Systematically Exploring the Behavior of Control Programs

Jason Croft, *UIUC*
Ratul Mahajan, *Microsoft Research*
Matthew Caesar, *UIUC*
Madan Musuvathi, *Microsoft Research*

Abstract— Many networked systems today, ranging from home automation networks to global wide-area networks, are operated using centralized control programs. Bugs in such programs pose serious risks to system security and stability. We develop a new technique to systematically explore the behavior of control programs. Because control programs depend intimately on absolute and relative timing of inputs, a key challenge that we face is to systematically handle time. We develop an approach that models programs as timed automata and incorporates novel mechanisms to enable scalable and comprehensive exploration. We implement our approach in a tool called DeLorean and apply it to real control programs for home automation and software-defined networks. DeLorean is able to finds bugs in these programs as well as provide significantly better code coverage—up to 94% compared to 76% for existing techniques.

1 Introduction

Control programs that orchestrate the actions of "dumb" devices are becoming increasingly popular. The number of such devices, including locks, thermostats, motion sensors, and packet forwarding switches, is projected to grow beyond 50 billion by 2020 [10]. In a similar vein, software-defined networking (SDN) has become a multi-billion dollar market.

While control programs for these devices may be specified using simple languages (e.g., ISY [14], in the case of home automation), reasoning about their correctness is an incredibly complex task. The programs can have complex interactions across rules due to shared variables and device states. Further, time plays an important role in program behavior, as the behavior can change with the time of day or the time between occurrences of certain events. System behavior is often programmed directly to depend on time, in terms of policies (e.g., different actions during day vs. night) and protocol behavior (e.g., DHCP leases). Therefore, the behavior of these control programs is hard to verify by running the program a few times (e.g., during development) and, as a result, many bugs are discovered in production. These bugs can compromise the safety, security, and efficiency of the system.

One method of uncovering bugs is to systematically explore program behavior using model checking. However, prior work [6, 12, 15, 18, 22] does not address an important aspect of program behavior, specifically time. Instead, these tools abstract away time and, as a result, assume timers of different periods can fire at any time and in any order. Similarly, comparisons involving time can nondeterministically return true or false. Such an imprecise analysis of time is unacceptable for control programs because, as we show later, it generates many states that are not reachable in practice. This can force developers to sort through many false positive bugs reported by these tools. Furthermore, by abstracting time, these tools preclude developers from verifying correctness properties involving time (e.g., that timers fire at the correct time and under the correct conditions). Tools that use coarse heuristics to model time [23] eliminate false positives at the expense of incomplete exploration.

Accurately modeling time when exploring program behavior is a non-trivial problem. The challenge arises because events can occur at any time. To explore all possible behaviors, in theory, we must study all possible events occurring at all possible times. However, this is an ill-defined concept since time is continuous. We describe in §2.1 why circumventing this issue by naively discretizing time is unsatisfactory.

We investigate the use of timed automata (TA) [2] to systematically explore the behavior of control programs. TAs have been previously used to verify models of real-time systems. A TA is a finite state machine extended with real-valued (not discrete) virtual clocks. TA transitions can specify constraints on clock variables. For instance, a timeout transition should happen only when a particular clock variable is greater than a constant. The analyzability of TAs arises from the fact that, under certain conditions on 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."

Prior work [4, 24] on exploring temporal behavior with TAs analyzes only an abstract model of a program or system. However, errors can be introduced in the model if it does not faithfully capture the behavior of the program, and the model can "drift" as the program evolves during development [18]. In this paper, we focus on using TAs to verify temporal properties of actual code. In particular, we ask: *can TAs be used to analyze executable programs? If so, what are the limitations of applying this theory to practice?*

Exploring the temporal behavior of a program without the need to first derive a TA introduces several new challenges. A TA-based exploration requires the set of temporal constraints that appear in the program. We develop a method to extract this information using program analysis [16]. As with many verification techniques, TA-based exploration inherits the state space explosion problem. As prior work is limited to exploring only abstract models, most heuristics to reduce the state space assume full knowledge of the model and cannot be used in our exploration. We develop three new techniques to boost exploration efficiency: (i) reducing the number of clock variables in the program, to cut down the number of regions we must explore; (ii) exploring the program as multiple, independent control loops; and (iii) predicting the response of the program to certain events, reducing the number of times we must execute the program.

