Property-Driven Runtime Resolution of Feature Interactions

Santhana Gopalan Raghavan[1], Kosuke Watanabe[2], Eunsuk Kang[3](✉), Chung-Wei Lin[4], Zhihao Jiang[5], and Shinichi Shiraishi[2]

[1] University of Southern California, Los Angeles, USA
santhanr@usc.edu

[2] Toyota InfoTechnology Center, Mountain View, USA
{kwatanabe,sshiraishi}@us.toyota-itc.com

[3] Carnegie Mellon University, Pittsburgh, USA
eskang@cmu.edu

[4] National Taiwan University, Taipei, Taiwan
cwlin@csie.ntu.edu.tw

[5] ShanghaiTech University, Shanghai, China
jiangzhh@shanghaitech.edu.cn

Abstract. The feature interaction problem occurs when two or more features interact and possibly conflict with each other in unexpected ways, resulting in undesirable system behaviors. Common approaches to resolving feature interactions are based on priorities, which are ineffective in scenarios where the set of features may evolve past the design phase, and where desirability of features may change dynamically depending on the state of the environment. This paper introduces a *property-driven* approach to feature-interaction resolution, where a desired system property is leveraged to determine which feature action should be enabled at a given context. Compared to existing approaches, our approach is capable of (1) providing resolutions even if the system evolves with new or modified features, and (2) handling complex resolution scenarios where the preference of one feature over the others may change dynamically. We demonstrate the effectiveness of our approach through a case study involving resolution of safety-critical features in an intelligent vehicle.

1 Introduction

The *feature interaction* problem occurs when two or more features interact and possibly conflict with each other in unexpected ways, resulting in undesirable system behaviors [3]. Feature interactions are becoming an important issue in emerging domains such as the Internet of Things and intelligent automotive systems, where the outcome of an unexpected interaction may pose significant safety or security risks [8,16,26]. For instance, a pair of independent safety features in a vehicle may attempt to send conflicting acceleration commands to the engine controller, possibly violating a safety requirement that would have been satisfied if each feature had existed in isolation.

C. Colombo and M. Leucker (Eds.): RV 2018, LNCS 11237, pp. 316–333, 2018.
https://doi.org/10.1007/978-3-030-03769-7_18

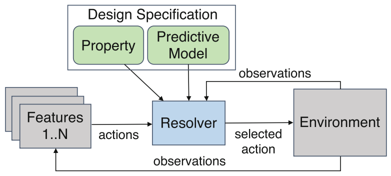


Fig. 1. Overview of the proposed resolution framework.

Most common approaches to resolving conflicts between features leverage some notion of *priorities* [4,5,13,24,28]. Typically, a total or partially-ordered ranking of features is determined at design time, and an arbitration procedure is applied at runtime to enable the actions of the highest ranking feature when a conflict occurs. However, a priority-based resolution strategy suffers from two major shortcomings. First, in certain domains, it may be difficult or impossible to predict the set of potential features that may be integrated into a system. Many modern vehicles, for example, are designed with a capability to download new applications or modify existing ones through over-the-air (OTA) updates, and the architecture of an in-vehicle software system is likely to evolve well beyond its deployment into the market.

Second, certain types of resolution decisions are *context-dependent*, in that the most desirable feature may depend on the state of the surrounding environment at a particular time. For instance, to reduce the risk of collision, a feature that results in increased acceleration may sometimes be preferable to one that attempts to reduce speed (e.g., in scenarios where a vehicle is being tailgated by another speeding vehicle within an unsafe distance). A static resolution strategy that always favors certain features over the others is insufficient to support this type of dynamic resolution, where contextual information plays a crucial role.

This paper introduces a novel *property-driven* approach to feature-interaction resolution that is designed to address these two shortcomings. The high-level overview of the proposed approach is shown in Fig. 1. Along with a set of feature actions, our resolver takes three different types of inputs: (1) a desired *property* to be fulfilled (e.g., "The distance to the preceding vehicle must be at least some minimum value"), (2) a *predictive model* that describes how the system and its environment evolve given a particular action (e.g., a model of changes in velocity over time given an acceleration), and (3) a set of *observations* that represent the current *context*, i.e., the state of the environment (e.g., velocities of the surrounding vehicles). The resolver then uses the model to evaluate potential consequences of each action and determine which of the conflicting features should be enabled to satisfy the property in the current context.

Instead of relying on a pre-determined priority list, our approach decouples resolution decisions from the presence of particular features and thus, is capable of providing resolutions even if the system evolves with new or modified features. In addition, since our approach does not rely on fixed resolution strategies, it is capable of handling complex resolution scenarios where the preference of one feature over the others may change dynamically depending on their satisfactions of the property in a given context.

In particular, we are interested in investigating the problem of feature interaction in *cyber-physical systems* (CPS), where the behavior of the system and the environment can be represented as an evolution of continuous variables (e.g., velocity or distance) over time. To express properties about these types of systems, we adopt Signal Temporal Logic (STL) as the underlying property specification language [15]. Sometimes, multiple actions may satisfy the desired property in a given context and thus, cannot be distinguished from each other. To resolve this issue, we leverage the notion of *robustness* of satisfaction [11] as a quantitative metric to measure how well the property is satisfied by a given action and distinguish competing features that both (dis-)satisfy the property.

We have built a prototype implementation of the proposed resolution framework as a part of an in-house simulation environment for designing and testing vehicle systems. To demonstrate the effectiveness of our approach, we applied this framework to a case study involving a set of safety features from the automotive domain. The outcome of this study shows that our approach can effectively resolve conflicts among features and ensure that the system performs the actions that are most satisfactory with respect to a given safety property.

This paper makes the following contributions:

– A novel, *property-driven* approach to feature-interaction resolution, which applies the notion of the *robustness* of property satisfaction to resolve conflicts among competing features (Sect. 4),
– A prototype implementation of the proposed approach (Sect. 5.1), and
– A case study demonstrating the effectiveness of the approach on a set of automotive safety features (Sect. 5.2).

The paper concludes with a discussion of the related work (Sect. 6), current limitations with the proposed framework and potential directions for further extending the property-driven approach (Sect. 7).

