Composing Features and Resolving Interactions

Jonathan D. Hay
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1 Canada
jd2hay@se.uwaterloo.ca

Joanne M. Atlee
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1 Canada
jmatlee@se.uwaterloo.ca

ABSTRACT

One of the accepted techniques for developing and maintaining feature-rich applications is to treat each feature as a separate concern. However, most features are not separate concerns because they override and extend the same basic service. That is, "independent" features are coupled to one another through the system's basic service. As a result, seemingly unrelated features subtly interfere with each other when trying to override the system behaviour in different directions. The problem is how to coordinate features' access to the service's shared variables.

This paper proposes coordinating features via feature composition. We model each feature as a separate labelled-transition system and define a *conflict-free (CF) composition* operator that prevents enabled transitions from synchronizing if they interact: if several features' transitions are simultaneously enabled but have conflicting actions, a non-conflicting subset of the enabled transitions are synchronized in the composition. We also define a *conflict- and violation-free (CVF) composition* operator that prevents enabled transitions from executing if they violate features' invariants. Both composition operators use priorities among features to decide whether to synchronize transitions.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques, Assertions, Invariants*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*process models*; D.2.12 [**Software Engineering**]: Interoperability; D.2.11 [**Software Engineering**]: Software Architecture—*domain-specific architectures*

General Terms

Design

Keywords

Modularity, incremental programming, feature interaction, conflict resolution

1. INTRODUCTION

Software engineering techniques such as separation of concerns and incremental programming help software developers to build feature-rich applications by allowing them to consider new features as independent increments of the application's basic service. However, separately developed features can subtly interfere with each other because they enhance the same basic service and manipulate the same service variables. In general, a *feature interaction* occurs whenever one feature affects the behaviour of another [11].

Not all interactions are bad. Many are planned. For example, features are designed to interact with the basic service and any features that they override and extend. Conflicts occur when seemingly unrelated features trip over each other while trying to simultaneously override system behaviour. The problem is that while features are treated as separate concerns, they are coupled to one another through the service that they extend and their access to the shared service is not well coordinated.

The *feature interaction problem* is how to rapidly develop and deploy new features without disrupting the functionality of existing features. Approaches to this problem include the following

Eliminating interactions during design. The idea is to formally specify features as separate entities, compose the specifications, search the composition for interactions, and resolve interactions by modifying the feature specifications [1, 3, 6, 8, 15, 24]. However, features cannot be re-designed *independently* to eliminate interactions. Also, if the system is evolving, many of the features under consideration may already be implemented and not open to re-design [30]. Instead, most design-time resolutions specify how groups of features behave together, either by adding exception clauses to features [3, 29], adding supervisors that explicitly interleave the actions of feature combinations [2, 26], adding rules or features that explicitly define the behaviour of feature combinations [5, 29], etc. The problem with such resolution techniques is that they don't scale to systems that have hundreds or thousands of features because the number of feature combinations to consider grows exponentially. Worse, by specifying the behaviour of feature combinations, they counteract the benefits of separation of concerns, making it difficult to resolve interactions with future features or to customize resolutions to an individual user's needs.

Preventing interactions via architectural constraints. An architecture can help avoid interactions by constraining and coordinating the features' access to the service's variables and resources. Distributed Feature Coordination (DFC) [20] organizes features in a pipe-and-filter architecture where the messages that are passed among the features act as tokens that enable features to react to the current system state. However, pipe-and-filter and layered architectures [6, 7] tend to serialize features' reactions to each event. This may over-constrain the features' access to the basic service and cause features to miss key events that they would otherwise react to. Also, it means each feature specification must indicate which input events are terminated at the feature (i.e, are not forwarded to the next feature), and such decisions may need to be modified when new features are introduced to the system [9]. Agent-based architectures [23, 32] and patterns [27] allow features more flexible access to the basic service and shared resources. However, architectural solutions cannot prevent interactions that violate features' constraints.

Resolving interactions at run-time. If an interaction is not eliminated during design nor avoided by the system's architecture, it must be detected and resolved at run-time. Some resolutions are intentionally deferred until run-time – until it can be determined that an interaction has actually occurred and corrective action is needed. Once detected, interactions can be resolved by ranking features by priority [19, 22], negotiating compromises[18], involving the user [16], rolling back conflicting actions [9, 22], disabling feature activation [19], terminating features [22], and terminating the application.

Our goal is to develop a feature specification and composition model that has the advantages of all three approaches: a model that enforces modular feature development, co-ordinates the features' accesses to the basic service, and constrains features' behaviours only when they conflict. We want to maintain the illusion that features are separate concerns because the illusion simplifies the application's design. The illusion also decreases the time-to-market for new features, because features can be developed in parallel or contracted out to third-party programmers. This emphasis on modularity rules out conflict resolution techniques that modify service or feature specifications or that specify the behaviour of feature combinations. However, the trick in addressing the feature interaction problem is not in finding resolutions, but in developing general mechanisms to detect and resolve interactions that are efficient and that apply in a majority of cases [21].

This paper introduces a model for specifying and composing features that detects and resolves interactions during feature composition. The system's state is abstracted as a set of relational *assertions* about the system's observable behaviour. For example, assertions in a telephony system include established voice connections, partial connections, connections on hold, etc. Features are specified as labelled transition systems that model the features' effects on system behaviour as changes to and constraints on the values of the assertion variables. Thus, feature interactions manifest themselves as conflicts over the values of these assertions. We propose composition operators that resolve conflicting actions and violations of constraints, using feature priorities to favour one feature over another.

The paper is organized as follows. First, we describe the types of assertions we use to model system behaviour and our notation for specifying feature behaviour in terms of the features' effects on assertions. We then incrementally present our prioritized composition operators and apply them to a few telephony features. Although all of the examples used in this paper involve telephony services, features, and interactions, our model can be applied to any feature-rich reactive system.

2. ASSERTIONS

The state of a system is modelled as a set of relational *assertions* that characterize key information about the system's observable behaviour. Example assertions in a telephony system are relations on connections **C**, users **U**, resources **R**, time **T**, money **$**, etc. Assertions about a connection $c \in \mathbf{C}$ relate all of the users, billing information, shared resources, and other information associated with a particular call.

As services and features execute, they monitor and modify the assertions' values. We say that a service or feature *asserts* a new relationship when it adds a relationship instance to an assertion variable (e.g., asserting a new connection would add a new connection element to the relation of established connections). A service or feature *retracts* a relationship when it removes a relationship instance from an assertion.

For convenience, we divide assertions into *state assertions*, *event assertions*, and *ordinary assertions*.

