\ominus$*rel(A,B) retracts* the assertion and removes the pair from *rel*. Similar notation specifies assertions made to set variables. For example, the call model asserts and retracts assertions about **vconn** as it makes and breaks voice connections.

Features specify constraints on system behaviour as first-order predicate sentences on the variables.

- Set and relational variables are atomic predicates
- If P and Q are both sets or relations, then so are: $P \cup Q$, $P \cap Q$, $P \setminus Q$
- If P is a relation, then:

$$dom(P) = \{s \mid (s,t) \in P\}, \quad ran(P) = \{t \mid (s,t) \in P\}$$
$$P^{-1} = \{(t,s) \mid (s,t) \in P\}, \quad P(s) = \{t \mid (s,t) \in P\}$$

Formulae are defined inductively:

- If u is a term, and P and Q are predicates, then the following are atomic formulae: $u \in P, P = Q, P \subset Q, P \subseteq Q$.
- If f and g are formulae, then so are: $\neg f, f \vee g, f \wedge g, f \rightarrow g, f \leftrightarrow g$.
- If f is a formula, x is a term variable, and P is a predicate, then the following are formulae: $\forall x \in P(f), \exists x \in P(f)$.

Features assert or retract constraint assertions as a side effect of their state transitions. For example, our 911 feature asserts that it cannot be put on hold:

$$\models 911 \notin ran(hold)$$

State	Input Event	Next State	Output Event	Assertions
Ready	$*\xrightarrow{O}_{TCM}$AuthTerm(*)	Test		
Test	$\triangleright_{TCS}$Match	Ready	$_{TCS}$AuthTerm$\xrightarrow{M}_{TCM}$ Ready($\triangleright$TermDenied)	
	$\triangleright_{TCS}$NoMatch	Ready		$\oplus$tcs(s,d) $\models$vconn^{-1}(d)$\subset$tcs^{-1}(d)
	$*\xrightarrow{O}_{TCM}$Ready(*)	Ready		
	$*\xrightarrow{R}_{TCM}$HuntFacility(*)	LateTest		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 3
Partial specification of Terminating Call Screening

State	Input Event	Next State	Output Event	Assertions
Null	$*\xrightarrow{O}_{OCM}$Active	Ready		
Ready	$\Downarrow_U$Flashhook	Holding	NewCall$_{3WC}$(OCM)	$\oplus$hold(s,d)
	$*\xrightarrow{O}_{OCM}$Ready	Null		
	$*\xrightarrow{O}_{OCM}$Exception	Null		
Holding	$*\xrightarrow{O}_{P(3WC)}$CallCancelled	Ready		$\ominus$hold(s,d)
	$*\xrightarrow{O}_{P(3WC)}$Conference	Conference		$\ominus$hold(s,d)
	$*\xrightarrow{O}_{OCM}$Ready	Null		$\ominus$hold(s,d)
	$*\xrightarrow{O}_{OCM}$Exception	Null		$\ominus$hold(s,d)
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4
Partial specification of Three-Way Calling

The specification for feature Terminating Call Screening (TCS) exhibits different aspects of our notation. The feature starts when the Terminating Call Model is activated (row 1) [4]. The next two transitions are activated by internal events produced when the feature tests the caller's number. If the number is on the screening list, then the feature outputs a Modify Transition Event that forces the Terminating Call Model (TCM) to reject the call (row 2). Otherwise (row 3), the feature allows the call to proceed, but it records that the test has been performed (first assertion) and it constrains the remainder of the call to voice connections that have passed the screening test (second assertion). If the call terminates, then the feature will also terminate (row 4). If the call model tries to authorize the call before the feature completes its test, then the feature suspends the call model by not granting its transition request (row 5); eventually, the feature will decide whether or not to screen the call and will issue a Modify Transition Event to resume the call model in the appropriate state (not shown).

[4] In state-transition events, event (*) is a wildcard that matches all events.

