Variable-Specific Resolutions for Feature Interactions

Cecylia Bocovich
University of Waterloo
Waterloo, ON, Canada
cbocovic@uwaterloo.ca

Joanne M. Atlee
University of Waterloo
Waterloo, ON, Canada
jmatlee@uwaterloo.ca

ABSTRACT

Systems assembled from independently developed features suffer from *feature interactions*, in which features affect one another's behaviour in surprising ways. The *feature-interaction problem* states that the number of potential interactions is exponential in the number of features in a system. Resolution strategies offer general strategies that resolve entire classes of interactions, thereby reducing the work of the developer who is charged with the task of resolving interactions. In this paper, we focus on resolving interactions due to conflict. We present an approach, language, and implementation based on resolution modules in which the developer can specify an appropriate resolution for each variable under conflict. We performed a case study involving 24 automotive features, and found that the number of resolutions to be specified was much smaller than the number of possible feature interactions (6 resolutions for 24 features), that what constitutes an appropriate resolution strategy is different for different variables, and that the subset of situation calculus we used was sufficient to construct nontrivial resolution strategies for six distinct output variables.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; D.2.10 [**Software Engineering**]: Design; D.2.12 [**Software Engineering**]: Interoperability

General Terms

Design, Reliability

Keywords

Feature interaction, conflict resolution

1. INTRODUCTION

Software systems are growing in size and complexity. The "scale" of large-scale systems no longer refers simply to large codebases but also to extreme *variability*, including increasingly larger sets of requirements, greater optionality and customization, and greater varieties of execution platforms. Such complexity is partly addressed through decomposition. In **feature-oriented software development**, a system's functionality is decomposed into features, where each **feature** is an identifiable unit of functionality or variation. Users view features as system capabilities (e.g., cut, copy, and paste; Caller ID; Cruise Control) that may be optional, selectable, or tariffable. Software developers use features as the basis for incremental software development, characterizing software releases in terms of the new features introduced or updates to existing features. Feature orientation is particularly important in *software product lines*, in which families of similar software products (e.g., smartphones) are managed and evolved in terms of their features, and where individual products are differentiated by their (optional) features [30]. Feature orientation also has the added benefit that features can serve as a shared vocabulary among diverse stakeholders (e.g., marketers, customers, other engineers) in a way that other types of software fragments–such as modules, objects, or components–cannot.

Although features are considered individually, they are often not truly separate concerns and problems arise when developers try to integrate them into a coherent product. A **feature interaction** occurs when one feature affects the behaviour of another. For example, the software controllers for the braking features on the 2010 Toyota Prius interacted badly, reducing drivers' overall ability to brake and leading to multiple crashes and injuries [29]. To be safe, software developers must consider how features interact and must resolve undesired interactions. Because features are optional in many products, or can be turned on and off dynamically, the number of feature combinations to be analyzed and managed is exponential in the number of features. As a result, a software team finds that the development of new features is eventually dominated by the **Feature Interaction Problem**: the need to analyze, resolve, and verify interactions [4].

Researchers have proposed a number of general-purpose strategies for resolving interactions, such as resolution by feature priority [14, 18, 24], feature precedence [2, 7, 17], negotiating compromises [12], involving the user [10], rolling back conflicting actions [24], disabling feature activation [16], terminating features [24], and terminating the application. These strategies address the scalability aspect of the Feature Interaction Problem directly by providing a default strategy for resolving interactions, thereby reducing the number of interactions that need to be individually addressed

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2635927

by the developers. However, most of these strategies are coarse grained (e.g., based on the priority or precedence of the features themselves rather than the features' interacting actions); they provide suboptimal win/lose resolutions in which some features' actions are sacrificed in favour of other features' actions; and they often require an upfront total or partial ordering on features [32].

In this paper, we propose a new approach to resolving feature interactions that (1) maintains feature modularity and obliviousness, (2) allows the developer to specify a generic resolution strategy for each output variable, and (3) is agnostic to the number and to the specifics of features in the system or the number of possible feature interactions. Each feature specifies actions and outputs, in response to system inputs and environmental conditions. All of the actions that apply to a particular output variable are input to a resolution module designed for that variable, and the resolution module produces a conflict-free sequence of actions that are based on the input actions and the developer-provided resolution strategy. Examples of simplified resolution strategies include assigning an output variable to the *average* or the *minimum* or the *maximum* of the values specified by the features' actions. A key benefit of this approach is that the default resolutions for conflicting actions on a variable are appropriate for that variable.

The contributions of this paper are as follows:

- We introduce a new approach to resolving features' interactions, in which the resolution strategies are specific to the variables being acted on.
- We present an implementation of the approach, in which feature actions and resolution strategies are encoded in situation calculus [25] and are executed by a GOLOG intrepreter [23]. We identify sufficient and necessary conditions on the developer-provided resolutions that ensure that a resolution has desired properties (e.g., is deterministic, is conflict-free, terminates).
- We performed a case study in which we used our approach and implementation to model the actions of 24 automotive features and to specify appropriate resolutions for 6 distinct output variables. The results of the case study demonstrate that different output variables require different resolution strategies. The case study also assessed the expressiveness of situation calculus as a suitable language for encoding feature actions and resolution strategies.

Our paper is organized as follows. In the next section, we give an overview of feature-oriented requirements modelling and the feature-interaction problem. In Section 3, we describe our approach to resolving feature interactions in terms of resolution strategies per output variable, including how to encode feature actions and resolution strategies in situation calculus. In Section 4, we prove that the developer-provided resolutions have desired properties (e.g., are deterministic, are conflict-free, terminate), and we present the results of our case study. In Section 5, we review the advantages of our resolution and discuss future work. Section 6 summarizes related work, and Section 7 concludes our work.

2. PRELIMINARIES

Throughout this paper, we use examples from the automotive domain. Each feature extends a Basic Driving Service (BDS), which serves as a base system [6]. An automotive system comprises BDS and a subset of optional features.

2.1 Feature-Oriented Requirements

We are primarily concerned with the requirements stage of feature-oriented software development. Behavioural requirements for each feature are modelled independently and then composed into a system. There are many ways to model feature behaviour, but we focus on state-machine approaches [13]. The language we describe here is based on the Feature-Oriented Requirements Modelling Language (FORML) [26] as it provides a rich syntax for expressing feature behaviour.