State and event assertions differ from ordinary assertions in that there are extra constraints on how and when they change value.

2.1 States

State assertions are relations that describe the control state of a service or a feature. Every executing service and feature has a current state, and is thus represented as an element in some state assertion variable.

Consider Plain Old Telephone Service (POTS), which provides voice connections between pairs of users. We model POTS as two coordinating services: O_POTS, which provides service to callers, and T_POTS, which provides service to callees[1]. A caller o in connection c is in exactly one of the following states of O_POTS. In the following, the left column lists states and the right column gives the states' definitions. We distinguish among callers ($o, o1, o2, \ldots \in$ **U**), callees ($t, t1, t2, \ldots \in$ **U**), and arbitrary users ($u, u1, \ldots \in$ **U**) who may be either callers or callees.

o_newcall(c, o)	Start a new call for caller o
o_dial(c, o)	Wait for o to dial number
o_analyze(c, o, n)	Analyze dialed number n
o_request(c, o, t)	Request a connection
o_wait(c, o, t)	Wait for response to request
o_ring(c, o, t)	Alert o that t's phone is ringing
u_voice(c, u)	Voice connection is established
u_dead(c, u)	Connection is dead; wait for hang up
u_disconn(c, u)	User hangs up

A callee t in connection c is in exactly one of the following states of T_POTS. (Note that the last three states are identical to the last three states of O_POTS.)

t_newcall(c, o, t)	Process new call request
t_ring(c, o, t)	t's phone is ringing
u_voice(c, t)	Voice connection is established
u_dead(c, t)	Connection is dead; wait for hang up
u_disconn(c, t)	User hangs up

[1]The names O_POTS and T_POTS refer to the names of the basic call model, Originating Call Model and Terminating Call Model, used in telephony. The Originating Call Model is so called because it "originates" calls on behalf of the caller. The Terminating Call Model is so called because a call request may establish several network connections between the caller's exchange and the callee's exchange, but eventually "terminates" at and is handled by this service at the callee's exchange.

In addition, a feature specification may introduce state assertions that reflect the various stages of its execution. For example a Call Waiting feature executing on behalf of a callee t is in exactly one of the following state assertions.

cw_newcall(c, t)	Accept second call
cw_notify(c, t)	Notify t of second call
cw_twocalls(c1,c2,t)	User t connected to two calls

The values of state assertions are constrained so that an executing service or feature is always in exactly one of its state assertions. Thus, if a service's (feature's) state is asserted, all of its other state assertions are implicitly retracted.

2.2 Events

Event assertions are relations that model events, messages, and other transient assertions. Examples include:

OffHook(u)	User picks up the phone's handset
OnHook(u)	User hangs up
Dial(o,t)	User o dials user t's number
DialTone(u)	DialTone emitted to user u
CW_Notify(u)	User u is notified of a Call Waiting call

Event assertions are distinguished from ordinary assertions because they are transient. They are asserted for one system state – long enough for the services and features to react to their occurrence – and then are automatically retracted.

2.3 Ordinary Assertions

(Ordinary) assertions are relations that model observable properties of the system. Some assertions model POTS-specific or feature-specific data. For example, POTS keeps track of which phone numbers are valid and which users are allowed to initiate or receive calls; Call Forwarding keeps track of forwarded numbers; etc.

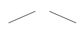
CanCall(u)	User u may originate calls
CanReceive(u)	User u may receive calls
InService(u)	User u's phone number is in service
OCS(o, t)	User o screens calls made to user t
TCS(t, o)	User t screens calls from user o
CFNA(t, t2, n)	Unanswered calls to t are forwarded to $t2$ after n time units

Other assertions model observable properties of the system's execution.

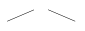
Conn(c, u)	User u participates in connection c
Alias(u1,u2)	$u1$ and $u2$ refer to the same user[2]
Redirect(t, t2)	Calls to t are redirected to user $t2$
Reveal(c, u1, u2)	$u1$'s phone number is revealed to $u2$.
Hold(c, u)	User u places connection c on hold.
CWHold(c, u)	User u puts connection c on hold to answer second call
3WayHold(c, u)	User u puts connection c on hold to start second call

As features introduce variations of the same ordinary assertion, it becomes necessary to organize the assertions into generalization hierarchies. For example, Call Waiting (CW) and Three-Way Calling (3WC) both have stages in their execution where a user places one connection on hold. By modelling feature-specific assertions CWHold and 3WCHold, each feature can assert or retract its respective hold assertion without affecting hold assertions issued by other features. The advantage of having an abstract Hold assertion is that a feature like 911 can forbid a connection from being put on hold without listing all of the forbidden feature-specific assertions. Otherwise, we would need to modify 911 every time a new feature introduced a new type of hold.

An assertion hierarchy defines concrete assertions as instantiations of abstract assertions.



Childless nodes represent concrete assertion variables that can be directly manipulated by services and features. Internal nodes represent abstract assertions whose values cannot be explicitly asserted or retracted. Instead, an abstract relationship is indirectly asserted whenever one of its concrete instantiations is asserted, and is retracted when all of its descendents are retracted. Example hierarchical assertions include:



Orig: callers in a connection
Term: callees in a connection

CFNA: call forward (no answer)
CFB: call forward (when busy)
VoiceMail: Voice Mail

In our approach, the introduction of new abstract assertions poses the greatest threat to modular feature development because it reflects that a new observable property has been added to the system. Adding a new abstract assertion does not necessitate changes to existing features, but it does present an opportunity to enhance existing features so that they take advantage of the new shared variable. For example, the specification for Call Waiting introduces all of the assertions it needs to specify how its user receives and alternates between two calls. However, the introduction of an abstract Reveal assertion may lead to a new Call Waiting Display feature that displays the second caller's number. In general, proactive feature designers will attempt to invent new features whenever a new abstract assertion is introduced, by integrating the new assertion into existing features.

3. SPECIFICATIONS

Specifications describe how services and features affect assertion values. They consist of sets of transition rules whose guards describe assertion valuations that the service or feature reacts to and whose actions describe the transitions' corresponding reactions.

Definition 1. A *service* or *feature* is a labelled transition system $(A, \longrightarrow)$, where

- A is a finite set of state, event, and ordinary assertions
- $\longrightarrow$ is a set of transitions with labels g, Δ, where
  - g is a *guard* that characterizes the states in which the transition is enabled. Guards are predicates defined by the following grammar:

    $g ::= \mathrm{R}(e,e) \mid \mathbf{!}\mathrm{R}(e,e) \mid +\mathrm{R}(e,e) \mid -\mathrm{R}(e,e) \mid g,g \mid g|g$
    $e ::= \mathrm{x} \mid !\mathrm{x} \mid *$

    where $\mathrm{R} \in A$ is an assertion variable, x is an element in R's domain or range, and $*$ is a wildcard that matches

[2] A user is identified by his telephone number; if a user has more than one number, he is treated as a user who has multiple aliases.

any element in R's domain or range. The operators '**,**' and '|' specify conjunction and disjunction, respectively, with disjunction having higher precedence than conjunction.

