Synthesis-Based Resolution of Feature Interactions in Cyber-Physical Systems

Benjamin Gafford
Institute for Software Research
Carnegie Mellon University
gafford@cmu.edu

Tobias Dürschmid
Institute for Software Research
Carnegie Mellon University
duerschmid@cmu.edu

Gabriel A. Moreno
Software Engineering Institute
Carnegie Mellon University
gmoreno@sei.cmu.edu

Eunsuk Kang
Institute for Software Research
Carnegie Mellon University
eskang@cmu.edu

ABSTRACT

The *feature interaction problem* arises when two or more independent features interact with each other in an undesirable manner. Feature interactions remain a challenging and important problem in emerging domains of cyber-physical systems (CPS), such as intelligent vehicles, unmanned aerial vehicles (UAVs) and the Internet of Things (IoT), where the outcome of an unexpected interaction may result in a safety failure. Existing approaches to resolving feature interactions rely on priority lists or fixed strategies, but may not be effective in scenarios where none of the competing feature actions are satisfactory with respect to system requirements. This paper proposes a novel *synthesis-based* approach to resolution, where a conflict among features is resolved by *synthesizing* an action that best satisfies the specification of desirable system behaviors in the given environmental context. Unlike existing resolution methods, our approach is capable of producing a desirable system outcome even when none of the conflicting actions are satisfactory. The effectiveness of the proposed approach is demonstrated using a case study involving interactions among safety-critical features in an autonomous drone.

ACM Reference Format:
Benjamin Gafford, Tobias Dürschmid, Gabriel A. Moreno, and Eunsuk Kang. 2020. Synthesis-Based Resolution of Feature Interactions in Cyber-Physical Systems. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416630

1 INTRODUCTION

The *feature interaction* problem occurs when two or more independent features interact with each other in unanticipated ways, possibly leading to an undesirable system outcome. Although feature interactions have been long studied by the software engineering community [11, 35, 48], they remain a challenging and important problem in emerging domains of cyber-physical systems (CPS), such as intelligent vehicles, unmanned aerial vehicles (UAVs) and the Internet of Things (IoT), where the outcome of an unexpected interaction may result in a safety failure [20, 32, 47]. For example, unexpected interactions between a pair of braking features has been found to be one of the contributing factors to the incidents involving unintended acceleration on Toyota vehicles [45].

Managing feature interactions typically involve two tasks: *detection* and *resolution*. In this paper, we specifically focus on the problem of resolution at the *run time*. Once a conflict among features is detected during the execution of a system, the goal of a run-time resolution method is to select the most desirable out of the actions proposed by the competing features.

A major challenge in designing effective resolution mechanisms is defining what it means for an action to be more *desirable* than others. Existing approaches to resolution rely on some notion of *priorities* [12, 13, 27, 50] (e.g., emergency braking should always be selected over other features) or *variable-specific strategies* [9, 49] (e.g., the feature that results in the lowest acceleration output is considered safest and thus should be selected). Both types of approaches, however, face limitations when managing feature interactions in modern CPS with a highly dynamic, unpredictable environment.

First, the desirability of a feature action may be *context-dependent*; i.e., whether or not one action is more desirable than the others depends on the current conditions of the environment. For example, although decreasing the speed of a vehicle is often associated with reducing the risk of a collision, in certain traffic scenarios, it may actually be safer to accelerate (e.g., when another vehicle is trailing within an unsafe distance). Since it is difficult to predict these types of scenarios at design-time, a resolution method that uses a priority list or fixed strategy may sometimes fail to produce a desirable system behavior.

Second, in certain situations, none of the competing feature actions may be desirable with respect to the system requirements. This is a fundamental challenge with resolution: When a pair of features have conflicting objectives, being forced to select one over the other means temporarily sacrificing the objective of the latter feature. However, when both of these features perform critical tasks (e.g., safety functions), this "winner-takes-all" approach may cause a negative impact on the system (e.g., a safety violation) regardless of which feature is selected.

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416630

To address these challenges, this paper proposes a *synthesis-based* approach for resolving unexpected feature interactions that arise in CPS. In this approach, the *desirability* of a feature action is defined in terms of how well the system would satisfy its desired requirements if that particular action were to be executed in the given environment. In particular, we leverage a formal specification notation called *signal temporal logic* (STL) [31]—an extension of linear temporal logic that is well-suited for describing behaviors of CPS—along with the notion of *robustness* of property satisfaction [24] to precisely define the desirability of an action as a quantitative metric. This enables a formulation of resolution as an *optimization* problem: Given a specification of desired system behaviors (specified as STL properties) and competing actions in a given context, the goal is to select the action that achieves the highest level of satisfaction. Furthermore, our approach explores actions beyond the given ones and resolves a conflict by *synthesizing* a new action that is more desirable than any of the given actions.

We have implemented a prototype of our resolution method as part of the flight control software for autonomous drones (in particular, using PX4 [23], a popular open source flight control software for drones). To demonstrate the effectiveness of our method, we performed simulations of an autonomous drone with four active features that perform distinct but sometimes conflicting safety tasks. Our evaluation shows that compared to existing methods, our resolution approach is significantly better at satisfying multi-dimensional objectives, avoiding the violation of safety requirements, and producing smoother and more stable action sequences.

The contributions of this paper are as follows:

- A *synthesis-based* approach to feature interaction resolution, where the problem of selecting the most desirable action is formulated as *synthesizing* the action that best satisfies the objectives of the competing features,
- A system architecture and an algorithm for synthesizing actions, and a prototype implementation on a control system for real-world autonomous drones,
- A case study that demonstrates the effectiveness of the proposed approach on an autonomous drone with four safety features.

The rest of the paper is structured as follows. We first introduce a running example involving an autonomous drone that illustrates the challenges with existing methods for resolution and how our approach addresses them (Section 2). Then, we describe the concepts of STL and robustness of property satisfaction (Section 3), which serves as the theoretical basis for our resolution method (Section 4). We then describe an evaluation of our approach using a case study involving simulations of an autonomous drone (Section 5). Finally, we discuss related works (Section 6) and conclude with a discussion of the scope and limitations of our approach and future directions (Section 7).

2 EXAMPLE

Consider an autonomous drone (shown in Figure 1) that is controlled by on-board autopilot software, which periodically generates a command to update the velocity and direction of its movement. The ego drone (i.e., the drone being controlled) has two safety features installed: (1) the *boundary enforcer*, which monitors and overrides the autopilot command (when necessary) to prevent the

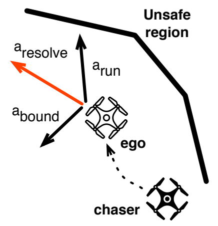


Figure 1: A scenario where two safety features on the ego drone are activated at the same time and generate conflicting commands (a_{bound} and a_{run}).

drone from flying into an unsafe region, and (2) the *runaway enforcer*, which monitors the movement of a trailing drone (e.g., *chaser* in Figure 1) and attempts to maneuver the ego drone away from the former. As the ego drone moves closer to the boundary while being chased by another drone, both features become triggered and generate conflicting commands: The boundary enforcer generates an action (a_{bound} in Figure 1) to move the ego away from the unsafe region, while the action from the runaway enforcer (a_{run}) pushes the ego towards the boundary in attempt to distance it from the chaser. The goal of resolution is to determine the final action to be performed by the system given such a conflict among the features.

Existing methods. One possible approach to resolving this conflict is to use some notion of *priorities* among features. The drone operator, for example, may designate the runaway enforcer as having a higher priority, and the software controller may be configured to disregard the actions from other features (including the boundary enforcer). This decision results in system behaviors that attempt to satisfy the requirement of the highest-ranked feature while disregarding those of the lower-ranked ones.

This type of "winner-takes-all" approach, however, may not be suitable in situations where such a strict ordering among features does not exist, or all of the conflicting features play a critical role in maintaining the system safety and performance. For instance, when the runaway enforcer is always favored, the ego drone may end up flying outside the safe boundary in order to evade the chaser. A more desirable outcome would be one where the ego drone behaves in a manner that attempts to satisfy the requirements of both features by evading the chaser while staying within the boundary.

Proposed method. Our resolution approach, in comparison, takes into account the requirements of all conflicting features and produces an action that is satisfactory with respect to all of them. The key idea is to (1) make explicit the desirable system behaviors that the features are designed to produce and (2) allow the system designer to specify the overall specification of the system as a combination of the requirements of individual features.

For instance, suppose that the two safety enforcers on the drone are designed to achieve the following requirements:

- Boundary enforcer: "Maintain minimum time-to-intercept T_{min} seconds away from the boundary."
- Runaway enforcer: "Maintain minimum safe distance D_{min} meters between the ego drone and another, trailing drone."

