# Back to the future: Forecasting program behavior in automated homes

Jason Croft[1], Ratul Mahajan[2], Matthew Caesar[1] and Madan Musuvathi[2]

[1]University of Illinois at Urbana-Champaign

[2]Microsoft Research

**Abstract—** Networked devices such as locks and thermostats are now cheaply available, which is accelerating the adoption of home automation. But unintended behaviors of programs that control automated homes can pose severe problems, including security risks (e.g., accidental disarming of alarms). We facilitate predictable control of automated homes by letting users "fast forward" their program and observe its possible future behaviors under different sequences of inputs and environmental conditions. A key challenge that we face, which is not addressed by existing program exploration techniques, is to systematically handle time because home automation programs depend intimately on absolute and relative timing of inputs. We develop an approach that models programs as timed automata and incorporates novel mechanisms to enable scalable and comprehensive exploration of their behavior. We implement our approach in a tool called DeLorean and apply it to 10 real home automation programs. We find that it can fast forward these programs 3.6 times to 36K times faster than real time and uncover unintended behaviors in them.

## 1. Introduction

With the advent of cheap networked devices (e.g., remotely controllable locks, lights, thermostats, and motion sensors), automated homes are becoming mainstream [24]. The behavior of such homes is controlled based on a script that is specified by the user. Such scripts have multiple rules, where each rule specifies the actions that the home should conduct in response to certain events (e.g., motion is sensed) or at certain times of the day (e.g., at sunset). The actions include, for instance, changing the state of devices (e.g., turn on lights) and variables, and they may depend on the time of day as well as environmental conditions (e.g., temperature).

Today, many users have great difficulty in predictably controlling their home automation (HA) systems because the control scripts may not behave as they expect. HA forums are alight with a myriad of problems that users face [17, 9, 25], and a similar picture is painted by systematic studies [5, 21]. Living with home automation is full of surprises, where the home behaves in unintended or unexpected ways. These surprises are not only a matter of annoyance but also pose security and monetary risks (e.g., unintended unlocking of the door, improperly activated thermostat). They occur because control scripts can be complex, with complex interactions across rules due to shared variable and device states. The collective behavior of the rules is also hard to verify (e.g., during specification) because it varies with time of day and environmental conditions. Users thus understandably find it hard to correctly reason about all possible behaviors.

The challenge faced by users is illustrated well by what one person told us: *"At one point I had a rule that would turn on the heat, disarm the alarm, turn on some lights, etc. at 8am in the morning on weekdays. What I didn't consider was the fact that I wouldn't want this to happen when I was on vacation. I came home from vacation to find a warm, inviting, insecure, well lit house that had been that way for a week. I didn't realize until then that I needed the morning setup process to only apply when the alarm was set in sleep mode which I set every night before I go to bed. That's just one example, but the point is that it has taken me literally YEARS of these types of mistakes to iron out all the kinks."*

We posit that we can help users in predictably controlling their automated homes by providing a primitive to "fast forward" their homes. Using this primitive, users will be able to observe all possible behaviors, under different times of day and environmental conditions, that the home can exhibit when operating as per a given script. If an unexpected behavior is observed, they can update their script and repeat the process. This exploration should of course be purely virtual; the actual state of the devices in the home should not be impacted.

The primary challenge in developing a practical fast forwarding primitive is correctly handling time. The behavior of an HA system depends heavily on time, not only on exactly when certain events occur but also on the relative timing of events. To explore all possible behaviors, in theory, we need to study all possible events at all possible times. Worse, even that is an ill-defined concept since time is continuous. We describe in §3 why circumventing this issue by discretizing time is unsatisfactory.

Prior work on systematically exploring program behavior, i.e., model checking [14, 23, 18, 27], does not address this challenge and instead abstracts away time. For instance, it assumes that timers of different periods can fire at any time in any order. Similarly, comparisons involving time can nondeterministically return true or false. However, such an imprecise analysis of time is unacceptable for HA systems because, as we show later, it gener-

ates many states that are not reachable in practice.

In this paper, we develop a novel approach that uses timed automata (TA) [2][1] to fast forward programs. A TA is a finite state machine extended with virtual clock variables. These virtual clocks are real valued (i.e., not discrete) and advance at the same rate as wall clock time. A TA transition can specify constraints on the clock variables and is taken only when the constraint is satisfied. For instance, one can specify that a timeout transition happens only when a particular clock variable is greater than a constant. The analyzability of TAs arises from the fact that, under certain conditions on the clock constraints, one can define a finite number of *regions* [2]. All program states within a region are equivalent with respect to the untimed behavior of a system. Thus, "all possible times" can be safely translated to "all possible regions."

While TAs have been extensively used for verifying real-time systems [4, 28], this paper is motivated by the need to analyze executable programs and not abstract models of them. Instead, we explore the timed behavior of a program without the need to first derive its entire TA. This exploration requires the set of temporal constraints that appear in the program, which we extract using symbolic execution [19] of program source code. We also develop new techniques to boost the efficiency of this exploration.

We implement our approach in a tool called DeLorean and use it to explore 10 real home automation programs. We find that we can fast forward these programs at a rate that is 3.6 times to 36K times faster than real time, while maintaining temporal consistency. In two of the programs that we inspected in detail, we found two instances each of what appear to be unintended program behaviors.

While this paper is presented in the context of HA control programs, our approach is general. In future work, we plan to apply it to other kinds of systems where systematically modeling temporal behavior is important, e.g., OpenFlow controllers [7] and security protocols [10].

## 2. Motivation

Studies of home automation (HA) and forum discussions reveal that users have serious difficulties in predictably controlling their automation technology [5, 17, 9, 25, 21]. To understand the kinds of the problems that users face today, it helps to understand the nature of HA systems. As illustrated in Figure 1, an HA system is usually composed of a controller and several devices such as light switches, motion sensors, locks, alarms, etc. The controller receives notifications from the devices (e.g., when motion is sensed), can poll them for their current state (e.g., current temperature), and can send them commands (e.g., turn on the light switch). It uses these capabilities to coordinate the devices based on a control script


Figure 1: An example home automation system

that is provided by the user.[2] In some HA systems, simple aspects of control may be delegated to individual devices (e.g., a light should turn on when a motion sensor broadcasts that motion was detected), but even those systems operate per a centralized control script.

While the scripting languages of different HA systems differ, the control loop of all systems can be viewed as a set of rules. Each rule has a trigger and associated actions. A trigger is either an event in the environment (e.g., sensed motion, toggled light switch) or a firing timer. Actions include setting the state of a device (e.g., turn on the light) or a variable and setting timers. Actions can be conditioned on device state, variable and timer values, and time of the day. They can also include sleep commands.

