Continuous Variable-Specific Resolutions of Feature Interactions

M. Hadi Zibaeenejad
University of Waterloo
mhzbibaee@uwaterloo.ca

Chi Zhang
University of Waterloo
c335zhan@uwaterloo.ca

Joanne M. Atlee
University of Waterloo
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 a 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 paths. 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.

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. Feature modularity enables rapid development of new features; and facilitates assembly of new systems from 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 [34]. There has been considerable research into the automatic detection of feature interactions [8], but the real complexity lies in their resolution. Most interactions (although not all [2]) 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 [6].

The most effective solutions to the Feature Interaction Problem are feature-oriented architectures that coordinate the communications and interactions among plug-in feature modules [19, 23, 24, 35, 36]. The architectures enable rapid development, integration, and deployment of new features because they preserve the modularity of distinct features. 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 – but they do so at the expense of optimal resolutions of interactions. Interactions are resolved at runtime by the architecture’s coordination strategies, typically 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 favour of other features’
actions; and they often require an upfront total or partial ordering on features [38].

Bocovich and Atlee [5] 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. A simple resolution strategy for one output variable might take the minimum of the features’ actions, whereas the strategy for another variable might be the average of the features’ actions. The 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 (cf. feature priority); and (3) are specific to the variables being assigned (cf. a uniform strategy like feature priority). However, the approach fails to consider the quality of default resolutions over a sequence of features’ actions, or how to coordinate the resolution of interactions among output variables that are coupled.

Our work extends their architecture in a number of ways:

- Better separation of functionality that is amenable to default resolutions of interactions, from functionality that needs to be optimized (e.g., control logic).
- Data sharing among resolution modules enables resolution of interactions involving coupled variables.
- Our feature-action language is richer to promote smooth resolutions over consecutive execution steps.
- 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 modelled in Simulink fifteen 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 3, we describe extensions to this architecture, to support continuous variable-specific resolution (CVSR) interactions over a sequence of execution steps, and co-resolution of interactions on tightly-coupled variables. In Section 4, we present the implementation of our architecture, our variable-specific resolution strategies, and the results of the case study. Section 5 summarizes related work, and Section 6 concludes the paper.

2 VARIABLE-SPECIFIC RESOLUTION (VSR)

The architectural model of variable-specific resolution (VSR) is depicted in Figure 1. 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 system output (e.g., a separate resolution module for brake pressure, instantiated for 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 (Figure 2) continuously monitors the car’s speed and compares it to the driver’s desired cruising speed; if corrective action is required, the feature sends actions to the car’s engine throttle system or the braking system. Normally, the features’ actions would flow directly to the output variables. In VSR, actions flow to the variables’ corresponding resolution modules. Each resolution module performs a pre-programmed variable-specific resolution strategy on the actions it receives, and issues a resolved action(s) to its corresponding output variable.

In VSR, the developer is responsible for devising a resolution strategy for each 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 [5]. 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. Actions can include meta-data that allow resolution strategies to give preference to predefined action categories (e.g., driver actions vs. emergency actions, safety actions, non-safety actions), but strategies do not give preference to specific actions or features. 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.

Figure 1: Architectural model for variable-specific resolutions

Figure 2: A model of a Cruise Control feature that combines its feature logic with its control logic.
When applied continuously over a sequence of feature’s 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
- **Electronic Stability Control (ESC)** keeps the vehicle from skidding

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

### 3.1 Feature vs. Control Logic

In VSR [5], 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 issue, respectively, optimized actions on the same output variable or actuator (e.g., engine throttle) at the same time, it is unlikely that an engineer could devise an appropriate resolution strategy that (1) resolves the interaction among the multiple optimized actions, such that (2) the resolution is itself optimal.

Instead, we propose separating feature logic from control logic, such that feature logic is responsible for computing a feature’s intended effects on the system, and control logic is responsible for realizing and optimizing the intended effects 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, and resolution modules resolve conflicts on attribute variables (thus both features and resolution modules are agnostic to the system’s actuators). The control logic transforms the actions on attribute variables into optimized control signals to actuators.