A system's behaviour is expressed in terms of its reactions to changes and conditions in its environment, and its actions on environmental variables. **Monitored variables** such as *car.speed* represent environmental phenomena that are sensed by or act as inputs to the system. Changes to monitored variables prompt reactions in the system behaviour. **Controlled variables** represent environmental phenomena that are controlled or affected by system outputs. For example, the variable *car.acceleration* represents the current acceleration of the vehicle and is affected by actuators such as the throttle or vehicle brakes. Often, related variables are packaged into objects. These variables are referred to as object attributes. For example, *acceleration* and *speed* are both attributes of a *car* object. This relationship between objects and attributes is expressed using a dot notation (as in *car.speed*). Together, the values of world variables comprise the **world state**. Of particular importance is the current world state, ws_c, which represents the current valuations of all environmental variables. The current value of attribute a of the object o is denoted $ws_c :: o.a$. Changes in variable values between the previous world state, ws_p, and the current world state, ws_c, represent events.

A feature's behaviour is modelled as a state machine, called a **feature machine**. Figure 1 shows feature machines for BDS and two optional features, Cruise Control (CC) and Speed-Limit Control (SLC). A state in a feature machine reflects the current state of a feature's execution. States may be hierarchical, containing several sub-states that more finely describe feature behaviour. Superstates may also contain concurrent regions that execute in parallel. For example, the CC feature has Active and Inactive superstates that reflect the feature's conditional behaviour. The Active state contains three concurrent regions responsible for monitoring and controlling different phenomena. A world state reflects the execution states of all feature machines, as well as valuations of all monitored and controlled variables.

Transitions between states may be labelled with an identifier id, a triggering event te, a guard condition gc, and one or more actions $a_1, \ldots, a_n$.

$$id : te\ [gc]/a_1 \ldots a_n$$

A transition from state s_1 to s_2 is enabled if the machine is currently in state s_1, the guard condition evaluates to true, and the triggering event occurs. For example, in the CC feature machine, the transition between Inactive and Active states executes only if the triggering event *ccActivate* occurs. Triggering events refer to changes in variable values between the previous world state, ws_p, and the current world state, ws_c. Guard conditions are boolean expressions over current

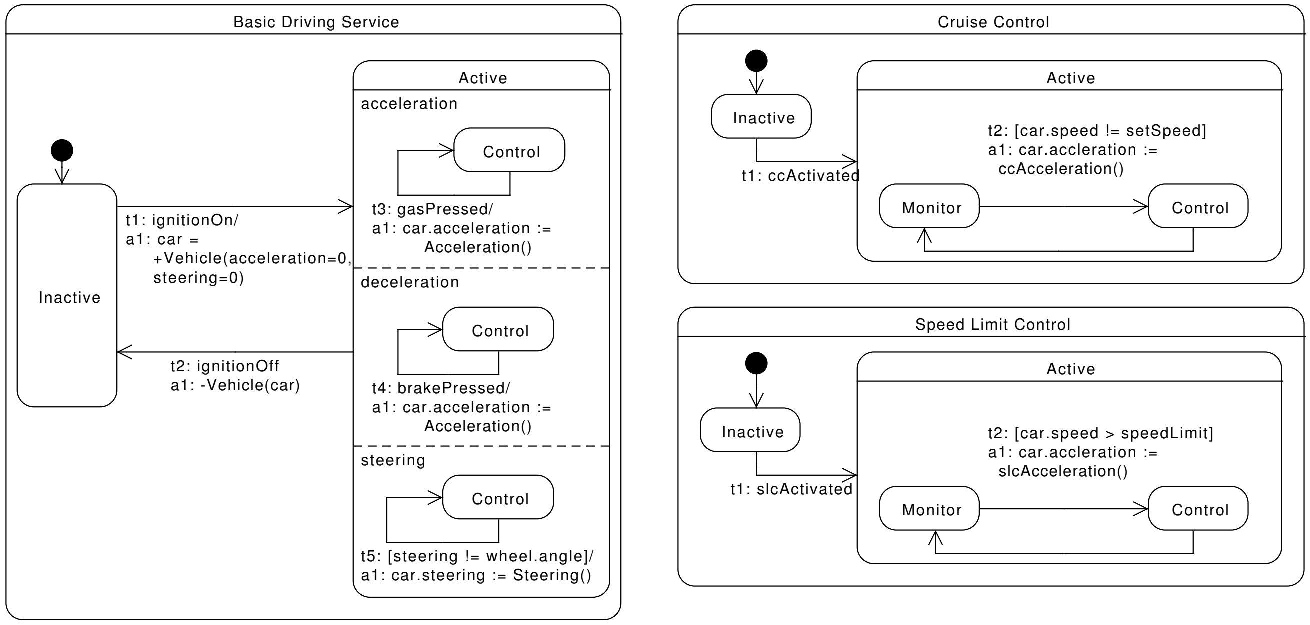


Figure 1: Behaviour model of BDS and some automotive features

values of variables in ws_c. A feature action corresponds to a prescribed change to the current world state.

Features affect the world state of a system by executing actions on controlled variables. An action can add an object to the world, remove an object from the world, or assign a value to a controlled variable. For example, many automotive features modify vehicle acceleration to maintain driver preferences or respond to safety concerns. This is modelled by actions that assign values to the controlled variable *car.acceleration*. Often, the details of calculating these values are abstracted as uninterpreted functions:

$$a1: \ car.acceleration := CCAcceleration()$$

An execution step of a feature machine consists of the execution of all concurrently enabled transitions and their actions. The behaviour of a system is the parallel execution of its feature machines. An execution step of a system is the simultaneous execution of each feature's transitions. The resulting world state is determined by the new states in each feature machine and the effects of the transitions' actions on the controlled variables.

2.2 Feature Interactions due to Conflicts

Composing independently developed features naturally leads to feature interactions. The literature [6] lists multiple types of feature interactions, but in this paper we are concerned with conflicts that occur in a single execution step. A **conflict** [20] occurs when the set of actions in an execution step are impossible to execute simultaneously (e.g., incompatible assignments to the same controlled variable).

In Figure 1, we see the potential for conflict when both SLC and CC are Active and $setSpeed > speedLimit$: SLC will try to decrease acceleration at the same time that CC tries to increase it:

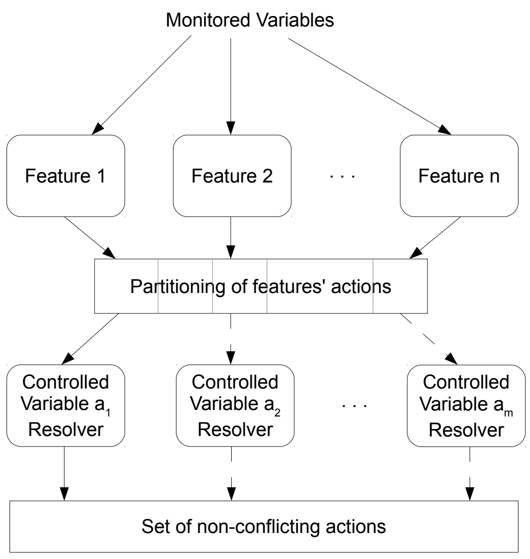
$$CC.t2.a1 : car.acceleration := ccAcceleration()$$
$$SLC.t2.a1 : car.acceleration := slcAcceleration()$$

3. RESOLUTION

Our aim is to resolve feature interactions in a way that addresses key aspects of the feature interaction problem. We developed a strategy with the following high-level goals.

