Systematically Debugging IoT Control System Correctness for Building Automation

Chieh-Jan Mike Liang[‡], Lei Bu[◇], Zhao Li[⋆‡], Junbei Zhang[⋆‡],
Shi Han[‡], Börje F. Karlsson[‡], Dongmei Zhang[‡], Feng Zhao[‡]

[‡]Microsoft Research [◇]Nanjing University [⋆]University of Science and Technology of China

ABSTRACT

Advances and standards in Internet of Things (IoT) have simplified the realization of building automation. However, non-expert IoT users still lack tools that can help them to ensure the underlying control system correctness: user-programmable logics match the user intention. In fact, non-expert IoT users lack the necessary know-how of domain experts. This paper presents our experience in running a building automation service based on the Salus framework. Complementing efforts that simply verify the IoT control system correctness, Salus takes novel steps to tackle practical challenges in automated debugging of identified policy violations, for non-expert IoT users. First, Salus leverages formal methods to localize faulty user-programmable logics. Second, to debug these identified faults, Salus selectively transforms the control system logics into a set of parameterized equations, which can then be solved by popular model checking tools or SMT (Satisfiability Modulo Theories) solvers. Through office deployments, user studies, and public datasets, we demonstrate the usefulness of Salus in systematically debugging the correctness of IoT control systems for building automation.

CCS Concepts

•Human-centered computing → Ambient intelligence; •Software and its engineering → Software testing and debugging;

Keywords

policy verification; policy violation debugging; IoT

1. INTRODUCTION

Behind the realization of building automation is a user-programmable control software system, which governs complex interactions among IoT-enabled sensors and devices. Interestingly, leveraging decades of experience from the wireless sensor networks (WSN) community, most industry solutions adopt the event-driven programming paradigm, where users author sets of rules to associate triggering sensor events and triggered device actions. These automation rules are also known as IFTTT-style rules (If This Then That). Millions of IoT users are exposed to IFTTT-style rules [27, 46], through popular vendor-specific products [9, 28] and integration services [2, 6]

Although automation rules such as IFTTT-style rules can intuitively and easily express the user-intended building automation, ensuring the underlying control system *correctness* (w.r.t. user expectations) is actually a difficult task. This observation is crucial as IoT-enabled actuation can be liable to our physical ill-being [17, 37]. One challenge is that, the space of reachable system states quickly grows with the number of IoT-enabled sensors and devices deployed, and the amount of time the system runs for. Most importantly, many IoT users are *non-experts* – people who lack the necessary know-how and tools to ensure the control system would behave as intended.

While academic communities have studied the problem of formally verifying system against policies [7, 18, 26, 30], these efforts typically rely on domain experts to manually debug faulty logics or error traces. Recently, the cyber-physical system (CPS) community started to verify building automation behavior. DepSys [35] checks for conflicting actuation triggered by multiple rules concurrently, but it does not verify system behavior against user expectations (i.e., policies). To this end, SIFT [31] demonstrated the feasibility of automatically verifying IoT control system correctness w.r.t. user policies. However, SIFT lacks mechanisms for users to subsequently understand the lengthy error trace and debug identified policy violations. Our studies suggest that many users give up on debugging after 45 seconds.

This paper presents our experience in running a building automation service based on the **Salus** framework. Complementing efforts that simply verify the IoT control system correctness, Salus tackles practical challenges in enabling *automated debugging* of identified policy violations for non-expert IoT users. While design principles are language-agnostic, Salus currently assumes the logic is sets of event-driven IFTTT-style rules, and user expectations to verify are specified as policies in conjunction of conditions.

Salus implements the following **design principles** to au-

Lei Bu and Chieh-Jan Mike Liang are corresponding authors. Feng Zhao is now at Haier.

BuildSys '16, November 16-17, 2016, Palo Alto, CA, USA

DOI: http://dx.doi.org/10.1145/2993422.2993426

```
IF room.temp < 18    THEN room.fireplace = on;
IF room.temp > 24    THEN room.fireplace = off;
IF room.CO > 180     THEN room.fireplace = off;
IF room.temp > 28    THEN house.hvac = off;
IF house.hvac == on THEN room.fireplace = off;
```

Listing 1: An automation task/app that controls the living room temperature.

```
bedroom.occupancy == TRUE
 AND bedroom.env_brightness <= 31.5
 AND bedroom.light.switch = ON
```

Listing 2: An example of user policies for the energy efficiency.

tomate debugging policy violations.

First, to effectively localize faulty user-programmable logics, Salus leverages formal methods. Specifically, for each policy violation, the counterexample from model checking indicates both system initializations (e.g., initial building states) and state transitions (e.g., automation rules). These information can be the starting point for the subsequent fix formulation. While bounded code exploration [31] could potentially be used in place of model checking, it has false negatives as it uncovers only policy violations happening within some k time steps. Furthermore, estimating k has been proven difficult.

Second, to efficiently formulate potential fixes for faulty user-programmable logics, Salus selectively transforms control system automation rules into a set of parametrized equations. Doing so can leverage state-of-the-art SMT (Satisfiability Modulo Theories) solvers to find satisfiable solutions, which are then filtered based on preferences to formulate personalized fix suggestions. Finally, these fix suggestions are presented to the user.

In summary, this paper makes the following **contributions**. We identify manual debugging as the pain point for non-expert users to ensure the control system correctness of building automation. And, we take the first step to enable automated debugging for policy violations, with two novel components in the Salus framework. Salus cleverly reformulates the problem statement into sub-problems, where advances in formal methods or SMT solvers can be leveraged. Results show Salus's fix recommendations have a user acceptance rate of 91%. And, it can debug 82% more policy violations while saving about 87.47% of debugging time, as compared to manual inspection.

2. BACKGROUND AND MOTIVATIONS

Before we present our approach to debug IoT control systems for building automation, this section introduces two common inputs of debugging: user-programmable logics (or automation rules), and policies. Then, it discusses the current debugging practice.

2.1 Event-driven Programming Paradigm

Building on decades of experience in the WSN community, most building automation services adopt event-triggered rules for user-programmable logics. Popularized by IFTTT.com [2], these automation rules are also known as IFTTT-style rules. An IFTTT-style rule (referred to as if-this-then-that) has the following semantic: `IF some event occurs THEN perform some action`. The triggered event comes from IoT devices attached to the IoT control system. In Salus, the automation task for a building appliance can consist of several IFTTT-style rules, and Listing 1 shows a real-world example. Salus executes rules in the authored order to resolve conflicting device commands issued by multiple rules (e.g., turning `room.fireplace` on and off at the same time).