As a concrete example, consider automotive features Cruise Control, Headway Control, and Speed Limit Control (in Figure 3). The feature logic for each feature determines a desired value for the vehicle’s speed (an attribute variable). A resolution module 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 resolved 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

If attribute variables have dependencies, their resolution modules cannot independently decide their values. For example, a vehicle’s steering angle and its speed need to be considered together, for the vehicle to turn safely without skidding.

When attribute variables are tightly coupled but are realized by different actuators, the CVSR architecture allows limited sharing of information among resolution modules (see Figure 4(a)). In our running example, the output of the steering angle resolution module flows to the speed resolution module, so that the latter can 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 (e.g., from the previous execution step or the sensors).

A different type of dependency exists between attribute variables that are realized by the same actuator. For example, Anti-Lock Braking (ABS) limits wheel slip, and Electronic Stability Control (ESC) prevents vehicle skidding. The two attribute variables are unrelated. However, actions on both attributes are realized, at least in part, by the braking system, and the realizations can conflict. CVSR defers the resolution of such interactions to the actuator’s logic for the engine throttle system or braking system, where each control-logic module is optimized for its respective actuator.

![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.](image)

![Figure 4: (a) Coupled attributes realized by different actuators, (b) Uncoupled attributes realized by the same actuator](image)
control-logic module (see Figure 4(b)). Thus, the resolution module for each coupled attribute variable sends its output to their common control-logic module, and the control-logic module coordinates and optimizes their realizations.

In summary, dependencies among attribute variables impinge on our objective to modularize resolution strategies. The impact is minor when dealing with tightly-coupled attribute variables: one of the coupled variable’s resolution modules is unchanged; but its output is shared with the other variable’s resolution module; both modules remain feature agnostic. In contrast, when attribute variables are realized by the same actuator, there is no possibility for feature-agnostic resolutions of interactions because the control-logic module needs to know the contexts of the actions it is realizing, in order to produce an optimized control signal for the actuator.

3.3 Actuator Features
The introduction of attribute variables allows features to be agnostic to the actual output devices used to realize their actions. However, some features are designed to enhance specific actuators, and they 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 and 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 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, in CVSR, actuator features are not coordinated by default resolution strategies; instead CVSR delegates their coordination to the control-logic modules.

Some system features (e.g., emergency features) may also have behaviours that must be optimized. In our running example, Electronic Stability Control activates when the vehicle skids due to oversteering or understeering; its actions are realized through precise application of brake forces to specific wheels. In CVSR, system features that must be optimized are coordinated directly by control-logic modules, which produce optimal resolutions and control signals for the actuators.

Thus, in CVSR, an execution step of the system proceeds as follows: (1) system features react to sensor inputs by acting on attribute variables, (2) their interactions are resolved according to default resolution strategies implemented in feature-agnostic resolution modules (one per attribute variable), (3) the resolved actions on attribute variables are realized by feature-agnostic control-logic modules as control signals destined for the system’s actuators, (4) actuator and emergency features enhance these control signals, (5) a second set of feature-knowledgeable control-logic modules resolve and optimize enhanced signals from multiple actuator features, and
(6) the optimized control signals are sent to the actuators. Figure 5 depicts the CVSR architecture applied to an extended version of our running example.

### 3.4 Feature Action Language

In VSR [5], the feature action language is limited to (1) assignments of new values to output variables and (2) meta-data that indicates whether an action is a driver action, an emergency action, a safety action, or a non-safety action. This information is sufficient to resolve interactions that occur in a single execution step [5]; but when the same resolution strategies are applied to feature actions in a sequence of execution steps, the result is turbulence and thrashing. To improve the quality of interaction resolutions over the course of an execution path, 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 in how the attribute’s value should change, such as target acceleration).

3. **Attribute bounds:** In some cases, a feature’s intent is to enforce a constraint [19]. 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 6 shows a simulation of Control Cruise (maintaining 100 km/hr) and SLC (adhering to a 90 km/hr speed limit). Resolutions fluctuate more when the features’ actions are target values of speed (solid line), compared to resolutions when SLC’s action is an upper 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 4, 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

CVSR is the outcome of our attempt to apply VSR to continuously executing features. We started with six features (shaded in Table 1), devised resolution modules for their attribute variables, and simulated the system of features and resolution modules; as we encountered challenges (posed by control logic, quality of resolutions, task features), we devised solutions that make up CVSR.