Given feature requirements in a formal specification notation (in our case, STL), our resolution method attempts to synthesize an action that best achieves all of these requirements. In the scenario depicted in Figure 1, neither of the two given actions a_{run} and a_{bound}

is satisfactory; moving away from the boundary will increase the chance of being intercepted by the chaser, and vice-versa. However, instead of taking either one of these actions, our controller synthesizes a new action, $\mathrm{a}_{\mathrm{resolve}}$, that moves the drone in a way that is consistent with both requirements.

Since our approach does not use a priority list or fixed strategies, it can handle conflict scenarios where the desirability of one feature changes dynamically depending on its satisfaction of the system specification in the current environmental context. In addition, by considering actions beyond those generated by the features, our approach is capable of resolving conflicts and maintaining a desirable system outcome even when none of the competing actions are satisfactory with respect to the specification.

3 PRELIMINARIES

Signals. In our approach, the behavior of a system is modeled using the concept of a *signal*. A signal over domain D is a function $\mathbf{s} : T \longrightarrow D$, where *time domain* T is a possibly infinite set of real numbers that represent points in time ($T \subseteq \mathbb{R}_{\geq 0}$). The value of a signal is a tuple of k real numbers that correspond to different state variables; i.e., $D \subseteq \mathbb{R}^k$ and $\mathbf{s}(t) = (v_1, ..., v_k)$. As a shorthand, the subscript notation $\mathbf{s}_i(t)$ denotes the i-th component of the signal at time t (for $1 \leq i \leq k$).

Signal Temporal Logic. STL [31] is an extension of linear temporal logic (LTL) [38] with an ability to specify system states as real-valued *signals* evolving over *time*. Since CPS requirements typically involve timing properties (e.g., "the drone must re-enter the safe region within the next 3 seconds") as well as continuous state variables (e.g., drone speed), STL is well-suited for specifying CPS behaviors. 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 $f_u(\mathbf{s}_1(t), ..., \mathbf{s}_k(t)) \geq 0$ for a k-tuple signal $\mathbf{s} = (\mathbf{s}_1, ..., \mathbf{s}_k)$ at time t. Informally, $\varphi_1 \mathbf{U}_{[a,b]} \varphi_2$ means that φ_1 must hold until φ_2 becomes true within the interval $[t+a, t+b]$. The until operator $\mathbf{U}$ can be used to express two other common temporal operators—eventually ($\mathbf{F}$) and always ($\mathbf{G}$): $\mathbf{F}_{[a,b]}\varphi = \mathtt{True}\mathbf{U}_{[\mathrm{a,b}]}\varphi$ and $\mathbf{G}_{[a,b]}\varphi = \neg\mathbf{F}_{[a,b]}\neg\varphi$.

Robustness. In a typical verification process, a specification language such as LTL is used to define a desired property of a system, and a tool is used to check whether the system satisfies the property. In addition to this *binary* notion of satisfaction, it is useful to be able to reason about how "close" the system is from satisfying or violating a property. To support such a *quantitative* notion of satisfaction, STL has been extended with the concept of *robustness* [22, 24].

Informally, the robustness of signal $\mathbf{s}$ with respect to STL formula φ at time t, denoted $\rho(\varphi, \mathbf{s}, t)$, is a value that represents the difference between the actual signal value and the threshold at which the system violates φ. For example, given property φ that says "the distance between the drone and the boundary to an unsafe region must be at least 3.0 meters", $\rho(\varphi, \mathbf{s}, t)$ represents how far beyond 3.0 meters the drone is able to maintain its distance.

Robustness can be used to compare different executions of a system in terms of their desirability. Suppose that we are given two signals $\mathbf{s}_1$ and $\mathbf{s}_2$ (representing two distinct system executions) such that $\rho(\varphi, \mathbf{s}_1, t) = 1.0m$ and $\rho(\varphi, \mathbf{s}_2, t) = 2.0m$. Arguably, the execution represented by $\mathbf{s}_2$ is a safer, more desirable scenario, since the system maintains a greater distance to the boundary.

Formally, robustness is defined over STL formulas as follows:

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

where $\mathbf{inf}_{x \in X} f(x)$ is the greatest lower bound of some function f over domain X (and $\mathbf{sup}$ the least upper bound). Since each predicate in STL is of the form $u \equiv f_u(\mathbf{s}_1(t), ..., \mathbf{s}_k(t)) > 0$, robustness intuitively captures how far the signal deviates above (or below) 0. Robustness for $\mathbf{G}_{[a,b]}\varphi$ represents the value within interval $[a, b]$ at which the system is the furthest away from satisfying φ (or closest to violating it). Similarly, robustness for $\mathbf{F}_{[a,b]}\varphi$ represents the value at the time between $[a, b]$ at which the system achieves the highest satisfaction of φ.

Example. Suppose that the objective of the boundary enforcer is to fulfill the following safety requirement: "The time before the ego drone crosses the boundary of a safe region must be at least some minimum threshold T_{min} seconds for the next 3 time units". This specification can be written as STL property $\varphi \equiv \mathbf{G}_{[0,3]}(tti - T_{min} \geq 0)$, where tti is a state variable that stores the current time-to-intercept (TTI); i.e., the time before crossing the boundary.

One possible execution of the drone is represented by the following signal: $\mathbf{s}^A = \{(0, (4.0)), (1, (3.5)), (2, (4.0)), (3, (4.5))\}$ (note that $\mathbf{s}^A$ here is a single-tuple signal with only one state variable $tti = \mathbf{s}^A_1(t)$; in practice, a signal would have a number of other variables, such as the drone coordinates, velocity, and acceleration). In particular, this signal depicts the behavior of the drone where its TTI approaches 3.5s (at time step 1) before pulling away to 4.5s (at time step 4). Suppose that $T_{min} = 2.5s$. Then, the robustness of signal $\mathbf{s}^A$ with respect to the above property at the beginning of the execution, denoted by $\rho(\varphi, \mathbf{s}^A, 0)$, can be computed as 1.0 (= 3.5 - T_{min}). Intuitively, this robustness value quantifies how close the system comes to violating the property.

Consider an alternative execution, represented by signal $\mathbf{s}^B = \{(0, (5.2)), (1, (2.9)), (2, (3.7)), (3, (4.7))\}$, where $\rho(\varphi, \mathbf{s}^B, 0) = 0.4$. Although the property holds over both executions (robustness values are positive for both), $\mathbf{s}^A$ represents arguably the safer and more desirable behavior, since the drone does not come as close to crossing the boundary as it does in $\mathbf{s}^B$. This example demonstrates how the concept of robustness in STL can be used to distinguish system behaviors that would be considered equivalent under the binary notion of satisfaction (as in LTL, for example). As described in the following section, this quantitative notion of satisfaction plays a key role in our resolution mechanism.

4 RESOLUTION APPROACH

A high-level overview of the proposed framework for feature interaction resolution is shown in Figure 2. In our modeling approach, a system consists of a *controller* that interacts with the *environment*. The controller contains one or more *features*, each of which receives information about the environment from various sensors

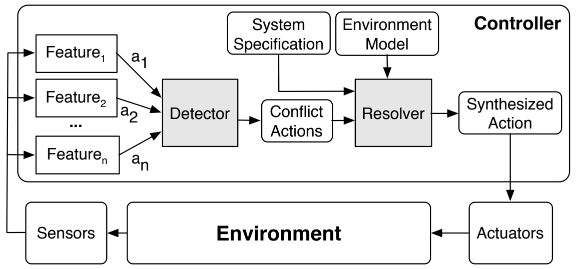


Figure 2: Overview of the proposed resolution framework.

and generates an *action* to influence the latter through an actuator. For instance, a feature in an autonomous drone may be a safety enforcer (e.g., the boundary enforcer from Section 2) that is activated under certain environment conditions and generates an action needed to fulfill a requirement (e.g., keep the drone within the safe region). The *detector* component in the controller monitors the actions generated by the features and determines the subset of the actions that are potentially in conflict with each other. These *conflict actions* are then passed onto the *resolver*.

The resolver is configured with two types of information: the *system specification* that describes the requirements of the features (expressed in STL), and the *environment model*, which is used by the resolver to predict how the environment evolves over time given a particular system action. During resolution, the resolver generates a space of candidate actions and evaluates each of them in terms of its robustness (i.e., how well it would satisfy the specification if it were to be executed on the environment). The action with the highest robustness value is selected and fed to the actuator, which then performs the corresponding control action on the environment.

Detection. Detecting feature conflicts is an important part of feature interactions management, but is not the focus of this paper and thus omitted from discussion. Briefly, we use the *variable-specific* method proposed in the prior works ([9, 49]), where a pair of features that involve inconsistent modifications to the same actuator variable (e.g., acceleration) is considered to be in a conflict.