2.2 Policies for Intended System Behavior

Policies (or specifications) are the common practice to express the intended control system behavior, which should not be violated. And, formal verification is one approach to decide whether the control system would violate any policy. Policies are written as the conjunction of conditions, and Listing 2 illustrates with a case of room brightness control. At the underlying system level, we note that Salus implements linear temporal logic (LTL) [16] to express the temporal constraints. The default LTL operator in Salus is Global (i.e., always), as it is most commonly used in automation scenarios. The decoupling between human-readable policies and complexities in formal verification allows Salus users to focus only at the level of building automation.

2.3 Debugging Control Systems

To debug a policy violation, the common industry practice is to present domain experts with a counterexample. As an error trace, the counterexample provides two pieces of information describing how the violation was reached: (1) control system initial state values, and (2) a path of control system state transitions.

Unfortunately, relying on non-expert IoT users to understand the counterexample is not feasible. First, the counterexample shows how a policy violation happens, but not why. For example, suppose Listing 1 violates the user intention that only one of HVAC and fireplace should be on at a time, it is not obvious which rules contribute to the policy violation. Second, understanding a counterexample can take as much effort as preparing a program for model checking. Our user study suggests that, given the counterexample, non-expert users typically give up debugging in less than one minute. While there are efforts to help users to understand the counterexample [7], they do not help users in fixing.

3. OUR APPROACH

As Figure 1 shows, Salus complements existing efforts that simply verify the IoT control system correctness. In case of verification failure, Salus automatically debugs and formulates solutions for the user. User inputs are automation tasks encoded in IFTTT-style rules (c.f. §2.1), user policies encoded in conjugation of conditions (c.f. §2.2), and descriptions of devices and environment (c.f. §4.1 and §4.2).

With the violated user policy identified by the existing policy verification efforts, the *Fault Localization engine* (FaultLoc) tries to localize the faulty user-programmable logics. This information is subsequently used to formulate fixes. It includes how the violating state can be reached from some initial system state values, through some sequence of state transitions. This step relies on symbolic model checkers to perform unbounded verification, which achieves verification

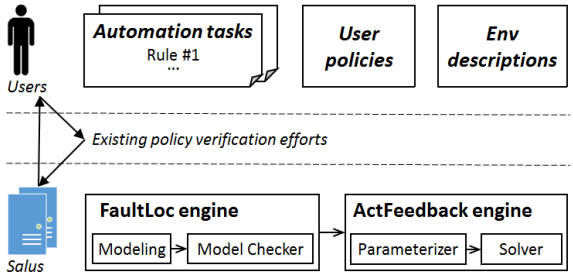

Figure 1: Salus framework complements existing policy verification efforts for IoT control systems, to enable automated debugging of faulty user-programmable logics. Salus builds on advances in formal methods and solvers to achieve this goal.

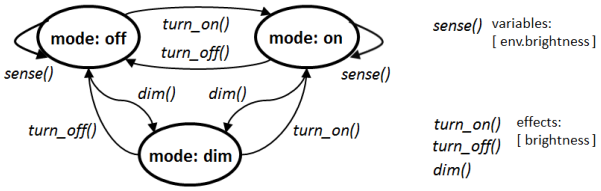

Figure 2: Example FSM model of a light with a brightness sensor.

completeness at the expense of slightly increased computation overhead. Model checking needs FSM (finite state machine) models for the IoT control system to be verified, and §4 describes the three aspects that Salus tries to automatically model: *devices*, *environment*, and *time*.

Based on the output of FaultLoc, the *Actionable Feedback engine* (ActFeedback) tries to formulate two types of fix suggestions for the localized user-programmable logics. First, it attempts to determine whether any automation rule parameters could be changed. Second, it attempts to see if additional automation rules could be introduced. Then, based on the user preference, ActFeedback formulates a solution which are then presented to the user for approval.

4. FAULT LOCALIZATION

The FaultLoc engine relies on unbounded model checkers to localize faulty user-programmable logics. In the case of building automation, running model checking requires three aspects to be modeled: *IoT devices*, *environment*, and *time*. While the common practice in related areas is for domain experts to manually build scenario-specific models, we attempt to automate this process for non-expert IoT users. This section discusses what schema information is needed, and how these schema information can be collected.

4.1 Device Modeling

Figure 2 shows the FSM model generated for a light device, with transitions between `ON` and `OFF` states. In general terms, a FSM contains a finite set of states, the initial State, variables, a finite set of synchronous events, and a set of state transitions.

To generate the FSM model, we argue that each device should be associated with a profile of standardized schema that encodes the necessary device specification. The following pieces of data schema are necessary: (1) A set of device *Working Modes*, which will be translated into states in the FSM. One of them marked as Starting Mode. (2) A set of device *Variables*, which will be translated into a set of FSM variables. These represent values read from the environment, or internal data kept on the device. (3) A set of device *Triggers*, which describes the conditions of working model transitions. Such rules will be translated into *Transitions* in the FSM. (4) A list of device *Commands*, which describes what kind of action can be executed by it. Such command interfaces will be translated into synchronous labels in the FSM, which can help to connect FSM models of different devices. (5) A set of *Effects* for commands that actuate on the environment. Effects describe which domains are affected upon executing an automation command.

Preparing the device schema information above requires an understanding of device specifications, and thus this task should ideally be carried out by manufacturers or experts. Fortunately, there is a lot of momentum in maintaining device schema. First, many cloud-based IoT device registries are available from industry [5, 38] and community [47]. These registries typically follow the JSON data format, which are sufficiently flexible to encode schema for a wide range of devices. Second, many device discovery standards [1, 22] allow IoT devices to advertise their own schema.

4.2 Environment Modeling

Salus creates one FSM model for each environment variable such as temperature. States in the FSM model represent discrete values the variable can take on.