To evaluate CVSR, we performed a case study involving a larger set of features (not known a priori) to assess whether CVSR default...
Table 1: Case study automotive features. Shaded features are those used to devise CVSR. Bold features are PreScan features.

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

resolutions are acceptable when applied to a sequence of execution steps in which features’ actions continuously interact with each other. Determination of acceptability is not straightforward because runtime resolutions to feature interactions, by definition, cause the system behaviour to deviate from the features’ collective intentions; thus feature specifications cannot serve as strict correctness criteria for assessing system behaviour. Instead, we used a combination of criteria. Firstly, we considered the degree to which resolutions adhere to the original intent and behaviours of interacting features. Secondly, we used a set of global properties (some from the automotive literature) 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 m/s², except due to emergency actions [21].
- The vehicle’s lateral acceleration does not exceed 1.176m/s² except due to emergency actions [10].

Thirdly, we used the degree of turbulence in variable values as a means of assessing whether the system’s behaviour is smooth and continuous. 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.

We used the MatLab toolbox PreScan as our simulation environment. PreScan is state of the art for designing and model-in-the-loop simulations of Advanced Driver Assistance Systems (ADAS) and their effects on vehicle dynamics [17, 20]. PreScan provides facilities for creating simulation scenarios from a database of road sections, infrastructure components (trees, buildings, and traffic signs), actors (cars, trucks, bikes, and pedestrians), weather conditions (rain, snow, and fog) and light sources (sun, headlights, and lampposts), but not road gradient. Scenarios are created manually. Recent research automatically generates test scenarios from a specification of the relevant search space and goals [1, 33]. We have not investigated its applicability to our scenario generation. In simulation, sensor data is generated at a frequency of 100 Hz.

In PreScan, actuators are built-in, which precludes our case study from adding actuator features that modify actuator behaviour. Fortunately, this limitation does not affect our evaluation of CVSR’s resolutions, because CVSR does not attempt to resolve interactions among actuator features. Also, PreScan does not provide realistic data on the performance overhead of our architecture or latency of default resolutions of feature interactions. Thus, our evaluation of CVSR focuses only on the quality of its resolutions to feature interactions.

---

1In [5], VSR resolutions were evaluated only with respect to feature actions in a single execution step.

2https://www.tassinternational.com/prescan
The case study involves 22 Bosch ADAS and active-safety features taken from public documents. 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 PreScan 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. Of the 22 original features, 15 features (listed in Table 1) are applicable to our simulation infrastructure. The six features with which we started are among the 15 case-study features. The boldface features are provided by PreScan; all other case-study features were modelled in Simulink by the second author. The third column of Table 1 indicates each feature’s outputs, and the last column indicates the feature’s associated resolution modules.

4.1 Programmed Resolution Modules

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

4.1.1 Steering-Angle Resolution Module. Our default strategy for resolving conflicts over actions on the vehicle’s steering angle is given in Algorithm 4.1:

```plaintext
4.1.2 Speed Resolution Module. Our default strategy for resolving conflicts over actions on the vehicle’s speed is given in Algorithm 4.2:
```

<table>
<thead>
<tr>
<th>Basic resolution strategy</th>
<th>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).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task features</td>
<td>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).</td>
</tr>
<tr>
<td>Constraints/bounds</td>
<td>The final output Angle is no greater than the minimum input bound (line 19–20), except for emergency actions (line 6–7).</td>
</tr>
</tbody>
</table>