We implement our approach for TA-based exploration in a tool called DeLorean and evaluate it in two diverse domains: home automation (HA) and SDNs. We explore 10 real HA programs and 3 SDN applications. Though DeLorean does not completely bridge the gap between the theory and practice of exploring real code with TAs, we see measurable benefit. DeLorean finds bugs uncovered by existing verification tools and new bugs that cannot be uncovered with existing techniques. We find we can achieve higher fidelity in exploring behavior, resulting in improved state and code coverage. We achieve up to 94% code coverage, compared to 76% in existing techniques that explore temporal behavior [23].

2 Background and Motivation

Many networked systems today are logically centralized. An HA system is composed of a controller and devices such as light switches, motion sensors, and locks. 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. Similarly, in SDNs, a controller manages the operation of switches by configuring them to forward packets as desired. The switches inform the controller when they receive packets for which forwarding actions have not been configured.

At the core of logically centralized control systems is a control program that determines its behavior. While the implementation languages for different systems and domains are different, control programs have a common structure. Their operation can be understood in terms of 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, arrival of a packet) or a firing timer. Actions include setting a device state (e.g., turn on the light, installing a new rule in a switch) or a variable and setting timers. Actions can be conditioned on device state, variable and timer values, and time of the day. Programs are single-threaded and each rule runs until completion before another is processed.

Figure 1 shows an example program 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 filter 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) and sets a timer for 5 minutes. Rule 3 is triggered when this timer fires, and turns off the light if the current time is between 6 AM and 6 PM.

2.1 Reasoning about Program Correctness

The correctness of control programs can be hard to reason about. Even if individual rules are simple, reasoning about the program as a whole can be difficult because of complex interactions across rules. These interactions arise from 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 dependency and the number of possible sequences makes predicting program behavior difficult.

As an example, even the simple program in Figure 1 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 sensed motion or by the user, triggering Rule 2. Then, the user walks on to the front porch at 9:04:50 PM, triggering Rule 1. This user might expect the light to stay on for at least 5 minutes, but it goes off unexpectedly 10 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.

Control programs are not the only ones whose correctness is difficult to reason about; the same holds true for almost all real-world programs such as network protocols and distributed systems. As a result, in a range of settings, researchers have developed a variety of techniques and corresponding tools, called model checkers, to automatically explore program behavior [6, 18].

Complex dependence on time. The behavior of control programs can depend intimately on time, both absolute time and the relative timing of triggers. For instance, the behavior of the program in Figure 1 depends on the time

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

Figure 1: An example home automation program.

```
1  Trigger0:
2    timeTrigger0 = Now;
3    timeTrigger1 = Now;
4    trigger1Seen = false;
5  Trigger1:
6    if (Now - timeTrigger0 < 5 secs)
7      trigger1Seen = true;
8  Trigger2:
9    if (trigger1Seen)
10     if (Now - timeTrigger1 < 2 secs)
11       DoOneThing();
12     else
13       DoAnotherThing();
```

Figure 2: An example home automation program.

of day and on how close in time two motion events fire.

Existing model checkers do not systematically model time. Most [6, 15, 18] perform *untimed model checking*. They completely abstract time (in the interest of scalability) and do not maintain temporal consistency. During exploration, calls to gettimeofday() return random values and timers can fire in any order, regardless of their values. This can lead to many false positives (§5.4), i.e., bad states that will not arise in practice. False positives can be highly problematic, and often worse than missing errors [23], because they can send developers on a wild goose chase. Equally important for our context, since time is abstracted, untimed model checkers cannot verify time-related properties of a system, which are of prime interest for control programs.

One model checker that maintains temporal consistency is MoDist [23]. It has a global virtual clock, which is used to return values for gettimeofday(). Timers are fired in order and, when they do, the virtual clock is advanced accordingly. It uses static analysis of program source to infer all timers, including implicit timers. (Line 2 of Figure 1 represents an implicit timer that is set in Line 5 to expire in 1 second. Line 10 checks if the timer has fired, and the program behaves differently for the two cases.) During exploration, MoDist explores two cases, one in which the timer has expired and one in which it has not. In each, the clock value is set appropriately to a value that is consistent with the explored case.