Since device actuation has impacts on the environment variable, transitions in the environmental model are triggered by state transitions in the device model. An example is turning on a heater would increase the temperature in the same space. Symbolic model checkers can support such behavior with parallel state transitions (e.g., asynchronous system in NuSMV checker) to change both the device model and the environment model. As the mapping from device actuation to environment variable is necessary, Salus represents this information by associating each device action to its affected domains. This approach is similar to the DoST ontology [14].

To model the building characteristics, Salus allows the user to specify what IoT devices and sensors are in the same logical space. A physical space (e.g., a large and open office space) is divided into one or several logical spaces. And, devices can influence the readings of sensors in the same logical space. This modeling methodology minimizes the user overhead in modeling the indoor environment while maintaining the fidelity, as most devices cannot influence sensors that are very far away.

4.3 Time Modeling

Salus can support temporal behavior modeling (e.g., a heater gradually heats up a space) with one simple extension: defining state transitions per time unit. For example, for each time unit that a heater is on, the room environment model transitions to a state corresponding to the amount of heat accumulated so far.

While the efficient realization of temporal behavior modeling is out of scope for this paper, our future work investigates how hybrid automaton [25] can reduce the modeling overhead.

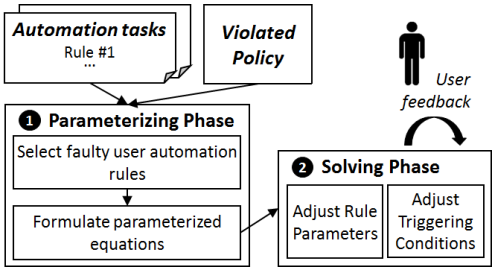

Figure 3: The ActFeedback engine suggests rule fixes which adjust parameters and triggering conditions.

5. FIX FORMULATION

With the faulty user-programmable logics localized by the FaultLoc engine, the ActFeedback engine automatically formulates potential fixes for the control system. Two types of fixes are attempted by the ActFeedback engine: changing automation rules' parameters or thresholds (c.f. §5.1), and adding or deleting rules' triggering conditions (c.f. §5.2).

Technically, as Figure 3 illustrates, the ActFeedback engine formulates fixes through two phases. First, the *parameterizing phase* cleverly frames the IoT control system as a set of parametrized equations, whose right-side equality is constraints of the violated policy. Then, to solve these parameterized equations, the *solving phase* re-purposes model checkers' ability to solve for unknown parameters. And, it subsequently translates these parameter values into fix recommendations, and presents to the users for approval.

Intuitively, the parameterizing phase highlights the search space where satisfiable solutions could be found. As some building automation deployments can have a lot of components (e.g., device/environment models, policies, and automation rules), exploring the entire search space can incur a significant overhead. To reduce the search space, the ActFeedback engine parametrizes only the following automation rules: (1) automation rules that share devices or domains with the violated policy, and (2) automation rules that the FaultLoc found to be part of the state transitions leading to the policy violation.

We note that, mathematically speaking, a system of parameterized equations can have many satisfiable solutions. In such cases, the ActFeedback engine presents users one fix recommendation at a time, and users can iterate through recommendations. It is also possible to learn user preferences and prioritize recommendations accordingly, and this section discusses such approaches.

The rest of the section discusses approaches and solver strategies that the ActFeedback engine uses to formulate both types of fix suggestions.

5.1 Rule Parameter Adjustment

The first form of rule fix suggestions is by adjusting rule parameters. Considering the following automation rule:

```
IF kitchen.CO2 > 1000 THEN kitchen.fan = on
```

If the enforced safety policy is (`kitchen.CO2` $\leq$ `1000`), or "kitchen CO_2 level cannot exceed 1000 ppm, globally", the fan could be turned on later than expected. While this particular problem can be addressed by modifying the CO_2 threshold, the challenge is to automate this process.

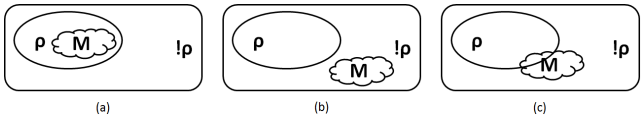

Figure 4: Possible logical relations between spaces represented by the control system behavior, M, and policies, ρ. The control system behavior satisfies policies in case (a), but not (b) and (c).

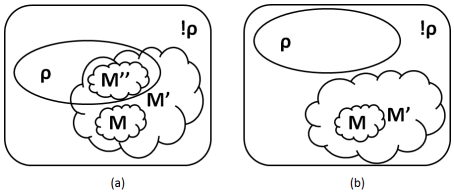

Figure 5: Ideally, the ActFeedback engine should find M'', a solution space that overlaps with the space of policies. This is possible in case (a) – changing automation rules enlarges the original control system behavior space, M, to M', and selecting solutions reduces M' to M''. However, this is not possible in case (b).

The basic idea is to re-use model checkers' ability of solving parametrized equations – this is typically for finding counterexamples of policy violations. Imagine an equation – where the left side represents a chain of system transitions (partially dictated by app rules), and the right side represents conditions of a policy – instead of fixating rule parameters as traditionally done, we fixate system initial values and make rule parameters free. Then, model checkers can solve for these free variables that do not satisfy the negation of a violated policy. The negation is necessary as model checkers find counterexamples, not satisfied examples. For the sake of discussion, we assume unbounded model checkers that do not put a bound on verification coverage.

Next, we expand the basic idea above with formal notations. Given an IoT system model M and a specification ρ, we say $M \vDash \rho$ if all behaviors of M fall into ρ. As shown in Figure 4.A, if the rounded rectangle is the complete set, the ellipse is the set of behaviors that satisfies ρ, and the cloud shape is the behavior set of M, the cloud shape should be a subset of ρ. However, if $M \nvDash \rho$, the state space could be like Figures 4.B or 4.C.

Intuitively, fixing a violation of an IoT system means changing the state space of M from either Figure 4.B or 4.C to that from Figure 4.A. The first step is to parametrize the original rule set, by substituting all parameter values with free variables. Continuing the kitchen example above, the parametrized rule below has a free variable X taking a value between [0, 2500]:

```
IF kitchen.CO2 > X THEN kitchen.fan = on
```

Since the value of x is changed from a concrete value 1,000 to a range of [0, 2500], many new behaviors are introduced into the system. Therefore the behavior space of M is a subset of M'. The behavior space of M' may or may not overlap with that of ρ. Using Figure 4.C as an example, Figure 5.A and 5.B illustrate the relations among M', M, and others.