**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 ∈ req[]: current_task=r; Perm[r]=1;
15.         For all i ∈ req[]\r: Perm[i]=0;
16.      else
17.         Angle = average(Targets[]);
18.      end
19.   if Angle > minimum(Bounds[]) then
20.      Angle = minimum(Bounds[]);
21. end
```

4.1.2 Speed Resolution Module. Our default strategy for resolving conflicts over actions on the vehicle’s speed is given in Algorithm 4.2:

**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 lowest 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 1.176m/s² (line 4).

**Constraints/bounds:** The algorithm computes the smallest constraint among the given Bounds[] and the computed speed bound
Algorithm 4.2: Speed resolution

\textbf{Input:} Targets[], Derivs[], Bounds[], Emerg[], NextAngle, CurSpeed,
\textbf{Input:} reqs[], rels[]

\textbf{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[]);

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

4.1.3 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 4.3:

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

\textbf{Input:} Targets[], reqs[], rels[]

\textbf{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[]);

4.2 Simulation Results

We first simulated features in isolation, to ensure that our evaluation of the quality of CVSR resolutions are not negatively affected by errors in the feature implementations. For each feature, we systematically devised a test suite that (1) tests the feature maintaining a steady behaviour, (2) tests the feature transitioning between behaviours (e.g., CC targeting a cruising speed vs. maintaining that speed), and (3) tests the feature at the boundary of a transition (e.g., CC when the vehicle speed is at or just below the cruising speed).

We then simulated feature combinations to evaluate the quality of default resolutions to feature interaction. Our simulation strategy was to create scenarios that guided the simulations towards expected interactions. For each pair of features, we identified potential interactions, and for each potential interaction we devised normal scenarios that stay clear of interactions (to assess default resolutions in the absence of interactions) transition scenarios that exercise the interaction boundary scenarios that barely interact.

Each scenario was simulated ten times, and the results were compared against the features’ intent and our quantitative and qualitative criteria (see the beginning of this 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 vehicle down more quickly and reestablishes a safe distance to the preceding car more quickly (Figure 7).

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 suite of transition scenarios and boundary scenarios, and our results show that, in all scenarios, the resolution strategy that does not make use of attribute bounds has greater turbulence in variables’
We use the following scenario as a stress test of our case study, when a car traveling at speed 80 km/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 9, 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.

### 4.4 Threat to Validity

The features were implemented by the authors. We attempted to reduce bias by basing features on descriptions from online sources. Differences between our implementations and industrial versions of these features may affect feature interactions, which in turn may threaten the validity of our results.

Evaluation by simulation is expensive and is limited by the time it takes to design, implement, and run simulation scenarios. Thus, our case study comprises only 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. Moreover, while CVSR is designed to apply generally to feature-oriented systems, including dynamic systems, it has been evaluated only on features from the automotive domain. There is nothing automotive-specific about our architecture, and we hypothesize that it would generalize to other feature-rich control systems. But this is not evaluated.

### 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 [3, 12, 14, 25].

Architectural approaches interconnect feature modules, and address interactions through conventions and protocols that coordinate the features’ executions and sharing of resources. Coordination is typically based on feature priority [11, 16, 19, 22] or precedence [4, 9, 23, 38], 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

---

For models see https://ece.uwaterloo.ca/~mhzibaee/fse

---

Figure 8: Constrain speed according to steering angle. The adjustments in speed are highlighted by rectangles.
of one feature 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 [18], arbitration [15, 24, 31], rollback [7, 29], involving the user [13], feature suspension [30], and feature termination [29]. Most approaches either require a centralized authority that is responsible for detecting and resolving conflicts (e.g., arbitration, 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. [26, 27] 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.

In previous work [5], we proposed a variable-specific resolution (VSR) 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.

In automotive software, feature interactions are typically resolved at design time, with each resolution treated as a bug fix that "fixes" how features behave in combination in a particular context (a la the Prius example at the start of this paper). This approach does not scale once the number of features grows to be in the high three digits. More recently, arbiters (a type of resolution module) have been used to resolve conflicts among a small number of features [31]. In these approaches, the arbiter either resolves interactions at runtime using feature priority or encodes an ad-hoc, interaction-specific resolution (for instance, [28] investigates the interactions among adaptive cruise control, 'speed limit follower' and 'curve speed control' features and designs an arbiter that ensures a smooth speed profile in transitioning between different situations). However, ad-hoc runtime resolution suffers the same problem of scalability as the original approach of resolving all interactions at design-time; and feature-priority resolution assumes that features have the same relative priority in all contexts, which is not always true [32].

6 CONCLUSION

We presented continuous variable-specific 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 execution paths.
REFERENCES