While MoDist's approach does not produce false positives, it does not comprehensively explore all possible behaviors because exploring both cases for timers is not enough. Consider the simple example in Figure 2. This program has three triggers: Trigger0 resets the control loops; Trigger1 is considered as seen if it occurs within 5 seconds of Trigger0; and Trigger2 does different things depending on whether it occurs within 2 seconds of Trigger0. Assume that when Trigger0 fires, the virtual clock time of MoDist is T (seconds). While exploring Trigger1, to cover both cases MoDist will select one virtual clock time in the range $[T, T+5)$ and one greater than $T+5$. But now it has a problem: while exploring Trigger2, it can only explore one of the two branches (Line 10 or 12) and not both. If it had picked $T+1$ in the first case, it cannot explore the path on Line 13; if it had picked $T+3$, it cannot explore the path on Line 11.

Note that at the point of exploring Trigger1, MoDist has no reason to believe the specific selection in the range $[T, T+5)$ matters. All choices lead to the same program state and paths, and only later the choice has an impact. This is just one simple example; in reality, temporal constraints in the program can be highly complex (e.g., the same timer may drive behavior in multiple places).

Without systematic modeling of the temporal behavior of the program, the only way MoDist can explore all possible program behaviors is to explore all possible times of all possible triggers. But "all possible times" is ill-defined because time is continuous. We could discretize time and assume events happen only at discrete moments. But picking a granularity of discretization is tricky—if it is too fine, the exploration will have too much overhead as we would explore 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. Simply picking the smallest time-related constant in the program is also not enough [2]. Thus, there appears no satisfactory way to pick a granularity that works for all events and programs.

We thus systematically reason about time by exploring the control program as a timed automaton (TA) [2]. This lets us carve time into equivalence regions such that the exact timing of events within a region is immaterial. Thus, instead of exploring all possible times, it suffices to explore all possible regions.

Time-bound correctness properties. Untimed model checkers find violations of properties such as liveness (i.e., the system will *eventually* enter a good state) and safety (i.e., the system *never* enters a bad state). Since these tools abstract time, they cannot verify properties involving concrete time. For example, consider an SDN program caching mappings of ports to MAC addresses. Entries should expire a certain period after their last access. An untimed model checker can prove the entry ex-

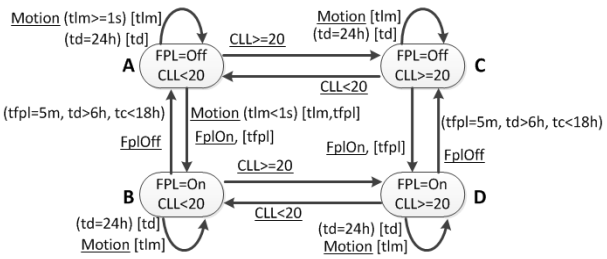

Figure 3: A TA for the program in Figure 1.

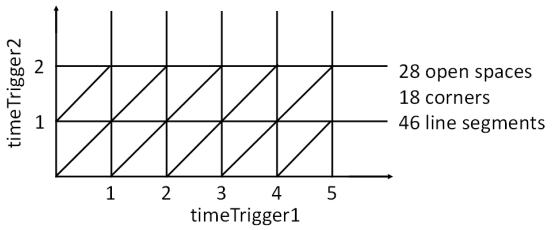

Figure 4: Time regions for the example in Figure 2.

pires after a certain sequence of events, but not that it expires after a certain period of time. To prove such a property, we must prove *time-bound liveness*. Furthermore, untimed model checkers cannot verify correctness properties based on absolute time, such as a light turning on at the right time or never being on at a certain time. In our evaluation of HA programs, we find three bugs (P9-1, P9-2, and P10-1 in §5.5) with this type of invariant.

2.2 Timed Automata

To reason systematically about time, we use timed automata [2] to guide our exploration. TAs are finite state machines extended with real-valued virtual clocks (VC), where a VC represents time elapsed since an event. (Wall clock time is simply one possible VC, which measures elapsed time since Jan. 1, 1970.) The state of a TA is the combination of the state of the underlying finite state machine together with the values of all VCs. A TA transition changes the machine state and may reset one or more 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 program in Figure 1. 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 VCs 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 where 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.

In general, systematic exploration of TAs is infeasible, even in theory, as VCs hold non-discrete real values. However, the seminal work on TAs [2] shows that an exhaustive exploration is feasible provided the 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 the possible behaviors of the TA can be discretized into regions, such that the (uncountably many) states in a region behave the same with respect to the correctness properties of the TA. How such regions emerge can be intuitively understood if one observes that for the TA in Figure 3, after a motion event, the future behaviors are determined by whether the succeeding motion event occurs before or after 1 sec. The exact timing of the second motion event is not critical. Regions exist in multi-dimensional space where each dimension corresponds to one VC, and a point in the space represents concrete values of all the VCs. Regions encompass a set of points such that the exact point is immaterial for the purposes of comprehensive exploration.