- Δ is a set of actions that specify changes to assertion values when the transition executes. Actions have the following grammar:

  a ::= +R(i,i) $\mid$ –R(d,d) $\mid$ a, a
  i ::= x
  d ::= x $\mid$!x $\mid$ $*$

  The operator '**,**' specifies action composition[3].

The guard R(x,y) holds whenever (x,y)∈R, and its complement **!**R(x,y) holds whenever (x,y)∉ R. Wildcard elements $*$ and !x are used to write existentially quantified guards:

$$\begin{array}{lcl} \text{R(x,*)} & \equiv & \exists y.((x,y) \in R) \\ \textbf{!}\text{R(x,*)} & \equiv & \neg\exists y.((x,y) \in R) \\ \text{R(x,!z)} & \equiv & \exists y.(y \neq z \wedge (x,y) \in R) \\ \textbf{!}\text{R(x,!z)} & \equiv & \neg\exists y.(y \neq z \wedge (x,y) \in R) \end{array}$$

Guards +R(x,y) and –R(x,y) hold when the predicate R(x,y) has just changed value: +R(x,y) holds whenever R(x,y) is true but was false in the previous state, and –R(x,y) holds whenever R(x,y) is false but was true in the previous state.

Actions either add new elements to or remove elements from assertion relations. Action +R(x,y) adds the element-pair (x,y) to assertion R, and –R(x,y) removes the element-pair (x,y) from the assertion. Quantified actions specify the removal of several elements from the assertion: action –R(x,*) removes from the relation all pairs whose domain is x. And action –R(x,!z) removes from the relation all pairs whose domain is x – except for the pair (x,z), whose membership in R is not affected by this action.

To improve the readability of service and feature specifications, we use delimiters and fonts to distinguish between state, event, and ordinary assertions: states are surrounded by square brackets [ ], events are italicized, and ordinary assertions are not delimited.

Example 1. O_POTS
The following are a few rules from the service specification of O_POTS.

Offhook(o), **!**Conn(*,o) $\longrightarrow$ [+o_newcall(c,o)], +Orig(c,o)
[+o_newcall(c,o)], CanCall(o)$\longrightarrow$[+o_dialing(c,o)],+*DialTone(o)*
[o_dialing(c,o)], *Dialed(o,t)* $\longrightarrow$ [+o_analyze(c,o,t)]
[+o_analyze(c,o,t)], InService(t) $\longrightarrow$ [+o_request(c,o,t)]
[+o_analyze(c,o,t)], **!**InService(t) $\longrightarrow$
[+o_dead(c,o)],+*WrongNumber(o)*

The first rule is triggered if the caller picks up the phone's handset and is not already involved in a call. As soon as the call enters state [o_newcall], the service determines whether the caller has permission to initiate calls. If so, the service proceeds to state [o_dialing] and a dial tone is issued. The third rule is triggered when the caller dials a number. The fourth and fifth rules describe the behaviours when the dialed number is and is not a valid number in service, respectively; in the latter case, an announcement is made to the caller. □

[3]While the grammars for guards and actions are expressed in terms of binary relations, assertion variables can also be sets and relations of higher order.

Scoping rules restrict which assertions the services and features may access. A service specifications may reference and modify only the assertions that it declares (i.e., the basic service's states, events, and assertions). A feature specification may reference and modify the new assertions that it declares (i.e., the feature's states, new input and output events, and assertions for new types of connections, screening lists, etc.) plus the service's assertions, plus the abstract assertions. Thus, features override the behaviour of the basic service by manipulating the abstract and service assertions; and features extend the system's behaviour by introducing new assertions and manipulating their values. If a feature overrides or extends the behaviour of another feature (e.g., Call Display Blocking overrides Call Number Display), it can access the second feature's assertions as if the second feature were part of its underlying service. In general, features should not access each other's declared assertions.

Example 2. Originating Call Screening
Originating Call Screening is invoked as soon as the caller's service enters state [o_analyze] to test the dialed number.

[+o_analyze(c,o,t)], OCS(o,t) $\longrightarrow$ [+o_dead(c,o)], *OSCDeny(o)*

If the dialed number is on the caller's screening list, then the feature terminates the call (by forcing the O_POTS service to transition to state [o_dead]) and sends an error message to the caller. □

Definition 2. Semantics
The *system state* of a service or feature is an assertion valuation $\mathcal{A}$ of the service's (feature's) assertions A. The *semantics* of a service (feature) is defined by a transition relation between system states. A transition $\mathcal{A} \xrightarrow{g\Delta} \mathcal{A}'$ executes if the assertion valuation $\mathcal{A}$ satisfies the transition's guards:

$$\mathcal{A} \models g$$

and if the actions Δ, when applied to assertion valuation $\mathcal{A}$, produce the new assertion valuation $\mathcal{A}'$:

$$\Delta\mathcal{A} = \mathcal{A}'$$

We derive for each transition a predicate that characterizes how the transition's actions are expected to affect assertion values. For example, if a transition's actions assert +R(x,y), then we expect $(x, y) \in R$ to hold after the transition executes, and if the actions assert –R(x,y), then we expect $(x, y) \notin R$ to hold. The effects e_Δ of a transition's actions Δ is the conjunction of such relation-membership tests, one test for every element added or removed from a relation.

To ensure that the actions applied by a single transition do not interfere with each other, we impose a correctness criterion on specifications that states a transition's effects must hold in the transition's destination state

CORRECTNESS CRITERION 1. *For all transitions $\mathcal{A} \xrightarrow{g\Delta} \mathcal{A}'$,*
$$\Delta\mathcal{A} \models e_\Delta$$

4. FEATURE COMPOSITION

A complete telephony system consists of several services and features executing in parallel. At any point in time, there could be several instances of the basic service, each executing on behalf of some user, and several instances of various features, each overriding or extending some service instance. In practice, we are interested in composing only the services and features pertaining to groups of related calls (i.e., calls that have common users).