2 Motivating Example

Modern vehicles are equipped with a set of safety features called *advanced driver-assistance systems* (ADAS). One common ADAS feature is called *cruise control* (CC), which is intended to automatically maintain the speed of the *ego* vehicle (i.e., the vehicle being controlled) to the driver-set speed. To achieve its objective, CC sends an acceleration request to the engine controller, which, in turn, generates a corresponding actuator command to increase the engine torque until the vehicle reaches the desired acceleration.

Another ADAS feature, called speed limit control (SLC), is designed to automatically reduce the speed of the vehicle to a legal limit that is obtained from the surrounding environment (e.g., by detecting a speed limit sign or a GPS location). SLC operates by sending a sequence of requests to the brake controller until the vehicle reaches the desired speed limit.

One desirable safety property of the ego vehicle can be stated as:

P1: *The time to collision (TTC) between this vehicle and a nearby vehicle must always be above TTC_{min}.*

where TTC_{min} is some constant threshold determined by automotive engineers (enough time for a driver to react; e.g., 5 s). The actual time-to-collision at a given moment depends on the acceleration, velocity, and distance between a pair of vehicles, and computed using information from on-board sensors.

Conflict Scenario. Consider a scenario with three vehicles sharing a single lane, as shown in Fig. 2. For the purpose of this example, vehicle B is designated to be the system that we wish to control, and the leading and following vehicles (A and C) are considered to be part of the environment. Initially, vehicle A is moving at a constant speed of 60 km per hour (km/h), and B decides to catch up to A from its initial speed of 40 km/h by enabling CC.

Suppose that vehicle B approaches an area with a speed limit of 40 km/h, and the SLC feature begins sending brake requests in order to limit the vehicle acceleration. This results in one type of *feature conflict*: Two independent features (i.e., CC and SLC), each trying to achieve its own goal, attempt to manipulate the same system variable (i.e., acceleration) in an inconsistent manner.

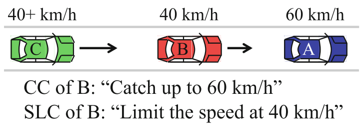

40+ km/h
40 km/h
60 km/h
C
B
A
CC of B: "Catch up to 60 km/h"
SLC of B: "Limit the speed at 40 km/h"

Fig. 2. Sample driving scenario.

Existing Methods. One way to resolve this conflict is to assign to each feature a *priority rank* that indicates the level of criticality, and have the feature with the highest priority (e.g., SLC) be selected over those with lower ranks when a conflict arises (e.g., CC). An alternative approach is to design and assign a specific resolution strategy to each system variable that may be manipulated by multiple features. For instance, one possible strategy, given multiple features that attempt to manipulate the acceleration, may select the one that results in the lowest acceleration (e.g., SLC)—the reasoning being that the slower the vehicle speed is, the safer it is likely to be.

While the latter approach has the advantage that it is feature-agnostic, it may still lead to unsafe outcomes in scenarios that the specific resolution strategy is not designed to handle. For instance, suppose that the following vehicle C begins to rapidly accelerate and exceed the speed of vehicle B. As vehicle C approaches B within an unsafe distance (thus, reducing the TTC between the two vehicles), the safer action to take in this scenarios is arguably to increase, not decrease, the acceleration of vehicle B to avoid a possible collision.

Proposed Method. Our approach to resolution, in comparison, evaluates the feature actions *with respect to the property* and selects the one that is most likely to satisfy it. For instance, as vehicle C speeds towards B from the rear and the TTC approaches the safe threshold (TTC_{min}), our resolver determines that accelerating the vehicle is more likely to satisfy the above property (**P1**) and selects CC over SLC.

Suppose, however, that as vehicle B speeds up towards 60 km/h, the leading vehicle A begins to slow down, and the TTC between A and B begins approaching the safety threshold (thus increasing the chance of collision ahead). Under this circumstance, the resolver determines that the safer action (as determined by property **P1**) is to decelerate vehicle B, and chooses SLC over CC.

Challenges. Note that the desirability of a feature may change depending on the context; i.e., a feature action may satisfy a desired property in one scenario while failing to satisfy it in a different scenario. To make this type of *context-dependent* decision, the system must explicitly take into account the information about the current and future states of the environment. Furthermore, if none of the competing features satisfies the property (or if all of them do), the system must still be able to make a meaningful choice between them. In the following sections of the paper, we introduce how our resolution framework leverages (1) a model of the environment to evaluate the desirability of a feature within a given context, and (2) the notion of *robustness* of satisfaction to select the action that is *most satisfactory* with respect to the given property.

3 Background

We are interested in designing a resolution framework to ensure the safety of CPSs, which share two common characteristics: (1) timing is often an important part of system requirements (e.g., "the vehicle must come to a full stop within the next 3 s"), and (2) certain aspects of system states are best captured using continuous domains (e.g., velocity). To express properties about such a system, we adopt a formal specification logic called *signal temporal logic* (STL) [15].

Behavior as Signals. In this approach, the state of a system and its evolution over time is captured using the notion of a *signal.* A signal over domain D is a function $\mathbf{s} : T \rightarrow D$, where *time domain* T is a finite or infinite set of real numbers that represent a particular point in time ($T \subseteq \mathbb{R}_{\geq 0}$). A typical system consists of multiple state variables, and so the value of a signal is represented as a tuple of k real numbers ($D \subseteq \mathbb{R}^k$); i.e., $\mathbf{s}(t) = (v_1, \ldots, v_k)$. For convenience, we use the subscript notation $\mathbf{s}_i(t)$ to denote the i-th component of the signal at time t (for $1 \leq i \leq k$).

Example. Suppose that the state of a vehicle at time t is modeled as tuple $\mathbf{s}(t) = (v, a)$, where v and a correspond to the velocity and acceleration of the vehicle, respectively. The signal $\mathbf{s} = \{(t_0, (30.0, 2.5)),\ (t_1, (32.5, 2.5)),\ (t_2, (35.0, 2.5))\}$ depicts a behavior of the vehicle as it speeds up from 30 to 35 km/h at a constant acceleration over a finite time sequence$\langle t_0, t_1, t_2 \rangle$.