The size of the region is proportional to the greatest common denominator (GCD) of constants in clock constraints, and hence the exploration will be faster if the GCD is larger. Regions get exponentially smaller as more VCs are included in the TA, because the plane for each pair of VCs divides open spaces into two parts. 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; and 2) exploring exactly one *delay transition* in which there is no state transition but all VCs advance by the same amount. This amount is such that the time progresses to the immediately succeeding region. Figure 4 shows the regions for the example in Figure 2, which has two VCs. The constants in the clock constraints are 5 and 2, and thus the GCD is 1. We get 92 regions in this example.

3 Our Approach

Our goal is to systematically explore control program behavior. From a starting time and state, we want to predict all possible program behaviors. The exploration should be virtual so the actual state of the devices is not impacted. The output should be the set of unique states the system can be in, along with the sequence of events (i.e., triggers, actions) leading to that state.

3.1 Introducing Virtual Clocks

Control programs do not contain explicit references to VCs, but as mentioned previously, all time-related activ-

ities in effect manipulate VCs.

There are three kinds of time-related activities in control programs. The first is measuring the relative time between two events of interest (e.g., consecutive motion events). Here, a variable (e.g., timeLastMotion in Figure 1) is used to store the time of the first event, which is then subtracted from the wall clock time of the second event. To express this as 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 1). 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, where actions for a rule are taken after a delay (e.g., turn on fan, sleep 30 seconds, turn it off). We express this by introducing a new timer and new rule. The actions of the new rule corresponds 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. 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 the systems we study.

3.2 Systematically Exploring Behavior

Given a control program and its starting state as input, our goal is to explore a given duration of wall clock time. A duration must be specified since wall clock time is unbounded and has an infinite number of regions. For programs with no dependency on the wall clock (e.g., many SDN applications), no duration is needed.

We assume that the program can be modeled as a TA. That is, all timers and variables that store time in program 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 different systems that we study in §5 and §6. The scripting languages of some of these systems cannot even express complex clock operations.

However, our exploration does not assume a TA has been derived from the actual code. Existing methods [4, 24] can explore the behavior of a TA, but deriving the entire TA corresponding to the program may not be feasible. Even the smallest of control programs can have extremely large TAs, as it needs to capture the program logic and its response to possible events.

Thus, we explore the TA dynamically, akin to how FSA-based model checkers dynamically explore the FSA

```
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:          So = Compute(Si, e);
10:         if !Similar(So, (US ∪ ES)) then
11:             US.push(So);
12:         end if
13:     end for
14:     if Si.EnTimers = φ then
15:         delay = DelayForNextRegion(Si.Region);
16:         if Si.WC + delay > EndWC then
17:             continue;
18:         end if
19:         So = Si.AdvanceAllVCs(delay);
20:         for all timer in So.Timers do
21:             if timer.dueTime >= So.WC then
22:                 So.EnTimers.Push(timer);
23:             end if
24:         end for
25:         if !Similar(So, (US ∪ ES)) then
26:             US.push((So, t));
27:         end if
28:     end if
29: end while
```

Figure 5: Pseudocode for basic TA exploration.

instead of deriving the complete FSA of the program. From a starting program state, we repeatedly derive successor states resulting from triggers or delay transitions. For delay transitions, we must know the timed regions in advance to compute the delay amount. Fortunately, constructing regions does not require the complete TA, but only the constraints on the values of VCs [2]. We extract these constraints using analysis of program source.

Figure 5 shows how we comprehensively explore program behavior. Assume we want to explore FFDuration of behavior, starting from the program state S_0. Program state includes the values of (non-time) variables, VCs, and enabled timers (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. If a successor state is not similar to any previously seen state, we add it to the queue. Two states are similar if their variable values and set of enabled timers are identical and if their VC values map to the same region; VC values need not 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. This represents a period of time where nothing happens and time advances to the succeeding region. States with enabled timers need to fire all enabled timers before time can progress. We ig-

nore the successor if this delay takes us past the end time (i.e., starting wall clock time + specified duration). 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.

3.3 Achieving Scalable Exploration

The basic TA-based exploration above correctly handles time but is too slow to be practical. We use three general techniques to make it practical:

Predicting successor states Our first technique reduces the time to obtain successor states of a state being explored. We must first define the notion of *clock personality*. Two program states have the same clock personality if their values of all the VCs are equivalent with respect to all the clock constrains of the TA. Two program states can have the same clock personalities even if they are not in the same region.

