Continuous Variable-Specific Resolutions of Feature Interactions

M. H. Zibaeenejad
University of Waterloo
Waterloo, ON, Canada
mhzibaee@uwaterloo.ca

Chi Zhang
University of Waterloo
Waterloo, ON, Canada
c335zhan@uwaterloo.ca

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

ABSTRACT

Systems that are assembled from independently developed features suffer from **feature interactions**, in which features affect one another's behaviour in surprising ways. The **Feature Interaction Problem** results from trying to implement an appropriate resolution for each interaction within each possible context, because the number of possible contexts to consider increases exponentially with the number of features in the system. Resolution strategies aim to combat the Feature Interaction Problem by offering default strategies that resolve entire classes of interactions, thereby reducing the work needed to resolve lots of interactions. However most such approaches employ coarse-grained resolution strategies (e.g., feature priority) or a centralized arbitrator.

Our work focuses on employing variable-specific default-resolution strategies that aim to resolve at runtime features' conflicting actions on system's outputs. In this paper, we extend prior work to enable co-resolution of interactions on coupled output variables and to promote smooth continuous resolutions over execution sequences. We implemented our approach within the PreScan simulator and performed a case study involving 15 automotive features; this entailed our devising and implementing three resolution strategies for three output variables. The results of the case study show that the approach produces smooth and continuous resolutions of interactions throughout interesting scenarios.

KEYWORDS

Feature interactions, software integration, internet of things

ACM Reference format:
M. H. Zibaeenejad, Chi Zhang, and Joanne M. Atlee. 2016. Continuous Variable-Specific Resolutions of Feature Interactions. In *Proceedings of 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, 4–8 September, 2017 (ESEC/FSE 2017),* **??** pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In **feature-oriented software development**, a system's functionality is decomposed into features, where each feature is an identifiable unit of functionality that may be optional or dynamically selectable (e.g., Cruise Control). Feature-orientation helps to address software complexity through decomposition into (feature) modules that can be considered, developed, and evolved independently. Software developers use features as the basis for incremental software development, where each software release is described in terms of new or updated features (as well as bug fixes). Feature modularity enables rapid development and integration of new features; and facilitates development of new systems through assembly of existing modules, possibly from multiple sources.

Although features are often treated as separate concerns, a feature may behave differently when integrated with other features in the system. 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 [**?**]. There has been considerable research into the automatic detection of feature interactions [**?**], but the real complexity lies in their resolution. Most interactions (although not all [**?**]) be *detected* through *pair-wise* analysis of features, which is quadratic in the number of features. However, each interaction needs to be analyzed and possibly resolved; and the ideal resolution depends on the presence of other features. For example, the desired speed of a car depends on the driver's actions (e.g., depression of the accelerator pedal) as well as which features related to speed have been activated (e.g., cruise control, speed-limit adherence, collision avoidance, stability control). Thus, each new feature comprises its core functionality plus some (possibly exponential) number of exceptions, each implementing desired behaviour in the context of a particular combination of other activated features. As the number of features grows, a software team finds that the development of a new feature is eventually dominated by the **Feature Interaction Problem**: the need to analyze feature combinations, resolve interactions, and verify resolutions [**?**].

The most effective solutions to the Feature Interaction Problem are feature-oriented architectures that coordinate the communications and interactions among feature modules [**? ? ? ? ?**]. The architectures enable rapid development, integration, and deployment of new features because they preserve the modularity of distinct features – but they do so at the expense of optimal resolutions of interactions. Interactions are resolved at runtime by the architecture's

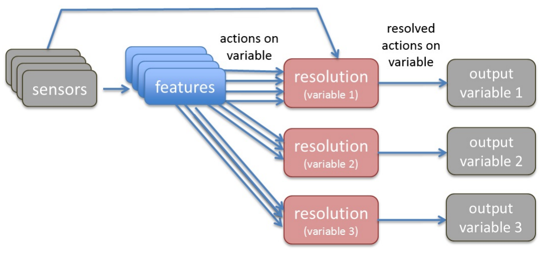

Figure 1: Architectural model for variable-specific resolutions

coordination strategies, which are usually based on feature priority or precedence, runtime arbitration, or negotiation. Such approaches for loose integration and coordination of features are becoming increasingly important, as they enable developers (and consumers!) to interconnect components and devices without considering their interactions. However, most of these resolution strategies are coarse grained, in that they are based on the priority or precedence of the features themselves rather than the features' interacting actions. As such, they provide suboptimal win/lose resolutions in which some features' actions are sacrificed in favor of other features' actions; and they often require an upfront total or partial ordering on features [**?**].

Bocovich and Atlee [**?**] have proposed a feature-oriented architecture with variable-specific resolutions, in which the developer can devise, for each system output variable, an appropriate default strategy for resolving conflicting actions on that variable. As simple examples, a resolution strategy for one output variable might be the *minimum* of the features' actions, whereas the strategy for another output variable might be the *average* of the features' actions. Their approach has several advantages, including that resolution strategies (1) are agnostic to the particular features, or even the number of features, involved in an interaction; (2) are fine-grained (c.f., feature priority); and (3) are specific to the variables being assigned (c.f., a uniform strategy like feature priority). However, their approach failed to consider the quality of default resolutions over a sequence of features' actions, or how resolution strategies might interact if their respective output variables are coupled.

Our work extends their architecture in a number of ways:

- Feature logic is separated from control logic, to distinguish between interactions that are amenable to (sub-optimal) default resolutions strategies versus computations that need to be optimal.
- **System features**, whose realizations are independent of specific input and output devices, are treated separately from **actuator features**, which apply to particular actuators.
- Non-cyclic information flows between resolution modules enable co-resolutions of interactions involving tightly-coupled variables.
- Our feature-action language is richer, and inputs to the resolution modules are richer, to promote smooth resolutions over consecutive execution steps.