1. Maintain the advantages of feature-oriented software development. This includes feature modularity and obliviousness to the presence of other features.
2. Enable conflict-free feature composition. Feature composition should resolve feature interactions if they are present and should preserve feature behaviour in the absence of interactions.
3. Allow resolutions to be based on all conflicting actions rather than on the features that perform them. This limits the impact that adding or removing features has on the specification of resolutions.
4. Resolutions should be agnostic to the number of features in the system and the number of features in an interaction. In addition, the number of resolutions specified by developers should be small with respect to the number of interactions and should not grow linearly or super-linearly with the number of features.
5. The resolutions we devise should be deterministic and total. Determinism guarantees that, given a current world state and a set of changes to monitored variables, there is only one possible next world state. As a result, system behaviour is predictable. Totality guarantees that there will always be a valid next world state.

We first give an overview of our approach to resolving feature interactions due to conflicting actions and describe how it fits into the execution model of a system composed of feature machines. We then present the details of our implementation and provide examples of resolutions in the automotive domain.



Monitored Variables
Feature 1
Feature 2
Feature n
Partitioning of features' actions
Controlled Variable a_1 Resolver
Controlled Variable a_2 Resolver
Controlled Variable a_m Resolver
Set of non-conflicting actions

Figure 2: Architectural model

3.1 Overview

We draw inspiration for our approach from the Software Cost Reduction (SCR) [15] requirements method. SCR specifications follow a dataflow execution model [19], in which requirements are represented as a directed graph. Each node in the graph is a function that calculates the current value of a particular variable. Edges indicate the flow of data between nodes. A node executes its function as soon as all of its input data are available along incoming directed edges, and it outputs the result along outgoing directed edge(s) to the next node(s).

Thus, an SCR specification defines a unique function for each controlled variable. This function calculates what the value of the variable will be at the end of an execution step. Each function takes as input the current values of all monitored variables, the most recently computed values of local variables, and the current modes of all mode classes (analogous to the current states in a state-machine) and deterministically calculates the next value of the variable for which it is defined. There are no conflicts among variable assignments in an SCR specification because they are resolved during specification: each function effectively encodes all contributions to a controlled variable's next value, and computes a single next value.

In our approach, we similarly define a unique function (called a **resolution module** or **resolver**) for each controlled variable that computes the variable's next value. However, in our approach, the sources of the inputs to a function (that is, the *features* that are executing actions) do not need to be known in advance. Each controlled-variable resolver takes as input the features' *actions* on the variable and employs a feature-independent resolution strategy to assign a conflict-free value to the variable.

Figure 2 depicts the architectural structure of our approach. We define a feature module for each feature. In each execution step, the feature machines in their modules execute in parallel, reacting to changes in the values of monitored variables. Each feature module outputs the set of actions on the transitions that execute in that step. The actions are partitioned according to the controlled variable that they modify. For each controlled variable in the current world state, we define a resolution module that is given as input all of the features' actions on the controlled variable. The resolver computes and outputs a sequence of actions to be executed atomically on the variable.

The next world state, ws_n, is the result of executing in parallel the output actions of each resolver, together with the next states of each feature machine. Thus, a complete execution step proceeds as follows:

1. Changes occur to the values of one or more monitored variables.
2. Feature machines react in parallel by executing transitions and outputting transition actions.
3. Our resolution introduces a third phase in the execution step, in which the feature-machine actions pass through resolver modules, one resolver per controlled variable. The output of each resolver is a sequence actions to be performed atomically.
4. The next world state, ws_n, is determined by the result of performing the transitions in step 2 and the feature actions produced in step 3.

There are a couple of details about the resolution modules worth noting. First, our resolvers differ from the controlled-variable functions in SCR in that they take as input not only the current values of monitored variables, but also an arbitrary number of possibly conflicting actions on the controlled variable in question. In contrast, SCR functions take only values as input.

Second, recall that a feature's actions can add a new object to the world or remove an object from the world. As objects are added (and removed), resolution modules for the objects' attributes are instantiated (and removed). Thus, the set of resolvers in the system is dynamic.

3.2 Details

In this section, we focus on the details of the resolution modules. We describe a language that is suitable for expressing the inputs and outputs of a resolver as well as for specifying a resolver's resolution strategy. We then provide an implementation that uses situation calculus and a subset of the GOLOG programming language.

3.2.1 Input Action Language

Our resolution language needs to be rich enough to encode the inputs to a resolver. This includes values of monitored variables and feature actions. An assignment assigns a variable to the value of an expression. An expression may be a simple value, or it may be a computation on other variables or uninterpreted functions[1]. The resolution language must be able to encode any relevant information about assignment expressions, as the inputs to a resolver.

3.2.2 Resolution Language

How conflicting assignments to a controlled variable are resolved depends on multiple factors, including the variable's

[1]The details of uninterpreted functions may be specified later in development.

Table 1: Domain independent situation calculus symbols

Symbol	Type	Description
S_c	constant	starting state
$\mathrm{do}(a, s)$	function	result of performing action a in s
$\mathrm{Poss}(a, s)$	predicate	a can be performed in s

range of values, the system domain, and the effect the variable has on the behaviour of the system or on the system's environment. As such, the developer or domain expert are in the best position to determine the most appropriate resolution strategy for conflicting actions on a controlled variable. Our goal is to provide a language that is suitable for them to program appropriate resolutions.

Consider two features, A and B, that control vehicle stability. Feature A monitors lane markings and detects that the vehicle has veered too far to the left and compensates by turning the vehicle to the right. Simultaneously, feature B monitors oscillation (the sway of the vehicle), detects that the vehicle is over-steering to the right, and attempts to correct this by turning the vehicle to the left. In order to achieve maximum stability, an appropriate resolution needs to consider the actions from both features. A reasonable resolution might be to set *car.steering* to the average of the values assigned by features A and B.

In general, the resolution language needs to be expressive enough to reason about a collection of actions and compute a conflict-free resolution. Examples of resolution strategies include computing the minimum, average, or sum of the assignment expressions that are output by the feature modules. Each resolution module considers only the *actions* on its controlled variable, and not the sources of the actions. Such strategies incorporate *all assignments* in the computation of the result value and thus offer an alternative to a win/lose resolution in which only one feature's actions (e.g., those of the highest-priority feature) are preserved in the final resolution.