If M' does not overlap with ρ (c.f. Figure 5.B), it means the system can never satisfy the specification. However, if

M' overlaps with ρ (c.f. Figure 5.A), then some configurations of rule parameters can satisfy the specification. And, the challenge is in finding these configurations, as represented by M''. Clearly, if M'' exists, $M'' \vDash \rho$, in other words $M'' \nvDash \neg\rho$. Therefore, the way we try to get the configuration of M'' is asking the model checker to check whether $M' \vDash \neg\rho$. If the result is false, the model checker will return a concrete counterexample trace τ, in which all the parametrized variables will be assigned to a concrete value. As τ falls in ρ, the configuration in τ makes a potential candidate for M''.

To realize this procedure, the ActFeedback engine concretizes the parametrized model M' by the configuration in τ, and it gets a new model M_τ. If $M_\tau \vDash \rho$, then we have found a fix for the original system, as the new configuration now satisfies the specification. If $M_\tau \nvDash \rho$, we can start the procedure again looking for a new potential fix. In order to avoid configurations in τ which have been already tried, we simply add new clauses into the parametrized system to ensure parameters can not be set to them again. For instance, suppose we have k parametrized variables $v_1\, v_2 \ldots v_k$, we just add an invariant into the model M' that $\neg((v_1 == v_{1_\tau}) \&\& (v_2 == v_{2_\tau}) \ldots \&\& (v_k == v_{k_\tau}))$. As the system is a finite state machine (i.e., the number of potential configurations is limited), this procedure is guaranteed to terminate.

Finally, in the case of $M_\tau \vDash \rho$, M_τ needs to be verified again by the Verification engine. This is because the configuration in τ may only work with the specific initial state in τ, not all the other initial states.

5.2 Triggering Condition Adjustment

Another form of fix suggestions is by adjusting triggering conditions (i.e., add or delete). Considering the following automation rule:

```
IF livingroom.temp <= 18 THEN livingroom.heater = on
```

If the policy is `(livingroom.occupancy == false AND livingroom.heater == off)`, or "if no one is in the living room, then heater should always be off", the violation can only be fixed by introducing additional conditions to the automation rule.

Similar to rule parameter adjustment, we adjust triggering conditions by parameterizing the existence of each IF subclause. We introduce a boolean variable λ_i for each subclause c_i in the conditions policies provide hints on. For example, parameterizing the living room rule above would give:

```
IF λ1  ⇒ (livingroom.temp <= X) AND
   λ2  ⇒ (livingroom.occupancy == Y)
THEN livingroom.heater = on
```

The basic idea is as follows. If $M_p \nvDash \neg\rho$, we will get a counterexample trace τ, with concrete values for each λ_i. If λ_i is $TRUE$, the sub-clause c_i has to be true to make the IF expression stands, which means the sub-condition c_i should be kept in the rule. Otherwise, c_i will not affect the behavior of τ, as it has been "short-circuited" by λ_i. In this case, c_i can be removed.

Similar to how we implement adjusting rule parameters, we concretize the parametrized model by the configuration in τ. If $\lambda_i == TRUE$, we keep the sub condition c_i, otherwise, we remove it. Then, we check whether the resulting model M_τ is a valid fix. We note that, as the number of λ_i is finite, the procedure is guaranteed to terminate. Lastly, to identify possible IF sub-clause to add, the ActFeedback engine takes hints from violated policies, by considering all conditions in the policies but not in the rules.

5.3 Extensions and Additional Considerations

Prioritizing Solutions. We acknowledge that the solution space of a set of parameterized equations can be large. And, it is challenging to decide the best fitting recommendation for a user. Considering the kitchen fan example in §5.1, while any CO_2 threshold between 0 and 999 would theoretically not violate the policy, a more conservative user would prefer a relatively lower threshold than others. Salus tackles this problem by getting hints from users in two ways. First, Salus prioritizes the recommendation whose parameters are closest to the original. In the kitchen fan example, Salus would first recommend the CO_2 threshold at 999. Second, while asking for a new fix recommendation, users can explicitly specify their preferred range of values for individual IoT devices.

Alternative Parameter Solving Technique with SMT. This section so far builds on general model checking techniques to find potential fixes. While this procedure can handle a wide spectrum of specifications, e.g., linear temporal logic (LTL) and computation tree logic (CTL), the computational overhead can grow fast with the complexity of equations to solve (c.f. §7.3).

For trivial policies without LTL and CTL syntax, SMT (Satisfiability Modulo Theories) solvers can be the alternative approach for faster parameter solving. Technically, we systematically encode all transitions within a bounded k number of transitions, into a set of parametrized SMT formulas F_k. Then, we ask the SMT solver (e.g., Z3 [19]) to test whether F_k can be satisfiable w.r.t. constraints of policy. If yes, then a solution naturally becomes a possible rule fix.

The challenge of such bounded checking is in choosing the bound, k. In our procedure, when general model checking gets a counterexample trace τ, the length of τ, l_τ, could be a good candidate for k. In other words, it makes sense to use SMT to fix the state space of the model reachable in l_τ steps.

6. IMPLEMENTATION

The Salus prototype has $\approx 13k$ lines spanning over system components, algorithms, and a front-end graphical UI (GUI) tool. Components are implemented as a set of C# and Python modules that communicate via RESTful interfaces and a TCP-based control plane. The Salus prototype currently supports interfacing with networked Philips Hue [42], Belkin WeMo [9], and AllJoyn [1] devices.

The GUI tool acts as the control hub. In addition to visualizing discovered devices, the tool offers configurations for (1) the environment where automation takes place (i.e., house and office), (2) user policies, and (3) deployed IoT apps. After physical devices are installed and user IoT apps are submitted, the GUI tool automatically contacts appropriate system components to verify. For any identified policy violations, it subsequently calls system components to start debugging. Finally, the GUI tool presents fix recommendations to users for approval.