Figure 2 shows an example script with three rules. Assume that the user wants to turn on the front porch light when motion is detected and it is dark out, and to automatically turn off this light after 5 minutes if it is daytime. Rule 1 is triggered when motion is detected by the front porch motion sensor. It turns on the light if motion is detected twice within 1 second and the light level sensed by a light meter is less than 20. The first condition is a heuristic to help weed out false positives in motion sensing, and the second ensures that light is turned on only when it is dark. Rule 1 also updates the time when motion was last detected. Rule 2 is triggered when the front porch light goes from off to on (either programmatically or through human action). It sets a timer for 5 minutes. Rule 3 is triggered when this timer fires, and it turns off the light if the current time is between 6 AM and 6 PM.

The behavior of HA systems is difficult for users to predict and control because of two factors:

**1. Complex interactions across rules.** Even if users can reason correctly about individual rules, it can still be hard for them to reason about the script as a whole because of complex interactions across rules. These interactions arise because of shared state across rules due to the state of variables and devices. Thus, the program's current behavior depends not only on the current trigger but also on the current state, which in turn is a function of the sequence and timings of rules triggered in the past. This dependence and the number of possible sequences makes it difficult to predict the behavior of the script.

[1]This timed automata is different from the timed automata proposed by Lynch and Vaandrager [22].

[2]By user, we refer to the person that is configuring the HA system, who may be a resident of the home or a hired professional.

```
1   /* Rule 1 */
2   motionFrontPorch.Detected:
3       if (Now - timeLastMotion < 1 secs
4           && lightMeter.LightLevel < 20)
5           FrontPorchLight.Set(On);
6       timeLastMotion = Now;
7
8   /* Rule 2 */
9   frontPorchLight.StateChange:
10      if (frontPorchLightState == On)
11          timerFrontPorchLight.Reset(5 mins);
12
13  /* Rule 3 */
14  timerFrontPorchLight.Fired:
15      if (Now.Hour > 6 AM && Now.Hour < 6 PM)
16          FrontPorchLight.Set(Off);
```

Figure 2: An example home automation script

**2. Verification difficulty.** Another difficulty arises from the fact that the behavior of the system depends on environmental factors (e.g., light level) and time of day. How the program behaves when it is being configured and tested can differ substantially from how it might behave in the future. This dependence makes verifying program behavior difficult because it is hard to emulate all possible combinations of future conditions.

As an example, even the simple script in Figure 2 has a behavior that may not be expected by the user. Suppose the light is turned from off to on at 9:00 PM either due to motion sensing or by the user, triggering Rule 2. Then, the user walks on to the front porch at 9:04:50 PM, which triggers Rule 1. This user might expect the light to stay on for at least 5 minutes, but the light goes off unexpectedly ten seconds later (at 9:05 PM). The fix here is of course to reset the timer in Rule 1, but that may not be apparent to the user until this behavior is encountered in practice.

## 3. Goals and challenges

Our goal is to help users predictably control their homes by providing a "fast forwarding" primitive. Given a starting time and device states, this primitive predicts possible behaviors of a control script, for the specified duration in the future. It systematically explores all possible sequences of triggers and environmental conditions that can yield different behaviors. The exploration is virtual in that the actual state of the devices is not impacted. Its output is the set of unique device states that the home can be in, along with the sequence of events (i.e., triggers, environmental conditions and actions) that lead to that state.

This primitive can help users identify if the home can be in any unintended state (e.g., alarm is armed but the back door is unlocked). In addition to unintended states, the fast forwarding primitive can also help identify unintended behaviors (e.g., light goes off a second after turning on) along any of the explored sequences of events. If an unintended state or behavior is found, the associated sequence of events that led to it can help identify the flaw in the script. At that point, the script can be updated and the process repeated.

In a home with many devices, it may not be feasible to browse through all possible states. In this paper, we assume that users can express state or behavior invariants using the same tools that they use to express the control script.[3] These invariants can correspond either to general safety conditions or to anomalies observed in practice—the former lends confidence in the configuration of the home, and the latter helps diagnose the observed anomalies. Inspection of our HA scripts suggests that users often know the invariants they want to maintain (§7.5). In future work, we will investigate if reasonable invariants can be automatically generated based on the functionality and location of devices in the home.

The difficulty in building a practical fast forwarding primitive for an HA control system is the enormity of the search space of possible sequences of events. This stems from two factors:

**1. Dependence on time.** The behavior of an HA system depends intimately on time, both on the absolute time and the relative timing of triggers. For instance, the behavior of the script in Figure 2 depends on the time of day and on how close in time two motion events fire. For a comprehensive exploration, all possible timings must be considered.

Further, "all possible times" is ill-defined because time is continuous. We could discretize time and assume that events happen only at discrete moments. But picking the granularity of discretization is tricky—if it is too fine, the exploration will have too much overhead as we would be exploring too many event occurrences; if it is too coarse, the exploration will miss event sequences that occur at finer granularity in practice and lead to different behaviors.[4] Without an ability to correctly reason about time, there appears to be no satisfactory way to pick a granularity that works for all events and programs [2].

**2. Dependence of external factors.** The behavior of an HA system depends also on external factors that are not captured in program variables or triggers. These external factors could be values obtained through network services (e.g., querying a Web service for local temperature) or values sensed by devices in the home (e.g., light level). For a comprehensive evaluation, all combinations of values of external factors must be explored. So, if a program depends on external temperature and light level, for every trigger, its response must be explored with all combinations of temperature and light levels.

As mentioned in §1, existing work on exploring program behavior does not address these challenges. We describe our approach next.

[3] Most users, residents or professionals, who configure HA scripts today are fairly technically savvy [5, 21].

[4] We initially tried this discretization-based approach but had to abandon it after we ran into these problems.

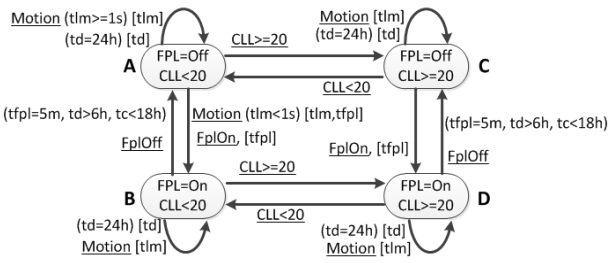


Figure 3: A TA for the program in Figure 2

## 4. Our approach

We leverage timed automata [2] to reason systematically about time and use program source code analysis techniques to scalably explore the impact of external factors. Our approach is not specific to HA systems and can be used to explore the behavior of any control program where correctly modeling time is important. We outline our approach in this section and describe in the next section how it is used in DeLorean to fast forward HA systems. We first provide a brief background on TAs.

### 4.1 Background on timed automata

TAs are finite state machines extended with real-valued virtual clocks (VC). The state of a TA consists of the state of the underlying finite state machine together with the values of all VCs. Thus, a TA contains an unbounded number of states. A TA transition changes the machine state and resets a set of VCs. Each transition specifies a set of clock constraints and is enabled from states that satisfy the constraint.