4.1 Environment Model

A key element of our approach is an executable model of the environment that is used to evaluate each candidate action in terms of how well it would satisfy the given specification if it were to be selected as the final action. In particular, the environment is modeled as a transition system $M = (Q, \mathcal{A}, \delta, Q_o)$, where:

- $Q \subseteq \mathbb{R}^k$ is the set of states, each representing one possible configuration of a k-tuple signal (i.e., $q = (v_1, ..., v_k)$ for some $q \in Q$ and $v_1, ..., v_k \in \mathbb{R}$),
- $\mathcal{A}$ is the set of controller actions,
- $\delta : Q \times \mathcal{A} \rightarrow Q$ is the transition function that describes how the system moves from one state to another by performing an action, and
- $Q_0 \subseteq Q$ is the set of initial states.

In our running example, the environment model describes how the state of the ego drone (i.e., its location, direction, and velocity) changes in response to a flight control action. More precisely, each state q can be represented as tuple $(loc_x, loc_y, loc_z, vel_x, vel_y, vel_z)$ where $loc_{x,y,z} \in \mathbb{R}$ encodes the x, y, z coordinates of the ego drone, and $vel_{x,y,z} \in \mathbb{R}$ encodes the current velocity of the drone.

One type of action performed by the controller is setVelocity(v_x, v_y, v_z), which sets the new target velocity of the drone to v_x, v_y, and v_z (for x, y, z components, respectively). Then, the transition function δ is defined such that for any state $q' = \delta(q, a)$ and $a =$ setVelocity(v_x, v_y, v_z):

$$q' = (q.loc_x + q.vel_x, q.loc_y + q.vel_y, q.loc_z + q.vel_z,$$
$$q.vel_x + a_x, q.vel_y + a_y, q.vel_z + a_z)$$

where a_x is the amount of acceleration that is applied to bring $q.vel_x$ to v_x (and similarly for a_y and a_z).

Conceptually, an environment model can be regarded as a machine that generates different signals based on the actions performed by the controller. In particular, an execution trace $q_0, q_1, ..., q_l$ can be converted into signal $\mathbf{s}$, where for each $t \leq l$, $\mathbf{s}(t) = q_t$.

4.2 System Specification

To recall, one of the major challenges in resolution is that some of the features are designed to fulfill requirements that inevitably conflict with each other in certain situations (e.g., the scenario in Figure 1). One approach is to select one of these conflicting features as the "winner" and disable the rest; however, this effectively amounts to disregarding the requirements of the "losing" features. Instead, our goal is to consider all of the (possibly conflicting) requirements and synthesize an action that satisfies them as much as possible. To support this type of resolution, we allow the overall system specification to be expressed as a combination of the requirements of individual features.

More precisely, a feature $f \in F$ is modeled as a function that takes a set of observable variables and produce an action. Each feature is associated with a requirement R_f that describes the objective that the feature is designed to achieve. For instance, the requirement of the boundary enforcer may be stated as "Maintain a safe minimum distance D_{min} to the designated boundary". In our approach, each feature requirement is specified as an STL formula (e.g., $R_{\text{boundary}} \equiv \mathbf{G}_{[0,3]}(tti - T_{min} \geq 0)$).

Then, given n features $f_1, f_2, ..., f_n$, the overall *system specification* is defined as a set of *weighted* feature requirements, as follows:

$$R_{sys} = \{(w_1, R_1), (w_2, R_2), ..., (w_n, R_n)\}$$

where $w_1, w_2, ..., w_n \in \mathbb{R}_{\geq 0}$ are *weighting factors* that together add up to 1.0. The weights allow the system engineer to specify the relative importance of the feature requirements. For example, assigning equal weights will intuitively configure the system to resolve a conflict by selecting an action that attempts to satisfy all of the feature requirements. Similarly, if desired, the engineer could assign a weight of 1.0 to one of the requirements, effectively turning our resolver into a priority-based one. The weights could also be modified at runtime, allowing the engineer to dynamically adjust the importance of requirements as needed over time.

4.3 Synthesis-Based Resolution

4.3.1 Resolution as Optimization. The goal of resolution is to resolve feature conflicts in a way to ensure that the resulting system behavior is desirable with respect to the feature requirements. To define desirability as a metric, we leverage the notion of robustness introduced in Section 3. Formally, given a set of feature requirements, $R_1, R_2, ..., R_n$ as STL formulas, the overall desirability of a system behavior (represented as signal $\mathbf{s}$) is the *weighted sum* of the robustness of the individual requirements:

$$\rho_{sys} = w_1\rho(R_1, \mathbf{s}, t) + w_2\rho(R_2, \mathbf{s}, t) + ... + w_n\rho(R_n, \mathbf{s}, t)$$

We call ρ_{sys} the *global robustness* of the system.

For action $a \in \mathcal{A}$ and signal $\mathbf{s}_{curr}$ that describes the execution of the system up to the present point in time, let $\mathbf{s}^a$ be the signal that results from performing a for the next k steps of the system execution (i.e., $\mathbf{s}^a$ is an extension of $\mathbf{s}_{curr}$). Then, the problem of synthesizing an action for resolution can be formulated as follows:

$$\arg\max_{a \in \mathcal{A}} \left(w_1\rho(R_1, \mathbf{s}^a, t) + w_2\rho(R_2, \mathbf{s}^a, t) + ... + w_n\rho(R_n, \mathbf{s}^a, t)\right)$$

In other words, the resolution task involves finding action a that would result in a signal that maximizes the global robustness.

4.3.2 Normalization. Computing ρ_{sys} by simply adding robustness values for different requirements, as stated above, is not semantically valid, due to the following reasons. First, recall from Section 3 that the robustness for STL expression φ is computed based on the atomic functions over signal values (i.e., $\rho(u, \mathbf{s}, t) = f_u(\mathbf{s}_1(t), .., \mathbf{s}_k(t))$). Since different feature requirements are generally expressed over different signal functions, their robustness values cannot be directly compared or combined. Second, although robustness values are real numbers, operations over them (e.g., addition and subtraction) may not be semantically meaningful. For example, the robustness value of -1, signifying that the associated STL formula φ is violated by the margin of -1, may not be as "bad" as $\rho = 1$ is "good".

Before computing the global robustness as a sum of robustness for individual feature requirements, we first perform *normalization* of these values into a common, semantically meaningful domain. In particular, our normalization method is designed to achieve the following list of desiderata:

- The same robustness value produced by two different signal functions should have roughly the same magnitude: i.e., $\rho(u_1, \mathbf{s}_1, t) \sim \rho(u_2, \mathbf{s}_2, t) \sim ... \sim \rho(u_k, \mathbf{s}_k, t)$.
- Addition and subtraction of robustness values should also be meaningful; e.g., $(-1) + 1 \approx 0$, which represents the threshold at which the system satisfies a STL formula.

To enable normalization, our resolution framework is configured at design time with a pair of robustness bounds, ρ_{max} and ρ_{min}, for each signal function; i.e.; for some $\rho(u, \mathbf{s}, t) = f_u(\mathbf{s}_1(t), .., \mathbf{s}_k(t))$ and any pair of signal $\mathbf{s}$ and time t,

$$\rho_{min} \leq \rho(u, \mathbf{s}, t) \leq \rho_{max}$$

Intuitively, ρ_{min} represents the amount of violation that the system may be willing to tolerate and recover from (i.e., below this robustness threshold, the system behavior may be considered unacceptable or invalid); similarly, ρ_{max} represents the maximum degree of property satisfaction that is meaningful. These lower and upper robustness values are highly application-specific and thus are provided by the system engineer (an example of these values will be discussed in Section 4.3.4).

Given these ρ_{max} and ρ_{min} values, original robustness x is transformed into a normalized value within $[-1, 1]$ using the following *min-max scaling* function:

$$scale(x) = \begin{cases} \dfrac{x - \rho_{min}}{-\rho_{min}} - 1 & x < 0 \\ \dfrac{x}{\rho_{max}} & x \geq 0 \end{cases}$$

Based on our experience, however, we found this scaling alone insufficient for encoding the desirability of system behaviors. In most CPS systems, bringing the system from the state of violating a requirement (i.e., negative ρ) to satisfying it (i.e., positive ρ) is more important than improving the degree of satisfaction (i.e., further increase ρ that is already positive). To drive the resolver towards making decisions that mitigate violations, we further improve our normalization so that (1) negative robustness values are weighted more than positive ones, and (2) major requirement violations are treated exponentially worse than minor violations. In particular, we map the scaled robustness values from $[-1, 1]$ to $[-2, 1]$ using the following function, which penalizes violations of a property while maintaining linearity for positive robustness values:

$$penalize(x) = \begin{cases} x - \dfrac{\alpha^{-x} - 1}{\alpha - 1} & x < 0 \\ x & x \geq 0 \end{cases}$$

where α is some constant greater than 1. Higher α values result in steeper exponential curves; for our case study discussed in Section 5, we used $\alpha = 32$, to create a sufficiently exponential curve that severely penalizes increasing requirements violations without overpenalizing minor requirements violations. Such a curve improves the resolution system's ability to minimally violate certain requirements in order to prevent extreme violations of other requirements.