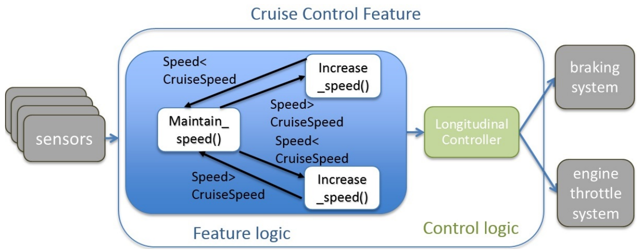

Figure 2: A model of a Cruise Control feature that combines its feature logic with its control logic.

- There is advice on how to accommodate **task features**, which have long-term actions.

We have evaluated our work within the PreScan simulator for advanced driver assistance systems (ADAS). We have implemented fourteen automotive features, have devised and implemented three default resolution strategies for the features' outputs, and have plugged these modules into our architecture. The results of the case study show that the approach produces smooth and continuous resolutions of interactions throughout interesting scenarios.

The rest of the paper is organized as follows. We first give an overview of the original work that promotes variable-specific resolutions (VSR) to feature interactions. In Section **??**, we describe extensions to this architecture, to support continuous variable-specific resolution (CVSR) of interactions over an execution sequence and co-resolution of interactions on tightly-coupled variables. In Section **??**, we present the implementation of our architecture, our variable-specific resolution strategies, and the results of the case study. Section **??** summarizes related work, and Section **??** concludes the paper.

2 VARIABLE-SPECIFIC RESOLUTION (VSR)

The architectural model of **variable-specific resolution (VSR)** is depicted in Figure **??**. Features are implemented as separate modules that execute in parallel, each reacting to current sensor values and issuing actions to the system's output variables. A resolution module is defined for each of the system's output variables (e.g., a separate resolution module for the brake pressure on each wheel of a vehicle). The role of the resolution modules is to resolve features' conflicting actions on the modules' corresponding output variables.

A system's execution is an iteration of periodic functions. In each iteration, the feature modules react to system inputs from sensors, by issuing actions on output variables. For example, a cruise control feature in a car (Figure **??**) will continuously monitor the car's speed and compare the current speed to the driver's desired cruising speed; if corrective action is required, the feature sends actions to the car's engine throttle system or to the braking system. Normally, the features' actions would flow directly to the output variables; instead, they flow to the variables' corresponding resolution modules. Every input to the resolution module includes meta-data regarding the action category, i.e., driver action, emergency action, safety action, non-safety action. Each resolution module inputs an arbitrary set of actions on some output variable, performs a pre-programmed variable-specific resolution strategy on those actions, and issues a resolved action or series of actions to its corresponding output variable.

In VSR, the developer is responsible for devising a resolution strategy for each output variable (or type of output variable). A primary benefit of this approach is that the number of resolution modules to be developed is typically much smaller than the number of possible optimal resolutions.[1] A second benefit is that resolution strategies do not depend on which features are active in the system or on the specific interactions to be resolved; thus, new features can introduce new actions on existing output variables without requiring any modifications to the resolution modules (although new features that introduce *new* output variables would require the development of new variable-specific resolution modules). As such, the VSR approach addresses the Feature Interaction Problem by drastically reducing the amount of work to be performed by the developer to resolve feature interactions.

However, the VSR architecture is a partial solution that only considers how singleton feature interactions are resolved. It raises questions about the effectiveness of default resolution strategies when applied continuously over a sequence of features' actions. Also, VSR does not accommodate the default resolution of interactions that involve coupled variables.

3 CONTINUOUS VSR

In this section, we introduce continuous variable-specific resolution (CVSR), which consists of a number of extensions to the original VSR architecture that are devised to resolve interactions effectively and continuously over a sequence of execution steps. We illustrate our work with a running example involving five advanced driver assistance systems (ADAS) and features:

Cruise Control (CC) keeps vehicle speed at driver-set value
Speed Limit Control (SLC) keeps vehicle speed below speed limit
Headway Control (HC) keeps a safe distance to the vehicle ahead
Anti-lock Braking System (ABS) keeps the wheels from locking
Traction Control System (TCS) keeps the wheels from spinning

Many of these features manipulate the same variables, increasing the likelihood of undesired interactions among the features. Specifically, all of these features have a direct or indirect impact on a vehicle's speed.

3.1 Feature vs. Control Logic

In previous works on resolving feature interactions at runtime cite, features operate directly on output variables and actuators. However, in dynamic systems like vehicle control, actions are highly optimized by nonlinear controllers that account for vehicle dynamics, environmental conditions (e.g., gradients in the road), fuel efficiency, and other performance factors. If multiple features were to attempt to control the same output variable or actuator (e.g., engine throttle) at the same time, with each feature issuing an optimized action on that actuator, it is unlikely that an engineer could devise an appropriate resolution strategy that (1) resolves an interaction among the multiple optimized actions, such that (2) the resolution is itself optimal.

[1] In the worst case, each feature interaction would require unique resolutions for every combination of active features.

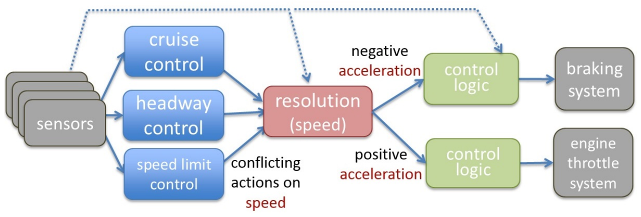

Figure 3: An overview of the CVSR architecture: feature-logic modules act on attribute variables; variable-specific resolution modules resolve conflicting actions on attributes; and control logic realizes and optimizes resolved actions.

Instead, we advocate separating features' intents from how those intents are realized through commands to actuators. In particular, we propose separation of **feature logic** from **control logic**, where feature logic is responsible for computing a feature's desired effects on the system, and control logic is responsible for realizing that effect and for optimizing its realization, through commands on output devices and actuators. To achieve this separation, we introduce **attribute variables**, which are abstract representations of system characteristics and outputs. A feature's intent is expressed as actions on attribute variables (thus actions are agnostic to the system's concrete actuators); and resolution modules resolve conflicts on attribute variables (thus their computations are also agnostic to the system's actuators). The control logic transforms the actions on attribute variables into optimized control signals to actuators.