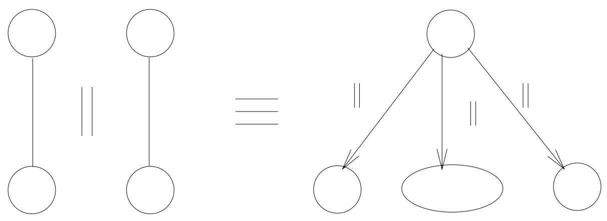


Figure 1: Composition of parallel features

When composing features, we want to synchronize enabled transitions to maximize the number of features that react to each system state. *AND-synchronization* [4] satisfies this requirement: it synchronizes transitions exactly when the conjunction of the transitions' guards is enabled. Thus, if transitions t_1 and t_2 from parallel features (Figure 1) are both enabled by assertion valuation $\mathcal{A}$, then they execute synchronously and modify assertion values simultaneously. Otherwise, one transition, or the other, or neither executes.

Definition 3. Composition with AND-synchronization synchronizes two transitions $t_1 = \mathcal{A} \xrightarrow{g_1, \Delta_1} \mathcal{A}'$ and $t_2 = \mathcal{A} \xrightarrow{g_2, \Delta_2} \mathcal{A}''$ iff the transitions' guards are simultaneously enabled (the $\circ$ operator denotes function composition). A transition t executes alone ($\| t$) if it is the only enabled transition.

$$\begin{aligned} t_1 \| t_2 &= g_1 \wedge g_2, \;\; \Delta_1 \circ \Delta_2 \\ \| t_1 &= g_1 \wedge \neg g_2, \;\; \Delta_1 \\ \| t_2 &= g_2 \wedge \neg g_1, \;\; \Delta_2 \end{aligned}$$

Note that the enabling conditions of transitions $\| t_1$ and $\| t_2$ give synchronized transitions priority over interleaved transitions.

The problem with this definition for feature composition is that synchronized transitions may assert conflicting actions.

Definition 4. A *feature interaction* occurs when two transitions $t_1 = \mathcal{A} \xrightarrow{g_1, \Delta_1} \mathcal{A}'$ and $t_2 = \mathcal{A} \xrightarrow{g_2, \Delta_2} \mathcal{A}''$ are synchronized, but their actions' expected effects are not realized:

$$\begin{aligned} \mathcal{A} &\models g_1 \wedge g_2 \\ \Delta_1 \Delta_2 \mathcal{A} &\not\models e_{\Delta_1} \wedge e_{\Delta_2} \end{aligned}$$

The actions in Δ_1 and Δ_2 counteract one another, either by asserting that some service or feature must transition into different states, or by asserting and retracting the same assertion. The result is that the effects e_{Δ_1} and e_{Δ_2} are not both realized after the actions are executed. The composition order of action sets Δ_1 and Δ_2 may affect which effects predicate is not realized, but it does not affect the value of the predicates' conjunction.

4.1 Conflict-Free Composition

We propose an alternate rule for synchronizing transitions called *CF-synchronization* that allows features to simultaneously react to a particular call situation, but *disables* transition combinations that conflict. We assume a total priority ordering on features, and we use these feature priorities to select which of the conflicting transitions will execute.

Consider two rules t_1 and t_2, where t_2's feature has priority over t_1's feature. We allow transitions t_1 and t_2 to synchronize if and only if their guards are simultaneously enabled and their actions do not conflict. If the transitions conflict, then transition t_2, from the higher priority feature, executes alone; it also executes alone if only its guard is enabled by the current assertion valuation. Transition t_1, from the lowest priority feature, executes alone only when it is the sole transition enabled by the current assertion valuation.

Definition 5. Composition with CF-synchronization synchronizes two transitions $t_1 = \mathcal{A} \xrightarrow{g_1, \Delta_1} \mathcal{A}'$ and $t_2 = \mathcal{A} \xrightarrow{g_2, \Delta_2} \mathcal{A}''$ iff their guards are enabled and their actions do not conflict. A transition t executes alone ($\ll t$) if it conflicts with lower-priority features or if it is the only enabled transition.

$$\begin{aligned} t_1 \ll t_2 &= g_1 \wedge g_2 \wedge (\Delta_1 \Delta_2 \mathcal{A} \models e_{\Delta_1} \wedge e_{\Delta_2}), \;\; \Delta_1 \circ \Delta_2 \\ \ll t_2 &= g_2 \wedge (\neg g_1 \mid (\Delta_1 \Delta_2 \mathcal{A} \not\models e_{\Delta_1} \wedge e_{\Delta_2})), \;\; \Delta_2 \\ \ll t_1 &= g_1 \wedge \neg g_2, \;\; \Delta_1 \end{aligned}$$

The interaction check in the above guard expressions simply evaluates a number of relation-membership tests and thus is not computationally expensive.

In general, given a collection of transitions enabled by an assertion valuation $\mathcal{A}$, we can find a maximal combination of non-conflicting transitions by applying the transitions in order of decreasing priority. Each transition is considered in turn and is selected for execution if and only if its actions do not conflict with the actions of transitions that have already been selected (since the selected transitions are of higher priority and would win any conflict). In the end, the set of transitions selected for execution is a maximal, non-conflicting subset of the enabled transitions. This algorithm allows the maximum number of features to react to a system state, while avoiding conflict.

Generalized Algorithm
CF Composition

```
Input: $\mathcal{A}$ {the current assertion valuation}
       $\mathcal{T}$ {prioritized set of enabled transitions}

Algorithm:
    Let $\mathcal{A}' = \mathcal{A}$   {next assertion valuation}
    Let $\mathcal{T}' = \emptyset$   {set of non-conflicting transitions}
    Let $E = true$ {effects of non-conflicting transitions}

    For each $t \in \mathcal{T}$ in order of decreasing priority
        If $\Delta_t \mathcal{A}' \models E \wedge e_{\Delta_t}$ then
            $\mathcal{A}' = \Delta_t \mathcal{A}'$
            $\mathcal{T}' = \mathcal{T}' \cup \{t\}$
            $E = E \wedge e_{\Delta_t}$
        { else ignore t }

Output: $\mathcal{A}', \mathcal{T}'$
```

5. WEAK INVARIANTS

The above feature model and composition operator resolves only those interactions in which features attempt to simultaneously modify the same assertion variables. More subtle feature interactions occur when one feature makes what are supposed to be persistent changes to the assertion variables, but another feature at a later time modifies those variables. To reflect features' intentions to constrain assertion values beyond the execution of a single transition, we augment our feature specification notation by adding *weak invariants* to transition rules. Weak invariants specify constraints on assertion values that are expected to hold until the invariants are explicitly retracted by the feature that asserted them. We call the invariants *weak* because they can be overridden by transitions of higher-priority features.