Then, fully normalized robustness values for atomic STL expressions are computed as follows:

$$\rho(u, \mathbf{s}, t) = penalize(scale(f_u(\mathbf{s}_1(t), .., \mathbf{s}_k(t))))$$

Robustness for other types of STL expressions (e.g., $\wedge$, $\vee$, $\mathbf{G}$, $\mathbf{F}$) are computed the usual way as described in Section 3. Then, normalized robustness values for different feature requirements can be added to compute the global robustness ρ_{sys}.

4.3.3 Algorithm. The resolution algorithm (Algorithm 1) takes four inputs: the set of conflicting actions, $\mathcal{A}_c$, the system specification, $R_{sys} = \{(w_1, R_1), (w_2, R_2), ..., (w_n, R_n)\}$, a model of the environment, M, and a signal $\mathbf{s}$ that describes the history of the system execution up to the current time.

Search space and sampling. The first step of the algorithm is to identify a finite set of candidate actions to be evaluated (line 3). This involves two tasks: (1) defining the search space given the set of conflicting actions and (2) sampling the elements of this space to be included in the candidate set $\mathcal{A}_{cand}$.

Since $\mathcal{A}$ may contain a large (possibly infinite) number of actions, it is infeasible to search this entire space. Instead, to bound the search space, we apply a heuristic based on the following idea: Given action a_i generated by some feature f_i (for $1 \leq i \leq n$), actions that are "closer" to a_i are more likely to fulfill the requirement of f_i than those that are further away from it. For example, given action

```
Input: $\mathcal{A}_c$:= conflicting actions; $R_{sys}$ := system specification;
       $M$ := environment model; $\mathbf{s}$ := current signal
1  fun resolve($\mathcal{A}_c, R_{sys}, M, \mathbf{s}$)
2      $\rho_{opt} := -\infty, a_{opt} := none$
3      $\mathcal{A}_{cand} := genCandidateActions(\mathcal{A}_c, G)$
4      for $a \in \mathcal{A}_{cand}$ do
5          $\mathbf{s}^a := execute(M, a, \mathbf{s}, window(R_{sys}))$
6          $\rho_a := computeGlobalRubustness(R_{sys}, \mathbf{s}^a)$
7          if $a_{opt} = none \lor \rho_a > \rho_{opt}$ then
8              $a_{opt} := a$
9              $\rho_{opt} := \rho_a$
10         end
11     end
12     return $a_{opt}$
13 end
```

Algorithm 1: Synthesis-based resolution algorithm.

a from the boundary enforcer that moves the drone in a particular direction, other actions that have the similar velocity vector as *a* are more likely to keep the drone within the safe region. Building on this idea, our heuristic is to narrow the search space to those actions that are located *between* the actions in $\mathcal{A}_c$—thus, minimizing the sum of distances to the actions generated by the features.

More formally, we assume that the set of actions $\mathcal{A}$ can be represented by a metric space (M, d) where M is an ordered set of elements in $\mathcal{A}$ and $d : M \times M \rightarrow \mathbb{R}$ is a metric function that computes a non-negative distance between any pair of elements in M. This assumption is reasonable in the context of CPS, where actions are typically used to manipulate continuous variables (e.g., velocity vector, steering angle, acceleration). Let a_{min} and a_{max} be the smallest and largest elements, respectively, in the given set of conflicting actions, $\mathcal{A}_c$. Then, the space of candidate actions is defined as the set of all actions between the a_{min} and a_{max}:

$$\mathcal{A}_{space} = \{a \in \mathcal{A} \mid a_{min} \leq a \leq a_{max}\}$$

This set, in general, may be infinite, and thus we further perform a sampling of this space to select a finite number of candidate solutions to evaluate. There are a number of different methods of sampling, and our approach does not prescribe a particular method. However, since resolution is performed during runtime, one important consideration is the number of the sampled actions: The overhead caused by robustness evaluation may interfere with system operations, depending on the timing properties of the controller (e.g., how frequent the controller issues an actuator command).

In our approach, we allow the system engineer to control the number of the sampled actions by providing an integer value for the sampling *granularity* (input G on line 3), which represents the number of samples per unit distance. Given this value, our sampling method partitions $\mathcal{A}_{space}$ into equally sized sets and samples one solution from each, resulting in $|\mathcal{A}_{cand}| = G * (a_{max} - a_{min})$. The more fine-grained the sampling is, the more likely that the search will find the best solution in $\mathcal{A}_{space}$; the engineer may adjust G depending on the amount of overhead that is acceptable to a particular application.

Signal estimation. Each of the candidate actions, $a \in \mathcal{A}_{cand}$, is then evaluated for its global robustness as defined by R_{sys}. To do this, the environment model is executed to extend $\mathbf{s}$ to a new signal, $\mathbf{s}^a$, which describes the estimated evolution of the system state assuming that action *a* is executed for some number of steps (line 5). The length of execution, also called the *window* of execution, is the minimum length of a signal needed to evaluate the robustness of a STL formula. To compute this, we adopt the method from a prior work ([39]) that analyzes the structure of formula φ to determine its window (ω), as follows:

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

In particular, $window(R_{sys})$ on line 5 is determined as the maximum of the windows of the STL formulas that describe the individual feature requirements.

Given the estimated signal $\mathbf{s}^a$, the global robustness ρ_a is computed and then compared against those of the other actions (lines 6 to 7). After evaluating $\mathcal{A}_{cand}$, the resolution algorithm terminates by returning an action that yields the highest global robustness value (line 12) among the candidate actions.

4.3.4 Example. In our running example, the overall action space $\mathcal{A}$ is defined as the set of all possible instances of action setVelocity(v_x, v_y, v_z) with concrete values for v_x, v_y, and v_z (which corresponds to the x, y, z components of the velocity vector, respectively). The distance between a pair of actions is defined as the Euclidean distance between the two vectors. We assume that the maximum speed of a drone is 2 m/s (i.e., $\sqrt{v_x^2 + v_y^2 + v_z^2} \leq 2.0$).

Suppose that the ego drone periodically keeps track of its own location and velocity as well as those of another non-friendly ("enemy") drone in its vicinity, as represented by the following signal:

$$\mathbf{s}(t) = (loc_{ego}, loc_{enemy}, vel_{ego}, vel_{enemy})$$

where loc_{ego} itself is a 3-tuple that contains the x, y, z coordinates of the ego drone (and similarly for other elements of the signal).

The drone is configured with two safety features: the boundary and runaway enforcers, as introduced in Section 2. The requirements of the two features are specified in STL as follows:

$$R_b : \mathbf{G}_{[0,1]}(timeToIntercept(\mathbf{s}(t)) - 1.5 \geq 0)$$
$$R_r : \mathbf{G}_{[0,1]}(distanceToEnemy(\mathbf{s}(t)) - 4.0 \geq 0)$$

where *timeToIntercept* is a function that returns the time (in seconds) before the ego drone crosses a designated boundary into an unsafe region, and *distanceToEnemy* returns the distance between the ego and the enemy drone. We assume that the features are assigned equal weights ($w_b = w_r = 0.5$).

Informally, R_b states that for the following 1 second after the controller issues a new actuator command[1], the drone must maintain at least a minimum time-to-intercept (TTI) of 1.5 seconds to the boundary. To normalize the robustness value ρ_b for requirement R_b, we use $\rho_{b_{max}}$=1.5, and $\rho_{b_{min}}$=−2.5. The upper bound of 1.5 signifies that being 3.0s (= 1.5 + 1.5) away from the boundary is sufficiently far enough that increasing the TTI even further is not critical for the boundary enforcer. On the other hand, once TTI falls below -1.0s (= -2.5 + 1.5), then the drone has already veered

[1]Since the controller is executed periodically at a much higher than 1 second, this window is sufficient to ensure that the safety property is maintained continuously throughout the entire drone mission.

deep into the unsafe region and should be considered as being in the least desirable state possible.

Similarly, the requirement for the runaway enforcer states that the ego drone must maintain a distance of at least 4.0 meters away from the enemy drone. We use $\rho_{r_{max}}$=4.0, and $\rho_{r_{min}}$=−4.0 as the upper and lower bounds on the robustness of the requirement R_r.