With the control logic separated out of features' actions, the task of devising a default resolution strategy for resolving interactions among conflicting feature actions becomes possible: a resolution module is associated with each attribute variables (rather than with each output variable) to resolve conflicting actions on attribute variables; and then control logic is used to realize and optimize the resolved actions on attribute variables. As a concrete example, consider automotive features Cruise Control, Headway Control, and Speed Limit Control (in Figure **??**). The feature logic for each of these features determines a desired value for the vehicle's *speed* (an attribute variable). A resolution module for the attribute variable *speed* takes as input the features' actions on vehicle *speed* and computes a vehicle *acceleration* (i.e., the acceleration needed to achieve the resolved *speed* value). Depending on whether the resolved acceleration is positive or negative, the action is forwarded to the control logic for the engine throttle system or braking system, where each control-logic module is optimized for its respective actuator.

By separating feature logic from control logic, we are able to distinguish between features' intents, whose interactions are likely to be amenable to default resolution; versus the optimized realizations of feature actions, which are nonlinear in nature, and less likely to be amenable to default resolutions.

3.2 Dependencies Among Attribute Variables

When there are dependencies between attribute variables, their corresponding resolution modules cannot decide their values independently. For example, a vehicle's *steering angle* and its *speed* need to be considered together, if the vehicle is to turn safely without skidding.

In the case where attribute variables are tightly coupled but are realized by different actuators, the CVSR architecture allows for limited sharing of information among resolution modules, so that a resolution strategy for one attribute variable may use the projected next value of a related attribute variable (see Figure **??**(a)). For instance, to accommodate the dependency between *steering* and *speed* variables in our experiments, the output of the *steering angle* resolution module flows to the *speed* resolution module, to be used to calculate a safe turning speed. Information flows among the resolution modules must be acyclic to avoid a circular dependency. If a dependency cannot be made acyclic, then either the coupled attribute variables must be set by a single combined resolution module, or one of the resolution modules must make do with stale values of the coupled attributes (e.g., from the previous execution step or from the sensors).

A different type of dependency exists between uncoupled attribute variables that are realized by the same actuator. For example, Anti-Lock Braking (ABS) limits *wheel slip*, and Traction Control (TCS), controls *wheel spinning*. The two attribute variables are unrelated, which means that there are no interactions to resolve with respect to actions on the attributes. However, actions on both attributes are realized, at least in part, by the brake actuator, and the realizations can conflict with each other. CVSR defers the resolution of such interactions to the actuator's control-logic module (see Figure **??**(b)). Thus, the resolution module for each coupled attribute variable sends its output to their common control-logic module, and the control-logic module is responsible for coordinating and optimizing their realizations.

In general, dependencies among attribute variables impinge on our objective to modularize default strategies for resolving feature interactions. The impact is minor when dealing with tightly-coupled attribute variables: one of the coupled variable's resolution modules remains agnostic of features and other resolutions modules; whereas the resolution module for the second variable is designed to take into account the projected next value of the first variable. However, in the case of unrelated attribute variables that are realized by the same actuator, there is no possibility for default resolution of their interactions. The control-logic module needs to know the context of the actions it is realizing, in order to produce an optimized control signal for the actuator.

3.3 Actuator Features

The decomposition of features into feature logic and control logic and the introduction of attribute variables allows features to be agnostic to the actual output devices used to realize their actions; and allows resolution modules to resolve interactions on attribute variables without knowing how the resolved actions will be realized. However, some features are designed to enhance specific actuators; obviously, these features cannot be actuator agnostic. In our running example, Anti-Lock Braking (ABS) is an actuator feature. The intent of ABS is to improve the braking actuator behaviour so that wheel slip is minimized – by applying brake pressure intermittently. On the other hand, Cruise Control, Headway Control, and Speed Limit Control are features that manipulate characteristics of the whole system (i.e., speed, distance to the preceding vehicle) without regard to how those characteristics are realized through actuators.

Therefore, we distinguish between two feature types:

- **Actuator features**: features that enhance and extend the behaviour of an actuator.
- **System features**: actuator-agnostic features.

CVSR is designed to ease the resolution of interactions among system features only. Because actuator features are designed to extend and improve the behaviours of actuators, and because actuators and actuator features typically need to behave optimally, it is unlikely that one can devise an acceptable feature-agnostic default strategy for resolving interactions that involve actuator features. Thus, we exclude them from the set of features that CVSR coordinates by default resolution strategies, and instead CVSR delegates their coordination to the control-logic modules.

We recognize that some system features (e.g., emergency features) may also have behaviours that must be optimized. In our running example, Traction Control activates when a car is out of control due to *wheel slippage*; its actions are realized through precise reduction of engine torque and application of specific brake forces to individual wheels. In CVSR, system features whose behaviours need to be optimized are treated as actuator features: they are coordinated by control-logic modules, which are responsible for producing optimal resolutions and control signals for the actuators.

Thus, in CVSR, each execution step of the system consists of the following sub-steps: (1) system features react to sensor inputs by acting on attribute variables, (2) their (possibly conflicting) interactions are resolved according default resolution strategies implemented in resolution modules (one per attribute variable), (3) the resolved actions on attribute variables are realized by control-logic modules as optimized control signals destined for the system's actuators, (4) actuator and emergency features enhance these control signals, (5) a second set of control-logic modules resolve and optimize enhanced signals from multiple actuator features, and (6) the optimized control signals are sent to the actuators. An example of the CVSR architecture as it applies to an extended version of our running example of automotive system and actuator features is given in Figure **??**.