Even when it is possible to specify variable-specific resolutions, priorities sometimes still play a role and our resolution language should support them. For example, in automotive systems, it is common to give higher priority to actions that preserve driver safety, such as Speed Limit Control (SLC), compared to actions that maintain driver-set preferences, such as Cruise Control (CC). Additionally, we may wish resolutions to prioritize driver actions over feature actions. As such, we categorize actions as being driver-controlled, safety, or non-safety and devise a resolution language that supports reasoning about action categories as well as the values of assignment expressions themselves. We stress that these resolutions depend on an action priority, not a priority ordering on features.

3.2.3 Situation Calculus

Situation calculus [25] is a first-order language that is well-suited to expressing actions, domain-knowledge, and the effects that actions have on the current domain state. We chose to use situation calculus as a proof-of-concept implementation language because it naturally supports our requirements with respect to both the input action language and the resolution language.

Situation calculus constructs are grouped into three basic categories: situations, fluents, and actions. A **situation** is a first-order term that represents a world state, or a valuation of all variables. A situation is the result of performing a sequence of actions on a defined starting state[2], S_c.

Actions in situation calculus are first-order logic terms that reflect a prescribed change to a situation, or world state. These actions may take one or more arguments as inputs. For our purposes, situation-calculus actions are analogous to feature actions. Performing an action a on a situation S_c is expressed using the special function *do*, and results in a new situation $s_n = do(a, S_c)$.

Fluents are functions and predicates that take a situation as one of their arguments; they are referred to as fluents because their valuations depend on the situation to which they are applied. Fluents can be used to refer to the values of variables in a particular world state. For example, the functional fluent $carSpeed(s)$ returns the value of the car's speed in the world state represented by situation s.

The developer uses a combination of situations, actions, and fluents to specify allowable steps in the execution of a system. These specifications are the axioms that constitute a domain theory $\mathcal{D}$. Starting-state axioms are assertions on the values of fluents in a starting state S_c. Successor-state axioms define the effects of performing an action a in a situation s. Precondition axioms specify whether an action a may be performed in a given situation s; they are expressed with the special predicate $Poss(a, s)$. Table 1 contains a summary of special situation-calculus constructs.

3.2.4 Encoding Inputs to a Resolution Module

The inputs to a resolution module for a controlled variable $o.a$ are (1) the values of monitored variables at the start of the execution step (i.e., the values of the monitored variables in the world state ws_c), and (2) the set of actions that are output by all the feature modules and that assign values to $o.a$.

The developer encodes the inputs to a resolution module as starting-state axioms, which assert constraints on the valuations of fluents in the starting situation S_c. The starting situation of each resolver will reflect the current world state, ws_c. Given a monitored variable m in ws_c, the developer asserts the relational fluent $m(v, S_c)$ to be true if the value of m in ws_c is equal to v ($ws_c :: m = v$). To express the set, in list form, of input actions, she asserts another relational fluent $assignRqst(L, S_c)$ where L represents the set of assignment expressions output by the feature modules. That is, if the features perform the actions $o.a := e_1, \ldots, o.a := e_n$, then $L = [e_1, \ldots, e_n]$. Note that the number n of actions output by the feature modules depends on the transitions that execute in the feature machines and varies between execution steps.

To distinguish between safety, driver, and non-safety actions, the developer may define *assignRqst* fluents for each category. The fluents $assignRqstSafety(L, S_c)$, $assignRqstDriver(L, S_c)$, and $assignRqstNonSafety(L, S_c)$ correspond to lists of assignment expressions partitioned according to the above categories. In general, the developer may define a fluent for every level of prioritization she wishes to express.

[2]We deviate from the traditional situation-calculus terminology of "initial-state" to avoid confusion with the concept of the initial state for a state-machine model.

Table 2: Domain independent resolution symbols

Symbol	Type	Description
empty(L)	Predicate	list L is empty
member(L, e)	Predicate	element e is in list L
append(L, e)	Function	append element e to list L
remove(L, e)	Function	remove element e from list L
average(L, v)	Predicate	average value in L is v
minimum(L, v)	Predicate	minimum value in L is v
maximum(L, v)	Predicate	maximum value in L is v
sum(L, v)	Predicate	sum of values in L is v
$<, >, =, \neq$	Predicate	equality and inequality
$+, -, *, /$	Function	arithmetic operations

Example 1. Suppose a developer is responsible for programming the resolution module for the controlled variable *car.acceleration*. Relevant monitored variables include *car.speed* and *car.acceleration*. The developer distinguishes between three levels of prioritization: safety, non-safety, and driver features.

In a particular world-state ws_c, the Cruise Control feature module outputs the action *car.acceleration* := *ccAcceleration*(), Speed Limit Control outputs the action *car.acceleration* := *slcAcceleration*(), and the driver presses the accelerator to produce the action *car.acceleration* := *Acceleration*()+5 The starting-state axioms that encode these inputs are:

$$car.acceleration(ws_c :: car.acceleration, S_c)$$
$$assignRqstSafety([slcAcceleration()], S_c)$$
$$assignRqstDriver([Accleration() + 5], S_c)$$
$$assignRqstNonSafety([ccAcceleration], S_c)$$

3.2.5 Encoding the Resolutions of a Resolver

Each resolution module is responsible for assigning a value to one controlled variable. Therefore, the developer defines only one situation-calculus action $assign(v)$ per resolver; the action assigns the value v to the controlled variable in question.

The developer uses precondition axioms to specify the output of a resolution module. A precondition axiom dictates the conditions under which a situation-calculus action may be performed in a given situation. The special predicate, $Poss(a, s)$ signifies that the action a may be performed in situation s. We characterize these precondition axioms using the fluents described above. Some helper predicates and functions that we deem useful for specifying resolutions that are listed in Table 2. This list is by no means exhaustive — the developer may define any first-order logic predicate and function needed to provide appropriate resolutions for her domain.

Example 2. Recall the acceleration example discussed in Example 1. There are several safety and convenience features that modify the controlled variable *car.acceleration*. These include Cruise Control (CC) and Speed Limit Control (SLC), which aim to keep the monitored variable *car.speed* at a driver-set preference and below the speed limit of the road, respectively. Additionally, the headway control (HC) feature changes the vehicle's acceleration in response to upcoming obstructions or other cars on the road; and the driver can affect vehicle acceleration by pressing her foot on the accelerator pedal.

Based on our understanding of how these features ought to interoperate with each other, we devised the following resolution[3]. Our resolution considers three different levels of priority: driver, safety, and non-safety actions. Driver actions to modify the car's acceleration have the highest priority, followed by actions from safety features, followed by actions from non-safety features. If there are multiple driver-related input actions, the resolver module will assign the minimum value. If there are no driver-related actions, then safety actions will be considered. If there is more than one safety action, then the minimum value is selected to be the output action. For example, if there are no driver-related actions and the two safety features, Speed-Limit control and Headway Control, both contribute input actions *car.acceleration* := e_1 and *car.acceleration* := e_2, then the output action will be the minimum of these two values.