The specification of Three-Way Calling (3WC) uses sibling features, one for the original call and one for the call to the third party. Table 4 shows a partial specification of the first sibling. The feature is enabled when the call model enters state Active and a voice connection is established (row 1). The feature is activated by a command from the user (row 2), at which point this sibling places the call on hold (assertion) and initiates, for the third-party call, a new half-call and the second sibling (output event). The second sibling (not shown) is responsible for interacting with the user to establish the third-party call and for joining the two calls into a conference call. If successful, the second sibling will transition to its Conference state, and the original sibling will respond by transitioning to Conference and releasing the held call (row 6). If instead the third-party call is unsuccessful, then the second sibling will transition to its CallCancelled state, and the original sibling will respond by releasing the held call and terminating the feature (row 5). As with most features, the feature terminates whenever the call terminates (rows 3, 4, 7, 8).

4 Scenario Selection

We have developed a suite of tools that analyze features arranged in a particular *call configuration*, in which features execute on particular half-calls in a particular order. Most interactions can be found by analyzing pairs of features: given a set of features, if pairs of features do not interact, then larger subsets usually do not interact. Thus to detect interactions among a set of features, we typically only analyze pairs of features. How many call configurations we analyze depends on the number of contexts in which the features can execute and the number of calls the features can affect.

If two features F and G can execute in the same half-call, with either OCM or TCM, and in either order, then there are at least four different call configurations to analyze (see Figure 6). The order of invocation determines the features' positions in the half-call configuration: the most recently activated feature is given highest priority because we assume it is freshest in the user's mind and is thus the intended destination of the user's input. The other two configurations correspond to the cases where each feature executes on one of the call's half-calls.

If feature F is made up of sibling features (e.g., Three-Way Calling) and G is not, then there are 12 possible call configurations to analyze (Figure 7). If F is composed of more than two siblings, or if both F and G are composed of siblings (e.g., if we want to analyze Call Waiting and Three-Way Calling), then the number of call configurations could be larger.

We rarely analyze so many configurations because most features cannot exe-

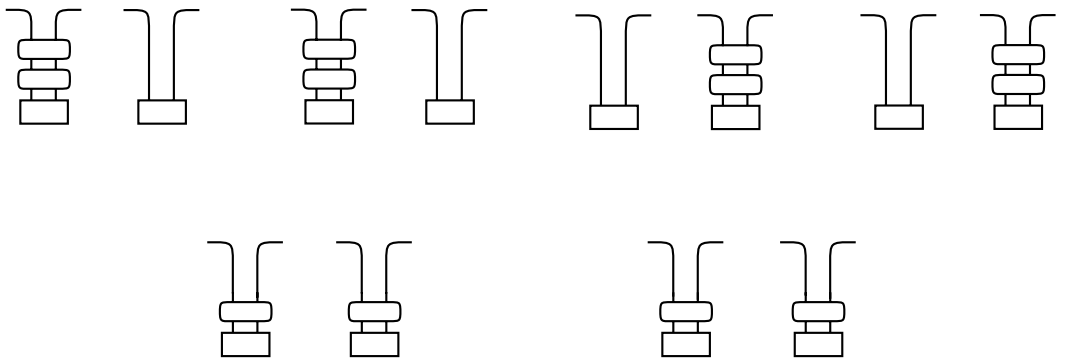

Fig. 6. Call configurations: features F and G execute in the same call.