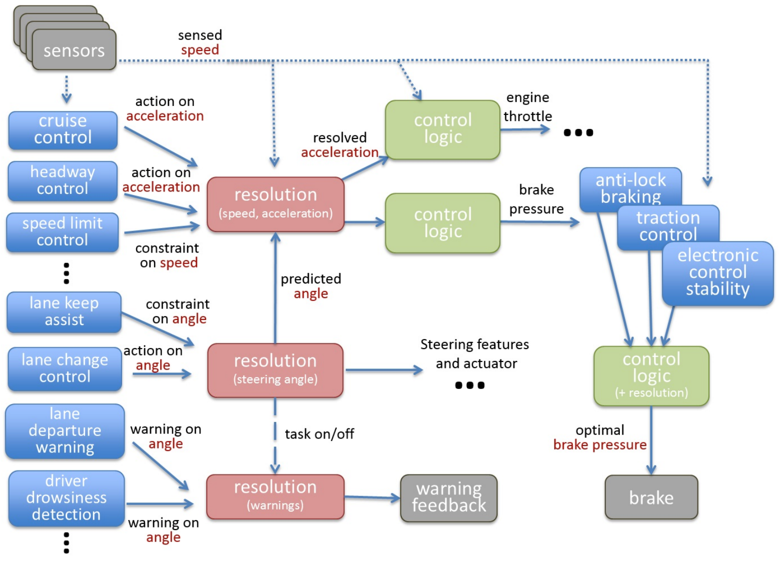

Figure 4: A partial model of the CVSR architecture instantiated with ADAS system and actuator features.

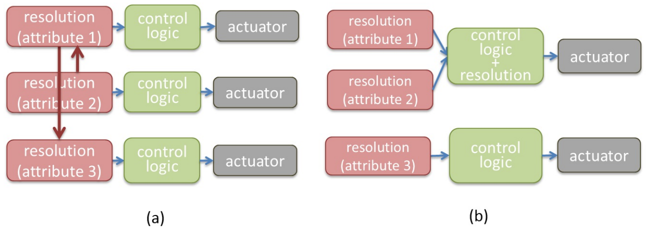

Figure 5: (a) Coupled attributes realized by different actuators, (b) Uncoupled attributes realized by the same actuator

3.4 Feature Action Language

In VSR [?], the feature action language is limited to assigning new values to output variables. Each feature action is (1) the target value of one variable, and (2) meta-data that indicates whether the action is a driver action, an emergency action, a safety action, or a non-safety action. This information is sufficient to resolve interactions effectively among a single set of feature actions [?]; but it results in turbulence and thrashing when the same resolutions are applied to multiple sets of feature actions over time. To improve the quality of interaction resolutions over an execution sequence, CVSR extends the feature action language to include not only target values of attribute variables, but also target derivatives of attribute values, and simple constraints on attribute values:

(1) **Target Attribute Value:** The target value of an attribute variable is still the truest reflection of a feature's intent with respect to a particular system attribute.

(2) **Target Derivative of Attribute:** For attribute variables that represent dynamic system attributes (e.g., *speed*), resolution modules produce smoother, more continuous behaviour if they resolve conflicts over target derivatives (i.e., target rates of change change) rather than over target values.

(3) **Attribute bounds:** In some cases, a feature's intent is to enforce a constraint [?]. For example, the intent of Speed Limit Control (*SLC*) is to keep a vehicle's *speed* at or below the current speed limit. If feature actions were limited to variable assignments, then a feature like *SLC* must continuously monitor the system and take corrective action whenever its intended constraint is violated. The effect on system behaviour is periodic violation and re-establishment of *SLC*'s *speed* constraint. To avoid such fluctuations in behaviour, CVSR extends the feature action language to allow lower and upper bounds on attribute values. The resolution module tries to produce resolved actions that satisfy given bounds, to avoid future corrective actions from features.

Figure **??** shows a simulation of Control Cruise (maintaining 100 km/hr) and *SLC* (adhering to a 90 km/hr speed

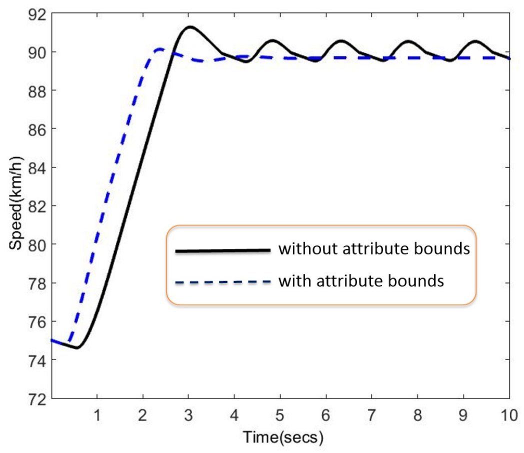

Figure 6: Car speed when Cruise Control is set to 100 km/hr and Speed Limit Control keeps the speed under 90 km/hr.

limit). Resolutions fluctuate more when the features' actions are target values of *speed* (solid line), compared to resolutions when SLC's action is a bound on *speed* (dashed line).

3.5 Task Features

So far, feature actions have been instantaneous: in each execution step of the system, features issue actions on attribute variables or control signals that are expected to be realized at the end of the execution step; and then in the next execution step, sensors report updated variable values and the features react by issuing new actions to be realized at the end of this next execution step; and so on. However, some feature actions are not instantaneous. A **task feature** is a feature that has at least one action that takes multiple execution steps to achieve and is non-interruptible. For example, Lane Changing (automatically move the vehicle to the centre of the next lane) and Automatic Park Assist (automatically perform parallel parking) are task features.

Because tasks should be completed without interruption, CVSR gives tasks priority over other feature actions. This requires some negotiation between a task feature and the resolution modules of the attribute variables that its task affects. Specifically, a task feature and the resolution modules need to agree on the initiation of a task: a task starts by seeking permission from the resolution modules of all the attribute variables that the task might act on; permission is granted when no other task is active and all warning signals are off. While the task is active, the affected resolution modules normally ignore actions from other features. Thus, during a lane change, the actions from Lane Keeping are ignored; and while parallel parking, the actions from Lane Keeping and Headway Control are ignored. The task ends by notifying resolution modules of the completion of the task. This protocol is discussed in more detail in Section **??**, where we present the resolution modules that we use in our studies.