Our resolution is expressed in situation calculus as follows:

$$\begin{aligned}
&Poss(assign(v), s) \equiv \\
&(\exists l.assignRqstDriver(l, s) \wedge minimum(l, v)) \vee \\
&(\forall l.(assignRqstDriver(l, s) \rightarrow empty(l)) \wedge \\
&\quad \exists l_2.assignRqstSafety(l_2, s) \wedge minimum(l_2, v)) \vee \\
&(\forall l.(assignRqstDriver(l, s) \vee assignRqstSafety(l, s) \rightarrow \\
&\quad empty(l))) \wedge (\exists l_3.assignRqstNonSafety(l_3, s) \wedge \\
&\quad minimum(l_3, s))
\end{aligned}$$

The interpreter will first see if there are any elements in the list of input driver actions and take the minimum value of this list. If the list of driver actions is empty, it will attempt to find the minimum value of the list of input safety actions. Finally, if there are no driver or safety actions, the interpreter will output the minimum value of non-safety actions.

As long as there is at least one enabled action, there will be a value v that satisfies the above formula. If there are no input actions, the resolver will not output any actions and the value of the controlled variable will not change. In Section 4, we discuss the necessary and sufficient conditions to ensure that resolutions are deterministic and total.

3.2.6 Implementation in GOLOG

GOLOG is a programming language for situation calculus. It provides procedures for outputting sequences of actions from a starting state that satisfy a collection of axioms. Given a domain theory $\mathcal{D}$, comprising situation calculus facts, fluents, and axioms; and given a starting state S_c and a GOLOG procedure δ, the GOLOG interpreter will find terminating situations S_n result from the program δ executing from the starting state S_c:

$$\mathcal{D} \models Do(\delta, S_c, S_n)$$

If the interpreter can find a situation S_n that satisfies this entailment, then the output of δ is a conflict-free sequence of actions to be applied to the resolver's controlled variable.

We now explain how a single resolution module is implemented in a combination of situation calculus and GOLOG.

[3]For the purposes of this paper, it does not matter whether or not we have a correct understanding of how feature interactions ought to be resolved. What matters is whether our proposed resolution language is expressive enough to specify interesting, non-trivial resolution strategies.

As we walk through the explanation, we will refer to the resolution of *car.acceleration* given in Examples 1 and 2.

At the very beginning of an execution step, the values of all monitored variables in the current and previous world states, ws_c and ws_p, are input to the feature machines. The changes in these variables will trigger transitions in the machines, resulting in actions on controlled variables. For example, an increase in vehicle speed may prompt SLC to assign a negative value to the car's acceleration if the vehicle's speed exceeds the speed limit of the road. All of the actions on *car.acceleration* from different features are then input to the same resolver, along with the current values of relevant monitored variables.

These inputs represent the state of the world before the resolution takes place and are encoded in situation calculus as starting-state axioms. In the *car.acceleration* example, the inputs to the resolution module are exactly the axioms given in example 1.

The resolution strategy for the resolver's controlled variable is encoded as a precondition axiom in situation calculus. This axiom, $Poss(assign(v), s)$, encodes when it is possible to execute an action of the form

$$car.acceleration := v.$$

The precondition axiom was given in Example 2. Note that its value depends on the evaluations of the starting state axioms given in example 1.

The GOLOG procedure specifies how the resolution strategy is used to determine the resolver's output. The GOLOG procedure to execute the acceleration resolution is:

$$\textbf{proc} \text{ resolve } \pi v. \text{ assign}(v)$$

This procedure nondeterministically explores all possible assignments v to the controlled variable of the resolver, looking for a value that satisfies the precondition axiom (i.e., the resolution strategy). The output of this procedure will be the result of performing a single action $S_n = do(assign(v), S_c)$. In simple cases, as in this example, the procedure invokes the resolution strategy once to determine a single output action. In more complicated cases, a developer may wish to iterate the strategy to output sequences of actions.

Example 3. As a separate example, there are a number of features that affect the direction in which the car is travelling. The controlled variable is *car.steering* and the affecting features include Lane Centring Control (LCC), and several stability features such as Traction Control (TC) and Stability Control (SC). The driver can also affect *car.steering* by rotating the steering wheel.

For this resolution, we distinguish only between driver and non-driver actions. Driver-related actions have the highest priority. If there are multiple assignments at the same priority level, the variable *car.steering* is set to the **average** of the assignment expressions. Thus, in this resolution, the actions from all features contribute to the outcome of the resolution. The inputs to the resolver are:

$$car.steering(ws_c :: car.steering, S_c)$$
$$assignRqstDriver(L, S_c)$$
$$assignRqstNonDriver(L, S_c)$$

The resolution is encoded as a precondition axiom that prioritizes driver-related actions and computes the average of the relevant steering actions:

$$\begin{aligned}&Poss(assign(v), s) \equiv\\ &\quad(\exists l.assignRqstDriver(l, s) \wedge average(l, v))\vee\\ &\quad(\forall l.(assignRqstDriver(l, s) \rightarrow empty(l))\wedge\\ &\qquad\exists l_2.assignRqstNonDriver(l_2, s) \wedge average(l_2, v))\end{aligned}$$

The GOLOG procedure

$$\textbf{proc} \text{ resolve } \pi v. \text{ assign}(v)$$

executes the resolver module for a given set of input actions and starting state S_c, and outputs a terminating state $S_n = do(assign(v), S_c)$.

The resolution examples discussed up to this point output a single action to be performed on a controlled variable. The next example uses iteration of a resolution strategy to output a sequence of actions. We use successor-state axioms in situation calculus to express the effects of a single iteration. For example, performing the action $assign(v_n)$ on the controlled variable *car.acceleration* will result in a new situation $do(assign(v_n), S_c)$ in which the value of *car.acceleration* is now v_n. We specify this effect in the domain theory $\mathcal{D}$ of the resolution with the successor-state axiom

$$car.acceleration(v_n, do(a, s)) \equiv a = assign(v_n)$$

We express the iteration as part of the GOLOG program.

Example 4. The variable *car.warningLightType* is controlled by alert features that try to get the driver's attention. If multiple features want to set a particular light to different values, one possible resolution is to satisfy all requests sequentially. For example, if one feature turns a light off and another turns the same light on, the resolution is to have the light blink on and off, to alert the driver to a possible conflict among the features associated with the warning light. The inputs to the resolution are:

$$car.warningLight(ws_c :: car.warningLight, S_c)$$
$$assignRqst(L, S_c)$$

Because the resolution is a sequence of more than one action, the developer needs to write the successor state axioms:

$$\begin{aligned}&car.warningLight(v, do(a, s)) \equiv a = assign(v)\\ &assignRqst(l, do(a, s)) \equiv a = assign(v)\wedge\\ &\qquad l = remove(k, v) \wedge modifyRqst(k, s)\end{aligned}$$

The first axiom encodes how the warning-light variable changes with each assignment. The second axiom specifies that as each assignment is made, the value of the assignment is removed from the list of feature actions. The warning-light assignments must be executed in some order.