Signal Temporal Logic (STL). STL is a logic designed for specifying and reasoning about the continuous behavior of a system over time [15]. STL is an extension of linear temporal logic (LTL) [20] with an ability to specify properties over real values and real time. An STL formula takes the following form:

$$\varphi := u \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathbf{U}_{[a,b]} \varphi_2$$

where $a < b$ for $a, b \in \mathbb{Q}_{\geq 0}$, and u is a predicate of the form $f_u(\mathbf{s}_1(t), \ldots, \mathbf{s}_k(t)) > 0$ for a k-tuple signal $\mathbf{s} = (\mathbf{s}_1, \ldots, \mathbf{s}_k)$ at time t. Informally, the meaning of the until operator at time t is that φ_1 must hold until φ_2 becomes true sometime within the interval $[t + a, t + b]$. The until operator $\mathbf{U}$ alone is sufficient to express two other types of temporal operators that are often useful in system specification—eventually ($\mathbf{F}$) and always ($\mathbf{G}$): $\mathbf{F}_{[a,b]}\varphi = \mathtt{True}\mathbf{U}_{[\mathtt{a},\mathtt{b}]}\varphi$ and $\mathbf{G}_{[a,b]}\varphi = \neg\mathbf{F}_{[a,b]}\neg\varphi$.

Robustness of Satisfaction. In a system whose behavior is captured using continuous variables, it is often useful to be able to talk about how *close* the system is from (dis-)satisfying a property. For instance, if a property says "the distance between the ego and preceding vehicles must be at least 3.0 m", it may be useful to know not only whether the vehicles satisfy this property, but also how far above 3 m they are apart (e.g., 3.1 m vs 5 m).

In prior works, STL has been extended to define this notion of closeness as the *robustness* of satisfaction [10,11]. Formally, the robustness of $\mathbf{s}$ with respect to STL formula φ at time t, denoted $\rho(\varphi, \mathbf{s}, t)$, is defined as follows:

$$\begin{aligned}
\rho(u, \mathbf{s}, t) &= f_u(\mathbf{s}_1(t), \ldots, \mathbf{s}_k(t)) \\
\rho(\neg\varphi, \mathbf{s}, t) &= -\rho(\varphi, \mathbf{s}, t) \\
\rho(\varphi_1 \wedge \varphi_2, \mathbf{s}, t) &= \mathbf{min}(\rho(\varphi_1, \mathbf{s}, t), \rho(\varphi_2, \mathbf{s}, t)) \\
\rho(\varphi_1 \mathbf{U}_{[a,b]}\varphi_2, \mathbf{s}, t) &= \mathbf{sup}_{t' \in [t+a, t+b]}\ \mathbf{min}(\rho(\varphi_2, \mathbf{s}, t'), \mathbf{inf}_{t'' \in [t, t']}\ \rho(\varphi_1, \mathbf{s}, t''))
\end{aligned}$$

where $\mathbf{inf}_{x \in X} f(x)$ returns the greatest lower bound of some function f over domain X (and similarly for $\mathbf{sup}$, the least upper bound). Given that each predicate in STL is of the form $u \equiv f_u(\mathbf{s}_1(t), \ldots, \mathbf{s}_k(t)) > 0$, robustness intuitively captures how far the signal deviates above (or below) 0.

Robustness for $\mathbf{G}$ and $\mathbf{F}$ properties can also be defined as:

$$\begin{aligned}
\rho(\mathbf{G}_{[a,b]}\varphi, \mathbf{s}, t) &= \mathbf{inf}_{t' \in [t+a, t+b]}\ \rho(\varphi, \mathbf{s}, t') \\
\rho(\mathbf{F}_{[a,b]}\varphi, \mathbf{s}, t) &= \mathbf{sup}_{t' \in [t+a, t+b]}\ \rho(\varphi, \mathbf{s}, t')
\end{aligned}$$

Informally, how well $\mathbf{s}$ satisfies $\mathbf{G}\varphi$ is defined to be the point at which the system is the furthest from satisfying φ (similarly, for $\mathbf{F}\varphi$, the point at which φ is satisfied "most well" by the system).

Example. Consider the property **P1** from Sect. 2, which says that the TTC between the ego vehicle and a nearby vehicle must always be above some predefined threshold (TTC_{min}). This property can be formulated as the following STL formula: $\mathbf{G}_{[0,3]}(ttc - TTC_{min} > 0)$ (for simplicity, let us assume that $\mathbf{s}$ is a single-tuple signal that only keeps track of TTCs over time; i.e., $ttc = \mathbf{s}_1(t)$). Suppose that we are given the following pair of signals, representing two different behaviors of the system:

$$\begin{aligned}
\mathbf{s}_X &= \{(0, (4.0)), (1, (3.5)), (2, (4.0)), (3, (4.5))\} \\
\mathbf{s}_Y &= \{(0, (4.0)), (1, (3.5)), (2, (3.0)), (3, (2.5))\}
\end{aligned}$$

Suppose $TTC_{min} = 4.0\,\mathrm{s}$. Then, $\rho(\mathbf{P1}, \mathbf{s}_X, 0) = -0.5$ and $\rho(\mathbf{P1}, \mathbf{s}_Y, 0) = -1.5$. Intuitively, under the scenario depicted by $\mathbf{s}_Y$, the ego vehicle comes closer to colliding with the neighboring vehicle than it does under $\mathbf{s}_X$. Thus, while the property is violated in both scenarios, $\mathbf{s}_X$ is arguably the more desirable outcome of the system.

4 Property-Driven Resolution

Given a set of conflicting feature actions, the goal of our resolution method is to determine which action is most satisfactory with respect to a desired property and allow that action to take place over the other competing actions. At high-level, for each action, our resolver generates a *predictive signal* that estimates how the system is likely to evolve over time given that particular action, and then computes its robustness with respect to the property. The resolver then selects the action with the highest robustness.

Scope. We are specifically interested in studying continuously running systems that must always stay within a safe state. Our goal is to ensure that interactions between features do not lead to a violation of *safety invariants*; i.e., properties that must be maintained by the system throughout its execution. Thus, our resolver takes an input property of form $\mathbf{G}_{[0,\infty]}\varphi$, where φ is any *bounded* STL formula. This restriction on the boundedness of φ is to ensure that prediction terminates after a finite number of steps; i.e., when the resolver performs prediction at each execution step, it only needs to look ahead a finite number of times in order to fully evaluate the robustness of the conflicting actions.