We build the FaultLoc engine on top of state-of-the-art formal verification tools: NuSMV [15] as the model checker

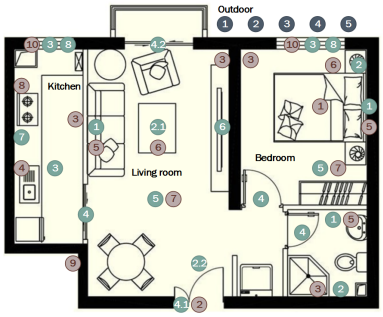

Figure 6: Floor plan of a "standard house", used in user studies to elicit app rule sets.

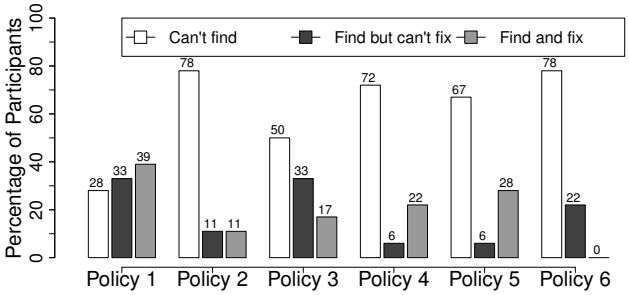

Figure 7: User study results suggest that manually finding and fixing policy violations are difficult for IoT users.

	Policy #					
(Unit is second)	1	2	3	4	5	6
Model checking	0.33	0.13	0.26	0.06	0.06	0.07
Sym. exec. testing	8.93	17.99	12.07	4.62	4.14	4.85

Table 1: Time needed to find each policy violation in User Study #2, with model checking (used by Salus) and symbolic execution based testing.

and Z3 [19] as the SMT solver. Both tools are available online.

The ActFeedback Engine currently has two implementations based on either model checkers (NuSMV) or SMT solvers (Z3). The former is the default because it can handle violations of policies written in LTL. But, Salus can be configured to use the latter for better performance in some cases (c.f. §7.3). Through the front-end graphical UI, users can either accept and apply a fix recommendation, or ask for another one. Moreover, users can specify that specific parameters or app rules should not be modified, and the engine then tries to find solutions that accommodate this request.

7. EVALUATION

Our key evaluation findings are as follows: **(i)** Formally ensuring the IoT control system correctness is an arduous task for humans. Even with a technical background, less than 25% of IoT users properly fix faulty user-programmable logics resulting in policy violations. **(ii)** Salus is able to localize faulty user-programmable logics for all intentionally induced policy violations and real-world case studies. **(iii)** Salus scales better than previous IoT-related verification efforts. Its formal model checking technique takes 98.29% less time on average, as compared to symbolic execution testing. **(iv)** Our fix suggestions have a user acceptance rate of 91%.

7.1 User Studies

To understand the pain points in ensuring IoT control system correctness, we conducted two 30-people user studies. Studies assumed the IoT-enabled apartment shown in Figure 6. The distributions of our 30 participants were as follows: 94% are male and 6% were female, 87% were 20 $\sim$ 35 years old and 13% were over 35 years old. These participants were either studying or working in the computer science field.

User Study #1 asked participants to write IFTTT-style automation rules for representative automation scenarios: brightness control, temperature control, environment personalization, occupants well-being, house security, energy efficiency, etc. An example is "write rules to control lights based on occupancy". In addition, participants were asked to describe events that should not happen, to provide us with a sense of user policies.

In User Study #2, given 29 IFTTT-style automation rules and six policies for the above apartment, participants tried to find and fix all six policy violations. Policy #1 is "bedroom temperature should be between 21 and 27 Celsius", #2 is "bedroom should not be bright or over-bright when someone is sleeping", #3 is "all lights should be off when no one is home", #4 is "entrance door should be locked when door RFID reading is not recognized", #5 is "balcony door should be locked if no one is home", and #6 is "kitchen gas level should never be danger". During the study, we also recorded the time taken by participants to answer each question.

We present motivating insights from user studies. Figure 7 shows manually localizing and debugging policy violations can be arduous. First, out of the six policy violations, Policy #1 matches our expectation of being the easiest to fix, with the highest percentage of participants (39%) being able to find and fix the violation correctly. For most questions, less than 25% of participants successfully completed. Second, while trying to fix policy violations, many participants did not realize that a fix might involve changing several rules, updating several triggering conditions, or considering several devices. For example, to properly fix Policy #3, participants need to ensure that automation rules exist for each light. And, to properly fix Policy #2, participants need to consider both inside and outside lighting sources. Third, we observed that many participants gave up debugging a policy violation after 45 seconds.

7.2 Fault Localization

The Fault Localization engine (FaultLoc) uses formal model checking to localize faulty user-programmable logics w.r.t. a policy violation. Compared to the symbolic execution based testing used by previous IoT-related efforts [31], formal model checking generally provides a stronger guarantee on false negatives and exhibit much less computation overhead. This section quantifies these two observations.

For comparison, we collected baseline results with Pex, a state-of-the-art symbolic execution tool [44] widely used in the academic community. To prepare inputs for Pex, we rewrote automation rules in the form of IF statements, and wrapped them around with a WHILE loop (to simulate continuously triggering app rules per time unit). In addition,

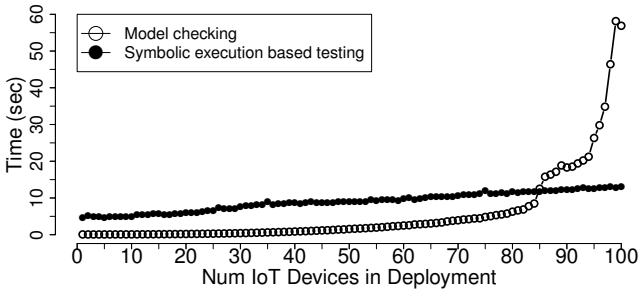

Figure 8: The runtime complexity of verifying one policy differs for techniques based on model checking and symbolic execution based testing.

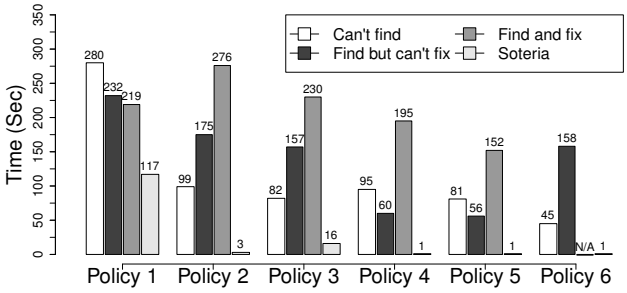

Figure 9: Salus can find and fix policy violations with 87.47% less time than human.

we rewrote policies as assertions at the end of the WHILE loop. Since Pex is bounded, we instructed Pex to stop if no policy violation is detected after 200 state transitions.