In our resolution, in each situation, the light is always assigned to the maximum light value in the list.

$$Poss(assign(v), s) \equiv modifyRqst(l, s) \wedge maximum(l, v)$$

This resolution is deterministic, and it intuitively sets the warning light to the strongest value first. The GOLOG procedure is iterates through the list of input actions until the list is empty:

```
proc resolve
    while $(modifyRqsts(l, now) \wedge \neg empty(l))$
        do $\pi n.assign(n)$
```

4. ANALYSIS

In this section, we demonstrate that our resolutions to feature interactions have the desired properties that we listed in the beginning of Section 3: that the resolutions are conflict-free, are deterministic and total, and preserve the features' actions in the absence of an interaction.

The most important goal of our work is to enable conflict-free feature composition. Recall that a feature interaction due to conflict occurs when two or more features attempt to simultaneously execute a set of incompatible actions. The resolution approach that we have devised eliminates conflicts by computing a conflict-free sequence of actions. Such a computation is performed for the actions on each controlled variable in each execution step of the system.

Theorem 1. *The set of action sequences output by the resolver modules are conflict-free.*

Proof. A feature may execute one or more actions in each step. These actions each belong to one of three categories: adding an object to the world, removing an object from the world, and modifying the value of a controlled variable. Possible conflicts occur when:

1. Two or more features attempt to set the value of the same controlled variable to different values, or

2. One feature attempts to remove an object while another feature attempts to modify it.

Case 1: All of the features' assignments to the same controlled variable are forwarded to the same resolution module. The resolver outputs one action or one sequence of incremental actions that computes the resolved value; thus the actions of an individual resolver do not interact. Furthermore, each resolver produces a resolution for a distinct controlled variable; thus the outputs of different the resolution modules do not conflict.

Case 2: An action to remove an object results in the removal of all resolver modules associated with the object's controlled variables. All attempts to assign a value to any of these variables are ignored. In this way, object removal has priority over assignment.

In both cases, the set of output actions is conflict-free. □

A second goal of our work is that resolutions should be deterministic and total: (1) for each set of features' actions, a resolver produces a unique sequence of assignments to the corresponding controlled variable; and (2) for any set of features' actions on a controlled variable, there exists a (possibly empty) sequence of conflict-free actions output by the variable's resolution module. Determinism and totality are not guaranteed by the approach itself and are dependent on the developers' implementations. Specifically, the developers' resolution modules must meet the following obligations.

$$\textbf{Ob 1}.\ \forall v_1.\forall v_2.\forall s.Poss(\text{assign}(v_1), s) \wedge Poss(\text{assign}(v_2), s) \rightarrow v_1 = v_2$$

$$\textbf{Ob 2}.\ \forall s.\exists v.Poss(assign(v), s)$$

Then there is always exactly one possible sequence of output actions from the resolver and one possible next world state ws_n.

Note that particular attention must be paid to computations that have loops: the developer must prove that every loop terminates:

$$\forall s.\exists s'.(\forall P.(\forall s_1.P(s_1, s_1)) \wedge (\forall s_1, s_2, s_3.P(s_1, s_2) \wedge \phi[s_2] \wedge Do(\delta_1, s_2, s_3) \supset P(s_1, s_3)) \supset P(s, s')) \wedge \neg\phi[s']$$

Theorem 2. *Given the set of situation-calculus facts, fluents, and axioms $\mathcal{D}$ for a resolution module,* **Ob 1** *on the precondition axiom in $\mathcal{D}$, and the corresponding resolution procedure δ, then the following entailment holds:*

$$\mathcal{D} \models \forall s_c, s, s'.Do(\delta, s_c, s) \wedge Do(\delta, s_c, s') \rightarrow s = s'$$

Proof. This is proven by structural induction on the GOLOG resolution program δ. □

Theorem 3. *Given the set of situation-calculus facts, fluents, and axioms $\mathcal{D}$, for a resolution module;* **Ob 2** *on the resolver's precondition axiom; the corresponding GOLOG program δ; and obligations for each loop in δ, then the following entailment holds:*

$$\mathcal{D} \models \forall s_c.\exists s.Do(\delta, s_c, s)$$

Proof. This is proven by structural induction on the GOLOG program δ. □

Finally, a resolution should preserve the functionalities of the features as much as possible. If the set of features' actions on a controlled variable in an execution step are non-conflicting, then the resolution should include all actions on that variable.

Ob 3. If in an execution step the features produce exactly one action $o.a := v$ on the controlled variable $o.a$ then $Poss(assign(v), s)$ for that variable.

Theorem 4. *If* **Ob 3** *holds for every resolver function, then feature functionality will be preserved in the absence of feature-interaction conflicts.*

Proof. This is proven by structural induction on the GOLOG program δ. □

4.1 Case Study

We conducted a case study to analyze the expressive power of our resolution language. We examined 24 automotive features and identified six different controlled variables that are modified by multiple features. In Section 3, we presented the resolutions for three of these variables; their resolutions were created during the development of our resolution language. The other three controlled variables described below serve as a test of our language's expressive power.

We now provide the details of the variables in our case study and their appropriate resolutions. We base these resolutions on feature specifications provided by our industrial partner, although the feature names we give are based on common features found on the internet.

4.1.1 Brake Pressure

There are three categories of actions that modify the controlled variable *brake.hydraulicPressure*. The first category of actions enhances driver preferences and inputs. For example, the Automatic Braking (AB) feature performs actions that minimize stopping distance when triggered by the driver applying a large amount of pressure to the brake pedal. The second category includes safety or stability actions such as those enacted by Trailer Stability (TS) and Stability Control (SC). These features apply brake pressure to each of the four wheels to maintain a straight vehicle path and counteract vehicle oscillation. The third category of actions maintains driver-set acceleration preferences or provides feedback to the driver. These are grouped into a non-safety action category.

We give the highest priority to safety and stability actions. There are two ways in which safety actions affect brake pressure. Some safety actions reduce vehicle speed by applying an even application of brake pressure to all four wheels. Other safety actions apply brake pressure unevenly with the goal of controlling vehicle oscillation or stability. Our resolution differentiates between these cases by observing the monitored variable *car.oscillation*. If *car.oscillation* is less than a low threshold value, the resolution sets *brake.hydraulicPressure* to the maximum of all assignment values. If it is greater than the threshold, we assume that multiple stability actions are working to correct the oscillation of the vehicle and our resolution sets *brake.hydraulicPressure* to the average of all assignment values.