4.1 Prediction

Predictive Model. For each competing action, the resolver generates a signal that predicts how the system evolves given this action, and then evaluates this signal for robustness. Let $\mathcal{V}$ be the set of variables that hold different kinds of system quantities (e.g., speed, distance). In our approach, the model used for prediction is encoded as a transition system $M = (Q, \mathcal{A}, \delta, q_o)$, where:

- $Q \subseteq \mathbb{R}^k$ is the set of states, represented as the configurations of a k-tuple signal (i.e., $q = (v_1, \ldots, v_k)$). In particular, the state consists of n *controlled* variables (v_i for $1 \leq i \leq n \leq k$); the remaining are called *monitored* variables (i.e., $\mathcal{V} = \mathcal{V}_{controlled} \cup \mathcal{V}_{monitored}$).
- $\mathcal{A}$ is the set of actions on controlled variables.
- $\delta : Q \times \mathcal{A} \rightarrow Q$ is the transition function that takes the system from one state to another on an action.
- $q_0 \in Q$ is the initial state of the system.

The notion of controlled and monitored variables is based on the *four variable* model by Parnas et al. [18]. A controlled variable represents the part of the environment that the system can manipulate (e.g., acceleration). A monitored variable, on the other hand, represents an observation about the environment that cannot be directly manipulated but may depend on one or more controlled variables (e.g., velocity of a vehicle, which depends on its acceleration).

Each action in $\mathcal{A}$ involves the assignment of a new value to a controlled variable. In every transition, the system performs one of the available actions to modify a particular controlled variable (for example, increasing the acceleration); the change to the controlled variable, in turn, determines the values of the monitor variables in the new state. We assume that a special action called $nop \in \mathcal{A}$ is defined for all controlled variables to represent the absence of change.

Intuitively, M is a machine that can be used to generate different signals of the system (each corresponding to one possible execution), depending on the choice of the action at every transition step: Given a particular sequence of states $q_0, q_1, \ldots, q_{i-1}$, $\mathbf{s}(t) = q_t$ for $0 \leq t \leq i-1$.

Example. Let us show how system dynamics (δ) can be specified using a set of actions and relationships among variables at consecutive time steps, t and t'. Consider a simplified version of the example from Sect. 2, with two vehicles, A and B (with B being the ego vehicle that we wish to control). The state of a predictive model for this system can be defined as (a_B, v_B, d_{AB}), where a_B is the sole controlled variable representing the acceleration of B, and the rest are monitored variables for the velocity of B and its distance to A, respectively.

The type of action for setting the acceleration of B is defined as follows:

$$setAccel(acc) \equiv a_B(t') := acc$$

where input parameter acc represents the new acceleration of the vehicle, and $:=$ denotes the assignment of a value to a controlled variable.

The dynamics of monitored variables are defined in terms of controlled variables (for simplicity, we assume that vehicle A maintains a constant speed):

$$\begin{aligned}
&v_B(t') = v_B(t) + a_B(t') * T_{step} \\
&d_{AB}(t') = d_{AB}(t) + v_A * T_{step} - (v_B(t) * T_{step} + 0.5 * a_B(t') * (T_{step}^2))
\end{aligned}$$

where t and t' are used to index into the value of a variable in the current and next state, respectively; T_{step} is a constant that represents the time elapsed between each system transition (e.g., 0.1 s). The concept of TTC, which appears in property **P1**, can be derived in terms of v_B and d_{AB}, and needs not be defined as its own monitored variable: $ttc(t') = d_{AB}(t')/(v_B(t') - v_A)$

Assumption. We assume that the model of the system used for prediction is *deterministic*; i.e., executing the model from a particular state with a given action and some number of steps always returns a unique signal. This simplification results in a desirable property that feature conflicts can be resolved in a deterministic manner.

```
1  fun $resolveAll(\varphi, M, \mathbf{s})$
2      $resolved := \{\}$
3      for $v \in \mathcal{V}_{controlled}$ do
4          $\mathcal{A}_c := detectConflicts(v, M)$
5          $resolved[v] := resolve(\mathcal{A}_c, \varphi, M, \mathbf{s})$
6      end
7      return $resolved$
8  end
9  fun $resolve(\mathcal{A}_c, \varphi, M, \mathbf{s})$
10     $rob := \{\}, a_{max} := none$
11     for $a \in \mathcal{A}_c$ do
12         $\mathbf{s}^a := execute(M, a, \mathbf{s}(t), window(\varphi))$
13         $rob[a] := \rho(\varphi, \mathbf{s}^a, t+1)$
14         if $a_{max} = none \vee rob[a] > rob[a_{max}]$ then
15             $a_{max} := a$
16         end
17     end
18     return $a_{max}$
19 end
```

Algorithm 1. Resolution Algorithm.

4.2 Resolution Algorithm

As shown earlier in Fig. 1, our resolver is placed between the set of available features and actuators that act on the system environment as requested by the feature actions. During each system execution cycle, the resolver performs the algorithm in Algorithm 1 to resolve potential conflicts and select system actions that are most likely to maintain a given invariant φ.

The resolver attempts to resolve conflicts associated with each controlled variable one-by-one (lines 3–6). For each of the conflicting actions $a \in \mathcal{A}_c$, the resolver predicts the effect of action a by executing the system model M for a time period that is sufficiently lengthy for evaluating how well a satisfies φ (lines 12–13). After all conflicting actions have been evaluated, the resolver selects the one with the highest robustness value to be performed by the system (line 18).

Conflict Detection. The first task in resolution is to determine the *conflict set* ($\mathcal{A}_c$, line 4)—the set of feature actions that may be in a potential conflict with each other. Since the focus of this work is on resolution, not detection, we omit the details of this step. At high-level, we adopt a *variable-specific* approach proposed by [27], where two features that attempt to modify the same controlled variable are deemed to be in a possible conflict. For instance, CC attempts to speed the vehicle up by increasing its acceleration while SLC attempts to do the opposite, and so the actions from these features are placed in the conflict set. In addition, based on the system dynamics ($M.\delta$), controlled variables that affect a common monitored variable are considered to be *coupled*; any pair of actions that modify two coupled controlled variables are also included in the conflict set.