A resolution module can abort a task in the event of a driver action or an emergency action from another feature (e.g., due to an impending collision). In such an event, the resolution module sends an abort message to the task feature, and the task feature notifies the other resolution modules that the task has completed. At present in CVSR, a task feature does not attempt to recover from an aborted task; instead, the driver, the emergency action, and other features work to react to the emergency situation. In the future, we plan to experiment with other reactions of task features to emergency situations, such forward commit or rollback.

4 CASE STUDY

To evaluate the quality of resolutions that result from using the CVSR approach, we conducted a simulation of the CVSR architecture populated with system features from the automotive domain and three resolution modules relevant to the features. We used the MatLab toolbox PreScan as our simulation tool.[2] PreScan is a physics-based simulation platform that is used in the automotive industry to develop Advanced Driver Assistance Systems (ADAS). Simulated vehicles can be equipped with a number of different sensors (e.g., camera, Lidar, radar, TIS). The simulation environment can be programmed with road segments (curvatures, speed limits, gradient) and actors (cars, bikes, pedestrians). Actuators are pre-built into the actors and cannot be directly manipulated. This limitation means that we cannot simulate actuator features such as ABS.

Our case-study features are taken from public documents that describe 22 Bosch driver-assistance and active-safety features.[3] Of these, four features (Anti-lock Braking, Traction Control, Active Steering, Electronic Stability Control) are excluded from the case study because they are actuator features; two additional features (Road Sign Assist, Driver Drowsiness Detection) are excluded because the PreScan simulator does not support the necessary sensors; one feature (Automatic Park Assist) is excluded because it is too complicated to implement and has no conflict interactions with the other features. Table **??** lists the remaining 15 features used in our case study and their descriptions. The third column indicates what the feature's outputs are, and the last column indicates what the associated resolution modules are.

4.1 Evaluation Criteria

To evaluate the quality of resolutions to feature interactions, we need to use multiple criteria. First and foremost is the degree to which a resolution adheres to the original intents and behaviours of the interacting features. However, given that a resolution, by its nature, changes the actions being performed by one or more interacting feature, we need secondary criteria for evaluating those resolutions that change features' actions.

There are global properties of the system that represent minimal criteria for correct behaviour of the overall system:

- The vehicle does not crash into other cars or obstacles.
- The vehicle stays on the road, and in the centre of the current lane except when changing lanes.
- The vehicle issues warning signals when a global property or feature property is violated.
- The vehicle's acceleration does not exceed 2.94 ms^2, except in due to emergency actions [**?**].
- The vehicle's lateral acceleration does not exceed $1.176 ms^2$ except due to emergency actions [**?**].

Lastly, variable values should not be *turbulent*, in order for sequences of resolutions to result in smooth and continuous system behaviour. A variable's value is deemed turbulent if it fluctuates around a target value. The higher the amplitude of the fluctuation, the worse the turbulence.

4.2 Programmed Resolution Modules

The essence of the CVSR approach are the resolution modules that are programmed to employ default resolutions strategies

[2]https://www.tassinternational.com/prescan

[3]https://goo.gl/4LSl3H

Feature Name	**Feature Functionality**	**Feature-Logic Output**	**Resolution Modules**
Speed Limit Control (SLC)	Automatically keeps the vehicle's speed at or below the road's speed limit.	Speed bound	Speed
Cruise Control (CC)	Maintains a desired speed of vehicle as set by the driver.	Acceleration	Speed
Headway Control (HC)	Keeps the vehicle within a safe distance behind the preceding vehicle.	Speed bound, acceleration	Speed
Traffic Jam Assist (TJA)	The car assumes the same speed as the car in front of it, following at a safe distance.	Acceleration	Speed
Pedestrian Protection Braking (PPB)	Automatically initiates emergency braking in case of impending collision with a pedestrian.	Acceleration	Speed
Predictive Braking System (PBS)	When warned of a collision, initiates partial braking to give the driver more time to react.	Acceleration	Speed
Emergency Braking System (EBS)	When warned of a collision, prepares the braking system for full braking, to be activated with any braking signal from the driver.	Acceleration	Speed
Lane Keeping Assist (LKA)	Issues appropriate steering commands to keep the vehicle in the center of the lane.	Steering angle	Steering
Lane Departure Warning (LDW)	Detects if the vehicle is about to move out of its lane, and warn the driver.	Warning	Warning
Collision Warning (CW)	Detects and warns the driver if the preceding vehicle is too close.	Warning	Warning
Pedestrian Protection Warning (PPW)	Detects an impending accident with pedestrians (who are in the same lane as the vehicle) and warn the driver of impending collision.	Warning	Warning
Lane Change Assist (LCA)	Detects and warns the driver if a vehicle is in the driver's blind spot or is approaching rapidly from the rear.	Warning	Warning
Rear Cross Traffic Alert (RCTA)	Detects and warns the driver if a vehicle crosses to the left or right behind the driver.	Warning	Warning
Parking Aid (PA)	Detects the presence of an object when the vehicle is moving in reverse, and warn the driver indicating the distance to the detected objects.	Warning	Warning
Lane Changing Control (LCC)	Automatically finishes a lane-change action.	Acceleration, steering angle	Speed, steering, warning

Table 1: Case study automotive system features.

that are most appropriate for resolving conflicts on their respective attribute variables. For our case study, we needed to devise three resolution modules for three attribute variables: *speed, steering angle,* and *warnings*.