The precondition axiom for the resolution is

$$\begin{aligned}
&Poss(assign(v), s) \equiv \\
&\quad (\exists l.assignRqstSafety(l, s) \wedge ((maximum(l, v) \wedge \\
&\quad\quad car.oscillation(s) < threshold) \vee (car.oscillation(s) \geq \\
&\quad\quad threshold \wedge average(l, v)))) \vee \\
&\quad (\forall l.(assignRqstSafety(l, s) \rightarrow empty(l)) \wedge \\
&\quad\quad (\exists l_2.assignRqstDriver(l_2, s) \wedge maximum(l_2, v))) \vee \\
&\quad (\forall l.(assignRqstSafety(l, s) \vee assignRqstDriver(l, s) \\
&\quad\quad \rightarrow empty(l)) \wedge (\exists l_3.assignRqstNonSafety(l_3, s) \\
&\quad\quad \wedge maximum(l_3, v)))
\end{aligned}$$

4.1.2 Warning Chime

Features such as Cruise Control (CC), Basic Braking (BB), Parking Brake (PB), Manual Park Brake (MPB), and Road Change Alert (RCA) use the controlled variable *alert.chimeType* to alert the driver. As the primary purpose of this variable is to capture the driver's attention, we operate under the assumption that values for this variable can be ranked from less to more urgent. We define the function $rankList(L)$ that takes a list of values and returns a corresponding list of rankings, and we define the function $getType(x)$ that returns the chime type that corresponds to the ranking x. The inputs to the resolution are encoded in the predicate $assignRqst(L, s_c)$.

The resolution will set *alert.chimeType* to the most urgent chime value. The precondition axiom is

$$\begin{aligned}
&Poss(assign(v), s) \equiv \exists l.assignRqst(l, s) \wedge r = rankList(l) \\
&\quad \wedge maximum(r, x) \wedge v = getType(x)
\end{aligned}$$

4.1.3 Air Flow Rate

There are several features that control cabin temperature and air quality. Each of these features modifies the variable *cabin.airFlowRate*. The Air Quality System (AQS) circulates air to reduce pollution levels, Air Conditioning (AC) and Heater Control (HC) use air flow to circulate cooler or warmer temperatures, and Air Recirculation (AR) recirculates air at the driver's request.

We prioritize actions that circulate air to improve air quality over actions that circulate air to improve the air temperature. Input actions for air-quality are encoded in the predicate $assignRqstQuality(L, s_c)$. Inputs actions for air-temperature are represented as $assignRqstTemp(L, s_c)$. In both cases, we set the air flow rate to be the maximum assigned value. The precondition axiom for this variable is

$$\begin{aligned}
&Poss(assign(v), s) \equiv \\
&\quad (\exists l.assignRqstQuality(l, s) \wedge maximum(l, v)) \vee \\
&\quad (\forall l.(assignRqstQuality(l, s) \rightarrow empty(l)) \wedge \\
&\quad\quad \exists l_2.assignRqstTemp(l_2, s) \wedge maximum(l_2, v))
\end{aligned}$$

5. DISCUSSION

In this section we summarize the results of our case study and the advantages of our approach.

5.1 Utility

One goal of our resolution approach is to provide the modeller with a language that is powerful enough to express resolution functions that are tailored to fit the domain and the behaviour of each controlled variable in a system. The purpose of our case study was to gauge the expressiveness of our approach by specifying variable-specific resolutions for a diverse set of controlled variables. We found that the variables in our case study called for unique variable-specific resolutions, and we were able to express all desired resolutions. Moreover, we were able to express resolutions, in which all features contribute to the resolution result and not just those with the highest priority. In the vehicle steering example, every feature's assignment to *car.steering* is used to compute the variable's next value. Thus, all features "win" in that resolution.

5.2 Scalability

One of the major advantages of our approach is the absence of a required priority scheme among features. The developer still has the option to define priorities between types of actions, as we did in our examples by grouping actions into safety and non-safety categories. However, this classification is coarse and does not require a total or partial ordering on all features. When adding new features or removing existing features from the system, the developer need only determine to which category the feature's actions belong. Classification decisions do not need to be revised as new features are added to the system.

Additionally, the developer does not need to know how many features modify each controlled variable. Our resolution approach is agnostic to the number of features in the system as well as the number of feature interactions that arise from their composition. Thus, features can be developed independently and can be added to the system incrementally. If a feature introduces a new type of controlled variable, the developer does need to introduce a new resolution module. However, our case study suggests that the introduction of new controlled variables are rare; we discovered a total of only 8 controlled variables, 6 of which are modified by more than one feature, in a group of 24 automotive features. The number of resolution modules that the developer writes can be further reduced by identifying the set of controlled variables that are assigned values by multiple features and writing resolvers only for these controlled variables rather than for all controlled variables. A simple static analysis can identify these variables.

5.3 Threats to Validity

The requirements documents on which we base our resolutions provide information on a subset of automotive features. We specified what we considered to be appropriate resolutions to conflicting assignments made by these features. It is possible that other features could modify the same variables in a way that would warrant a different resolution strategy. This would weaken our claim that the addition of features does not impact the resolution strategies. These claims should be validated with future case studies.

5.4 Future Research Directions

Some of the resolutions we considered suggest that there are dependencies among controlled variables. For example, *brake.hydralicPressure* affects *car.acceleration*. Our resolution does not consider dependencies or interactions between controlled variables. We conjecture that such interactions could be addressed by clustering related variables and resolving their conflicts or by imposing a partial ordering on controlled variables and using the resolutions of some variables as inputs to the resolver modules of others. We leave these investigations to future work.

There is also the possibility for race conditions when features execute at different speeds or asynchronously. For example, actions by the features ABS and cruise control can produce a shuddering effect on the vehicle by bouncing back-and-forth between increasing and decreasing the vehicle acceleration. It may be possible to mitigate flip-flops or reversals in assignments by remembering a short history of resolutions and taking them into account when computing new resolutions. We leave the investigation of this problem to future work.

While situation calculus and GOLOG serve as an intuitive proof-of-concept implementation of our resolution modules, we believe there to be simpler and faster languages to accomplish the same goal. By relying on a deterministic language, many of the proofs in the analysis section would be made much simpler. Future research in this area will be to develop and refine a more appropriate implementation language.

6. RELATED WORK

The majority of related work on resolving feature interactions relies on a priority ranking among features [8, 11, 14, 16, 18, 24]. Priority-based approaches need a total or partial ordering on features to support the resolution strategy. When a new feature is developed, its place in the priority ordering must be determined, making it difficult to add new features. In the case of a conflict, only the actions of the highest-priority feature are executed, blocking the behaviour of all other features. Our resolution considers all enabled actions, regardless of feature priority.