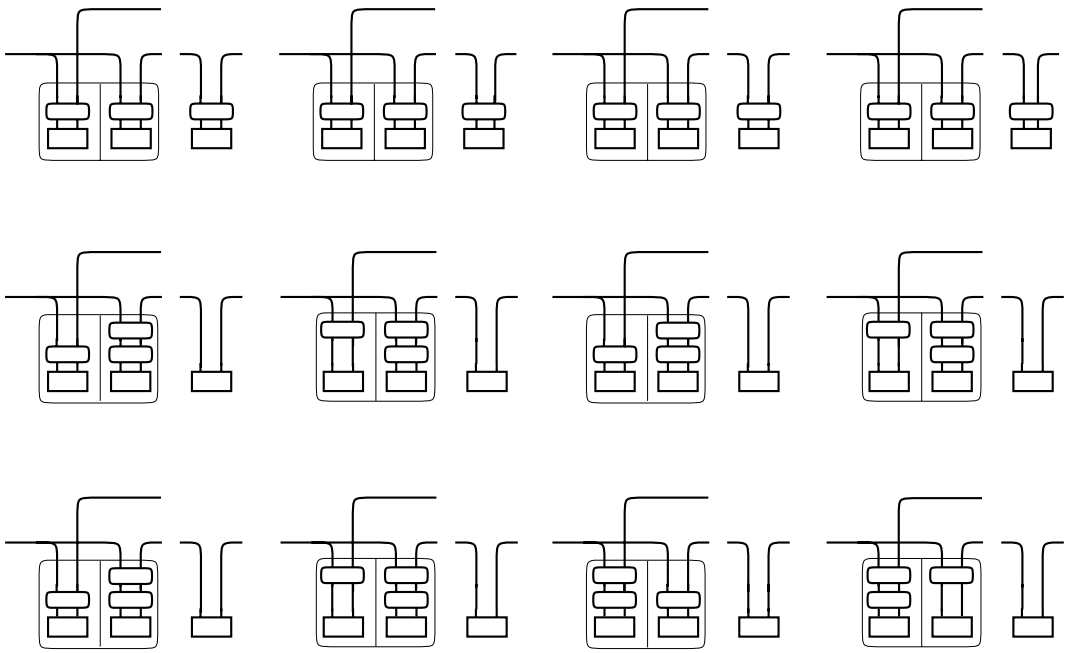

Fig. 7. Call configurations: feature F is initiated by the caller and spans two calls.

cute in arbitrary contexts. Terminating Call Screening always executes as part of the callee's half-call. The Call Waiting sibling that accepts additional calls always executes on a callee's half-call. Feature 911 has top priority no matter when it is activated. Each restriction reduces by half the number of configurations we need to consider. Note that call configurations do not depend on data values. Data are abstracted in specifications as input events (e.g., an internal event that decides whether or not a phone number is on a screening list) that provoke equivalence classes of behaviours.

5 Reachability Analysis

Our analysis tools take as input a call configuration and a set of feature specifications and produce as output a *reachability graph*, which depicts all of the configuration's execution paths. Each node in the reachability graph is a compound state made up of one state from each of the feature specifications. The initial node is composed of the features' initial states. Starting with the initial node, the tools determine for each node which input events the constituent states are ready to react to; and for each event, they compute the features' reaction as a compound transition of features' transitions to a new graph node. Each compound state is analyzed for interactions on-the-fly, as it is generated. But to alleviate the state-explosion problem, only stable states are stored as nodes in the resultant reachability graph.

5.1 Warning Messages

We have developed a number of tests that analyze reachable states and output messages to warn of possible interactions. A human analyst interprets the messages and decides whether the warnings reveal an interaction and, if so, whether the interaction is planned, harmless, or a conflict. Because every reachable node is tested, the analysis is exhaustive.

- An *event loss* warning is issued when a feature outputs a message event, but no feature or service reacts to it An *impossible state transition* warning is issued when a feature outputs a Modify Transition Event for an invalid transition. These *coordination warnings* reveal errors in planned interactions, where one feature fails to control the service or a second feature.
- A *modified state transition* warning is issued when a State Transition Request is denied or modified. The human analyst uses these warnings to verify planned interactions. If a Modified Transition has output events, then a *side-effect* warning is issued for every transition activated by these outputs.
- A *modified event* warning is issued when some feature intercepts or alters an input event that other features are ready to react to (e.g., when Three-Way Calling and Call Waiting are both ready to react to a flashhook event).
- A *reachability* warning is issued when one of the features being composed never enters one of its states (e.g., two users who unconditionally forward calls to each other and thus never reach a voice-connection state).
- A *modified data* warning is issued whenever a feature reacts to a modified message event (e.g., when Call Number Display receives caller ID information that was modified by Call Number Display Blocking). We explicate modified message events in specifications by marking the output events. The analysis tools track the marked events as they are passed among features.

- A *resource contention* warning is issued whenever the number of requested resources exceeds the number of available resources.
- An *assertion violation* indicates a constraint has been violated. Each node in the reachability graph is annotated with the variable values and constraints that hold in that state. The constraints are evaluated with respect to the variable values, and a warning is issued if a constraint evaluates to *false*.
- An *IN conflict* warning is issued whenever concurrent IN features disagree on either the requests sent to the service or the state in which call processing resumes (e.g., IN Freephone Routing and IN Freephone Billing request different connections and charge the call to different users).
- An *IN modified state transition* warning is issued when an IN feature resumes call processing in a new state (e.g., when IN Teen Line terminates a call attempted during restricted hours). This warning reveals interactions between the IN feature and other half-call features that would react to transitions in the call model that now may not occur.