Prediction Window. The resolver must simulate the effect of actions long enough to determine their robustness with respect to the given property. This duration depends on the structure and length of intervals in the property itself. For instance, consider the following formula:

$$\varphi_{recover} \equiv (ttc \leq TTC_{min} \Rightarrow \mathbf{F}_{[0,3]}(ttc > TTC_{min}))$$

which says that if TTC falls below TTC_{min}, it must be brought back above this safe minimum threshold within the next 3 time steps[1]. In order to determine the robustness of an action with respect to this property, the resolver must generate a predictive signal of at least length 4 by executing M. More generally, the *prediction window* for a bounded STL formula, φ, is defined as follows:

$$\omega(u) = 1 \qquad \omega(\neg\varphi) = \omega(\varphi)$$
$$\omega(\varphi_1 \wedge \varphi_2) = \mathbf{max}(\omega(\varphi_1), \omega(\varphi_2))$$
$$\omega(\varphi_1 \mathbf{U}_{[a,b]} \varphi_2) = \mathbf{max}(\omega(\varphi_1) + b - 1, \omega(\varphi_2) + b)$$

The intuition behind the prediction window for the until operator is as follows: Since φ_2 must become true in at most b future steps, and φ_1 needs to hold only until φ_2 turns true, the prediction task needs to estimate future states for only $b - 1$ steps (plus the number of steps needed to predict φ_1 itself) to determine the robustness of an action for φ_1 part of the $\mathbf{U}$ formula.

Model Execution. The *execution* function (line 12) simulates M to generate a sequence of future states by iteratively applying its transition function δ to the current state and action a. We assume that throughout the prediction window, the applied action remains fixed as a. An alternative approach would involve using models of the features to predict their future actions. Although this could result in more accurate predictions, it would also introduce an additional requirement that every feature comes with its own predictive model—which, based on our interactions with automotive engineers, is rather unrealistic (particularly since many features are developed and updated by third-party suppliers beyond the control of a car manufacturer). Thus, we believe that our design decision is crucial for making the proposed resolution method applicable in practice.

In our experience, we found that our approach is still effective at predicting the system evolution. Most automotive features perform actions that change the system state in a *gradual* manner (e.g., slowly adjust the acceleration); we observed that such actions do not deviate significantly during the prediction windows that we experimented with. In addition, the accumulative effect of inaccuracies is mitigated by repeatedly performing resolution with updated feature actions at each iteration. The frequency of resolution is a parameter that can be adjusted in our framework (further discussed in Sect. 5.3. **Performance**).

Example. Consider actions, $a_1 = setAccel(0.1 \text{ m/s}^2)$ and $a_2 = setAccel(-0.45 \text{ m/s}^2)$, generated by CC and SLC, respectively. Since both manipulate the acceleration of vehicle B, they are considered to be in conflict (i.e.,

[1] To match the syntax of STL, the inequalities can be rewritten to the form $f(\mathbf{s}(t)) > 0$.

$\mathcal{A}_c=\{a_1, a_2\}$). The current system state is given as $(a_B(t), v_B(t), d_{AB}(t)) = (1.0 \text{ m/s}^2, 60 \text{ km/h}, 12 \text{ m})$, with $v_A = 50$ km/h and $TTC_{min} = 5s$. Thus, $ttc = 12/((60-50)*0.2778) \approx 4.32 \leq TTC_{min}$, meaning vehicle B faces the risk of an impending collision with A. Recall the STL formula $\varphi_{recover}$ introduced earlier:

$$\varphi_{recover} \equiv (ttc \leq TTC_{min} \Rightarrow \mathbf{F}_{[0,3]}(ttc > TTC_{min}))$$

To evaluate the robustness of the two actions against this formula, the resolver generates the following predictive signals (for simplicity, we show ttc as the sole component of the signal instead of a_B, v_B, d_{AB}):

$$\mathbf{s}^{a_1} = \{(t_0,(3.19)),(t_1,(2.10)),(t_2,(1.05)),(t_3,(0.028))\}$$
$$\mathbf{s}^{a_2} = \{(t_0,(4.06)),(t_1,(3.91)),(t_2,(3.99)),(t_3,(4.59))\}$$

where t_0 is the first step in the future ($t_0 = t+1, t_1 = t_0+1\ldots$). According to the robustness semantics of STL, $\rho(\varphi, \mathbf{s}^{a_1}, t_0) = 3.19 - 5 = -1.81$ and $\rho(\varphi, \mathbf{s}^{a_2}, t_0) = 4.59 - 5 = -0.41$. Even though both actions do not satisfy the invariant, a_2 is arguably a safer choice, since it pulls the vehicle closer back to the TTC_{min} threshold. Thus, the resolver selects a_2 as the next action to be performed.

5 Case Study

We present a case study applying our resolution method to a set of conflicting safety features in an automotive system. In particular, our goal was to demonstrate that given a set of conflicting feature actions, our method is effective in selecting the action that is most satisfactory with respect to a desired property.

5.1 Implementation

We built a prototype implementation of the feature resolution framework as a part of an in-house simulation environment that we had been developing for vehicle design and testing. The environment consists of two main parts: (1) The driving simulator, built on top of the Unity engine, and (2) the vehicle control system, built as a suite of models in Simulink/MATLAB, each describing the behavior of a controller (e.g., brake controller) or an ADAS feature (e.g., SLC).

The simulator is responsible for animating a model of the traffic environment, while the control system describes the internal behavior of a vehicle. Each simulation run takes place on a traffic map (e.g., a highway road) with a set of vehicles configured with an initial location (2D coordinates), orientation, and velocity. At each simulation step, the simulator sends a message to the control system with the information about the current state of the environment (i.e., surrounding vehicle locations and speeds). Given this information, the control system determines the next control action to perform (e.g., reduce acceleration) and relays this decision back to the simulator, which then accordingly updates the state of the environment by using a built-in physics engine. For our study, we implemented the following features as Simulink models:

- Cruise control (CC): Gradually increases and maintains the vehicle speed to a set value by generating a sequence of acceleration requests.
- Speed limit control (SLC): Gradually reduces the vehicle speed to a context-dependent threshold by sending a sequence of partial braking requests and decreasing its acceleration.
- Automatic emergency braking (AEB): Brings the vehicle to a stop by sending a sequence of full braking requests, drastically reducing its acceleration.
- Partial braking assistance (PB): Gradually slows down the vehicle to maintain a minimum distance to the leading vehicle by generating a sequence of partial braking requests.

5.2 Experimental Setup

We tested our resolution framework on a number of scenarios involving three vehicles (A, B, C) traveling on a single lane, as shown in Fig. 2.

Predictive Model. The model used in our simulation is more complex than the one introduced throughout Sect. 4. The state of the system is represented as the following tuple: $(a_B, v_B, d_{AB}, d_{BC}, a_A, v_A, a_C, v_C)$. In addition to observing the distance between vehicles A and B, we also keep track of information about vehicle C (which trails B). Furthermore, we assume that the speeds of both A and C may also change over time, and this information is made available to B via vehicle-to-vehicle communication.