Suppose that the current state of the ego drone is as follows:

$$\mathbf{s}(t) = \big((8, 8, 10), (5.5, 5.5, 10), (1.2, 1.2, 0), (1.41, 1.41, 0)\big)$$

This signal shows that the ego drone is being chased by the enemy drone towards the top-right corner of the boundary. At this point, both enforcers become activated and generate actions to fulfill their individual requirements. In particular, the boundary enforcer generates setVelocity(−1.41, −1.41, 0), which would instruct the ego drone to fly towards the origin (0,0,0) at the maximum velocity of 2 meters per second. On the other hand, the runaway enforcer generates action setVelocity(1.41, 1.41, 0), intended to further distance the ego from the enemy drone.

Given this pair of conflicting actions, our resolver attempts to synthesize a new action that satisfies the requirements of both features, as described in Algorithm 1. The set of candidate actions $\mathcal{A}_{cand}$ are sampled from the space between the two conflicting actions: a_{min} = setVelocity(−1.41, −1.41, 0) and a_{max} = setVelocity(1.41, 1.41, 0). With $G = 10$ and the z-component fixed to 0, this would result in around 660 sampled actions. The resolver then evaluates each of these actions by executing it under environment model M, which estimates how the location and velocity of the ego drone evolves over time given the new velocity vector; the enemy drone is modeled as constantly updating its velocity in order to minimize its distance with the ego drone.

Executing the environment model for the window of execution (in this case, 1 second) with setVelocity(−2, 0, 0) generates a new estimated signal with the following values:

$$\mathbf{s}(t + 1) = \big((6, 8, 10), (6.91, 6.91, 10), (-2, 0, 0), (-1.28, 1.54, 0)\big)$$

Evaluating the normalized robustness values over this signal results in $\rho_r = -0.92$ and $\rho_b = 1$, which is combined using the given weights to yield $\rho_{sys} = 0.04$. After evaluating all of the sampled candidates, our resolver determines that setVelocity(−2, 0, 0) yields the highest ρ_{sys} and thus returns it as the final action to be executed by the flight controller.

5 EVALUATION

In this section, we describe the evaluation of our proposed resolution method. In particular, our goal is to answer the following research question: *Does the proposed synthesis-based approach to resolution result in more desirable system behaviors compared to existing methods?* We first describe a prototype implementation of our method as part of the flight control software for an autonomous drone, and then discuss a set of simulation-based experiments that demonstrate the effectiveness of our approach.

The code for our implementation and evaluation is available at: https://github.com/cps-sei/cps-synth-resolultion

5.1 Implementation

We implemented a prototype of our resolution framework on top of PX4, an open source flight control software in C++ used for a wide range of drones [23]. To run our experiments, we used the jMAVSim simulator (part of PX4), which simulates the physical dynamics of a drone as it reacts to commands from the flight control software.

Simulated scenarios. On top of PX4, we implemented the mission-level control logic that instructs the ego drone to perform the following mission: Fly to a number of waypoints, perform a reconnaissance task at each point, and return to the starting coordinate. A reconnaissance task involves dropping to some predefined altitude and flying in a figure-8 about the waypoint. To command the drone, the mission control software sends a velocity vector to the low-level flight control at a frequency of approximately 16.7Hz.

The environment model used for signal estimation was implemented as a C++ program that (1) takes the current signal and controller action as inputs and (2) produces a new set of signal values that describe how the environment evolves given this action (e.g., changes to the location and velocity of the enemy drone).

Features. Various safety features of the drone are implemented as components called *enforcers*, which are interposed between the mission controller and the low-level flight controller. Each enforcer monitors the action generated by the mission control for a potential safety violation (e.g., approaching the boundary within an unsafe distance) and if necessarily, overrides it with its own action that is intended to maneuver the drone to safety.

We implemented four safety features to test as part of our evaluation. Each feature is associated with certain operating conditions under which it becomes active and begins to generate actions that override those from the mission control. For instance, the boundary enforcer from the running example is activated when the distance between the drone and the boundary drops below a certain threshold. The following features were implemented:

- **Boundary enforcer** ensures that the drone stays within a predefined boundary, with its robustness value calculated based on signal function `timeToIntercept`.
- **Runaway enforcer** ensures that the drone maintains a minimum distance from a trailing drone, with its robustness calculated based on signal function `distanceToEnemy`.
- **Stealth enforcer** ensures that the drone stays above some minimum altitude $AL_{stealth}$ when flying through predefined regions, with its robustness calculated based on signal function `distanceToAltitude`, which measures the distance between the ego vehicle and some target altitude along the z-axis.
- **Flight enforcer** ensures that the drone does not crash into the ground by staying above minimum altitude AL_{ground}, with its robustness calculated based on function `distanceToAltitude`.

These features were chosen because they correspond to realistic features in autonomous drones and based on their requirements, could lead to interesting interactions.

5.2 Experimental Design

To evaluate our approach, we compared our resolution method (which we call *synthesis-based*, or **SynthR** for short) to the following two methods: (1) *priority-based* resolution (**PriorR**), which uses a fixed total ordering among the features to decide which one of the competing actions should be selected, and (2) *property-based* resolution (**PropR**), which evaluates each of the given conflicting

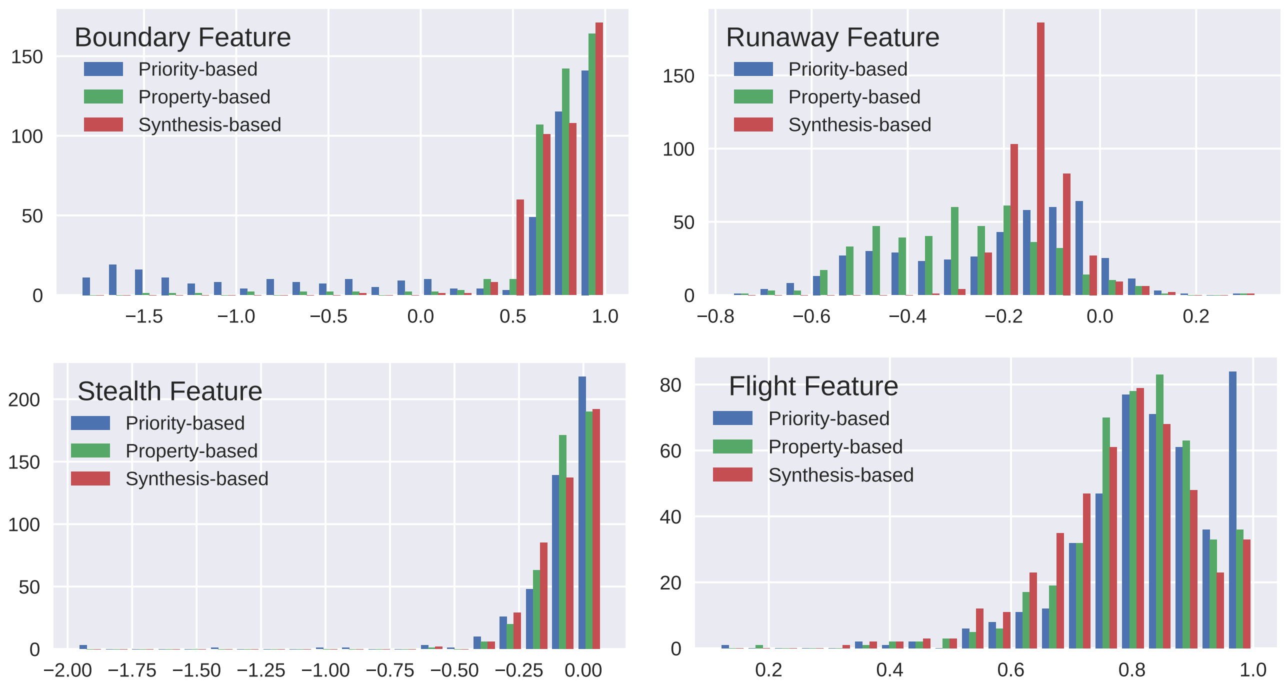


Figure 3: Histogram of robustness values for each resolution strategy. The X-axis represents the average robustness values for each run with the corresponding feature. The Y-axis represents the frequencies of those robustness values. Larger robustness values mean higher satisfaction of the feature requirement.

actions for global robustness ρ_{sys} and selects the one with the highest value, but does not involve synthesizing a new action.

To evaluate our approach, we devised the following hypotheses:

H_1: SynthR leads to a higher satisfaction of the feature requirements than PriorR.
H_2: SynthR method leads to a higher satisfaction of the feature requirements than PropR.