Definition 6. A *weak invariant* is a constraint on assertion values that is expected to hold from the time the invariant is asserted to the time it is retracted. Weak invariants are specified in three parts:

$$\textit{Name} = \textit{Condition} : \textit{Violation Event}$$

where *Name* is a unique identifier that may have parameters, the *Condition* specifies a constraint on assertion values, and the *Violation Event* is an event assertion that is output whenever the invariant prevents a violating action from occurring. Invariant conditions have the following grammar:

$I ::= \forall X.(I) \mid E$
$E ::= \mathrm{R}(e,e) \mid !\mathrm{R}(e,e) \mid E, E \mid E|E$
$e ::= \mathrm{x} \mid X$

where $\mathrm{R} \in A$ is an assertion variable, x is an element in R's domain or range, and X is a bound variable over elements in R's domain or range. The operators ' ,' and '|' specify conjunction and disjunction, respectively, with disjunction having higher precedence.

Weak invariants can be asserted or retracted as a side effect of a transition rule. Thus, we extend our specification model to include *actions on invariants*.

Definition 1. (Revised) A *service* or *feature* is a labelled transition system (A,$\longrightarrow$), where

- A is a finite set of state, event, and ordinary assertions
- $\longrightarrow$ is a set of transitions with labels g, Δ, δ where
  - g is a guard on assertions in A
  - Δ is a set of actions
  - δ is a set of *actions on invariants* that assert and retract weak invariant constraints

Weak invariants express the features' intentions to control assertion values. However, a feature's invariants can be overridden by the actions of a higher-priority feature. When this happens, the invariant is said to be *relaxed* – the invariant is violated, but remains asserted in order to keep lower-priority features from re-violating it. For example, a high-priority emergency call could violate a call screening invariant, but the relaxed invariant would continue to screen connection attempts made by lower-priority features.

We delimit weak invariants in transition rules with angle brackets $<>$, to distinguish them from guards and actions. An invariant named I may be asserted +<I> or retracted –<I> as a side effect of a service or feature transition.

Example 3. Originating Call Screening
The intention of the Originating Call Screening feature is to constrain which users a caller o can be connected to:

OSCInv(c,o)=$\forall U$.(!OSC(o,U)|!Conn(c,o)|!Conn(c,U)) :*OSCDeny*

Every user U is either not on o's screening list or not in a connection with o. Whenever the invariant prohibits a requested connection, an *OSCDeny* event is sent to the user whose feature tried to establish the screened connection. This invariant is enforced whenever user o is considered the originator of a connection, either via POTS or some other feature (e.g., Call Transfer):

+Orig(c,o) $\longrightarrow$ +<OSCInv(c,o)>
–Orig(c,o) $\longrightarrow$ –<OSCInv(c,o)> □

Example 4. 911
One of the constraints of being connected to a 911 operator is that only the 911 operator can put the call on hold or end the call.

911Hold(c) = $\forall U$.(!Hold(c,U)) : *911DenyHold*
911Conn(c) = $\forall U$.([u_voice(c,U)]) : *911DenyHangup*

These invariants are asserted whenever a user $u2$ establishes a connection with a 911 operator and are retracted whenever the 911 operator ends the call.

[+u_voice(c,u2)], Conn(c,u), 911(u) $\longrightarrow$
  [+911_Call(c,u)], +<911Conn(c), 911Hold(c)>
[911_Call(c,u)],*OnHook(u)* $\longrightarrow$ –<911Conn(c), 911Hold(c)> □

Because the set of asserted invariants varies and (as will be seen) affects the behaviour of services and features, we revise our definition of semantics to incorporate weak invariants.

Definition 2. (Revised) The *system state* of a service or feature is a pair ($\mathcal{A}$, $\mathcal{I}$), where $\mathcal{A}$ is a valuation of the service's and feature's assertions A, and $\mathcal{I}$ is a set of asserted invariants. We define a transition relation between system states, such that transition $(\mathcal{A},\mathcal{I}) \xrightarrow{g,\,\Delta,\,\delta} (\mathcal{A}',\mathcal{I}')$ executes if (1) the assertion valuation $\mathcal{A}$ satisfies the transition's guards:

$$\mathcal{A} \models g$$

(2) the actions Δ, when applied to assertion valuation $\mathcal{A}$, produce the new assertion valuation $\mathcal{A}'$:

$$\Delta\mathcal{A} = \mathcal{A}'$$

and (3) the actions on invariants δ, when applied to invariant set $\mathcal{I}$, produce the new invariant set $\mathcal{I}'$:

$$\delta\mathcal{I} = \mathcal{I}'$$

5.1 Violating (and Re-violating) Invariants

At this point, we would like to assert a second correctness criterion on service and feature specifications stating that a transition's actions must not interfere with the transition's own asserted invariants. One might try to express this correctness criterion by evaluating the invariants before and after the transition executes:

$$\left(\mathcal{A} \models \bigwedge_{J\in\delta} J\right) \rightarrow \left(\Delta\mathcal{A} \models \bigwedge_{J\in\delta} J\right)$$

However, if any of the invariants $J \in \delta$ is unsatisfied when the transition executes, the above expression will evaluate to *true*, whether or not the transition's pwn actions Δ violate the invariant. Similarly, if we try to use only the conclusion of the above expression, this subexpression will evaluate to *false* whenever δ contains an unsatisfied invariant. We are interested in detecting new violations in the presence of relaxed invariants, which means that we want to detect when *some instantiation* of an invariant *becomes false*.

To do this, we evaluate invariants with respect to a pair of consecutive assertion valuations $(\mathcal{A}, \mathcal{A}')$. We define a four-valued assignment function v_4 on atomic expressions R(x,y) in assertion valuation $\mathcal{A}'$, such that

$$\begin{array}{rlll} v_4(R(x,y)) = & \mathrm{T} & \text{iff} & (x,y)\in R \text{ and } (x,y)\in R' \\ v_4(R(x,y)) = & +\mathrm{T} & \text{iff} & (x,y)\notin R \text{ and } (x,y)\in R' \\ v_4(R(x,y)) = & +\mathrm{F} & \text{iff} & (x,y)\in R \text{ and } (x,y)\notin R' \\ v_4(R(x,y)) = & \mathrm{F} & \text{iff} & (x,y)\notin R \text{ and } (x,y)\notin R' \end{array}$$

where R denotes the value of a relation in assertion valuation $\mathcal{A}$ and R' denotes the value of the same relation in valuation $\mathcal{A}'$.

R(x,y)	!R(x,y)
T	F
+T	+F
+F	+T
F	T

,	T	+T	+F	F
T	T	+T	+F	F
+T	+T	+T	F	F
+F	+F	F	+F	F
F	F	F	F	F

|	T	+T	+F	F
T	T	T	T	T
+T	T	+T	T	+T
+F	T	T	+F	+F
F	T	+T	+F	F

Table 1: Four-valued-logic truth tables for non-membership, conjunction, and disjunction.