Figure 3 shows a TA that captures the behavior of the script in Figure 2. There are four states, corresponding to the Cartesian product of whether the front porch light (FPL) is on or off and the current light level (CLL). The TA uses three virtual clocks to capture the time since $i$) the last motion (tlm), $ii$) the light was turned on (tfpl), and $iii$) midnight (td). Transitions are labeled with their triggers (underlined), the clock constraints (in parenthesis), and the clocks that are reset (in brackets). Motion denotes motion, and FplOn and FplOff denote the physical acts of manipulating the light. Some transitions have multiple labels, one for each situation in which the TA can go from the source to the sink state. Transitions that have no triggers are taken as soon as the clock constraints are met.

The key property of TAs that is of interest to us is that they provide a systematic way to explore a TA if VC constraints obey certain conditions. The conditions are that arithmetic operations cannot be performed between two VCs and a VC cannot be involved in a multiplication or division operation. But adding or subtracting constants to VCs is allowed, and so is comparing two VCs (potentially after adding or subtracting constants).

Under these conditions all possible behaviors of the TA can be explored without the need to consider "all possible times." Time can be divided into regions of equivalence such that the exact timing of a trigger within the region does not matter. These regions are defined using constraints on VCs. How these regions emerge can be intuitively understood if one observes that what matters for the TA in Figure 3 is, after a motion event, whether the succeeding motion event occurs before or after 1 sec. The exact timing of the second motion event is not critical.

For space constraints, we do not describe here how these regions can be derived, but we point out their two properties that are relevant to our work. The size of the region is proportional to the greatest common denominator (GCD) of constants used across all clock constraints. Regions get exponentially smaller as VCs are added to the TA, because each VC imposes additional constraints on what can fall within a region.

Once the regions are known, fully exploring the TA's behavior requires 1) exploring all possible transitions, in response to all possible triggers, from the current state that can be taken given VC constraints; and 2) exploring exactly one *delay transition* in which there is no state transition but all the VCs advance by the same amount. This amount is such that the time progresses to the immediately succeeding region.

### 4.2 Exploring program behavior using TA

We are given a control program and its starting state as input, and our goal is to fast forward it for a given duration. The output of this process is all possible states that the program can be in after this duration.

We assume that the program can be modeled as a TA. In such modeling, all timers and variables that store time are, in effect, VCs. To leverage time regions, these VCs must satisfy the conditions mentioned above. We believe that these conditions are met in many contexts. They are certainly met in the eight different kinds of HA systems that we study in §7. The scripting languages of these systems cannot even express complex clock operations.

If we were given or if we could derive the entire TA corresponding to the program, we could explore its behavior using existing methods [28, 4]. What we have instead is a program. The TA corresponding to all but the smallest of control programs will be extremely large as it needs to capture the program logic and its response to possible events and environmental conditions.

Thus, we explore the TA dynamically (akin to how FSA-based model checkers dynamically explore the FSA instead of deriving the complete FSA of the program). Beginning from the starting program state, we repeatedly derive successor states that result from triggers or delay transitions. For delay transitions, we need to know the timed regions in advance, so we can compute the delay amount. Fortunately, constructing these regions does not require the complete TA; we only need the constraints on

```
1: EndWC=Time.Now + FFduration;          ▷ How long to explore
2: S0.WC = Time.Now;                     ▷ Set the wall clock
3: ES = {};                              ▷ explored states
4: US={S0};                              ▷ unexplored states
5: while US ≠ φ do
6:     Si = US.pop();
7:     ES.push(Si);
8:     for all e in Events, Si.EnTimers do
9:         for all env in Environments do
10:            So = Compute(Si, e, env);
11:            if !Similar(So, (US ∪ ES)) then
12:                US.push(So);
13:            end if
14:        end for
15:    end for
16:    if Si.EnTimers = φ then
17:        delay = DelayForNextRegion(Si.Region);
18:        if Si.WC + delay > EndWC then
19:            continue;
20:        end if
21:        So = Si.AdvanceAllVCs(delay);
22:        for all timer in So.Timers do
23:            if timer.dueTime >= So.WC then
24:                So.EnTimers.Push(timer);
25:            end if
26:        end for
27:        if !Similar(So, (US ∪ ES)) then
28:            US.push((So, t));
29:        end if
30:    end if
31: end while
```

Figure 4: Pseudocode for basic TA exploration.

the values of virtual clocks [2]. We extract these constraints using analysis of program source.

Figure 4 shows how we comprehensively explore program behavior. Assume that we want to fast forward for FFDuration and the starting state of the program is $S_0$. Program state includes the values of its (non-time) variables, VCs, and timers that are enabled (i.e., ready to fire). We do a breadth-first exploration using a queue of unexplored states. Obtaining all successors of a state entails firing all possible events and all enabled timers, under all possible combinations of values of environmental factors (Environments). If a successor state is not similar to any that had been seen before, we add it to unexplored states. Two states are similar if their variable values and set of enabled timers are identical and if their VC values map to the same time region; it is not necessary that the VC values be identical since the exact time within a region does not matter.

If the state being explored has no enabled timer, it is eligible for a delay transition that represents a period of time where nothing happens but time advances to the succeeding region; states with enabled timers need to fire those timers before time can progress. We ignore the successor if this delay takes us past EndTime. Otherwise, the successor state is computed by advancing all VCs. We treat wall clock time, which is virtualized during exploration, as any other VC except that it never resets; it tracks the progress of absolute time. We then check if any of the timers have been enabled because of this delay and mark them as such. The construction of time regions guarantees that no timers are skipped during the delay transition.

We make a few observations about the exploration above. First, it assumes that the environment is highly dynamic—the value of each factor can vary over its entire range at any given time and two proximate (in time) values can differ arbitrarily. In general, however, an environmental factor (e.g., temperature) may vary within a smaller range at a given time of day and it may also be slow moving such that two proximate values differ only by a small amount. It is straightforward to limit the exploration to more realistic models of environmental conditions. Building such models (e.g., through historical data) is a subject of future work.

Second, the exploration assumes that the processing of a trigger is instantaneous and deterministic, which is common in HA systems since actions in a rule are simple. If the system is non-deterministic, we can account for that in exploration just like existing model checkers by exploring the different executions that can transpire. Another potential source of non-determinism is race conditions between simultaneous triggers, but we already handle that since we explore all possible trigger sequences.

#### *4.2.1 Predicting successor states*

The basic TA-based exploration above correctly handles time but is too slow to be practical. We use three techniques to make it practical. Our first technique reduces the time to obtain successor states of a state being explored. To explain it, we first define the notion of *clock personality*. A program state's clock personality w.r.t. a clock constraint is 0 or 1, depending on whether its VC values satisfy the constraint. Its clock personality w.r.t. a set of constraints captures its personality w.r.t. individual constraints. Two sets of VC values can have identical personalities even if they are not in the same region.