To test these hypotheses, we conducted 1500 controlled experiments. To avoid a selection bias, we randomly sampled 500 starting conditions (e.g., locations of the waypoints and the starting positions of the ego and enemy drones). For each of them, we executed the drone mission under all three resolution methods.

One risk to consider was that environment variables, such as the enemy drone speed or the size of the boundary, might implicitly change the importance of each feature, which could introduce a situational bias to one method over the other. For example, a faster enemy drone might lead to persistent violations of the runaway feature. A method that optimizes for the runaway feature will perform very well in this specific case, however may perform poorly in other feature interactions. On the other hand, decreasing the boundary size might cause the boundary feature to become the limiting factor, biasing the results towards an implementation that is optimized for this situation. Since the hypothesis to test in this evaluation is about the general techniques instead of specific application-dependent situations, we created a large space of environment variables and controlled for them to introduce the least possible situational bias.

In addition, to avoid a predominant order between features, we also randomly modified the weights of the features between the runs. For the priority method, the total ordering between the features were determined from their weights. To bound the number of candidate actions evaluated by SynthR, we used a granularity of 10 samples per unit. For example, for the range of [-1, 1] in the x-component of the velocity vector, this would result in 20 candidate solutions being sampled uniformly across this space (and if the same range was set for all x, y, z-components, $20^3 = 8000$ samples).

Data collection. For each run, we measured the robustness of the feature requirements every 0.06 seconds. We also collected the number of feature interactions (i.e., the occurrence of a conflict between at least two features), allowing us to later compare the effectiveness of the methods with and without conflict.

5.3 Experimental Results

We evaluated and compared the effectiveness of the resolution methods in terms of the robustness that they were able to achieve during the simulated runs. Figure 3 shows the results of our experiments.

To evaluate the significance of the results, we used a *t-test*, which determines whether there is a statistically significant difference between the means of two distributions (in this case, the distribution of average robustness values collected over the simulation runs). Once an effect is determined by the t-test to be statistically significant, the practically more relevant metric is *effect size*, since it states how large the difference between a pair of methods is. In particular, we used Cohen's D [16], one of the most common ways to measure effect size. Cohen's D describes the difference between two means in terms of the standard deviation of the overall distribution.[2] For

[2]A value of approx. 0.2 is considered a small effect size. 0.5 is considered a medium effect size and 0.8 is considered a large effect size [16].

all measurements, we also built a regression model that controls for the starting conditions. Since both analysis methods resulted in the same conclusions, we focus on the results of the more intuitively understandable t-test as well as Cohen's D as effect size measure.

5.3.1 SynthR vs. PriorR. The data analysis results show that the SynthR method is significantly better at satisfying the requirements of multiple features and avoiding their violations than the PriorR method. This subsection will discuss this in more detail.

When comparing the weighted sum of the robustness values of all four features (i.e., global robustness), our results show that SynthR is significantly better than PriorR (Cohen's D = 0.403 with $p < 0.001$, meaning that the global robustness achieved by SynthR is almost 0.403 standard deviations larger).

One noteworthy observation, which can be seen in Figure 3, is that PriorR tends to result in negative robustness values more frequently. For example, note how the requirement of the boundary feature is often violated under PriorR, whereas SynthR rarely results in a violation. This is a consequence of the "winner-takes-all" nature of PriorR, which simply disregards the requirements of the lower-priority features and thus results in their violations.

Looking at the robustness of individual feature requirements, we observe that SynthR is significantly better than PriorR for both the boundary with medium to large effect size (Cohen's D = 0.737 medium to large effect size, $p < 0.001$) and runaway feature with a medium effect size (Cohen's D = 0.573, $p < 0.001$). This was a surprising outcome, since we had expected that for individual features, PriorR would perform just as well (if not better) than SynthR. This can be attributed to the fact that SynthR, by design, values the improvement of negative robustness values more than the improvement of positive robustness values in order to avoid requirement violations. So it trades off the improvement of a potentially violated requirements (in this case, boundary and runaway) with the a small compromise with an already highly satisfied requirement (in this case, flight).

The stealth feature shows no significant effect size for a difference between between the two resolution strategies (Cohen's D = -0.053, $p = 0.425$). For the flight feature, SynthR performs significantly worse with a small to medium negative effect size (Cohen's D = -0.43, $p < 0.001$). This can be attributed to the fact that conflicts involving these features are relatively rare, and in such cases, optimizing for one particular feature (as done by PriorR) may overall be just as effective as, if not better than, SynthR.

The larger effect size for the improvement in the boundary feature and runway outweighs the smaller effect size for a decrease in the flight feature. So we can conclude that, in our case study, SynthR shows a better satisfaction of the multi-dimensional requirements than PriorR. **This leads us to accepting hypothesis H_1.**

5.3.2 SynthR vs. PropR. Our results show that SynthR is significantly better than PropR while feature interactions are occurring, but only insignificantly better outside of these time periods.

To be more specific, when comparing PropR with SynthR with respect to global robustness, the difference is not statistically significant (Cohen's D = 0.12, $p = 0.07$). The primary reason for this is that during the parts of execution in which no feature conflict exists, all feature requirements are satisfied regardless of the selected resolution method. It is only during the period of interactions in which SynthR can be shown to achieve higher global robustness than PropR. This can be seen by conducting a time series analysis of the global robustness over each run.

We conducted a regression discontinuity analysis [8, 17, 44] to compare PropR with SynthR by evaluating how the level and slope of the global robustness time series of the two methods change over time. In this analysis, each individual mission is modelled as a time series with a data point for each time tick. All data points of the 1500 time series are then analyzed in one regression model that enables us to statistically compare the slopes of the robustness values during the presence of a feature interaction versus not. The resulting regression equation is:

$$\begin{aligned}\rho_{sys} \approx &Intervention + Synth * Intervention + Time + \\ &TimeAfterIntervention + Synth * TimeAfterIntervention + \\ &ENEMY\_DRONE\_SPEED + BOUNDARY\_SIZE\end{aligned}$$

$Intervention$ is a Boolean indicating whether a conflict is taking place; $Synth$ is a Boolean for whether the experiment was conducted with SynthR or not; $Time$ is the number of ticks passed since the start of the mission; $TimeAfterIntervention$ is the number of ticks after a conflict started, and $ENEMY\_DRONE\_SPEED$ and $BOUNDARY\_SIZE$ are the control variables for environment configurations. $Intervention$ tells us the average change in robustness values during the presence of a feature interaction versus not. $Synth * Intervention$ tells us the difference in change in robustness when using SynthR versus not. $TimeAfterIntervention$ tells us the average change in the *slope* of robustness when a conflict is taking place. Similarly, $Synth * TimeAfterIntervention$ constitutes the average difference in robustness slope for SynthR.

The entire regression model explains 30.1 % of the variance. The results are shown in Figure 4. All independent variables are statistically significant ($p < 0.001$ for each). Overall, the improvement achieved by SynthR is 0.077 robustness units (95 % confidence interval (CI) $[0.064, 0.089]$) when controlling for the mentioned variables. So there is a small improvement overall when using SynthR, if we disregard the presence or absence of feature interactions.

When a conflict occurs, the global robustness is reduced on average by 1.469 robustness units (CI $[1.485, 1.454]$), which is a significant difference. The average robustness difference for SynthR is 0.29 robustness units smaller (CI $[0.267, 0.313]$) than for PropR, which is about 20 % of the overall difference. Hence, we can conclude that SynthR performs significantly better in achieving higher global robustness than PropR when the system is encountering a conflict.

Furthermore, as shown in Figure 5, SynthR results in a smoother and more stable path with less extreme changes, which is a desirable property in a cyber-physical system.

In conclusion, in our case study, there is a large effect size for improvements achieved by SynthR during the presence of feature interactions, while there is relatively a small effect size during their absence. **This leads us to accepting hypothesis H_2.**

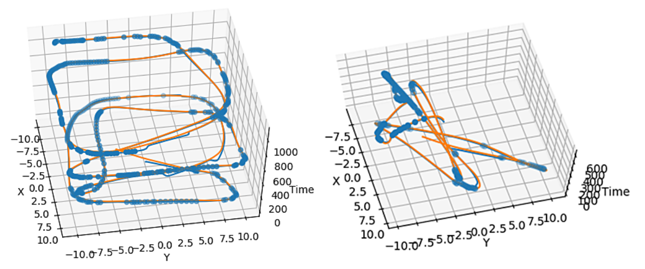
5.4 Performance Overhead

Although our results suggest that SynthR can achieve higher global robustness, it also involves more computation in comparison to the other two methods, mainly due to the evaluation of additional actions. Since PX4 requires the velocity vector to be updated at

	coeff.	**std err.**	R^2
Intercept	6.1152^{***}	0.028	-
Intervention	-1.4694^{***}	0.008	25.523 %
ENEMY_DRONE_SPEED	-0.6874^{***}	0.020	0.293 %
BOUNDARY_SIZE	0.0372^{***}	0.000	1.845 %
Synth	0.0767^{***}	0.007	0.349 %
Synth:Intervention	0.2900^{***}	0.012	0.133 %
TimeAfterInterv.	-0.0069^{***}	8.46e-05	1.642 %
Synth:TimeAfterInterv.	-0.0096^{***}	0.000	0.265 %
Time	0.0002^{***}	1.84e-05	0.029 %

$^{***}p < 0.001 \quad N = 409512 \quad R^2 = 30.1\,\%$

Figure 4: Results for the regression discontinuity analysis. The intervention is defined as the time frame in which feature interactions are happening. The synthesis-based resolution leads to higher satisfaction of the requirements, in particular during the time while feature interactions are happening.