The inputs to each resolution module are lists of actions and meta-data from an arbitrary number of features:

$Targets$ - array of features' target values
$Derivs$ - array of features' derivative targets
$Bounds$ - array of features' bounds on attribute variable
$Emerg$ - designated emergency actions
$CurValue$ - current attribute value as detected by sensors
RM - data from other resolution modules
$reqs$ - array of task requests from task features
$rels$ - array of task-complete notifications

The outputs of each resolution module is a resolved action on the attribute variable, or signals to task features:

$Action$ - resolved action on attribute variable
$Perm$ - array of responses to task initiation requests
$abort$ - signal to abort current task

In addition, there is an internal variable current_task, which designates a task feature that is currently executing a task.

In each execution step, the input arrays are uniformly indexed, such that actions and meta-data at index i are associated with the same feature. An array element with index i may be empty if there is no corresponding input for the corresponding feature. If the variable current_task is set, meaning that a task is currently executing, we will use current_task as an index into input arrays, to designate the inputs associated with the executing task feature.

4.2.1 *Steering-Angle Resolution Module.*

Our default strategy for resolving conflicts over actions on the vehicle's *steering angle* is given in Algorithm **??**:

Basic resolution strategy: Emergency actions have the highest priority (line 6). For non-emergency actions, the average value of the target angle values is used to balance the requirements of all of the steering features (line 17).

Task features: If no tasks are currently executing, then at most one request to initiate a task will be accepted (lines 12-15). Once a task is executing, then its actions have highest priority (line 10), until the task completes (lines 2-3) or is interrupted by an emergency action (lines 4-7).

Constraints/bounds: The final output Angle is no greater than the minimum input bound (line 19-20), except for emergency actions (line 6-7).

4.2.2 *Speed Resolution Module.*

Our default strategy for resolving conflicts over actions on the vehicle's *speed* is given in Algorithm **??**.

Basic resolution strategy: In automobiles, a smaller acceleration has a greater possibility of being 'safe' than a larger acceleration. Thus, the basic resolution strategy is to compute all of the target accelerations (the acceleration needed to achieve the lowest bound on speed (line 6), the acceleration needed to achieve the minimum target value (line 7), the minimum value among all of the target accelerations), and return the minimum of these (line 11).

Coupled variables: To keep the vehicle's lateral acceleration within an acceptable range, the speed resolution takes as input the projected value of the *steering angle* (provided by the steering resolution module), and calculates the upper bound on vehicle *speed* needed to maintain a lateral acceleration of less than $1.176 m/s^2$ (line 4).

Algorithm 4.1: Steering resolution

```
Input: Targets[],Bounds[],Emerg[],reqs[],rels[]
Output: Angle,Perm[],abort
1  Var: current_task
2  if rels[] is non-empty then
3      current_task = none;
4  if Emerg[] is non-empty then
5      abort current_task;
6      Angle = average(Emerg[]);
7      return;
8  switch current_task do
9      case not none do
10         Angle = Targets[current_task];
11         For all r ∈ reqs[]: Perm[r]=0;
12     case none do
13         if req[] is non-empty then
14             For some r ∈ reqs[]: current_task=r;Perm[r]=1;
15             For all i ∈ reqs[]\r: Perm[i]=0;
16         else
17             Angle = average(Targets[]);
18             }
19 if Angle > minimum(Bounds[]) then
20     Angle = minimum(Bounds[]);
```

Constraints/bounds: The algorithm computes the smallest constraint among the given Bounds[] and the computed *speed* bound needed to maintain a safe lateral acceleration (line 5); a (negative) target acceleration is computed to achieve this minimal bound if it is less than the vehicle's current *speed*.

Task features: The handling of tasks the same as in the *steering* resolution module, and is omitted here for brevity.

Algorithm 4.2: Speed resolution

```
Input: Targets[],Derivs[],Bounds[],Emerg[],NextAngle,CurSpeed,
Input: reqs[],rels[]
Output: Accel,Perm[],abort
1  if Emerg[] is non-empty then
2      Accel= min(Emerg[]);
3  else
4      Bound = Calc_Bound(NextAngle);
5      Bound = min(min(Bounds[]), Bound);
6      if Bound < CurSpeed then
7          Accel = (Bound - CurSpeed)/DefaultTime;
8      Accel2 = (min(Targets[]) - CurSpeed)/DefaultTime;
9      if Accel2 and Derivs[] is empty then
10         Accel = 0;
11     else
12         Accel =  min(Accel, Accel2, Derivs[]);
```

4.2.3 Warning Resolution Modules. For the sake of this case study, we assume that all the warnings related to the same attribute variable have the same display (for instance, a warning light or alarm for angle warnings and another one for speed warnings). Our resolution module for one such attribute (*steering angle*) is given in Algorithm **??**:

Basic resolution strategy: If any of the warning features for an attribute sends an alert, the warning display for that attribute should be on (line 10).

Task features: Permission to execute a task is granted only if the related attribute is not issuing a warning (lines 3-4). Otherwise, the handing of task requests and completions is similar to the other resolution modules.

Algorithm 4.3: Steering warning resolution

```
Input: Targets[],reqs[],rels[]
Output: Warning,Perm[],abort
1  if rels[] is non-empty then
2      current_task = none;
3  if max(Targets[])≠0 then
4      For all r ∈ reqs[]: Perm[r]=0
5  else
6      if reqs[] is non-empty then
7          For some r ∈ reqs[]: current_task=r, Perm[r]=1;
8          For all i ∈ reqs \ r: Perm[i]=0;
9      else
10         Warning = max(Targets[]);
```

4.3 Simulation Results

We first simulated features in isolation, to ensure that our evaluations of the quality of CVSR resolutions are not negatively affected by errors in the feature implementations. For each feature, we systemically devised a test suite comprising three types of test scenarios:

- **normal scenarios** that test a feature maintaining a behaviour (e.g., *CC* maintaining the current speed)
- **transition scenarios** that test a feature transitioning between two normal cases (e.g., *CC* targeting a cruising speed vs. maintaining that speed)
- **boundary scenarios** that test a feature at the boundary of a transition (e.g., *CC* behaviour when the vehicle speed is at or just below the cruising speed)

For each scenario, we determined whether feature logic was implemented correctly by comparing the simulation results against feature intent as well as against our quantitative and qualitative criteria (see Section **??**).