Table 1 shows the time needed to find individual policy violations in User Study #2 (based on a house with 41 devices). On average, Salus used 98.29% less time to find a policy violation than symbolic execution. In fact, for symbolic execution, the amount of time needed also depends on the number of system state transitions before a policy violation is triggered. For example, our empirical data suggest that Policy violation #2 was found at the 132^{nd} iteration, which is much larger than 17 iterations for Policy #5.

However, as the control system complexity scales up, symbolic execution might have faster verification time. Fortunately, building automation at homes and offices might not reach this scale. To measure the impact of the number of devices, we took Policy #1 from User Study #2, and incrementally added devices that periodically sample (e.g., clocks). Figure 8 illustrates that model checking has an exponential runtime complexity due to the state space growth. Since symbolic execution based testing is bounded, the linear growth is simply due to the overhead of simulating additional devices (at the expense of false negatives). Finally, for the scenario tested in Figure 8, while model checking is outperformed at 86 devices, most IoT deployments have much less devices in a logical space (e.g., living room).

7.3 Fix Formulation

This section evaluates the ActFeedback engine with two major metrics: (1) the user acceptance rate of fix recommendations, and (2) the computation overhead of generating fix recommendations.

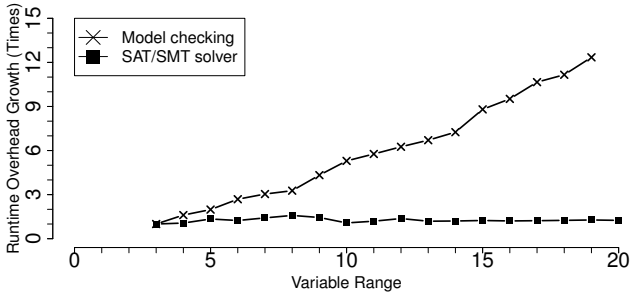

Figure 10: The impact of variable range on runtime complexity of generating a fix suggestion that updates four app rules. We compare techniques based on model checking and SMT solvers.

7.3.1 User Acceptance

User study #2 results suggest that our fix suggestions had a user acceptance rate of 91%. And, ∼9% of fix recommendations were rejected due to personal preferences, and users subsequently requested for another fix recommendation.

Figure 9 shows that Salus could find and fix policy violations with 87.47% less time, as compared to human. Interestingly, this figure also suggests that many users gave up after 45 seconds if they could not identify policy violations, and after 56 seconds if they cannot fix the identified policy violations. Therefore, the lack of automated debugging introduces a significant gap for which unsafe IoT control systems can creep in.

An interesting observation is the amount of time needed to generate fix suggestions varies from one policy to another. Policy #1 needed the most time. As we drill down in the microbenchmarks next, there are several factors behind this, which include the range of variables. Policy #1 depends on the room temperature, which has a wider range of states than binary variables such as door open/close.

7.3.2 Computation Overhead

We now discuss the ActFeedback engine's computation overhead. As mentioned in §5, fix recommendations are essentially solutions to a set of parametrized equations (formulated from sequences of automation rules) that satisfy a given constraint (formulated from policies). In addition to the complexity of equations (e.g., number of free variables, and variables' value range), the underlying solver technique also impacts the computation overhead. Our current system implementation can use either a general model checker (based on NuSMV) or an Satisfiability Modulo Theories (SMT) solver (based on Z3). As this section demonstrates, while the former supports a wider spectrum of policies, the latter scales more gracefully with the complexity of equations.

Figure 10 illustrates the runtime overhead to generate a policy violation fix that involves changing four app rules. To collect evaluation data, we use Policy #1 from User Study #2, and this policy concerns temperature whose variable range can be easily extended. Empirical results suggest that general model checkers cannot efficiently solve equations with variables of large range, and the runtime complexity grows faster than bounded model checker. On the other hand, being designed to solve equations efficiently, SMT solvers have a more graceful runtime complexity. In

fact, we also observe the same pattern while varying the number of free variables needed to solve.

By default, Salus currently uses model checking for generating violation fix recommendations. This way, it can support a wider spectrum of policies, which includes violations that happen after an unknown number of system transitions. However, users can switch to SMT solvers, if necessary.

8. DISCUSSION

This section discusses overarching issues related to the design of the Salus framework.

Model Limitations. As with all model checking efforts, behaviors that are not encoded in the models cannot be checked. To reduce human errors in manually constructing models, Salus automates model constructions as much as possible. However, our current implementation does not efficiently model some aspects of an IoT system. One such aspect is temporal properties, e.g., a heater takes time to pre-heat. While our current realization assumes each state transition takes up one time unit, the state space can grow quickly to slow down verification. Furthermore, the environment model does not fully consider all physics and dynamics. We plan to incorporate new techniques (e.g., hybrid automata [11]) to address these limitations.

Scalability in Practice. Both model checking and SMT solving have an increasing complexity w.r.t. the system size. For instance, the satisfiability problem is known to be NP-complete, which suggests no known way to efficiently locate a solution. Since Salus leverages the state of the art from the formal verification community, it inherits these limitations but also benefits from any advances. Furthermore, since buildings are physically divided into spaces and rooms, the number of devices to be considered at a time has a relatively lower complexity as compared to other control systems. Therefore, the runtime overhead from model checking and SMT solving does not incur significant user-perceivable delays in our case.

Support for Other Programming Paradigms. Our current implementation supports IFTTT-style app rules, due to their popularity and flexibility [36, 39, 45]. Since model checking is a language-agnostic concept, tools are also available for various other programming paradigms. However, depending on the language complexity, generating fix suggestions might require additional work. We plan to explore this direction in the future.

Learning User Preferences. While being out of scope for this paper, data mining could be one venue for Salus to learn user preferences and subsequently apply to formulating fix suggestions. We share our experience below to spark further exploration.