5.2 Examples

Suppose user C subscribes to TCS. Half-call analysis of a call configuration where C is a callee reveals the following warnings.

```
CONTROL WARNING: Modified State Transition detected -
        forced modification of feature TCM by TCS.
(From state: [Test/AuthTerm] to state: [Ready*/Ready*])

CONTROL WARNING: Modified State Transition detected -
        forced modification of feature TCM by TCS.
(From state: [LateTest/AuthTerm] to state: [Ready*/Ready*])

CONTROL WARNING: Modified State Transition detected -
        original request
               "r:AuthTerm:TCM:HuntFacility:TCM:CallPresented"
        was denied by Feature TCS at state "[Test/AuthTerm]".

CONTROL WARNING: Modified State Transition detected -
        forced modification of feature TCM by TCS.
(From state: [LateTest/AuthTerm] to state: [Ready*/HuntFacility*])
```

The first two messages warn that TCS may terminate the call (i.e., forcs the call model back to initial state Ready)[5]. The third message states that TCS, while in state Test, prevents the call model from prematurely authorizing the call. The fourth message states that TCS may force the call model to authorize the call. These warnings confirm all of the feature's planned interactions.

Suppose that user A calls user B, and then invokes 3WC to establish a conference call with user C. Half-call and call-level analyses of this configuration reveal the following side-effect warnings.

[5] [A/B] denotes a compound state where the feature is in state A and the call model is in state B. Names with asterisks are new states resulting from a transition.

```
SIDE-EFFECT WARNING: Output CallCleared:u:r result of forced modification
         of feature TCM by TCS.
         (From state: [Test/AuthTerm] to state: [Ready*/Ready*])

SIDE-EFFECT WARNING: Side Effect Interaction detected -
         [3WC/OCM] accepted message CallCleared:d:r in state
                [[SentCall/SendCall]/[Ready/Exception]]
```

The first warning states that the Modified Transition outputs event CallCleared towards the remote end of the call (`:u:r`). The second warning states that the originating half-call reacts to the CallCleared side-effect message by terminating both the third-party call and the feature.[6] In essence, these messages warn that the third-party call will be rejected if User C screens calls from User A. The human analyst determines that the rejected call conflicts with the intentions of 3WC and decides that a conflict interaction has occurred.

Call-level analysis of this call configuration also reveals an assertion violation:

```
Assertion Warning: at state [[Conference/Active]/[Ready*/Active]],
while the sets and relations are:
   vconn: <A,B>,
   vconn: <A,C>,
   vconn: <B,C>,
   tcs: <A,C>,
   the constraint is:  vconn~{C}:<=:tcs~{C}.
```

The constraint asserts that callee C may only be connected to callers who have passed C's call screening test; the constraint literally says that the inverse of vconn applied to C (which is the set {A,B}) should be a subset of the inverse of tcs applied to C (which is {A}). The human analyst determines that a conflict interaction occurs if User C screens calls from User B.

To help the analyst relate warning messages to interactions, we augmented the reachability analysis tools to produce all of the paths from the initial graph node to a specified compound state. The analyst can use this program to understand the scenarios that lead to a warning state.

5.3 The Feature Interaction Detection Contest

We entered our notation and analysis tools in the Feature Interaction Detection contest. We specified and analyzed all 12 contest features, though we had some problems mapping the Chisel diagrams, which have an event-based model, onto our state-transition-machine model. Our models are based on the AIN architecture and specify unobservable states between an input and its corresponding output, whereas Chisel diagrams describe only observable behaviour. Also, the Chisel specification for POTS is missing some failure

[6] [[A/B]/[C/D]] denotes a compound state in a call: [A/B] is a compound state in the originating half-call, and [C/D] is from the terminating half-call.