If two states $S1$ and $S2$ with the same clock personalities have identical variable values and enabled timers, then any stimulus (i.e., combination of trigger and environmental conditions) will have exactly the same effect on both states. Thus, it is necessary to compute the successor of only one such state, say $S1$. The successor of $S2$ can be obtained from the successor of $S1$ while retaining the clock values of $S2$ for all VCs except those that are reset by the current stimulus.

Computing a successor requires deserializing the parent's state, running the program, subjecting it to the stimulus, and serializing the successor's state. These are costly operations. In contrast, prediction requires only copying the state and modifying VC values.

Independent control loops Our second optimization is based on the observation that large control programs may often be composed of multiple, independent control loops manipulating different parts of the program state. For instance, thermostats and furnaces may be controlled by a climate control loop, and locks and alarms by a security loop. These two may manipulate different variables and clocks, but otherwise share no state. In such cases, we can explore the loops independently, instead of exploring them jointly. Separate exploration is faster since joint exploration considers the Cartesian product of the values of independent variables and clocks. We use taint tracking to identify independent loops.

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 transforming a control 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 [7]. 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, control programs often have daily action for different times of day (e.g., sunrise, one hour after sunrise). The straightforward translation is to introduce a new timer per unique activity. A more efficient method is to use one timer to conduct all such activities, using a method similar to the above — introduce an additional program variable to cycle through the different actions and reset it after the last action is conducted.

3.4 Theory-Practice Gap

Existing TA-based model checkers work with abstract models and assume the model is provided as input. In building the model incrementally and dynamically, we uncover several gaps in using TAs on real code.

A transition in a TA must occur instantaneously since time only progresses explicitly through a delay transition. In practice, however, the processing of an event (e.g., in response to motion occurring) may involve a non-trivial amount of time. In our implementation, we assume processing time is instantaneous, but propose a technique to handle events with non-trivial processing times. For each of these event handlers in the program, we can introduce a timer to expire after the expected processing time. When the timer is active and has not expired, the system is processing the event. If the timer is inactive, either because the timer has expired or has not been activated, the system is not processing the event.

In some systems, clocks may be created in response to events. In SDN programs, for example, flows installed in switches in response to a packet arrival at the controller introduce two new VCs—one each for the soft and hard timeouts. We can use symbolic execution to extract the clock constraints for all possible values of timeouts, but we cannot determine the number of occurrences of the event triggering this behavior. Rather, this is dependent on the number of times the event occurs along a path generating a specific state. We cannot add a new clock with a new constraint to the region construction, as we cannot change regions during exploration. Regions are constructed using GCD of the clock constraints and adding

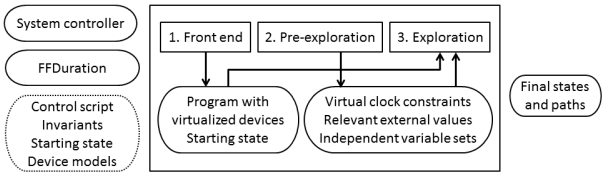

Figure 6: Overview of DeLorean.

a new clock mid-exploration could change the delay if the GCD changes. Instead, we pre-allocate a number of VCs, each with a specific constraint. During exploration, when a new VC needs to be created, it is allocated an available VC from a queue of pre-allocated VCs.

4 Design

We now describe the design of DeLorean, our TA-based model checker. As shown in Figure 6, the primary inputs to DeLorean is the control program—including a model of the controller (on which the program runs) and devices—and the duration of wall clock time to explore (FFDuration). If the program has no dependence on wall clock time, i.e., timers are relative, this parameter is not needed. The user can also specify three optional inputs. The first is a list of invariants on device states and system behavior, which should be satisfied at all times. These are specified in a manner similar to the *if* conditions in the rules and can include time. The second is the wall clock time. The third is the starting state of (a subset of) devices and variables from which exploration should begin.

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

DeLorean has three stages. First, the front end converts the program to one where clocks have been virtualized, using the method in §3.1. Second, pre-exploration analyzes this program to recover information required for the optimizations in §3.3. The final stage is the exploration itself.