User preferences can be derived from frequently observed user behavioral patterns, and each pattern can be encoded as the conjugation of event conditions. Such patterns can be constructed with the decision tree classifier. The output is a tree whose branches consist of intermediate nodes describing conditions on observed sensor data, and leaf nodes predicting a device's state. Decision trees have several characteristics that fit the case of IoT. First, decision trees do not assume prior knowledge of the learning graph topology (e.g., the precise set of sensors that would influence an actuation), especially as non-experts do not typically have domain knowledge of devices and physical phenomena. Second, each decision tree branch is already a conjugate of conditions.

Dataset	Number of devices	Avg daily data ptrs
CASAS-7	$M\times51$, $D\times14$, $H\times2$, $W\times2$, $T\times5$, $P\times1$	14,127
CASAS-8	$M\times51$, $D\times14$, $H\times2$, $W\times2$, $T\times5$, $P\times1$	12,071
CASAS-13	$L\times17$, $M\times44$, $T\times5$	6,126
HH120	$M\times11$, $MA\times4$, $L\times7$, $LS\times15$, $T\times4$, $D\times3$	3,605
HH122	$M\times19$, $MA\times5$, $T\times5$, $LS\times24$, $D\times4$	64,687
BJW-1	$L\times1$, $M\times1$, $T\times1$, $LI\times1$	85,107
BJW-2	$L\times1$, $M\times1$, $T\times1$, $LI\times1$	86,223

M: Motion. D: Door. H: Heater. W: Water flow.
T: Temperature. LS: Light sensors. P: Power.
L: Light. LI: Light intensity.
MA: Wide-area infrared motion sensors.

Table 2: The deployment set up for our office datasets and CASAS datasets.

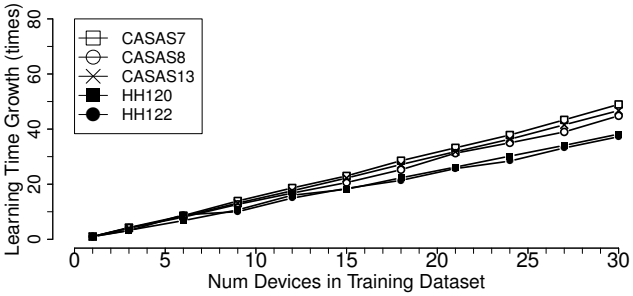

Figure 11: The deployment complexity impacts the time needed to learn the user preferences in device usage.

Constructing decision trees is an iterative process that builds layer by layer in a top-down fashion. It starts by choosing the most significant tuple of (*sensor*, *threshold*) and splits the dataset based on the threshold. Standard practice defines this significance by the information gain, or how well a (*sensor*, *threshold*) tuple predicts an actuating device state. Building subsequent layers of the decision tree follows the same process to split the remaining data. The process stops either when the decision tree already reaches four levels, or when the classification correct rate is 100%. From crawling $\approx 50,000$ shared recipes on IFTTT.com and app rules authored in our user study, we observe that typical human-readable rules do not exceed four conditions.

We next show preliminary results to illustrate the potential of learning user preferences in assisting Salus. We used seven deployment datasets (c.f. Table 2): two are from our office deployments (BJW-1 and BJW-2), and five are from CASAS [4] (CASAS-7, CASAS-8, CASAS-13, HH120, and HH122). Both office datasets have two months of data, with a sampling interval of five seconds. All CASAS datasets are around one week long, with a sampling interval varying between 10 and 300 seconds.

From 98 patterns mined, we manually inspected top 30 patterns of high confidence. 91.11% of these presented patterns were marked as being reasonable. The reason behind many cases of recommendation rejection is the incomplete-

ness of datasets. For example, if the living room heater had only `ON` state in the dataset, we would record the pattern `(livingroom.heater == on)`.

Both the deployment size can impact the training time. To quantify this aspect, we varied the number of devices in CASAS datasets. Figure 11 illustrates a linear relationship between the number of devices in the training set and the training time.

9. RELATED WORK

IoT Security and Testing. Most efforts from the security community target the IoT connectivity: protocol design, encryption or authentication [1, 3], privacy [34], etc. We argue that better programming support is another key to close the IoT safety gap, especially in ensuring the programmed behavior matches the intended behavior. In the context of IoT, SIFT [31] took the first step to demonstrate the feasibility of using symbolic execution techniques to automatically find safety problems. Building on this momentum, Salus tackles practical challenges in enabling automated debugging of identified policy violations for non-expert IoT users.

IoT Control System Verification. Many IoT platforms aim for a flexible programming interface and high usability, but relying on IoT users to debug problems without sufficient assistance can be difficult [10]. While HomeOS [21] allows priorities to be assigned to IoT apps, priorities are not sufficient if two apps do not form a clear hierarchy or distinction of priority. In addition, many efforts on running concurrent applications in sensor networks [33, 49] are either limited in resolving conflicts (e.g., imposing overly strong restrictions) or too complex to be comprehensible to non-experts. Finally, DepSys [35] places the burden of properly specifying app intents and dependencies on developers, and it does not address system-wide policy violations.

While SIFT [31] demonstrated the potential of formally verifying IoT apps, Salus addresses challenges in making formal verification practical in real world. These challenges include automated assistance for policy authoring and violation debugging. Furthermore, SIFT relies on symbolic execution based exploration that is bounded, but Salus adopts model checking to traverse the full state space. As a result, Salus can answer more complex questions, e.g., temporal behavior.

Fault Localization and Correction Assistance. Counterexamples (or traces) can be too long or too convoluted (e.g., if there are many processes involved) to make debugging failures a trivial task. Many efforts tried to add explanations to counterexamples. [8] processes traces and highlights causes of a violation, as the way to explain the failure to users. Significant attention has also been given to the extraction of additional information about the model from the trace to ease debugging [7, 13, 20, 23, 24, 30]. These range from calculating distance metrics between a counterexample and correct traces [24] to translating a counterexample into a formula where a maximal number of clauses can be discarded as not the source of faults [30]. While users need to manually map the findings of these approaches back to the original code, there are also efforts on improving the feedback by pointing out relevant code fragments [40].

Salus goes beyond counterexamples, and it tries to recommend IoT app rule fixes. While automatic repairing of problems has recently been an active area in the software testing community [12, 32, 41, 48], our approach cannot rely on non-expert users to write test cases or extract semantics from static analysis of programs. Some efforts focus on Boolean programs [29, 43], but IoT app interactions require more complicated theories such as integer arithmetic. Moreover, Salus needs to consider fix suggestions that do not significantly change intended automation.