The values of expressions !R(x,y), conjunctions, and disjunctions are defined in truth tables in Table 1. For convenience, we define an operator $\downarrow$ on universally quantified invariant expressions that is true whenever some instantiation of the expression evaluates to +F:

$$\downarrow \forall X.(I) \text{ iff } \exists X.(v_4(I) = +\mathrm{F})$$

Our correctness criterion, then, states that a transition's actions should not cause any instantiation of the transition's own asserted invariants to become false.

CORRECTNESS CRITERION 2. *For all invariants $+I \in \delta$ asserted by a transition $(\mathcal{A},\mathcal{I}) \xrightarrow{g,\Delta,\delta} (\mathcal{A}',\mathcal{I}')$,*

$$(\mathcal{A}, \Delta\mathcal{A}) \not\models \downarrow I$$

Correctness Criteria 1 and 2 ensure than any interactions detected and resolved when combining transitions are inter-feature interactions and not intra-feature interactions.

5.2 Conflict and Violation-Free Composition

We propose a second composition operator that supports *Conflict- and Violation-Free (CVF) synchronization.* This operator resolves four types of feature interactions:

action conflicts - This is the same interaction we had previously, in which two transitions' actions attempt to change assertion variables to conflicting values:

$$\Delta_1\Delta_2\mathcal{A} \not\models e_{\Delta_1} \wedge e_{\Delta_2}$$

As before, CVF composition resolves action conflicts by executing the maximal set of non-conflicting transitions, using feature priorities to select the transitions.

assertions violate invariant - This occurs when a rule asserts an invariant $+I \in \delta$ that does not hold in the current assertion valuation:

$$\mathcal{A} \not\models I$$

We allow such rules to execute and assert their weak invariants as relaxed invariants. Future actions may cause the relaxed invariants to become satisfied. More importantly, relaxed invariants can continue to prevent actions from lower-priority features from re-violating the invariant.

actions violate invariant - This occurs when the actions Δ_1 of an enabled rule violate either an invariant in the current state or an invariant asserted by a simultaneously enabled rule.

$$(\mathcal{A}, \Delta_1\mathcal{A}) \models \bigvee_{J \in \mathcal{I}} \downarrow J \qquad (\mathcal{A}, \Delta_1\Delta_2\mathcal{A}) \models \bigvee_{J \in \delta_2\mathcal{I}} \downarrow J$$

We use feature priorities to resolve these interactions. If the violating actions are from a rule whose feature has a priority equal to or higher than that of the violated invariant's feature, then the rule is executed and the invariant is relaxed. Otherwise, the rule is not selected for execution.

invariants conflict - A newly asserted invariant $+I \in \delta$ is said to conflict with other asserted invariants if it causes the conjunction of invariants in $\delta\mathcal{I}$ to become false:

$$\models \bigwedge_{J \in \mathcal{I}} J \quad \text{but} \quad \not\models \bigwedge_{J \in \delta\mathcal{I}} J$$

We allow such rules to execute for the same reason that we allow rules that assert violated invariants to execute: they may keep lower-priority features from performing actions that violate the invariants. Since the conflicting invariants have different priorities, the lower-priority invariants are the ones that are relaxed. But they are ready to take effect if the higher-priority invariants are retracted before they are.

Figure 2 contains the formal definition of CVF composition for two transitions. In general, for a given set of enabled transitions, we can find a maximal combination of non-conflicting, non-violating transitions by analyzing the transitions in order of decreasing priority. A transition is selected for execution if and only if its actions do not conflict with the actions of transitions that have already been selected and its actions do not violate any invariant asserted by a higher-priority feature. When all of the enabled transitions have been considered, the set of transitions selected for execution is our maximal set of synchronized transitions.

Generalized Algorithm
CVF Composition

```
Input:  $\mathcal{A}$  {the current assertion valuation}
        $\mathcal{I}$  {set of asserted invariants $I$, with priority $p(I)$}
        $\mathcal{T}$  {set of enabled transitions $t$, with priority $p(t)$}

Algorithm:
  Let $\mathcal{A}' = \mathcal{A}$  {next assertion valuation}
  Let $\mathcal{I}' = \mathcal{I}$  {next set of asserted invariants}
  Let $\mathcal{T}' = \emptyset$  {set of transitions selected for execution}
  Let $E = true$ {effects of the selected transitions}

  For each $t \in \mathcal{T}$ in order of decreasing priority
    If $\Delta_t\mathcal{A}' \models E \wedge e_{\Delta_t}$ then
      If $\forall I \in \mathcal{I}', (((\mathcal{A}, \Delta_t\mathcal{A}') \not\models \downarrow I)$ or $(p(t) > p(I)))$ then
        $\mathcal{T}' = \mathcal{T}' \cup \{t\}$
        $\mathcal{A}' = \Delta_t\mathcal{A}'$
        $\mathcal{I}' = \delta_t\mathcal{I}'$
        $E = E \wedge e_t$
      { else ignore t }
    { else ignore t }

Output: $\mathcal{A}', \mathcal{I}', \mathcal{T}'$
```

Example 5. Call Forward vs. Originating Call Screening
Call Forward on Busy (CFB) has a rule that redirects calls if its subscriber receives a call when he is already on the phone.

CFB:
CallRequest(c2,o,t), Conn(c,t), CFB(t,t2) $\rightarrow$ [+o_analyze(c2,o,t2)]

Rules for Originating Call Screening (OSC) appear in Examples 2 and 3. Suppose a caller o screens calls to user $t2$ and user t has forwarded his calls to $t2$. If caller o phones t while t is talking on the phone, CFB and T_POTS both react to the CallRequest event. However, because Call Forwarding has priority over T_POTS (features have priority over the basic service), the feature forwards the

Definition 3. Composition with CVF-synchronization synchronizes two transitions $t_1 = \mathcal{A} \xrightarrow{g_1,\Delta_1,\delta_1} \mathcal{A}'$ and $t_2 = \mathcal{A} \xrightarrow{g_2,\Delta_2,\delta_2} \mathcal{A}''$ iff (1) their guards are enabled, (2) their actions do not conflict with each other, and (3) their actions to not violate any invariants asserted by higher-priority features. Otherwise, the transition from the higher-priority feature, say t_2, executes alone ($\ll t_2$) if it is enabled and its actions do not violate any higher-priority invariants. Otherwise, the transition from the lower-priority feature executes alone, if it is enabled and its actions do not violate any higher-priority invariants. In the following, p is a function that maps a transition or invariant to the priority of the feature that executes the transition or asserts the invariant.