Consider two non-similar program states, $S1$ and $S2$ with identical variable values, enabled timers, and clock personalities w.r.t. to clock constraints in the program. In this case, we observe that the behavior of $S2$ to a stimulus (i.e., the combination of trigger and environmental conditions), will be identical to that of $S1$. This observation enables us to predict the successors of $S2$, instead of computing them, if we already obtained the successors of $S1$. The predicted successor of $S2$ for a given stimulus has the same variable values and enabled times as the corresponding successor of $S1$. It, however, does not inherit the VC values of $S1$'s successor. It retains its own VC values, except that all virtual clocks that were reset for $S1$ in response to the stimulus are reset.

Successor prediction is much faster than successor computation. Prediction requires a copy of state and modifying its VC values, which we can make even more efficient through appropriate data structures (§5). Computa-

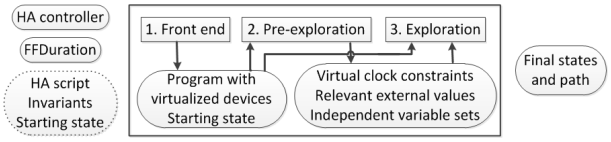


Figure 5: Overview of DeLorean

tion requires deserializing the parent's state, running the program and subjecting it to the stimulus, and then serializing the successor's state. These are costly operations.

#### 4.2.2 *Cutting down on environmental factors*

The following observation significantly reduces the number of environmental factors that must be explored. The values of environmental factors can vary over a wide range; in theory, each possible value should be explored as program behavior can differ for different values. But in practice, not every single value leads to a different outcome. A range of values can have an identical impact. For instance, the light level in Figure 2 can vary from 0 to 99; but instead of behaving differently for each possible value, what really matters is whether the light level is from 0-19 or 20-99. Using one value in each range is sufficient to explore all possible behaviors.

A special case of this observation is that some environmental factors may not matter at all for processing a trigger. For instance, light level does not matter for Rules 2 and 3 in Figure 2. We thus do not need to explore this factor at all when exploring their triggers.

We use symbolic execution [19] of the program source code, in which we process each rule to recover the combinations of values of external factors that lead to different outcomes when processing a rule. More details are presented in the next section.

#### 4.2.3 *Independent control loops*

Our third optimization is based on the observation that large control programs may often be composed of multiple, independent control loops that manipulate different parts of the program state. For instance, in the HA context, thermostats and furnaces may be controlled by a climate control loop, and locks and alarms may be controlled by a security control loop, and these two may manipulate different variables and clocks. In such cases, we can explore the loops independently, instead of exploring them jointly. Separate exploration is faster because joint exploration considers the Cartesian product of the values of independent variables and clocks. We use taint tracking to identify independent loops.

## 5. System design

We now describe the design of DeLorean which uses the approach above in the context of HA systems. The primary input that the user provides to DeLorean is the duration for which the system should be fast forwarded

| | Notifier | Non-notifier |
|---|---|---|
| Event sensor | Trigger | - |
| Value sensor | Trigger, prog. variable | Env. factor |
| Actor | Trigger, prog. variable | Env. factor |

Table 1: Device modeling in DeLorean.

(FFDuration). The user can also specify three optional inputs. The first is an alternative HA script whose behavior should be explored. By default, we extract and use the script currently running at the controller. The second is a list of invariants on device states and system behavior, which should be satisfied at all times. They are specified in a manner similar to the $if$ conditions in the rules. The third is the wall clock time and starting state of (a subset of) devices and variables from which exploration should begin. By default, current wall clock time and device states, extracted from the controller, are used.

The output of DeLorean is all the unique states that the devices can be in after FFDuration. If invariants are specified and are violated by any state during the exploration, we also output that state. In addition, DeLorean outputs the path that leads to each state, where the path is the timestamped sequence of triggers, along with the values of environmental factors during those firings.

As Figure 5 shows, the operation of DeLorean has three stages. First, the front end converts the HA script to a program in which devices and clocks have been virtualized. Second, pre-exploration analyzes this program to recover information required for the optimizations mentioned above. The final stage is exploration itself. While the front end is specific to the type of the HA system and its scripting language, the other two stages are completely independent. In fact, these stages are not even HA-specific and can be applied to any control program in which interactions with external world have been virtualized. We now describe each stage in more detail.

### 5.1 Stage 1: Front end

The main goal of the front end is to transform the HA script to a program in which devices have been virtualized and time-related activities have been expressed using VCs. The resulting program does not need access to the physical devices and can be run on any computer (e.g., in the cloud). The behavior of this program is otherwise identical to the HA script that runs on the HA controller with access to real devices.

#### 5.1.1 *Virtualizing devices*

When modeling devices in the program, our goal is to model their externally observable state and actions, not their internal behavior which can be arbitrarily complex. We model devices using one or more variables or environmental factors in the program. First consider simple devices such as light switches and motion sensors whose current state can be represented using one value. The

operation of such devices can be classified as: $i$) *event sensors* which sense events in the environment such as motion or smoke; $ii$) *value sensors* which sense environmental conditions such as temperature or light level; and $iii$) *actors* such as light switches and locks which do not sense the environment but their state can be changed programmatically or by a human. Devices can also be classified as those that send notifications when their state changes and those that need to be polled by the controller when their current value is needed. Event sensors always send notifications when events are detected.

Table 1 summarizes how we model each case. The simplest case is that of event sensors, which can appear only as a trigger in a rule. We do not need to do anything special for an event sensor. If its trigger does not already exist in the script, the script must be ignoring its events and we can ignore it as well. If the trigger exists, it becomes one of the triggers in the transformed program. For value sensors and actors that do not send notifications, we treat attempts to query their value as a query for an environmental factor. This query returns values in the valid range for that device which depends on its type (e.g., a dimmer varies between 0–99).

For value sensors or actors that send notifications, HA controllers keep an up-to-date view of its value which is used when the device state is queried. The value update process at the controller is automatic and not explicit in the script. To model it, we introduce an additional rule and program variable. The rule's trigger is a notification from the device, and its action updates the variable's value to the notified value. If a rule with the same trigger already exists in script, we merge the two rules by prepending the variable-update action to the existing rule's actions. Queries for device state read this variable.

More complex devices such as a thermostat, which has an actor functionality for target temperature and a value sensor functionality for current temperature, are modeled as multiple single-valued devices. We currently assume that the values of a device's sub-functions are independent, which works well for the devices that we have encountered in our work, but may not work for all devices.

A related limitation of our modeling is that we assume that there are no dependencies across device values in the home. But in practice, for instance, the value sensed by a light meter may be impacted by the state of a nearby light. In the future, we will extend our device models to include relationships between the multiple values of a given device and allow users to provide models of their home that capture relationships across devices.

Finally, our modeling strategy cannot model extensible devices such as smartphones and PCs. But such devices are incorporated in current HA systems in a limited manner, as a remote control for other devices. They activities they can trigger can also be triggered by humans. Our exploration already considers such triggers.