Properties. We tested our resolution approach on the following two properties:

$$\mathbf{P1} \equiv \mathbf{G}_{[0,\infty]}(ttc > TTC_{min})$$
$$\mathbf{P2} \equiv \mathbf{G}_{[0,\infty]}(ttc \leq TTC_{min} \Rightarrow \mathbf{F}_{[0,3]}(ttc > TTC_{min}))$$

Conceptually, **P2** can be considered a weaker form of **P1** that the system attempts to satisfy if **P1** is violated: When the TTC falls below a minimum threshold, the vehicle must recover back to a safer state within the next three seconds.

The definition of TTC is also more complex than the one introduced in Sect. 4, as we now take into account the distances between B and C as well as A and B. In particular, *ttc* between A, B, C is now defined to be the minimum of the TTCs between pairs of vehicles:

$$ttc = \mathbf{min}(ttc_{AB}, ttc_{BC}) \quad ttc_{AB} = d_{AB}/(v_B - v_A) \quad ttc_{BC} = d_{BC}/(v_C - v_B)$$

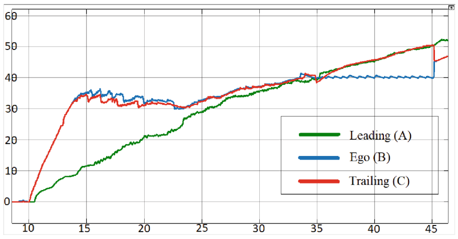
Intuitively, *ttc* represents the time to the *first* potential collision. Thus, by maximizing the TTC, our resolver can be regarded as attempting to delay the first impending collision as much as possible (giving the driver more time to react).

Simulation Scenarios. For each pairwise feature combination (e.g., CC vs SLC), we simulated the four distinct scenarios and observed the changes in vehicle speeds as well as the robustness of the features. Each scenario was executed twice, with and without our property-driven resolver activated. In the run without the resolver, the action that would result in a lower acceleration was selected over the other conflicting action—the rationale being that the slower the vehicle speed, the safer it is likely to be (the resolution strategy used by [27]).

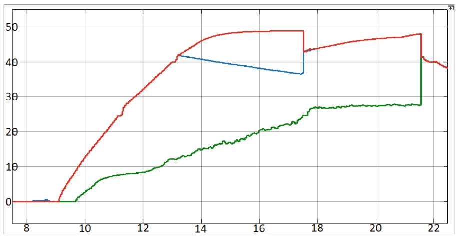
5.3 Simulation Results

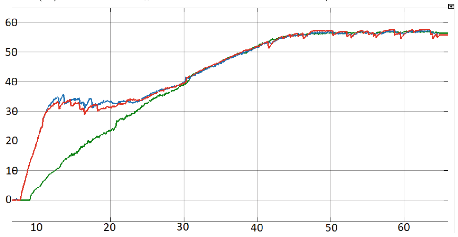
Figure 3 shows the results from two scenarios involving the following feature combinations: SLC vs CC and PB vs CC, with **P1** as the property. In addition to the plotted scenarios, we tested every other pairwise combinations of features; due to limited space, we discuss only these two in detail.

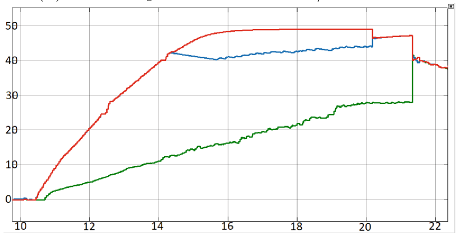
SLC vs CC. Figure 3(a) and (b) shows the speed changes in the vehicles when SLC and CC are enabled, without and with the resolver active, respectively. In

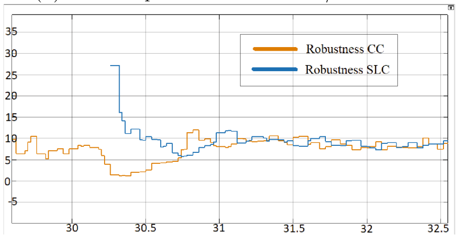


(a) Vehicle speeds: SLC vs CC w/o resolver.

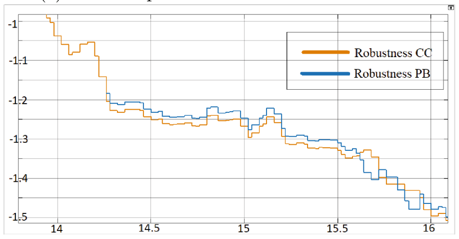

(d) Vehicle speeds: PB vs CC w/o resolver.


(b) Vehicle speeds: SLC vs CC w/o resolver.


(e) Vehicle speeds: PB vs CC with resolver.



(c) Robustness: SLC vs CC with resolver.



(f) Robustness: PB vs CC with resolver.

Fig. 3. Simulation results. The x-axis in every plot represents simulation time elapsed (seconds). In plots (a), (b), (d), (e), the unit of the y-axis is in km/h; in (c) and (f), the y-axis represents the robustness value (which is, in general, unitless). In both (c) and (f), CC is enabled from the initial state, while the other feature does not become activated until certain environmental conditions are met (e.g., vehicle exceeds a SLC limit)—at which its robustness value (in blue) begins to register. (Color figure online)

Fig. 3(a), without our resolver, the system always selects the feature that results in a lower acceleration (SLC). When the trailing vehicle C (in red) speeds up, the ego vehicle B (blue) is unable to maintain ttc_{BC} above the safe threshold and eventually ends up in a collision, around 45 s. Note that when a pair of vehicles collide, their velocities simultaneously become equal (as shown by the sudden drop and increase in the plot).

In Fig. 3(b), when the resolver is active, it selects the feature that is likely to result in a higher TTC. As vehicle C approaches B from behind, ttc_{BC} begins to decrease, and the resolver selects CC to allow B to speed away from C to a state with a higher TTC. Subsequently, as vehicle B accelerates towards the leading vehicle A (in green), ttc_{AB} begins to approach the threshold, and the resolver selects SLC as the more desirable action. The resolver keeps alternating between the two features in order to maintain both ttc_{AB} and ttc_{BC} above the threshold until all three vehicles stabilize to a similar speed.