To assess the benefit of using derivatives in resolution-module calculations, we compared resolution strategies that make use of target derivatives against resolution strategies that do not use target derivatives. We tested feature combinations on a test suite of transition scenarios and boundary scenarios, and we compared the simulation results according to how well they adhere to the features' intents and how well they meet our quantitative and qualitative requirements. Our results show that, in transition scenarios, the resolution strategies that make use of derivatives produce resolutions that more closely adhere to the features' intents. For example, in scenarios involving Headway Control and a preceding car, the resolution strategy that uses derivatives slows the

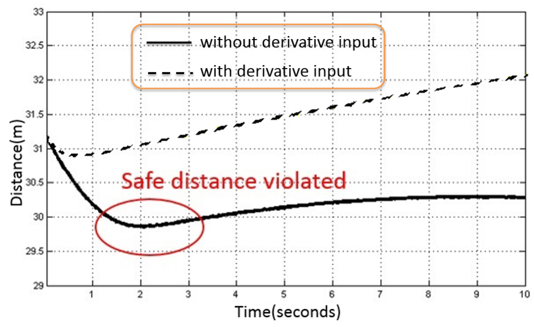

Figure 7: Improvement of resolutions due to the inclusion of derivatives in the resolution strategy.

vehicle down more quickly and reestablishes a safe distance to the preceding car more quickly (Figure **??**).

To assess the benefit of using attribute bounds in resolution-module calculations, we compared resolution strategies that make use of attribute bounds against resolution strategies that do not use attribute bounds. We simulated feature combinations on a test suites transition scenarios and boundary scenarios, our results show that, in all scenarios, the resolution strategy that does not make use of attribute bounds has greater turbulence in variables' values (with the amplitude of the turbulence greater in transition scenarios than in boundary scenarios). Whereas the resolution strategy that employs attribute bounds results in smoother resolutions of variable values over an execution (recall Figure **??**).

To assess the necessity of allowing information flows among resolution modules, we simulated *speed* and *steering* features in combination. Recall that in our implemented resolution modules, the projected next *steering angle* value is shared with the *speed* resolution module. We devised a test suite of normal scenarios and transition scenarios. Figure **??** shows the simulation results for a transition scenario in which the vehicle's lateral acceleration is exceeded: the vehicle enters a curve in the road at high speed, and the *steering* resolution turns the vehicle approximately 2° to keep the vehicle in the centre of the road. The changes in *steering angle* can be sudden, causing the lateral acceleration to exceed $1.176 m/s^2$; in response, the *speed* resolution modules decelerates the vehicle to reduce lateral acceleration.

Full simulation results can be found the second author's thesis (citation omitted).

4.4 Stress test

We use the following scenario as a stress test of our case study, to assess how a CVSR architecture performs in the presence of multiple feature interactions. The scenario comprises three phases: in phase one, the vehicle has a speed of 75 km/hr, Cruise Control (CC) is active and set to 100 km/hr, and the road has a speed limit of 90km/hr. As shown in the simulation results in Figure **??**, the vehicle's speed transitions smoothly up to the speed limit and stays there.

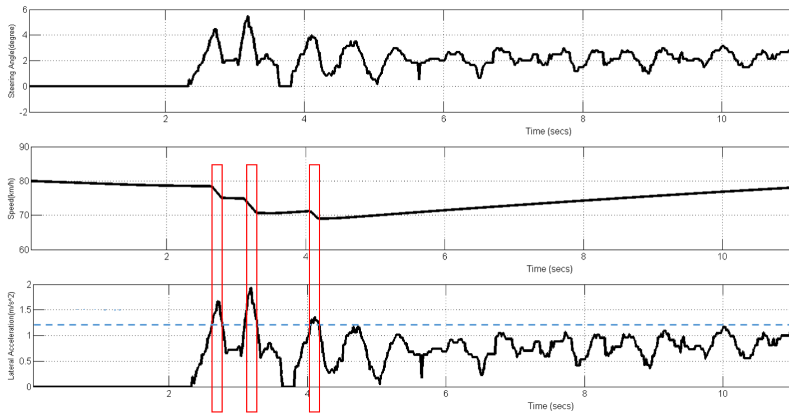

Figure 8: Constrain speed according to steering angle. The adjustments in speed are highlighted by rectangles.

In phase two of Figure **??**, the vehicle is cruising at the speed limit when a car traveling at speed 80km/hr cuts in front of the vehicle (11s into the simulation). Headway Control (HC) reacts by issuing speed and acceleration targets to slow down the vehicle, and the resolution (of CC, SLC, and HC actions) is to brake sharply until a safe distance is achieved with the preceding car (12s). At this point, HC issues a speed bound to maintain a safe distance, and the resolution is to increase vehicle speed to the features' minimum speed bound (i.e., the speed of the preceding vehicle). Eventually (15s), the driver invokes Lane Change Control (LCC) to change lanes and pass the preceding car. But there is another car in its blind spot, which means there is a Lane Change Assist (LCA) warning that prevents the steering resolution module from allowing the lane-change task.

In phase three of Figure **??**, the adjacent car leaves the vehicle's blind spot, which inactivates the LCA warning. When the driver re-invokes Lane Change Control, the resolution module permits the lane-change task. During lane changing, priority is given to LC actions (and actions of LCA, LDW, and DDD are ignored). Once the lane change is complete, the resolution module resumes default resolutions and the vehicle's speed smoothly progresses back to the speed limit.

4.5 Threat to Validity

The features were implemented by the authors, although they are based on descriptions from online sources. Differences between our implementations and an industrial vehicle's features may affect feature interactions, which in turn may threaten the validity of our results. Moreover, because significant effort is needed to implement features and to generate and run simulations, our case study comprises 15 features, and only one task feature and one pair of coupled variables. CVSR's approach to handling task features and dependencies among variables would benefit from further evaluation.