#### *5.1.2 Introducing virtual clocks*

HA scripts do not contain explicit references to VCs, but as mentioned previously, all time-related activities in effect manipulate VCs. We make these references explicit so time is properly handled in later stages.

There are three kinds of activities in HA scripts that are time related. The first is measuring the gap between two events of interest (e.g., consecutive motion events). Here, a variable (e.g., timeLastMotion in Figure 2) is used to store the time of the first event, which is then subtracted from the wall clock time of the second event to obtain the gap. To capture this activity using a VC, we set the VC to zero when the first event occurs; the value of the VC when the second event occurs yields the delay, since VCs progress at the same rate as the wall clock unless reset.

The second time-related activity is a timer (e.g., timerFrontPorchLight in Figure 2). To capture this activity using a VC, we reset the VC when the timer is set, and queue a timer trigger to fire after the desired delay, after removing any previously queued event.

The third time-related activity is a sleep call, in which some actions for a rule are taken after a delay (e.g., turn on fan, sleep 30 secs, turn it off). We capture this activity by introducing a new timer and a rule. The actions of the new rules correspond to post-sleep actions of the original rule. The sleep and post-sleep actions in the original rule are replaced by a timer that fires after the desired delay. Multiple sleep calls in a rule are handled similarly, by splitting across rules and using timers to link the rules. In our treatment of sleep calls, if the trigger for the original rule occurs again before the timer set by an earlier occurrence fires, the post-sleep actions that correspond to the earlier trigger will not be carried out (because the earlier timer event will be dequeued). This behavior is consistent with the semantics of HA systems.

**Reducing the number of clocks** The number of VCs in the program has a significant impact on exploration efficiency because the size of regions shrinks exponentially with it. When converting an HA script into a program, we should introduce the minimum number of VCs. We exploit two opportunities. First, consider cases where the actions in a rule have multiple sleeps, e.g., action1; sleep(5); action2; sleep(10); action3. Instead of using two timers (one per sleep), we can use only one because the two sleeps can never be active at the same time [11]. To retain the original dynamic behavior, we introduce a new program variable to track which actions should be taken when the timer fires. In the example above, when the rule is triggered, after action1 is taken, this variable is reset to 0 and the timer is set to fire after 5 seconds. When the timer fires: $i$) if the variable value is 0, action 2 is taken, the variable is set to 1, and the timer is set to fire after 10 seconds; $ii$) if the variable value is 1, action 3 is taken.

Second, HA scripts often have daily activities for several different times of the day (e.g., sunrise, an hour after sunrise, sunset, midnight, etc.). The straightforward way to translate such a script is to introduce a timer per unique time. But a more efficient way is to use just one timer to conduct all such activities, using a method similar to the above — introduce an additional program variable to cycle through the activities of the day. The variable's value is reset after the last activity is conducted.

### 5.2 Stage 2: Pre-exploration

This stage analyzes the program produced by the front end to recover the information needed for constructing timed regions and implementing the optimizations of §4.

It primarily uses symbolic execution [19] of program source. Symbolic execution simulates the execution of code using a symbolic value $\sigma_x$ to represent the value of each variable $x$. As the symbolic executor runs, it updates the symbolic store that maintains information about program variables. For example, after the assignment $y$=$2x$ the symbolic executor does not know the exact value of $y$ but has learned that $\sigma_y$=$2\sigma_x$. At branches, symbolic execution uses a constraint solver to determine the value of the guard expression, given the information in the symbolic store. The symbolic executor only explores the branch corresponding to the guard's value as returned by the constraint solver, ensuring that infeasible paths are ignored. If there is insufficient information to determine the guard's value, both branches are explored. In this way, a tree of all possible program execution paths is produced. Each path is summarized by a path condition that is the conjunction of branch choices made to go down that path.

We symbolically execute the program's main control loop (EventFired) which is the starting point for all processing activity. We configure the symbolic execution to treat the following entities as symbolic: program state (variables and clocks), the return value of each external function call, and the two parameters of EventFired. There is a separate symbol for each external function call instance even if the calls are to the same function; two calls to the same function can return different values.

The output of the symbolic executor is the set of possible paths for each possible trigger. In general, symbolic execution may not be able to cover all paths, but the code to process a trigger in HA programs tends to be simple. We find that all paths are covered for the programs we study in §7. If full coverage is not reached for a trigger, our exploration may miss some behaviors.

For each path, we obtain the $i$) constraints that must hold for the program to traverse that path, and $ii$) the program state that results after its traversal. The constraints and the resulting program state are in terms of input symbols, the entities we made symbolic in the configuration.

We can now recover the following information.

**Virtual clock constraints.** These are required for constructing time regions and for predicting successor states. We obtain them from the output of symbolic execution by taking the union of constraints on VCs along each path. Additionally, program statements that reset a timer $x$ to $k$ secs are essentially clock constraints of the form $x \geq k$. We extract such statements from the program source and add corresponding constraints to the set.

**Relevant values of environmental factors.** We need to identify which values of an environmental factor lead to different outcomes when processing a rule. For this, we look at the path constraints and resulting program states corresponding to the rule. If the input symbol for a factor is missing from all constraints and resulting program states, it is completely irrelevant for exploring the rule.

For factors that are relevant, constraints on the input symbols along each path reflect the ranges of values for each factor that take the program down the path. We can use any example values that satisfy the constraint, which we obtain using a constraint solver.

However, there is a distinction between traversing the same path and resulting in the same state. A program such as var=CurrTemp() traverses the same path for all values of CurrTemp(), but it produces different resulting states. In the presence of such data dependencies (as opposed to only control dependency), we must explore all values that satisfy the path constraint. We have not encountered such dependencies in our programs.[5]

**Independent control loops.** We can use the output of symbolic execution for taint tracking as well. We analyze the program state that results along each path. If the final value of a variable along any path is different from its (symbolic) input value, that variable is impacted along the path. This impact depends on the input symbols that appear in the output value (data dependency) and path constraints (control dependency). The variables corresponding to those input symbols are tainting the variable.

This taint information is used to identify independent sets of variables and VCs. Two variables or VCs are deemed dependent if either they taint each other in the program, or they occur together in a user-supplied invariant (as we must do a joint exploration in this case as well). After determining pairwise dependence, we compute the independent sets that cover all variables and VCs.

### 5.3 Stage 3: Exploration

This stage implements the method outlined in §4. It starts by running the program and initializing it to the starting state. We then take the checkpoint of the pro-

[5]The HA scripting languages that we are aware of do not allow the expression of such data dependencies. To "remember" the value of an external factor, one can create a control dependency. Instead of var=CurrTemp, we can use if (CurrTemp() $<$ 20) var=1; elsif (CurrTemp() $<$ 40) var=2; ..... This restores one-to-one mapping between paths and states.

gram, which involves serializing its internal state. The checkpoint captures the values of all variables in the program, including time related variables, and the times when various timers will fire. As mentioned previously, we also virtualize the wall clock time during exploration, with its initial value set to the desired start time.