The robustness values for the two features are shown in Fig. 3(c). It starts out as CC being the only enabled feature until vehicle B reaches a certain SLC-specific speed limit, at which the SLC feature is enabled (and its robustness, in blue, begins to appear on the plot). The oscillation between the robustness of SLC and CC shows the resolver attempting to maximize *ttc* by alternating between the two features, until a stable speed is established by the vehicles.

PB vs CC. Figure 3(d) and (e) shows the speed changes in the vehicles when PB and CC are enabled, without and with the resolver active, respectively. In Fig. 3(d), as vehicle B catches up to vehicle A within a set distance, PB is activated and begins generating requests for decelerating vehicle B, introducing a conflict with CC. Given the two competing features, the system without our resolver selects the one that would result in a lower acceleration—in this case, PB. Around 17.5 s, the trailing vehicle C approaches and ends up colliding with B. As the leading vehicle A is traveling at a sufficiently low speed, all of the vehicles eventually end up in a three-way collision.

In Fig. 3(e), as vehicle C approaches B, the resolver continuously alternates between CC and PB in order to maximize the current *ttc*. However, as vehicle A is traveling at a significantly lower speed than C is, the resolver is still unable to keep the minimum TTC between the vehicles. Subsequently, vehicle B collides with C (shortly after 20 s) and eventually with vehicle A (around 21 s).

This scenario shows an example where none of the actions is sufficient to prevent the system from violating the property. In Fig. 3(f), the robustness values for both PBS and CC drop below zero and continue to fall, despite the resolver's attempt to maximize TTC, eventually resulting in a three-way collision. However, the behavior resulting from our resolver (Fig. 3(e)) is still arguably more desirable than the outcome without the resolver (3(d)). In always selecting the action that maximizes TTC, the resolver effectively delays the time of the first collision as much as possible (17.5 s vs 21 s), giving the driver more time to react.

Effect on Properties. When we initially designed our experiments, we expected to see different simulation outcomes depending on the input property (**P1** vs **P2**). Surprisingly, however, we found that the results for both sets of simulation runs were very similar, regardless of the enabled features. Note that for the invariant in **P1**, the robustness of an action is defined as $\rho(ttc > TTC_{min}, \mathbf{s}, t) = ttc(t) - TTC_{min}$. Now, consider the robustness for the invariant in **P2**:

$$\begin{aligned}
&\rho(ttc \leq TTC_{min} \Rightarrow \mathbf{F}_{[0,3]}(ttc > TTC_{min}), \mathbf{s}, t) \\
&= \rho(ttc > TTC_{min} \vee \mathbf{F}_{[0,3]}(ttc > TTC_{min}), \mathbf{s}, t) \\
&= \mathbf{max}(\rho(ttc > TTC_{min}, \mathbf{s}, t), \rho(\mathbf{F}_{[0,3]}(ttc > TTC_{min}), \mathbf{s}, t)) \\
&= \mathbf{max}(\rho(ttc > TTC_{min}, \mathbf{s}, t), \mathbf{sup}_{t' \in [t,t+3]} \rho(ttc > TTC_{min}, \mathbf{s}, t')) \\
&= \mathbf{max}(ttc(t) - TTC_{min}, \mathbf{sup}_{t' \in [t,t+3]}(ttc(t') - TTC_{min}))
\end{aligned}$$

In other words, for **P2**, the resolver selects the action that tries to maximize TTC as much as possible during the period of the prediction window. Due to the robustness semantics, the resolver attempts to preemptively prevent TTC from falling below TTC_{min}, effectively establishing the same overall system behavior as it does under **P1** as the property. In comparison, under the conventional Boolean semantics of satisfaction, the resolver would treat competing actions equally until TTC falls below TTC_{min}. In effect, the robustness semantics of STL enables a more robust resolution of conflicts.

Performance. To assess the overhead incurred by resolution, we computed the ratio of the average simulation time with the resolver over that without the resolver. The overhead depends on the frequency of resolution, the number of features being evaluated, and the input property (which affects the prediction window). We selected the frequency of resolution to be 0.1 s, based on our estimates of how frequently messages are generated by typical electronic control units (ECUs). On average, with all of the four features enabled, the overhead was around 15.1% for **P1** and 17.8% for **P2**. The additional overhead from **P2** was due to the latter property having a larger prediction window, as expected.

It is difficult to accurately estimate how well our proposed resolver would perform in an actual vehicle. For our simulations, we are executing *models* of the features, controllers, and the environment in Simulink, which does not reflect realistic operating conditions. In a typical vehicle, these models would be realized as low-level embedded code running on ECUs or a dedicated hardware device (e.g., FPGA). In addition, in safety-critical systems like vehicles, *lookup tables* [1, 23, 29] are widely used to pre-compute and reuse the results of time-consuming operations (e.g., simulation of physics dynamics), which could be used to reduce the overhead introduced by the prediction step.

6 Related Work

Our work is most closely related to the *variable-specific* resolution approach introduced in [2, 27], which associates each system variable (e.g., speed) with a specific strategy for resolving conflicts between multiple actions (e.g., select the action that results in the smallest acceleration). Like ours, their approach decouples resolution decisions from the presence of particular features and is capable of handling feature addition or modification without having to modify the resolution strategies. However, since their approach still relies on fixed strategies, it may fail to produce a desirable outcome when the system runs into scenarios that are unanticipated by those strategies (as discussed in Sect. 2).

Griffeth and Velthuijsen proposed a runtime resolution method based on the notion of *negotiation*, where a central *mediator* is used to resolve conflicting actions among system agents [12]. Rather than attempting to satisfy a global system property, the goal of their resolution is different from ours, in that it attempts to come up with actions that all agents consider to be *acceptable*. Other resolution methods [4,5,13,17,24,28] rely on a priority or precedence ordering, and may not be suitable for systems where the set of features evolve over time.

STL has been leveraged for online monitoring of system properties [6,7,9]. The key difference is that monitoring attempts to detect a violation of a property *after* it has already occurred, whereas our resolution attempts to select an action that is least likely to lead to a violation *before* it occurs.

Runtime techniques for enforcing a desirable property by observing and possibly modifying system actions have been studied [19,25]. However, these approaches typically evaluate a single trace for property satisfaction, and do not involve a comparison of conflicting actions for their satisfaction or robustness.