	CFBL	CND	INFB	INFR	INTL	TCS	TWC	INCF	CW	INCC	RC	CELL
CFBL	–	–	2A	3C	–	2C	2A 3C	1A $1S_{IN}$	3C	1A $1S_{IN}$	–	–
CND		–	–	1C	–	1C	–	1C	–	$1S_{IN}$	–	–
INFB			–	1A $1T_{IN}$	–	–	3A	2A	2A	1A	–	1A
INFR				–	–	1C $1S_{IN}$	3A 1C	2A	1A 2C	2A	–	–
INTL					–	–	$1S_{IN}$	–	–	–	–	–
TCS						–	1A 1C	1C	1C	$1S_{IN}$	2C	–
TWC							2C	2A 2C	7C	3A $1S_{IN}$	1C	1A
INCF								1R	1A 3C	2A	1C	–
CW									2C	2A $1S_{IN}$	2C	1A
INCC										–	–	1A
RC											–	–
CELL												–

Interpretation of table entries:
1,2,3: Number of a particular type of interaction
C: Control Interaction – corresponds to occurrences of either control modifications or coordination warnings
A: Assertion Violation
R: Reachability Interaction
S_{IN}: IN side-effect – occurs when features miss a POTS state transition event because an adjunct feature resumes call processing at a different point-in-call
T_{IN}: IN conflict – occurs when multiple AIN features are activated at the same point-in-call but have different outcomes

Table 5
Results of the interaction analysis

scenarios (e.g., phone cannot initiate/accept calls, circuits busy, etc.). This hidden and missing behaviour is revealed in the Chisel feature specifications, which means that we needed to decide, for every contest feature, whether the feature diagram describes POTS-related behaviour and, if so, whether that behaviour belongs in POTS or whether the feature emulates POTS.

We detected 91 interactions between pairs of features. Table 5 shows the number and types of interactions we found. The contest organizers compared our list of interactions against theirs, which they compiled from manual inspections, and found 40 of our interactions correct, 14 incorrect, and 37 irrelevant (these were deemed interactions between a feature and POTS rather than interactions between features). We also missed 46 of their interactions.

One reason why so many of our interactions were ignored is we used a different definition of feature interaction. We assumed an interaction is a discrepancy between a feature's specified behaviour and its behaviour when combined with other features. The contest rules did not provide a definition, but from examining their list of interactions we deduce that the contest organizers were only interested in interactions that adversely affect the behaviour of some feature.

We missed some interactions because we did not strictly adhere to the Chisel specifications with respect to how features activated. For example the Chisel specification for the feature Cellular Charge (CELL) specifes the following behaviour: the caller goes offhook, the caller dials the CELL subscriber, the subscriber's phone rings (i.e., the subscriber's phone must not be busy), and

the subscriber answers the call. Call Forward Busy Line interacts with CELL because the subscriber is not charged for airtime when the call is forwarded, since the caller does not dial the subscriber's number. We missed this interaction because we assumed a CELL subscriber would be charged for airtime whenever the subscriber received and answered a call.

Other interactions were missed or rejected because we adopted the wrong interpretation of ambiguous specifications. For example, they rejected our interaction between IN Freephone Billing (INFB) and CELL because we wrongly assumed that the intention of feature INFB was to bill the callee for the entire cost of the call; thus, we thought that the CELL feature, when applied to caller, violated INFB's intention by charging the caller for airtime.

We have examined the organizer's partial list of interactions and have used their explanations to interpret specifications, resolve ambiguities, and identify call configurations that reveal interactions. Of the 51 interactions described in the partial list, we originally missed 19, but only 3 are undetectable in our model. In each of these, the features output contradictory call-model responses. For example, if a subscriber of IN Charge Call (INCC) calls a CFBL subscriber who is already on the phone, CFBL forwards the call and the caller hears an audible ring tone; but the INCC specification says the caller should hear a busy tone. In our model, the features do not output these tones; they each manipulate the call model, which outputs the tones. Since the call-model is in exactly one state at a time, it cannot simultaneously output both tones.

6 Related Work

Interaction-detection methods that have been proposed in the literature generally fall into one of three categories: state-based, logic-based, or hybrid models.

6.1 State-Based Models

State transition machines are the classic model used to specify telephony services and features. Feature specifications are typically composed into a reachability graph, and the graph is analyzed for feature-independent interactions, such as deadlock, livelock, non-determinism, and reachability.

Kelly et al. [9] and Combes and Pickin [7] use the Specification and Description Language (SDL), based on communicating finite state machines. POTS and features are specified as processes, and compositions are configurations of processes. In the second approach, they express feature requirements as event

traces in Message Sequence Charts (MSC) and detect interactions by testing that configurations satisfy the same MSC properties as their constituent features. Interactions are exposed by simulation and manual inspection.