For efficiently operating on program state, we maintain several pieces of additional information for each state. These include three different kinds of hashes: $i$) of their variable values and enabled timers; $ii$) of the time region the VC values fall in, which is a combination of the hashes on clock constraints that define the region; $iii$) of the clock personality of the VC values.

The combination of the first two hashes lets us quickly determine if two states are similar; we cannot do a direct comparison of checkpoints, and restoring and peeking inside the checkpoint for similarity checks is costly. The combination of the first and third hash lets us quickly determine if a state's successor can be predicted using another, already explored state.

We also maintain separately a table that contains the values of the VCs of a state. Many states differ only in terms of their VC values—the successor state after a delay transition differs from the parent only in terms of VC values, so does the successor that is predicted from another state. Maintaining this table separately lets us quickly obtain these successor states.

Maintaining this table also helps us reduce the memory footprint. Two states that differ only in their VC values can share the checkpoint, while having separate tables. However, this implies that the VC values in a table can be out of sync with those embedded in the checkpoint. Thus, when restoring a state, we update its VC values from the table, immediately after deserialization and before any other processing begins.

Finally, to quickly compute the amount of delay needed to advance to the successor region, we use the abstraction technique proposed by Clarke et al. [8]. In this technique the VC values are abstracted in terms of their integral and fractional components. One difference in our implementation, motivated by the need to maintain a precise virtualization of the wall clock time, is that we do not cut off the fractional components.

## 6. Implementation

We implemented DeLorean in C#. We developed front end modules for two HA systems—ISY [16] and ELK [12]. We chose these two because of their popularity. Of all the HA systems that we study in §7, ISY has the most expressive scripting language. For both systems, we parse scripts with a parser that was developed using ANTLR [3]. The parser produces a C# program that captures the behavior of the script and contains additional variables, rules, and actions needed for modeling devices.

| | type | #rules | #devs | SLoC | #VCs | GCD (s) | #trans |
|---|---|---|---|---|---|---|---|
| P1 | OmniPro | 6 | 3 | 59 | 2 | 7200 | 72 |
| P2 | Elk | 3 | 3 | 75 | 2 | 1800 | 123 |
| P3 | MiCasaVerde | 6 | 29 | 143 | 2 | 300 | 178 |
| P4 | Elk | 13 | 20 | 193 | 5 | 5 | 19.7M |
| P5 | ActiveHome | 35 | 6 | 216 | 14 | 5 | 78.7K |
| P6 | mControl | 10 | 19 | 221 | 4 | 5 | 51K |
| P7 | OmniIIe | 15 | 27 | 277 | 6 | 60 | 3.6M |
| P8 | HomeSeer | 21 | 28 | 393 | 10 | 2 | 8.1M |
| P9 | ISY | 25 | 51 | 462 | 6 | 60 | 121M |
| P10 | ISY | 90 | 39 | 867 | 6 | 10 | 256M |

Table 2: The HA programs we study

We use Pex [26] to symbolically execute the main event loop of this C# program. Pex is a modern symbolic execution engine that mixes concrete and symbolic execution ("concolic" execution) to boost path coverage and efficiency. The bulk of our code implements the exploration stage, and it was developed from scratch. We could not use one of the existing tools for exploring TAs [28, 4] because we do not have the complete TA for the program.

## 7. Evaluation

This section evaluates DeLorean. We show its performance, the benefit of its prediction-based optimization, compare it to alternative approaches, and finally show that it can help find unintended behaviors.

### 7.1 Dataset

We evaluate DeLorean using real HA scripts. We solicited these scripts on a mailing list for HA enthusiasts. We picked the 10 scripts shown in Table 2. We selected them for the diversity of HA systems and the number of rules and devices. We see that most installations have tens of rules and devices, with the maximums being 90 and 51. This points to the challenge users face today in predictably controlling their homes. Collectively, these installations had 19 different types of devices, including motion sensors (an event sensor), temperature sensors (a value sensor), sprinklers (an actor), and thermostats (a multi-value device).

The table shows the source lines of code (SLoC) and the number of VCs in the program obtained after transformation in the first stage. Systems for which we have not implemented a front end yet were transformed manually. We see that most installations have 5 or more VCs, indicating a heavy reliance on time.

The table also shows the GCD (greatest common denominator) across all constants in VC constraints in the program. The GCD can be coarsely thought of as the detail with which the program observes the passage of time. Since the size of the regions depends on it, it also heavily influences the exploration time.

### 7.2 Fast forwarding performance

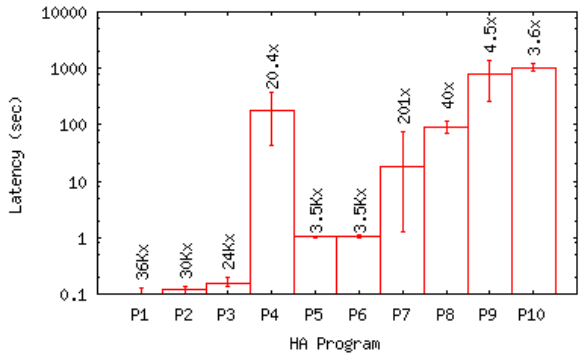


Figure 6: Latency of fast forwarding for an hour.

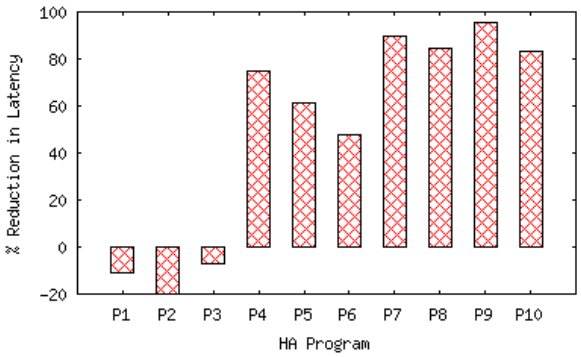


Figure 7: Benefit of predicting successor states.

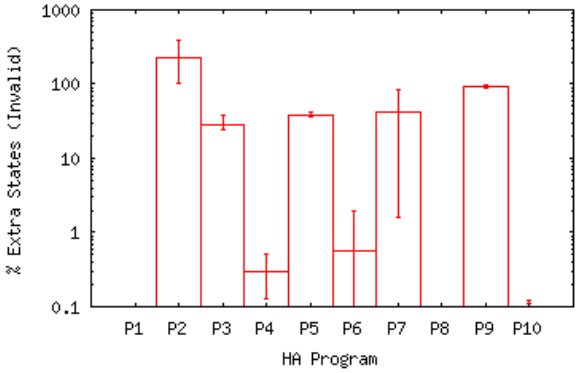


Figure 8: Invalid program states generated by untimed exploration.