$$t_1 \ll t_2 \;=\; g_1 \wedge g_2 \wedge (\Delta_1\Delta_2\mathcal{A} \models e_{\Delta_1} \wedge e_{\Delta_2}) \wedge \left((\mathcal{A},\, \Delta_1\Delta_2\mathcal{A}) \not\models \bigvee_{\substack{J \in \delta_2\mathcal{I}\, \wedge \\ p(J) > p(t_1)}} \downarrow J \right),\;\; \Delta_1 \circ \Delta_2,\;\; \delta_1 \circ \delta_2$$

$$\ll t_2 \;=\; g_2 \wedge \left(\neg g_1 \;\Big|\; \Delta_1\Delta_2\mathcal{A} \not\models e_{\Delta_1} \wedge e_{\Delta_2} \;\Big|\; (\mathcal{A},\, \Delta_1\Delta_2\mathcal{A}) \models \bigvee_{\substack{J \in \delta_2\mathcal{I}\, \wedge \\ p(J) > p(t_1)}} \downarrow J \right) \wedge \left((\mathcal{A},\, \Delta_2\mathcal{A}) \not\models \bigvee_{\substack{J \in \mathcal{I}\, \wedge \\ p(J) > p(t_2)}} \downarrow J \right),\;\; \Delta_2,\;\; \delta_2$$

$$\ll t_1 \;=\; g_1 \wedge \left(\neg g_2 \;\Big|\; (\mathcal{A}, \Delta_2\mathcal{A}) \models \bigvee_{\substack{J \in \mathcal{I}\, \wedge \\ p(J) > p(t_2)}} \downarrow J \right) \wedge \left((\mathcal{A},\, \Delta_1\mathcal{A}) \not\models \bigvee_{\substack{J \in \mathcal{I}\, \wedge \\ p(J) > p(t_1)}} \downarrow J \right),\;\; \Delta_1,\;\; \delta_1$$

Figure 2: Definition of Composition with Conflict- and Violation-Free Synchronization

call, by asserting that the caller return to state [o_analyze] with the forwarded number as the new call destination.

Now O_POTS and Originating Call Screening both want to react to this new system state where the caller has entered the [o_analyze] state:

O_POTS:
  [+o_analyze(c,o,t)], InService(t) $\longrightarrow$ [+o_request(c,o,t)]

OCS:
  [+o_analyze(c,o,t)], OCS(o,t) $\longrightarrow$ [+o_dead(c,o)], *OSCDeny(o)*

Since the call screening feature has priority over O_POTS, the call is terminated with an announcement to the caller indicating that their call was screened. □

Example 6. Call Display ≪ Call Display Blocking
Call Display (CD) reveals the phone number of the caller to the callee. (The first line declares assertion Display to be a descendent of abstract assertion Reveal.)

CD:
  Display : Reveal
  [+t_ring(c,o,t)] $\rightarrow$ +Display(c,o,t)

As soon as the caller lifts the handset, Call Display Blocking (CDB) asserts an invariant stating that the caller's number not be revealed to other parties of the connection.

CDB:
  CDBInv(c,o) = !Reveal(c,o,*) : *NumBlocked*
  +Orig(c,o) $\rightarrow$ +<CDBInv>

When caller o picks up the handset, O_POTS asserts +Orig(c,o); this addition to the system state triggers CDB's rule that asserts its invariant. If the caller dials user t, who subscribes to Call Display, and t's T_POTS enters state [t_ring], then CD attempts to execute its rule, which violates CDB's invariant. If Call Display Blocking has priority over Call Display, then CD's rule does not execute. Note that Call Display is an enhancement to T_POTS; it does not try to override any rule in T_POTS. Thus the callee's phone continues to ring whether or not Call Display executes its rule. □

6. DISCUSSION AND RELATED WORK

The main contributions of this work are the composition operators and how they resolve conflicts and prioritize invariant violations. Thus, while our specification notation resembles other relational languages like Prolog [25] and tuple-space coordination languages like Linda [17], our semantics for services and features executing in parallel are very different. Rather, our composition operators form the basis of a sophisticated coordination model [14] that dictates how concurrent agents synchronize and coordinate their actions.

Many researchers have looked at using formal methods to model components or viewpoints and compose them into coherent systems. To give a few examples, Zave and Jackson [31] propose writing different views of a specification in (possibly different) appropriate formalisms and conjoining the first-order representations of the partial specifications. Burns, Mataga, and Sutherland [8] model features as service transformers whose transitions are composed sequentially. Cameron and Lin [12] model features as labelled transition systems whose composition operator resembles composition with AND-synchronization [4]. These techniques promote modular feature development, but detect only those interactions that are similar to our action conflicts, and they do not address resolution. Our ideas on composition were inspired by Bornot, Sifakis, and Tripakis's work [4] on composing prioritized timed labelled transition systems. However, their goal was to maximize components' progress to meet real-time deadlines, and their models abstract away components' actions on system variables. Thus, our composition operations are very different from theirs.

We are certainly not the first to propose using priorities to resolve conflicts [2, 6, 13, 19, 20, 22, 27]. However, their results address only conflict interactions and do not resolve interactions that violate features' constraints. van Lamsveerde, Darimont, and Letier propose the notion of temporarily violating invariants [28, 29] as a manual technique for resolving conflicting requirements goals. To our knowledge, we are the first to provide a model and a means for automatically detecting and resolving constraint violations.

A promising approach to conflict resolution is preliminary work by Finkelstein, Gabbay, Hunter, Kramer, and Nuseibeh [16] on a meta-language for expressing generic rules to detect conflict and specify corrective action. While their work is aimed at resolving conflicts during the requirements phase of software development, one can imagine using such a language to map appropriate run-time resolutions to different types of conflict. In contrast, we currently hard-code such mappings into our composition operators.

7. CONCLUSION

In conclusion, this paper has described a compositional approach to adding new features to an existing application. It supports modular development of features and resolves conflicts and constraint violations during feature composition. While all of the examples in this paper come from telephony, our design model and composition operators could be used to build feature-rich applications in other domains. The design model is most useful in domains such as word-processing, games, travel reservation systems, banking services, etc., where a product's success is highly dependent on its feature set. The first step in applying the model is to identify the application's basic service and the set of assertions that represent the service's shared variables. All other aspects of the model are application independent.

We have used our specification model and CVF composition operator to specify and compose telephony features from Bellcore's IN Feature Interaction Benchmark [10]. We had no difficulty ranking the benchmark features by priority, such that the behaviours of the composed features seemed reasonable and desirable. We intend to run a more formal experiment in which CVF resolutions are compared with users' expectations for how a collection of features should behave.