Some priority-based approaches offer finer-grained resolutions. Laney et al. [21, 22] propose resolutions in which priorities are considered at the granularity of individual feature requirements. During feature composition, the developer specifies which aspects of a feature's behaviour may be left unsatisfied in the event of a conflict. Interactions are resolved on a case-by-case basis. Thus, this approach does not address the the feature interaction problem: the number of interactions to consider, resolve, and verify is potentially exponential in the number of features. In addition, the resolutions are win/lose in that only the highest-priority requirements are satisfied in case of a conflict.

Precedence-based resolution strategies [2, 7, 17], in which features are executed in a specified order, display similar problems to priority-based approaches. Features are given a total or partial precedence ordering, and the task of determining a precedence order for n features requires that the developer consider up to $n!$ orderings.

Some work has been done to mitigate the task of specifying priorities or precedences among large collections of features by categorizing features [32] and using automated detection of feature interactions to find acceptable orderings [32]. However, these approaches still suffer from course-grained resolutions based on feature priorities and offer only win/lose resolutions to feature interactions.

Griffeth and Velthuijsen reduce a developer's work by resolving conflicts through automated negotiation [12]. The general idea behind negotiation-based resolution is to offer alternative feature behaviours in the event of a conflict, to maintain the essential intent of the feature developer. This approach has been applied to multi-agent systems [27] using situation calculus as the action language. Negotiation requires multiple rounds of communication between negotiating agents that act on the behalf of features. Many safety-critical systems have strict timing requirements and cannot afford of multiple rounds of communication. Our approach resolves interactions in a single multi-phase execution step, by calculating variable-specific resolutions to conflicting assignments. These calculations are fast and each resolver is independent, so all resolvers may execute in parallel. Furthermore, features themselves do not need to interact with each other. This promotes feature modularity and obliviousness — key attributes of feature-oriented software development.

7. CONCLUSIONS AND FUTURE WORK

We have presented an approach for resolving feature interactions that addresses key aspects of the feature interaction problem by providing means for developers to specify an appropriate resolution strategy for each controlled variable rather than for each possible feature interaction. Our approach exhibits several advantages: the developer of a resolver module does not need to be aware of the number of features in the system, and there is no need to impose a partial ordering on features. This eases the task of adding features to the system

We show that the desired resolution for a controlled variable depends on the roles that the variable plays in overall system behaviour. Our approach allows for resolution strategies that are tailored to the specifics of each controlled variable. We provide evidence, in the form of a case study that different controlled variables warrant different resolution strategies.

8. REFERENCES

[1] E. Baroth and C. Hartsough. Visual object-oriented programming. chapter Visual Programming in the Real World, pages 21–42. Manning Publications Co., Greenwich, CT, USA, 1995.

[2] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[3] T. Bowen, C.-H. Chow, F. Dworak, N. Griffeth, and Y.-J. Lin. Views on the feature interaction problem. Technical Report Technical Memorandum TM-ARH-012849, Bellcore, Oct. 1988.

[4] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *Proceedings of the 7th International Conference on Software Engineering for Telecommunication Switching Systems (SETSS)*, pages 59–62, 1989.

[5] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Netw.*, 41(1):115–141, Jan. 2003.

[6] 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.

[7] A. Chavan, L. Yang, K. Ramachandran, and W. H. Leung. Resolving feature interaction with precedence lists in the feature language extensions. In *Proceedings of the 9th International Conference on Feature Interactions (ICFI)*, pages 114–128, 2007.

[8] Y.-L. Chen and S. Lafortune. Resolving feature interactions using modular supervisory control with priorities. In *Feature Interactions in Telecommunications Systems IV*, pages 108–121. IOS Press, 1997.

[9] Y.-L. Chen, S. Lafortune, and F. Lin. Priority assignment algorithms for resolving blocking in modular control of discrete event systems. In *Proceedings of the 35th IEEE Conference on Decision and Control,*, volume 3, pages 2743–2748, Dec 1996.

[10] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, Aug 1994.

[11] N. Fritsche. Runtime resolution of feature interactions in architectures with separated call and feature control. In *Feature Interactions in Telecommunications Systems III*, pages 43–63. IOS Press, 1995.

[12] N. Griffeth and H. Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In *Feature Interactions in Telecommunications Systems*, pages 217–235, 1994.

[13] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[14] J. Hay and J. Atlee. Composing features and resolving interactions. In *ACM SIGSOFT Foundations of Software Engineering (FSE)*, pages 110–119, 2000.

[15] C. L. Heitmeyer. Software cost reduction. Technical report, Naval Research Laboratory, 2002.

[16] S. Homayoon and H. Singh. Methods of addressing the interactions of intelligent network services with embedded switch services. *IEEE Communications Magazine*, 26(12):42–46, Dec 1988.

[17] 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.

[18] Y. Jia and J. Atlee. Run-time management of feature interactions. In *ICSE Workshop on Component-Based Software Engineering (CBSE)*, 2003.

[19] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004.

[20] A. L. Juarez-Dominguez, N. A. Day, and J. J. Joyce. Modelling feature interactions in the automotive domain. In *Proceedings of the 2008 International Workshop on Models in Software Engineering*, MiSE '08, pages 45–50, New York, NY, USA, 2008. ACM.

[21] R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh. Composing requirements using problem frames. In *12th IEEE International Proceedings of the Requirements Engineering Conference*, pages 122–131, Sept 2004.

[22] R. Laney, T. Tun, M. Jackson, and B. Nuseibeh. Composing features by managing inconsistent requirements. In *Proceedings of the 9th International Conference on Feature Interactions (ICFI)*, pages 129–144, 2007.

[23] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, 1997.

[24] D. Marples and E. Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *Feature Interactions in Telecommunications Systems V*, pages 115–134, 1998.

[25] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, pages 463–502. Edinburgh University Press, 1969.

[26] P. Shaker, J. Atlee, and S. Wang. A feature-oriented requirements modelling language. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 151–160, Sept 2012.

[27] S. Shapiro and Y. Lespérance. Modeling Multiagent Systems with CASL - A Feature Interaction Resolution Application. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII Agent Theories Architectures and Languages*, volume 1986 of *Lecture Notes in Computer Science*, pages 244–259. Springer Berlin Heidelberg, 2001.

[28] SWI-Prolog. Swi-prolog [online].

[29] U.S. National Highway Traffic Safety Administration. Safercar.gov [online].

[30] D. Weiss and R. Lai. *Software Product Line Engineering, A Family Based Development Process.* Addison Wesley, 1999.

[31] P. Zave. Requirements for evolving systems: A telecommunications perspective. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE)*, pages 2–9, 2001.

[32] P. A. Zimmer and J. M. Atlee. Ordering features by category. *Journal of Systems and Software*, 85(8):1782–1800, Aug. 2012.