The time it takes to fast forward an HA program is important because it reflects the amount of time the user will have to wait to determine if the input script behaves as intended. To study this performance, we run DeLorean over all 10 programs. We conduct 20 trials, each with randomly selected starting state and time (since program behavior depends on both). All experiments use an 8 Core 2.5Ghz Intel Xeon PC with 16GB RAM.

Figure 6 shows the time in seconds it takes to fast forward one hour. The error bars correspond to minimum and maximum time observed across trials, and the labels above the bars show the fast forwarding rate (i.e., one hour divided by the time taken).

We see that DeLorean can fast forward real programs by 3.6 times to 36K times faster than wall-clock time. A majority of the programs can be fast forwarded in under a minute, which can enable a fast explore-debug cycle.

We also see that the fast forwarding rate is roughly but not strictly dependent on LoC in the program. P4 has the second slowest rate, even though its LoC is fourth smallest. Here, the reason is that the fast forwarding speed depends on the biggest control loop in the program, and in P4's case this loop is bigger (more variables and clocks) than those of P5-P7.

To provide a sense of the complexity of fast forwarding, the last column in Table 2 shows the number of transitions that we needed to fast forward for an hour. Dividing by the latency of fast forwarding, we can estimate that DeLorean makes 200K transitions per second. As we will show below, being able to predict successor states is key to supporting this high rate.

### 7.3 Benefit of predicting successor states

Predicting successor states is a general optimization for dynamically exploring TAs. Its effectiveness, however, depends on how often we encounter non-similar states with identical clock personalities, variables, and enables timers. To evaluate it, Figure 7 plots the percentage reduction in latency when prediction is used compared to when it is not used. We see that for the smallest of our programs, prediction leads to slower exploration. This is because in such cases the overhead associated with checking for past states that can be used for prediction is greater than any benefit it brings.

However, for larger programs, prediction brings substantial benefit. For P9, prediction helps cut the exploration time by 95%. That is, exploring without prediction is slower by a factor of 20.

### 7.4 Comparison with alternatives

We now compare our TA-based systematic exploration to two other alternatives—untimed exploration and randomized testing. We show that the former generates many invalid states, and the latter misses many valid states.

#### *7.4.1 Untimed exploration*

As mentioned earlier, current model checkers ignore time and can thus generate invalid program states that will not be generated in real executions. If there were just a few invalid states, it is conceivable that users would be willing to put up with occasional incorrectness. However, we find that untimed exploration results in many incorrect states. Figure 8 shows the percentage of additional, invalid states produced by untimed model checking,[6] when

[6]This comparison based on invalid states alone hides one addi-

beginning from the same starting state as DeLorean and running until it cannot find any new states. Untimed exploration differs from DeLorean in three aspects: $i$) in addition to successors based on device notifications, each state has successors based on each queued timer, independent of the target time of the timer; $ii$) if a comparison to time is encountered during exploration both true and false possibilities are considered; $iii$) there are no delay transitions. The graph averages results over 10 paired trials with different starting inputs, and the error bars shows maximum and minimum percentage of invalid states.

We see that untimed exploration produces a significant number of invalid states. For most programs, the number of invalid states is of the same order as the number of valid states produced by DeLorean.

Closer inspection of results from untimed exploration provides insight into how some invalid states are produced. One common case is where devices such as lights are programmed to turn on in the evening, using a timer. Because timers can fire anytime, untimed model checking incorrectly predicts that the light can be off in the evening, which will not happen in practice. Another case is where certain actions are meant to occur in a sequence, e.g., open the garage door after key press and then close it 5 minutes later. With DeLorean, these actions are carried out in the right sequence, correctly predicting that the door is left in the closed state. But both possible sequences are explored by untimed exploration, one which incorrectly predicts that the garage door is left open.

These results confirm the limitation of using untimed exploration for HA systems. Untimed exploration can produce many false positives, that is, unintended states that will not arise in practice. Users will likely find it hard to separate these false positives from any true positives.

As an aside, comparing the number of needed transitions in timed and untimed exploration provides a gauge for the additional complexity that systematically handling time brings. Across all programs, we find that untimed exploration needs 82% fewer transitions compared to fast forwarding for an hour.

#### 7.4.2 *Randomized testing*

Another alternative to our approach is randomized testing in which the program is subjected to random sequences of triggers to uncover unintended states and behaviors. Unlike untimed exploration, such testing maintains temporal correctness but it does not systematically explore program behavior.

To evaluate the benefit of systematic exploration, we subjected our HA programs to randomized testing. We generated a random sequence as follows. At each step, we randomly choose an input from the set of delay plus device notifications. If a device notification is chosen, the corresponding rule in the program is processed. If there are environmental factors in the rule's actions, we pick their values randomly. If delay is chosen as the input, the amount of delay is randomly chosen between 0 and the GCD of the constants in clock constraints (to match the granularity of timed regions, which approximates the time granularity of program's operation). We then advance all VCs by this amount. If a timer becomes ready to fire along the way, we process its corresponding rule. We repeat such steps until the virtual wall clock time advances by an hour.

We generate and test the program with multiple sequences. For a fair comparison, we hold constant the CPU time taken for exploration—we conduct randomized testing for the amount of real time it takes for DeLorean to explore an hour from the same starting input.

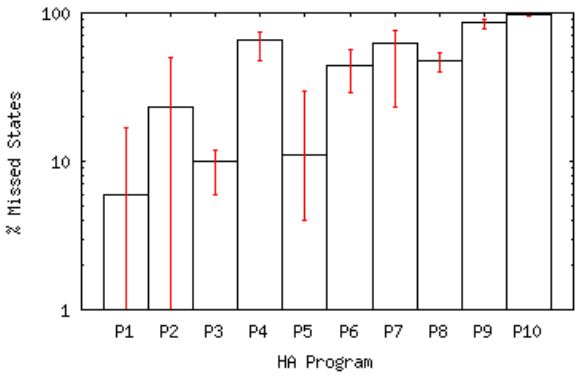


Figure 9: States missed by randomized testing.

Figure 9 plots the percentage of states that are reached by DeLorean but missed by randomized testing. The results are aggregated over 10 paired trials with different starting states and times, and the error bars denote the maximum and minimum. We see that a significant fraction of states can be missed with randomized testing of HA programs, which points to the need for systematic exploration to uncover unintended states and behaviors.

### 7.5 Unintended behaviors

Since we are not aware of the user intent that underlies our HA programs, we are unable to systematically examine how often unintended states or behaviors are found by DeLorean. But to informally gauge the ability of DeLorean to find unintended behaviors, we conduct an informal analysis based on comments on rules in the scripts. The comments can sometimes suggest invariants that the user wants to maintain.

We inspected comments in two of the scripts (P9, P10), and turned them into invariants for which DeLorean should report violations. We found four violations.