In preparation for this experiment, we are constructing a reachability analyzer that accepts a prioritized collection of features and computes the system's *reachability graph*, consisting of all the system states and state-transitions in the features' composition. The reachability graph will contain enough information for the analysis tool to report all of the features' conflicts and invariant violations as well as their resolutions. We also expect to be able to analyze the computed reachability graph to verify that features work correctly when composed with the underlying service specification and to check if desired invariants hold in various feature combinations.

8. ACKNOWLEDGEMENTS

We would like to acknowledge Tom Gray, Serge Mankovski, and Michael Weiss, of Mitel Corporation. Our discussions with them on resolving interactions at run-time helped us to formulate our ideas of composition. We thank Dan Berry for his comments on early versions of this work. We also gratefully acknowledge MITEL and Communications and Information Technology Ontario (CITO) for their financial support.

9. REFERENCES

[1] A. Aho, S. Gallagher, N. Griffeth, C. Schell, and D. Swayne. "SCF3^{TM}/Sculptor with Chisel: Requirements Engineering for Communications Services. In *International Workshop on Feature Interactions in Telecommunications Systems V*, pages 45–63, 1998.

[2] D. Amyot, L. Logrippo, R. Buhr, and T. Gray. "Use Case Maps for the Capture and Validation of Distributed Systems Requirements". In *International Symposium on Requirements Engineering*, 1999.

[3] J. Blom, B. Jonsson, and L. Kempe. Using Temporal Logic for Modular Specification of Telephone Services. In *International Workshop on Feature Interactions in Telecommunications Systems II*, pages 197–216, 1994.

[4] S. Bornot, J. Sifakis, and S. Tripakis. "Modelling Urgency in Timed Systems". In *International Symposium: Compositionality – The Significant Difference, LNCS 1536*, 1997.

[5] M. Boström and M. Engstedt. Feature Interaction Detection and Resolution in the Delphi framework. In *International Workshop on Feature Interactions in Telecommunications Systems III*, pages 157–172, 1995.

[6] K. Braithwaite and J. Atlee. "Towards Automated Detection of Feature Interactions". In *International Workshop on Feature Interactions in Telecommunications Systems II*, pages 36–59, 1994.

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

[8] G. Burns, P. Mataga, and I. Sutherland. Features as Service Transformers. In *International Workshop on Feature Interactions in Telecommunications Systems V*, pages 85–97, 1998.

[9] M. Cain. "Managing Run-Time Interactions Between Call-Processing Features". *IEEE Communications*, 30(2):44–50, February 1992.

[10] E. Cameron, N. Griffeth, Y. Lin, M. Nilson, W. Schnure, and H. Velthuijsen. A Feature Interaction Benchmark in IN and Beyond. Technical Report TM-TSV-021982, Network Systems Specifications Research, Bell Communications Research, September 1992.

[11] E. Cameron, N. Griffeth, Y. Lin, and H. Velthuijsen. "Definitions of Services, Features, and Feature Interactions", December 1992. Bellcore Memorandum for Discussion, presented at the International Workshop on Feature Interactions in Telecommunications Software Systems.

[12] E. Cameron and Y.-J. Lin. "A Real-Time Transition Model for Analyzing Behavioural Compatibility of Telecommunications Services". In *Proceedings of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, pages 101–111, 1991.

[13] Y.-L. Chen, S. Lafortune, and F. Lin. "Resolving Feature Interactions Using Modular Supervisory Control with Priorities". In *International Worksohp on Feature Interactions in Telecommunications Systems IV*, 1997.

[14] P. Ciancarini. “Coordination models and languages as software integrators”. *ACM Computing Surveys*, 28(2):300–302, June 1996.

[15] L. du Bousquet. Feature interaction detection using testing and model-checking. In *Formal Methods 99*, 1999.

[16] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. “Inconsistency Handling in Multiperspective Specifications”. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.

[17] D. Gelernter and N. Carriero. “Coordination languages and their significance”. *Communications of the ACM*, 35(2):97–107, February 1992.

[18] N. Griffeth and H. Velthuijsen. “The Negotiating Agents Approach to Runtime Feature Interaction Resolution”. In *International Workshop on Feature Interactions in Telecommunications Systems*, pages 217–235, 1994.

[19] S. Homayoon and H. Singh. “Methods of Addressing the Interactions of Intelligent Network Services with Embedded Switch Services”. *IEEE Communications*, 26(12):42–70, December 1998.

[20] M. Jackson and P. Zave. “Distributed Feature Composition: A Virtual Architecture for Telecommunications Services”. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.

[21] K. Kimbler. “Comprehensive approach to service interaction handling”. *Computer Networks and ISDN Systems*, 30:1363–1387, 1998.

[22] D. Marples and E. Magill. “The use of Rollback to prevent incorrect operations of Features in Intellent Network Based Systems”. In *International Workshop on Feature Interactions in Telecommunications Systems V*, pages 115–134, 1998.

[23] D. Pinard, M. Weiss, and T. Gray. “Issues in Using an Agent Framework for Converged Voice/Data Applications”. In *Practical Applications of Intelligent Agents and Multi-Agents*, 1997.

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

[25] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Mass., 1986.

[26] J. Thistle, R. Malhame, H.-H. Hoang, and S. Lafortune. “Feature Interaction Modelling, Detection and Resolution: A Supervisory Control Approach”. In *International Worksohp on Feature Interactions in Telecommunications Systems IV*, pages 93–107, 1997.

[27] G. Utas. “A Pattern Language of Feature Interaction”. In *International Workshop on Feature Interactions in Telecommunications Systems V*, pages 98–114, 1998.

[28] A. van Lamsweerde, R. Darimont, and E. Letier. “Managing Conflicts in Goal-Driven Requirements Engineering”. *IEEE Transaction on Software Engineering*, 24(11):908–926, November 1998.

[29] A. van Lamsweerde and E. Letier. “Integrating Obstacles in Goal-Driven Requirements Engineering”. In *Proceedings of the 20th International Conference on Software Engineering*, pages 53–63, 1998.

[30] P. Zave, February 2000. Presented at the IFIP WG 2.9 Workshop on Requirements Engineering.

[31] P. Zave and M. Jackson. “Conjunction as Composition”. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.

[32] I. Zibman, C. Woolf, P. O’Reilly, L. Strickland, D. Willis, and J. Visser. “Minimizing Feature Interactions: An Architecture and Processing Model Approach”. In *International Workshop on Feature Interactions in Telecommunications Systems III*, pages 65–83, 1995.