Our approach of using a predictive model to dynamically determine the safest of the conflicting actions is similar to *online planning* [21,22], which tackles the problem of periodically computing a desirable *policy* (i.e. which actions the system should take at a given state) during the execution.

7 Conclusions and Future Directions

This paper proposes an approach that leverages a desired property of the system to resolve conflicts between competing features at runtime. Based on our experience using this framework in-house, we believe that the property-oriented method is a promising approach, especially in emerging domains such as connected vehicles where the set of installed features may change frequently.

As discussed in Sect. 4.1, our predictive model assumes that the environment evolves in a deterministic manner given a system action. Probabilistic models (e.g., Markov decision processes) may be more suitable for accurately capturing the behavior of environmental agents (e.g., how other vehicles adjust their speeds). To this end, we plan to extend our resolution framework by adopting a stochastic notion of STL satisfaction [14]. We are also exploring the possibility of incorporating enforcement techniques into our framework to *synthesize* a new action to maintain a safety invariant if none of the given feature actions is satisfactory.

References

1. Arechiga, N., Dathathri, S., Vernekar, S., Kathare, N., Gao, S., Shiraishi, S.: Osiris: a tool for abstraction and verification of control software with lookup tables. In: Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, SCAV@CPSWeek 2017, Pittsburgh, PA, USA, 21 April 2017, pp. 11–18 (2017)
2. Bocovich, C., Atlee, J.M.: Variable-specific resolutions for feature interactions. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, 16–22 November 2014, pp. 553–563 (2014)
3. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. Comput. Netw. **41**(1), 115–141 (2003)
4. Chavan, A., Yang, L., Ramachandran, K., Leung, W.H.: Resolving feature interaction with precedence lists in the feature language extensions. In: Feature Interactions in Software and Communication Systems IX, International Conference on Feature Interactions in Software and Communication Systems, ICFI 2007, Grenoble, France, 3–5 September 2007, pp. 114–128 (2007)
5. Chen, Y., Lafortune, S., Lin, F.: Resolving feature interactions using modular supervisory control with priorities. In: Feature Interactions in Telecommunications Networks IV, Montréal, Canada, 17–19 June 1997, pp. 108–122 (1997)
6. Deshmukh, J.V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., Seshia, S.A.: Robust online monitoring of signal temporal logic. Formal Methods Syst. Des. **51**(1), 5–30 (2017)
7. Dokhanchi, A., Hoxha, B., Fainekos, G.: On-line monitoring for temporal logic robustness. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 231–246. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_19
8. Dominguez, A.L.J., Day, N.A., Joyce, J.J.: Modelling feature interactions in the automotive domain. In: International Workshop on Modeling in Software Engineering (MiSE), pp. 45–50 (2008)
9. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 264–279. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_19
10. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_9
11. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES/RV -2006. LNCS, vol. 4262, pp. 178–192. Springer, Heidelberg (2006). https://doi.org/10.1007/11940197_12
12. Griffeth, N.D., Velthuijsen, H.: The negotiating agents approach to runtime feature interaction resolution. In: Feature Interactions in Telecommunications Systems, Amsterdam, The Netherlands, 8–10 May 1994, pp. 217–235 (1994)
13. Hay, J.D., Atlee, J.M.: Composing features and resolving interactions. In: ACM SIGSOFT Symposium on Foundations of Software Engineering, Proceedings, San Diego, California, USA, 6–10 November 2000, pp. 110–119 (2000)

14. Li, J., Nuzzo, P., Sangiovanni-Vincentelli, A.L., Xi, Y., Li, D.: Stochastic contracts for cyber-physical system design under probabilistic requirements. In: Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, 29 September–02 October 2017, pp. 5–14 (2017)
15. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. Formal Techniques. Modelling and Analysis of Timed and Fault-Tolerant Systems, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
16. Metzger, A.: Feature interactions in embedded control systems. Comput. Netw. **45**(5), 625–644 (2004)
17. Nakamura, M., Igaki, H., Yoshimura, Y., Ikegami, K.: Considering online feature interaction detection and resolution for integrated services in home network system. In: ICFI, pp. 191–206. IOS Press (2009)
18. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. **25**(1), 41–61 (1995)
19. Pinisetty, S., Roop, P.S., Smyth, S., Tripakis, S., von Hanxleden, R.: Runtime enforcement of reactive systems using synchronous enforcers. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, 10–14 July 2017, pp. 80–89 (2017)
20. Pnueli, A.: The temporal logic of programs. In: Symposium on Foundations of Computer Science, SFCS 1977, pp. 46–57 (1977)
21. Ross, S., Pineau, J., Paquet, S., Chaib-draa, B.: Online planning algorithms for POMDPs. J. Artif. Intell. Res. **32**, 663–704 (2008)
22. Seuken, S., Zilberstein, S.: Formal models and algorithms for decentralized decision making under uncertainty. Auton. Agent. Multi-Agent Syst. **17**(2), 190–250 (2008)
23. Sundström, C., Frisk, E., Nielsen, L.: Diagnostic method combining the lookup tables and fault models applied on a hybrid electric vehicle. IEEE Trans. Control Syst. Technol. **24**(3), 1109–1117 (2016)
24. Tsang, S., Magill, E.H.: The network operator's perspective: detecting and resolving feature interaction problems. Comput. Netw. **30**(15), 1421–1441 (1998)
25. Wu, M., Zeng, H., Wang, C., Yu, H.: Safety guard: runtime enforcement for safety-critical cyber-physical systems: invited. In: Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18–22 2017, pp. 84:1–84:6 (2017)
26. Yarosh, L., Zave, P.: Locked or not?: Mental models of IoT feature interaction. In: Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, 06–11 May 2017, pp. 2993–2997 (2017)
27. Zibaeenejad, M.H., Zhang, C., Atlee, J.M.: Continuous variable-specific resolutions of feature interactions. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, 4–8 September 2017, pp. 408–418 (2017)
28. Zimmer, P.A., Atlee, J.M.: Ordering features by category. J. Syst. Softw. **85**(8), 1782–1800 (2012). https://doi.org/10.1016/j.jss.2012.03.025
29. Zurbriggen, F., Ott, T., Onder, C.H.: Fast and robust adaptation of lookup tables in internal combustion engines: feedback and feedforward controllers designed independently. Proc. Inst. Mech. Eng. Part D: J. Automob. Eng. **230**(6), 723–735 (2016)