Bruns et al. [4] specify services as a set of variables and a set of rules that modify variable values in response to input events. Feature specifications enhance service behaviour by declaring new events and rules. If several features and services are ready to react to the same event, their rules are applied one at a time, according to a priority scheme. Interactions are detecting by generating the specifications' reachability graph and performing special-purpose tests on nodes in the graph. For example, one test reveals *order interactions*, where the priority of activated rules affects the features' response to an input event. A second test reveals when reacting rules have conflicting outputs.

6.2 Logic-Based Models

Logic-based models typically consist of two parts: a set of state variables and a state transition relation. State variables describe the states of the service and features, and the transition relation describes allowable changes to variable values. Composition of feature specifications is by conjunction. The classes of detectable interactions are deadlock, livelock, assertion violations and nondeterminism.

Blom et al. [2] use temporal logic and the Z notation to specify services and features. The state variables are relations between users. The transition relation defines initial conditions, input events (predicates that describe communication between the system and its environment), and the system's response (changes to the state variables). Interactions are detected when several transition relations react to the same event but with conflicting effects. the system.

Ohta et al. [15] developed a notation called State Transition Rules (STR), where user states are predicates on phone numbers and transitions specify changes to predicate values in response to events. The composition of feature specifications is the union of the features' STR rules, where a priority scheme governs the application of rules. The current state is the union of the user states. Interactions are detected by comparing the states and transitions of individual features with those of the composite system and identifying unreachable states and transitions.

The state variables in the logic-based models represent the same information as our state-transition machine states and our assertions combined. The logic-based transition relations are as complex as our state transitions with their side effects. Thus, the expressiveness of the notations used in the two approaches is the same. One advantage of our hybrid model is the separation of the state-

transition relation from information that models more permanent effects on the call. For this reason, most practitioners prefer to use extended finite state machines to specify communication protocols.

6.3 Hybrid Models

Lin and Lin [11] propose a hybrid model that combines state transition machines and logic. *Procedural level specifications*, written in Promela, describe how a feature should operate, and *behavioral-level specifications*, written in temporal logic, describe what properties a feature must exhibit. Features are specified as independent entities and are used as building blocks to build different call configurations. Half-call, call, and call-group configurations can be analyzed. The Spin model checker is used to verify that the behavioural-level specifications hold for combinations of features. While this approach will detect reachability, control modifications, and assertion violation interactions, it cannot detect unexpected interactions due to data modification or to side effects of modified control flow.

Logrippo et al. [13] use the LOTOS specification language to model feature interactions. This method is based on the idea of feature intentions (similar to our assertions) which describe the asserted assumptions and properties that the features expect to hold. An intention verifier is used to synchronize with the actions in the system and to check whether a violation has occurred. Each system of features and each intention verifier is viewed as a process in process algebra and they can be joined together in parallel composition. An interaction is detected when a feature intention is violated in the system. This method uses reachability analysis, but the analysis is of paths in the reachability graph rather than states. Because exhaustive analysis of reachable paths is infeasible, search strategies must be suggested to reduce the search space. Interactions are detected by manual inspection of the traces produced from the tools. The interactions this method can detect would correspond to our control modification, data modification, assertions violation and reachability interactions.

Khoumsi [10] specifies POTS and features as extended finite state machines augmented with boolean variables. Propositional formulae over the variables act as guards on the transitions and can also be used to express invariant properties of a feature. The variables are updated by the transitions. Specifications are composed into a reachability graph, and the graph is searched for undesirable properties: deadlock, unreachable states, non-determinism, or violations of invariants. Our tools reveal a wider class of interaction types because, in addition to the above, our tools track the effects of manipulated data and state transitions.

7 Conclusion

Previously, our group used pure state-based models and analysis techniques to automatically detect feature-independent interactions [3, 12]. Logic notations are more expressive and can be used to specify and detect violations of feature-specific behaviour. However, it is difficult to formulate properties that are likely to reveal interactions [14]; one needs to know the possible assertion violations in order to encode and detect them. Hybrid models combine the advantages of both approaches. With the new model, we can detect feature-independent interactions without prior manual analysis of potential interactions, and we can model and check feature-specific assertions.