10. CONCLUSION

IoT control systems for building automation can run automation tasks authored by non-expert users. These users require assistance to ensure their user-programmable logics will match the intended behavior. Complementing efforts that simply verify the IoT control system correctness, Salus demonstrates the feasibility and benefits of automating debugging. The paper also discusses the potential of leveraging data mining as a future extension to the Salus framework. In addition, we plan to explore other formal method techniques, to improve the performance in handling temporal states and large-scale systems.

11. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for constructive reviews and Prof. Anand Sivasubramaniam for shepherding.

Lei Bu is supported by the National Key Basic Research Program of China (2014CB340703) and the National Natural Science Foundation of China (61561146394, 91318301, 61572249).

12. REFERENCES

[1] AllJoyn. http://www.alljoyn.org.
[2] IFTTT: Put the internet to work for you. http://ifttt.com.
[3] Thread. http://threadgroup.org.
[4] WSU CASAS Datasets. http://ailab.wsu.edu/casas/datasets.
[5] Amazon. Device Registry for AWS IoT. http://docs.aws.amazon.com/iot/latest/developerguide/thing-registry.html.
[6] Apple. HomeKit. http://developer.apple.com/homekit.
[7] T. Ball, M. Naik, and S. K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *POPL*, 2003.
[8] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. *Formal Methods in System Design*, 2012.
[9] Belkin. Wemo. http://www.belkin.com/us/Products/c/home-automation.
[10] A. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon. Home automation in the wild: challenges and opportunities. In *CHI*, 2011.
[11] L. Bu, Q. Wang, X. Chen, L. Wang, T. Zhang, J. Zhao, and X. Li. Toward Online Hybrid Systems Model Checking of Cyber-Physical SystemsŠ Time-Bounded Short-Run Behavior. In *ACM SIGBED Review*, 2011.
[12] S. Chandra, E. Torlak, S. Barman, and R. Bodík. Angelic Debugging. In *ICSE*, 2011.

[13] M. Chechik and A. Gurfinkel. A framework for counterexample generation and exploration. *STTT*, 9(5-6), 2007.

[14] C. Chen and S. Helal. A Device-Centric Approach to a Safer Internet of Things. In *NoME-IoT*, 2011.

[15] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*. Springer, 2002.

[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *Trans. PLS*, 8(2), 1986.

[17] CNN. Why It's So Easy To Hack Your Home. http://cnn.com/2013/08/14/opinion/schneier-hacking-baby-monitor/, 2013.

[18] J. Croft, R. Mahajan, M. Caesar, and M. Musuvathi. Systematically Exploring the Behavior of Control Programs. In *ATC*. USENIX, 2015.

[19] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[20] N. Dershowitz, Z. Hanna, and A. Nadel. A Scalable Algorithm for Minimal Unsatisfiable Core Extraction. In *SAT*. Springer, 2006.

[21] C. Dixon, R. Mahajan, S. Agarwal, A. B. Brush, B. Lee, S. Saroiu, and P. Bahl. An Operating System for the Home. In *NSDI*. USENIX, 2012.

[22] Google. Weave. http://developers.google.com/weave.

[23] A. Griesmayer, S. Staber, and R. Bloem. Automated Fault Localization for C Programs. In *ENTCS*. Elsevier, 2007.

[24] A. Groce. Error Explanation with Distance Metrics. In *TACAS*, 2004.

[25] T. A. Henzinger. The Theory of Hybrid Automata. In *LICS*, 1996.

[26] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. In *CAV*. Springer, 1997.

[27] Icontrol. State of the Smart Home 2015. Technical report, 2015.

[28] Insteon. Insteon. http://insteon.com.

[29] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and Fixing Faults. In *Journal of Computer and System Sciences*, 2012.

[30] M. Jose and R. Majumdar. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. *SIGPLAN Note*, 46(6):437–446, June 2011.

[31] C.-J. M. Liang, B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu. SIFT: Building an Internet of Safe Things. In *IPSN*, 2015.

[32] F. Logozzo and T. Ball. Modular and Verified Automatic Program Repair. In *OOPSLA*, 2012.

[33] P. J. Marrón, A. Lachenmann, D. Minder, J. Hahner, R. Sauter, and K. Rothermel. TinyCubus: A Flexible and Adaptive Framework Sensor Networks. In *EWSN*. IEEE, 2005.

[34] C. M. Medaglia and A. Serbanati. An Overview of Privacy and Security Issues in the Internet of Things. In *The Internet of Things*. Springer, 2010.

[35] S. Munir and J. A. Stankovic. DepSys: Dependency Aware integration of Cyber-Physical Systems for Smart Homes. In *ICCPS*, 2014.

[36] M. W. Newman, A. Elliott, and T. F. Smith. Providing an integrated user experience of networked media, devices, and services through end-user composition. In *Pervasive Computing*. Springer, 2008.

[37] P. Nolan and M. Adair. Untangling The Web Of Liability In The Internet Of Things. http://www.mhc.ie/latest/untangling-the-web-of-liability-in-the-internet-of-things.

[38] Nominet. Nominet IoT Registry. http://nominet.uk.

[39] J. F. Pane, C. Ratanamahatana, B. A. Myers, et al. Studying the language and structure in non-programmers' solutions to programming problems. *IJHCS*, 2001.

[40] C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *ISSTA*, 2011.

[41] M. Pezzè, M. C. Rinard, W. Weimer, and A. Zeller. Self-repairing Programs. In *Dagstuhl Reports*, 2011.

[42] Philips. Hue. http://www.meethue.com.

[43] R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic Generation of Local Repairs for Boolean Programs. In *FMCAD*, 2008.

[44] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *ESEC/FSE*, 2005.

[45] B. Ur, E. McManus, M. P. Y. Ho, and M. L. Littman. Practical Trigger-Action Programming in the Smart Home. In *CHI*. ACM, 2014.

[46] Verizon. State of the Market - The Internet of Things 2015. Technical report, 2015.

[47] W3C. Schema.org. http://schema.org.

[48] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated Fixing of Programs with Contracts. In *ISSTA*, 2010.

[49] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *SenSys*, 2006.