CVSR has been designed to apply generally to feature-oriented systems, including dynamic systems; but it has been evaluated only on features from the automotive domain. We

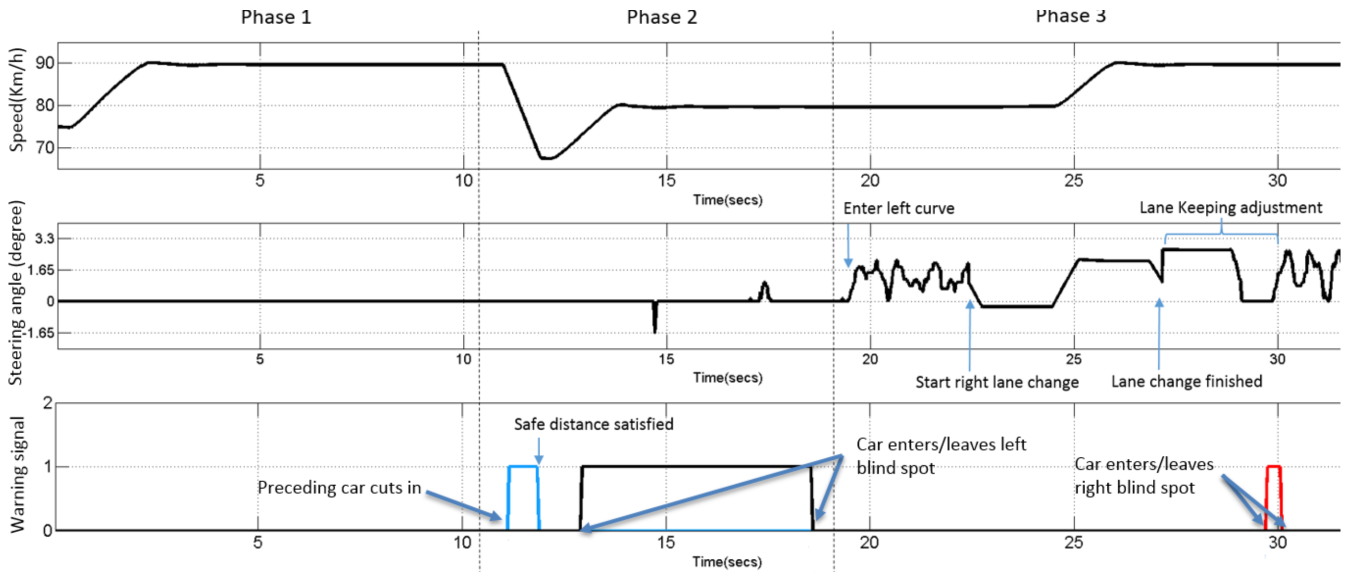

Figure 9: Simulation of a stress-test scenario.

cannot guarantee that the current architecture can be applied to other domains without additional case studies.

5 RELATED WORK

In general, there are three major approaches to addressing feature interactions: formal methods, software architectures, and online resolution techniques. Formal methods are mainly used to detect interactions among features. Mathematical models of features (e.g., logic or automata) and automated reasoning techniques (e.g., theorem proving or model checking) is used to detect interactions among features [? ? ? ?].

Architectural approaches interconnect feature modules, and address interactions through conventions and protocols that coordinate the features' executions and sharing of resources. Ccoordination is typically based on feature priority [? ? ? ? ?] or precedence [? ? ? ?], which orders the features' reactions to system inputs. Such approaches need a predetermined total or partial ordering on features, which can be time-consuming to derive; and there may not be a single ordering that is appropriate for resolving all feature conflicts. Moreover, such approaches favour the actions of one feature the the expense of other features, resulting in possibly sub-optimal resolutions. In contrast, CVSR resolutions consider all enabled actions, can be specialized to the variable being acted on, and do not require features orderings (or even prior knowledge of the features).

Online resolution techniques detect and resolve conflicts among features at runtime. Resolution strategies include feature priority, feature precedence, negotiation [?], arbitration [? ? ?], rollback [? ?], involving the user [?], feature suspension [?], and feature termination [?]. Most approaches either require a centralized authority that is responsible for detecting and resolving conflicts (e.g., arbitation, rollback), have high runtime costs (e.g., negotiation, involving the user), or result in coarse-grained resolutions.

Some approaches offer finer-grained resolutions. Laney et al. [? ?] 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, the number of interactions to consider and resolve is potentially exponential in the number of features.

The work closest to ours is variable-specific resolution (VSR) [?], which proposes an architecture that uses resolution modules to resolve conflicting actions at runtime. The resolution modules are variable specific, allowing engineers to devise an appropriate resolution strategy for each controlled variable rather than needing to devise a monolithic arbitrator that fixes all possible interactions. Our work extends VSR by (1) separating feature logic from control logic, and distinguishing between system features and actuator features, thereby clarifying which interactions are amenable to default resolutions and which need optimal resolutions; (2) extending the feature action language to include derivatives and bounds, resulting in higher-quality resolutions; (3) addressing interactions among variables that have interdependencies; and (4) accommodating task features. Automotive researchers [?] have independently proposed using speed limits as constraints to optimize the behaviour of Cruise Control, to achieve a smooth vehicle speed when transitioning between different speed-limit zones. Our work generalizes this idea, and uses bounds to prevent thrashing around a target attribute value.

6 CONCLUSION

We presented continuous variable default resolution (CVSR) for resolving at runtime interactions among ad-hoc combinations of features in a dynamic system. Our contributions over prior work include (1) separation of feature logic from control logic, (2) the distinction between system vs. actuator features, and how only system features are amenable to default resolutions, (3) extensions to the feature action language to include derivatives and constraints on attribute values, to promote smooth sequences of resolutions over time, and (4) preliminary support for task features. Lastly, we provide evidence in the form of a case study that CVSR can produce smooth continuous resolutions over executions paths.