**P9-1** A comment indicated that the lights in the back

tional limitation of untimed model checking. Untimed exploration is incapable of verifying program behaviors that depend on time (e.g., light turned off a second after turning on).

of the house should turn on if motion is detected in the evening, defined as the period from sunset to 11:35PM. But DeLorean found that the lights could be on even if there was no motion. When we inspected the triggers that produced this behavior, we found a rule that appears misprogrammed. Instead of using conjunction as the condition to turn on the light (sunset $<$ Now $<$ 11:35PM && MotionDetected), it was using disjunction (sunset $<$ Now $<$ 11:35PM || MotionDetected)

**P9-2** A comment indicated that front porch light should stay on from half hour after sunset until 2AM. There were two rules to implement this invariant, one that turned the light on at a half hour after sunset and one that turned it off at 2AM. But DeLorean found cases where the light was off in that time window. Inspection revealed that there was another rule in the script to turn off the light at 7:45PM. Thus, the invariant is violated if sunset occurs after 7:15PM (which does happen where the user of P9 resides). In HA scripts the time of the sunset is either configured statically, or fetched from a Web service. This scripts falls in the latter category, and we model sunset as an external factor and explored its entire valid range. Exploring values higher than 7:15PM uncovered the violation. An additional comment in the script suggests that the 7:45PM rule was meant to exist temporarily but the user forgot to remove it.

**P10-1** A comment indicated that the user wanted to turn on the dimmer switch in the master bath room when motion is detected. But we found instances where the motion occurred but the dimmer was not on. Inspection revealed that the user's detailed intent, implemented using two rules, was to turn on the dimmer half-way when motion occurs during the day, and to turn it on fully when its detected during night. But the way day and night time periods were defined left a 2 minute gap where nothing would happen in response to motion.

**P10-2** A comment indicated that the user wanted to treat three devices identically, that is, they should be either all on or all off. Inspection of a violation of this invariant showed that while three of the four rules that involved these devices correctly manipulated them as a group, one rule had left out one of the devices.

These findings suggest that DeLorean can indeed help users find unintended behaviors in their scripts. We also note that during this exercise, we found several commented out rules and comments that suggest users often "test drive" their rules to verify their behavior. DeLorean can help make this test drive systematic, by verifying behavior under a wider variety of conditions (time of day and environmental factors).

## 8. Related work

Our work builds on the progress that the research community has made towards verifying the behavior of real systems. Three threads of work are particularly relevant.

**Model checking programs** One class of techniques is model checking in which programs are modeled as an FSA and their behavior is comprehensively explored [14, 23, 18]. Recent work, like us, also combines model checking with symbolic execution [27, 6]. However, most model checking work ignores time. It assumes that timers can fire any time after they are set and all time comparisons can yield true or false with equal probability. This approach works well for programs that have a weak dependence of time, but the behavior of control programs that we study is intricately linked with time. Ignoring time in such programs can lead to exploring infeasible executions, and it cannot discover unexpected behaviors in which the mismatch is the time gap between events.

One exception is NICE, which studies OpenFlow applications whose behavior can vary considerably based on packet timings [7]. However, its treatment of time is not systematic and is domain-specific (e.g., injects packets without any delay and with usually high delay). Our work shows how time can be accounted for in model checking systematically and in a general manner.

**Model checking using TA** There has been much work on TA-based model checking in the real-time systems community. It includes developing efficient tools to explore the TA [4, 28] as well as transformations that speed explorations [15, 11]. This body of work assumes that the entire TA is known in advance, and it does not target program analysis. While we draw heavily on the insights from it, to our knowledge, our work is the first to use TA to model check programs. We describe general methods to dynamically and comprehensively explore program executions and techniques to optimize exploration.

**Other debugging techniques** Explicit state model checking, which we use in our work, is complementary to other program debugging approaches. For instance, record and replay [20] can help diagnose faults after-the-fact and is especially useful for non-deterministic systems; in contrast, we want to determine if faults can arise *in the future*. There has also been work on "what-if" analysis in IP networks, e.g., with the use of shadow configurations [1] and route prediction [13]. These works largely focus on computing the outcomes of configuration changes; in contract, we want to study the dynamic behavior of more general programs.

## 9. Conclusions

Mistakes in automation scripts can impact the comfort, safety, and efficiency of the home environment. We built DeLorean, a tool that can help users gain confidence in their script by virtually fast forwarding it to see its future behaviors under different times of day and environmental conditions. DeLorean is based on a new approach to systematically explore program behavior. It uses timed

automata, instead of finite state machines, as the basis of exploration, and it can thus correctly handle time. While developed in the context of home automation, this approach is general. In future work, we will use it to explore programs in other contexts where correctly modeling time is important [7, 10].

## 10. References


[1] R. Alimi, Y. Wang, and Y. Yang. Shadow configuration as a network management primitive. *ACM SIGCOMM*, August 2008.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 1994.

[3] ANTLR parser generator. `http://antlr.org/`.

[4] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL: A tool suite for automatic verification of real-time systems. *Hybrid Systems III*, 1996.

[5] A. J. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon. Home automation in the wild: Challenges and opportunities. In *CHI*, 2011.

[6] M. Canini, V. Jovanovic, D. Venzano, B. Spasojevic, O. Crameri, and D. Kostic. Toward online testing of federated and heterogeneous distributed systems. In *USENIX ATC*, 2011.

[7] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[8] E. M. Clarke, F. Lerda, and M. Talupur. An abstraction technique for real-time verification, 2007.

[9] Home automation news, reviews, forums. `http://cocoontech.com/forums/`.

[10] R. Corin, S. Etalle, P. H. Hartel, and A. Mader. Timed analysis of security protocols. *Computer Security*, 15(6), 2007.

[11] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Real-Time Systems Symposium*, 1996.

[12] ELK products, inc. `http://www.elkproducts.com/`.

[13] N. Feamster and J. Rexford. Network-wide prediction of BGP routes. *IEEE/ACM Trans. Networking*, April 2007.

[14] P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, 1997.

[15] M. Hendriks and K. G. Larsen. Exact acceleration of real-time model checking. In *Electronic Notes in Theoretical Computer Science*, 2002.

[16] Universal devices products/insteon/isy-99i series. `http://www.universal-devices.com/99i.htm`.

[17] Forum - questions and answers. `http://forum.universal-devices.com/viewforum.php?f=27`.

[18] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.

[19] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.

[20] T. Leblanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36, 1987.

[21] S. Lucero and S. Schatt. Home automation and security. ABI Research, 2009.

[22] N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. *Rex Workshop*, 1991.

[23] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A programatic approach to model checking real code. In *OSDI*, 2002.

[24] Home controls – analysis and forecasts. Park Associates, 2010.

[25] SmartHomeUsa.com home automation forums. `http://forums.smarthomeusa.com/`.

[26] N. Tillmann and J. de Halleux. Pex: White box test generation for .NET. In *Tests and Proofs (TAP)*, April 2008.

[27] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, 2009.

[28] S. Yovine. Kronos: A verification tool for real-time systems. *Int'l Journal of Software Tools for Technology Transfer*, 1997.