(a) Synthesis-based resolution **(b) Property-based resolution**

Figure 5: Time-spatial visualization of a representative run for both methods. Orange lines represent the flight path and the big blue dots represent the the points at which a conflict takes place. SynthR results in a smoother path.

least 2Hz, the operation of the drone may be affected negatively if resolution takes longer than the maximum allowed time between controller updates (i.e., 0.5 seconds). To test this, we measured the duration of each resolution task and checked whether this exceeded the maximum cycle length[3]. In summary, with the granularity of sampling used for our case study, we did not observe any overhead that was significant enough to violate the timing requirement on controller updates and negatively impact the drone operation.

In practice, to deploy our resolution method, the system engineer would take into account the hardware capacity and timing properties of the controller system, and would adjust the sampling granularity to ensure that resolution does not interfere with the drone operation.

5.5 Threats to Validity

Construct validity. When measuring properties of CPS, simulations are often the only practical evaluation technique. This constitutes a threat to construct validity, since every simulation omits some properties of the real world. However, the asserted safety properties measure execution logic instead of accuracy of physics. Further, any imprecision applies to all experiment instances to the same extent. So while our measurements might include a small error, it should not bias the results towards a particular conclusion of H_1 or H_2.

Internal validity. The practical constraint of partially sampling a space of configurations (i.e., actions) poses a threat to internal validity since the possibility of sampling exceptional cases can never be ruled out. This threat was addressed by sampling a large number of different cases and by paying close attention to statistical significance. However, the concrete numbers for effect sizes should not be over-interpreted, but rather only used to get an overall impression of how large the difference can be.

External validity. The performance measurements of the run time overhead are application-specific and hardware-dependent. On other hardware and for other applications the results may vary greatly. So with more restricted hardware and more complex computation, the implementation would need more performance tuning than our prototypical implementation. However, we believe in the conclusion that in most cases the improved ability to resolve feature conflicts outweighs the insignificant increase in computation.

Furthermore, the results of the case study might not necessarily generalize to other potential use cases of the presented approach. Although we believe that the presented case study embodies common characteristics of CPS, the results might differ for other kinds of CPS, features, or use case scenarios. The main claim of this evaluation is that the presented approach is able to significantly outperform existing work. The empirical evaluation explicitly does not make any claim about whether the measured effect sizes apply to other cases, or whether H_1 and H_2 hold true for a majority of CPS. However, since the proposed approach is also supported by a theoretical framework that outlines its advantages, we expect it to perform well on many other CPS that have to deal with conflicting features that need to be resolved during run-time.

6 RELATED WORK

Feature interactions. Feature interaction problems have been studied extensively by the software engineering community [3–7, 9, 11, 15, 35, 41, 42, 48, 49]. There are two major sub-problems in feature interactions: *detection* and *resolution*. Here, we mainly focus on the prior approaches to resolution.

Raghavan et al. [39] proposed a *property-driven* approach to resolution, where they also use the notion of robustness in STL to resolve conflicts between a set of competing feature actions. Our work improves on theirs in the following ways: (1) our approach allows a specification to be expressed as a weighted sum of robustness of properties that describe the requirements of conflicting features, whereas their approach is limited to a single global property (that is not tied to a particular feature), and (2) our approach is capable of synthesizing a new action, while their approach is limited to selecting one of the given feature actions. Thus, as demonstrated in our study, our approach is more effective at producing desirable system behaviors when there is no clear priority among the features, and all of their requirements are critical to system operation.

[3]The performance measurements were made on an iMac with an 3.8 GHz Intel Core i5 CPU and 16 GB RAM

Another related approach is the *negotiation-based* approach to run-time resolution [26], where a central *mediator* is used to resolve conflicts among multiple system agents. Their approach is similar in that the goal of negotiation is to devise actions that are considered *acceptable* to all agents involved (similar to how we attempt to synthesize an action that satisfies the requirements of the conflicting features). However, our approach significantly differs in the type of specifications used to describe desired system behaviors (constraint-based *policies* that describe acceptable system operations in their approach versus CPS properties in ours).

Our work is also related to the *variable-specific* approach introduced in [9, 49], where each actuator variable (e.g., acceleration) is associated with a resolution *strategy* that is designed to resolve conflicts over that variable (e.g., "select the action that results in the smallest acceleration value"). However, it may be difficult for the system engineer to anticipate all possible interaction scenarios (e.g., especially less common ones, like those in Figure 1) and devise a strategy that can produce a desirable system outcome under them. In comparison, since our approach does not rely on a fixed, design-time strategy, it is capable of handling scenarios that are unanticipated by the engineer (as long as the the model of the environment is general enough to cover those scenarios).

Prior approaches to resolution that use a priority or precedence ordering ([12, 13, 27, 34, 50]) are not effective for resolving interactions in systems where the desirability of a feature action is highly context-dependent (e.g., the runaway enforcer, normally the top-priority feature, may be of lower importance when the drone is about to cross the boundary into an unsafe region).

Runtime monitoring and verification. STL and other similar specification formalisms (e.g., metric temporal logic, or MTL [29]) has been used for online monitoring of system properties [18, 19, 21, 43]. Runtime techniques have also been developed for *enforcing* a desirable property by observing and possibly modifying system actions [1, 25, 37, 46]. However, these prior works focus on evaluating a *single* execution trace for property satisfaction, and do not involve a comparison of multiple actions or traces.

Self-adaptive systems. *Self-adaptive systems* is an active area of research on designing systems to be capable of adjusting to changes in their operating environment [10, 14, 30, 36, 40]. Our proposed resolution framework shares some of the common characteristics of a self-adaptive system, in that it relies on information collected by monitoring the state of the system and its environment to determine if it is necessary to take an action to maintain system requirements and to plan what action to take. In addition, it shares the use of a predictive model as in model-based approaches to proactive self-adaptation [2, 33] to evaluate the effect of candidate actions over the near future. Our approach supports the formal specification of the desired properties to be maintained using STL and provides a resolution approach when there are conflicts between them.

7 DISCUSSIONS AND FUTURE WORK

Role of existing features. Since our resolver works by effectively overriding the actions of the existing features, one may wonder whether there is any need for the existing features at all. One could, for example, conceive of a system architecture where a controller based on our synthesis method is used at all times to generate an action that is optimal with respect to a set of requirements. However, the synthesis procedure can be computationally expensive, and thus our method is likely to be more cost-effective in systems where conflicts are rare rather than being common events. In these systems, the existing features implementations will still be needed to control the behavior of the system as desired. Additionally, structuring the system as set of independent features, instead of having a single, monolithic controller that synthesizes actions, allows for the benefits of modularity as system requirements change over time.

Optimality of the approach. Since we randomly sample from the (possibly infinite) search space, our current method does not provide any theoretical guarantees about the optimality of the synthesized action, but only that it results in a higher global robustness than the given conflicting actions (assuming the accuracy of the environmental model used for signal estimation). However, our method may be augmented with search techniques such as stochastic gradient descent or simulated annealing to more efficiently explore the space and converge to an approximate global optimum.

Generality. Our approach makes several assumptions regarding the system that may not hold in general. We assume the environment can be effectively modeled within some prediction window, and that this prediction window is sufficient for approximating an optimal action. Additionally, designers must be able to determine a fixed maximum violation and satisfaction for each property. Future work could improve the normalization method to automate finding these parameters or better account for the different rates of change of robustness values across properties. Applying an exponential penalty to violations generalizes to any system where further violation of one property is worse than further satisfaction of another. When any violation constitutes a maximum violation, this can be captured by setting the maximum violation accordingly.