While we are disappointed in how our model and tools fared in the Feature Interaction Contest, many of our problems were due to ambiguous and oversimplified specifications rather than to weaknesses in our approach. Chisel diagrams nicely depict event sequences, but we question whether traces of observable events are sufficiently expressive and flexible to capture complex telephony behaviour [16].

A weakness of our approach is that some feature specifications are difficult to write because they are tightly coupled with other specifications. For example, sibling features must remain synchronized with one another. Also, many specifications must include transitions that react to seemingly unrelated events (e.g., the user hangs up) in order to be complete. The analysis tools detect coordination interactions, but this only help us ensure that specifications and planned interactions are correct before analyzing combinations of features.

Another problem is the number of messages the tools output. We could better classify the "warnings" that clearly identify interactions (e.g., modified transitions and assertion violations) from those that should be examined by a human analyst. We could also use a sifting tool, such as the Unix program *diff*, to highlight new and missing messages due to the addition of a new feature.

Acknowledgements

Our greatest thanks go to those who worked on previous incarnations of our analysis tools: Kenneth Braithwaite, Keith Pomakis, Pansy Au, and Maggie Cai. Graham Toppin and James Keast helped us reason about extending our model and analysis tools to deal with IN features. We also thank Tom Gray, Serge Mankovskii, Michael Weiss, all of Mitel Corporation, for their support, encouragement and expertise in telephone features and interactions, and the anonymous reviewers for their suggestions on how to improve the paper.

References

[1] P. K. Au and J. M. Atlee. Evaluation of a state-based model of feature interactions. In *Int. Workshop on Feature Interactions in Telecommunications IV*, pages 153–167, 1997.

[2] J. Blom, R. Bol, and L. Kempe. Automatic Detection of Feature Interactions in Temporal Logic. In *Int. Workshop on Feature Interactions in Telecommunications III*, pages 1–19, 1995.

[3] K. H. Braithwaite and J. M. Atlee. Towards Automated Detection of Feature Interactions. In *Int. Workshop on Feature Interactions in Telecommunications II*, pages 36–59, 1994.

[4] G. Burns, P. Mataga, and I. Sutherland. Features as Service Transformers. In *Int. Work. on Feature Interactions in Tele. V*, pages 85–97, 1998.

[5] M. Cai. *Assertion Analysis of Feature Interactions.* Master's thesis, Department of Computer Science, University of Waterloo, 1997.

[6] E.J. Cameron, et al. "Definitions of Services, Features, and Feature Interactions", December 1992. Bellcore Memorandum for Discussion.

[7] P. Combes and S. Pickin. Formalisation of a User View of Network and Services for Feature Interaction Detection. In *Int. Workshop on Feature Interactions in Telecommunications II*, pages 120–135, 1994.

[8] D.O. Keck and P.J. Kuehn. "The Feature and Service Interaction Problem in Telecommunications Systems: A Survey". *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998.

[9] B. Kelly, et al. Service Validation and Testing. In *Int. Workshop on Feature Interactions in Telecommunications III*, pages 173–184, 1995.

[10] A. Khoumsi. Detection and Resolution of Interactions Between Services of the Telephone Network. In *Int. Workshop on Feature Interactions in Telecommunications IV*, pages 78–106, 1997.

[11] F.J. Lin and Y.-J. Lin. A Building Block Approach to Detecting and Resolving Feature Interactions. In *Int. Workshop on Feature Interactions in Telecommunications II*, pages 86–119, 1994.

[12] K. Pomakis and J. Atlee. Reachability analysis of feature interactions. In *Int. Sym. on Software Testing and Analysis*, pages 216–223, 1996.

[13] B. Stepien and L. Logrippo. Representing and Verifying Intentions in Telephony Features Using Abstract Data Types. In *Int. Workshop on Feature Interactions in Telecommunications III*, pages 141–155, 1995.

[14] H. Velthuijsen. Issues of Non-Monotonicity in Feature-Interaction Detection. In *Int. Workshop on Feature Interactions in Telecommunications III*, pages 31–42, 1995.

[15] T. Yoneda and T. Ohta. A Formal Approach for Definition and Detection of Feature Interactions. In *Int. Workshop on Feature Interactions in Telecommunications V*, pages 202–216, 1998.

[16] P. Zave. *Services and Visualization – Towards User-Friendly Design*, chapter 'Calls considered harmful' and other observations: A tutorial on telephony, pages 8–27. Springer-Verlag, 1998.