Some specifics of our case study do not generalize to other domains. Our method of determining a search space is specific to our case study because all of our actions are defined by 3D velocity vectors, and we use this knowledge to define our search space. Different CPS with different action spaces may require different methods to define or search for actions, but the details of these methods are orthogonal to our high-level approach and remain as future work. In addition, the naive sample-and-test method used for action synthesis generalizes to any CPS. However, such a naive search method requires a greater number of candidate actions to be tested and therefore would be infeasible in domains where evaluating a candidate action is computationally expensive. As future work, leveraging optimal control theory [28] could more efficiently identify Pareto-optimal actions, and could potentially do so offline without needing to discretize and search the action space.

ACKNOWLEDGMENTS

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center, by Toyota Motor North America, Inc., and NSF Award No. 1852260: CMU REU Site in Software Engineering (REUSE). DM20-0193

REFERENCES

[1] Björn Andersson, Sagar Chaki, and Dionisio de Niz. 2017. Combining Symbolic Runtime Enforcers for Cyber-Physical Systems. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings.* 68–84.

[2] K. Angelopoulos, A. V. Papadopoulos, V. E. S. Souza, and J. Mylopoulos. 2016. Model Predictive Control for Software Systems with CobRA. In *2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS).* 35–46. https://doi.org/10.1109/SEAMS.2016.012

[3] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. 2010. Detecting Dependences and Interactions in Feature-Oriented Design. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010.* 161–170.

[4] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of feature interactions using feature-aware verification. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011.* 372–375.

[5] Sven Apel, Alexander von Rhein, Thomas Thüm, and Christian Kästner. 2013. Feature-interaction detection based on feature-based specifications. *Computer Networks* 57, 12 (2013), 2399–2409.

[6] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. 2013. Strategies for product-line verification: case studies and experiments. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013.* 482–491.

[7] Joanne M. Atlee, Uli Fahrenberg, and Axel Legay. 2015. Measuring Behaviour Interactions between Product-Line Features. In *3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, FormaliSE 2015, Florence, Italy, May 18, 2015.* 20–25.

[8] Howard S. Bloom. 2012. Modern Regression Discontinuity Analysis. *Journal of Research on Educational Effectiveness* 5, 1 (2012), 43–82. https://doi.org/10.1080/19345747.2011.578707

[9] Cecylia Bocovich and Joanne M. Atlee. 2014. 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, November 16 - 22, 2014.* 553–563.

[10] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. 2009. Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems.* Springer, 48–70.

[11] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41, 1 (2003), 115–141.

[12] A. Chavan, L. Yang, K. Ramachandran, and W. H. Leung. 2007. Resolving Feature Interaction with Precedence Lists in the Feature Language Extensions. In *Feature Interactions in Software and Communication Systems IX, International Co nference on Feature Interactions in Software and Communication Systems, ICFI 2007, 3-5 September 2007, Grenoble, France.* 114–128.

[13] Yi-Liang Chen, Stéphane Lafortune, and Feng Lin. 1997. Resolving Feature Interactions Using Modular Supervisory Control with Priorities. In *Feature Interactions in Telecommunications Networks IV, June 17-19, 1997, Montréal, Canada.* 108–122.

[14] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Dagstuhl Seminar Report.* 1–26.

[15] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010.* 335–344.

[16] Jacob Cohen. 1977. . Academic Press. https://doi.org/10.1016/B978-0-12-179060-8.50011-6

[17] Thomas D Cook and Donald T Campbell. 1979. The design and conduct of true experiments and quasi-experiments in field settings. In *Reproduced in part in Research in Organizations: Issues and Controversies.* Goodyear Publishing Company.

[18] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. 2017. Robust online monitoring of signal temporal logic. *Formal Methods in System Design* 51, 1 (2017), 5–30.

[19] Adel Dokhanchi, Bardh Hoxha, and Georgios E. Fainekos. 2014. On-Line Monitoring for Temporal Logic Robustness. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings.* 231–246.

[20] Alma L. Juarez Dominguez, Nancy A. Day, and Jeffrey J. Joyce. 2008. Modelling feature interactions in the automotive domain. In *International Workshop on Modeling in Software Engineering (MiSE).* 45–50.

[21] Alexandre Donzé, Thomas Ferrère, and Oded Maler. 2013. Efficient Robust Monitoring for STL. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings.* 264–279.

[22] Alexandre Donzé and Oded Maler. 2010. Robust Satisfaction of Temporal Logic over Real-Valued Signals. In *Formal Modeling and Analysis of Timed Systems - 8th International Conference, FORMATS 2010, Klosterneuburg, Austria, September 8-10, 2010. Proceedings.* 92–106.

[23] Dronecode Project. 2020. PX4 autopilot. https://px4.io.

[24] Georgios E. Fainekos and George J. Pappas. 2006. Robustness of Temporal Logic Specifications. In *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers.* 178–192.

[25] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. 2011. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design* 38, 3 (2011), 223–262.

[26] Nancy D. Griffeth and Hugo Velthuijsen. 1994. The negotiating agents approach to runtime feature interaction resolution. In *Feature Interactions in Telecommunications Systems, May 8-10, 1994, Amsterdam, The Netherlands.* 217–235.

[27] Jonathan D. Hay and Joanne M. Atlee. 2000. Composing features and resolving interactions. In *ACM SIGSOFT Symposium on Foundations of Software Engineering, an Diego, California, USA, November 6-10, 2000, Proceedings.* 110–119.

[28] D.E. Kirk. 2004. *Optimal Control Theory: An Introduction.* Dover Publications. https://books.google.com/books?id=fCh2SAtWIdwC

[29] Ron Koymans. 1990. Specifying real-time properties with metric temporal logic. *Real-time systems* 2, 4 (1990), 255–299.

[30] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. 2015. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing* 17 (2015), 184–206.

[31] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems.* Springer Berlin Heidelberg, 152–166.

[32] Andreas Metzger. 2004. Feature interactions in embedded control systems. *Computer Networks* 45, 5 (2004), 625–644.

[33] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. Proactive Self-Adaptation under Uncertainty: A Probabilistic Model Checking Approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015).* https://doi.org/10.1145/2786805.2786853

[34] Masahide Nakamura, Hiroshi Igaki, Yuhei Yoshimura, and Kousuke Ikegami. 2009. Considering Online Feature Interaction Detection and Resolution for Integrated Services in Home Network System. In *ICFI.* IOS Press, 191–206.

[35] Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. 2008. Feature interaction: The security threat from within software systems. *Progress in Informatics* 5 (2008), 75–89.

[36] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. 1999. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14, 3 (May 1999), 54–62.

[37] Srinivas Pinisetty, Partha S. Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. 2017. 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, July 10-14, 2017.* 80–89.

[38] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977.* 46–57.

[39] Santhana Gopalan Raghavan, Kosuke Watanabe, Eunsuk Kang, Chung-Wei Lin, Zhihao Jiang, and Shinichi Shiraishi. 2018. Property-Driven Runtime Resolution of Feature Interactions. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings.* 316–333.

[40] Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *TAAS* 4, 2 (2009), 14:1–14:42.

[41] Wolfgang Scholz, Thomas Thüm, Sven Apel, and Christian Lengauer. 2011. Automatic detection of feature interactions using the Java modeling language: an experience report. In *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011. Workshop Proceedings (Volume 2).* 7.

[42] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland.* 167–177.

[43] Prasanna Thati and Grigore Rosu. 2005. Monitoring Algorithms for Metric Temporal Logic Specifications. *Electr. Notes Theor. Comput. Sci.* 113 (2005), 145–162.

[44] Donald L Thistlethwaite and Donald T Campbell. 1960. Regression-discontinuity analysis: An alternative to the ex post facto experiment. *Journal of Educational psychology* 51, 6 (1960), 309. https://doi.org/10.1037/h0044319

[45] US NHTSA. 2010. ABS ECU Programming, 2010 Toyota Prius Recalls. https://www.nhtsa.gov/vehicle/2010/TOYOTA/PRIUS/4%252520DR/FWD#recalls.

[46] Meng Wu, Haibo Zeng, Chao Wang, and Huafeng Yu. 2017. 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*. 84:1–84:6.

[47] Lana Yarosh and Pamela Zave. 2017. 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, May 06-11, 2017*. 2993–2997.

[48] Pamela Zave. 1993. Feature Interactions and Formal Specifications in Telecommunications. *IEEE Computer* 26, 8 (1993), 20–30.

[49] Mohammad Hadi Zibaeenejad, Chi Zhang, and Joanne M. Atlee. 2017. 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, September 4-8, 2017*. 408–418.

[50] P. Ann Zimmer and Joanne M. Atlee. 2012. Ordering features by category. *Journal of Systems and Software* 85, 8 (2012), 1782–1800. https://doi.org/10.1016/j.jss.2012.03.025