4.1 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 in §3.3. Here, we use symbolic execution [16] of program source. Symbolic execution simulates the execution of code using a symbolic value σ_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 σ_y=2σ_x. At branches, symbolic execution uses a constraint solver to determine the value of the guard expression given the information in the store. The executor only explores the branch corresponding to the guard's value as returned by the constraint solver, ensuring infeasible paths are ignored. If there is insufficient information to determine the guard's value, both branches are explored. This produces a tree of all possible program execution paths. 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, which is the starting point for all processing activity. We configure the symbolic executor to treat the following entities as symbolic: program state (variables and clocks) and the parameters of the control loop. The output of the symbolic executor is the set of possible paths for each possible trigger. 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.

Independent control loops We also use the output of symbolic execution for taint tracking. We analyze the program state that results from 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.

We use this information to identify independent sets of variables and VCs. Two variables or VCs are deemed dependent if they either 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.

4.2 Exploration

This stage implements the method outlined in §3. To start, it runs the program and initializes the starting state. We then checkpoint the program by serializing its internal state. The checkpoint captures the values of all variables, including time related variables, and the times

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

Table 1: The HA programs we study.

when various timers will fire.

We maintain a table that contains the values of the VCs of a state. Many states differ only in VC values—the successor state after a delay transition differs from the parent only in VC values, so does the successor that is predicted from another state. Maintaining this table separately lets us quickly obtain these successor states. It also helps reduce the memory footprint, since two states that differ only in VC values can share the same checkpoint. 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 before any other processing.

4.3 Implementation

DeLorean is implemented with 10k+ lines of C# code. The bulk of this code implements the pre-exploration and exploration stage, which we developed from scratch. We could not use existing tools for exploring TAs [4, 24] because we do not have the complete TA for the program. Our implementation includes models of controllers and devices in the two domains we study—home automation (§ 5) and software-defined networks (§ 6).

For HA applications, we implemented front end modules for two systems—ISY [14] and ELK [9]. We chose these two because of their popularity. The front end parses ISY or ELK programs using ANTLR [3] and produces a C# program that captures the behavior of the program and contains additional variables, rules, and actions needed for modeling devices. As the state of these devices is typically simple and can be represented using boolean or integer variables, we can model the devices automatically from the ISY or ELK program.

The pre-exploration stage uses Pex [19] 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.

5 Case Study: HA Networks

To evaluate a TA-based exploration against existing verification techniques, we examine DeLorean in two environments: home automation networks and SDNs.

5.1 Domain-Specific Optimizations

A common behavior in HA is a dependence on environmental factors (e.g., temperature, light level) sensed by devices in the system. For a comprehensive evaluation, we must explore all combinations of values of external factors. To address this challenge, we build on prior work and combine symbolic execution with model checking [6]. We use symbolic execution of program source to infer equivalence classes of combinations of values of environmental factors. In Figure 1, for example, there are two equivalence classes, corresponding to light level values below or higher than 20. Then during exploration, we use one set of values per class, instead of having to explore all possible combinations of values. So, if a program depends on temperature and light level, for every trigger, its response must be explored with all combinations of temperature and light levels.

5.2 Dataset

We evaluate DeLorean using real HA programs. We solicited these programs on a mailing list for HA enthusiasts. We picked the 10 programs shown in Table 1. 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, temperature sensors, sprinklers, and thermostats.

The table shows the source lines of code (SLoC) and the number of VCs in the program after transformation in the first stage. Systems 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.

5.3 Exploration Performance

We run DeLorean over all 10 programs and 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. Table 2 shows the number of transitions and average CPU time needed to explore one hour of wall clock time for each program. We estimate DeLorean makes 200k transitions per second. Since HA programs depend on wall clock time, we can also measure the CPU time with respect to wall clock time. We also see that DeLorean can explore real programs 3.6 times to 36K times

	# Transitions	CPU Time (sec)	Reduction w/ Prediction
P1	72	0.10	-11.11%
P2	123	0.12	-20%
P3	178	0.15	-7.14%
P4	19.7M	176.10	75.16%
P5	78.7K	1.03	61.28%
P6	51K	1.04	48%
P7	36.M	17.87	89.53%
P8	8.1M	89.50	84.36%
P9	121M	793.90	95.24%
P10	256M	998.0	83.5%

Table 2: Performance for exploring one hour of wall clock time and the reduction in CPU time from predicting states.

faster than wall clock time.

An important element in obtaining quick explorations for these programs is predicting successor states. While this is a general optimization for dynamically exploring TAs, its effectiveness depends on how often we encounter non-similar states with identical clock personalities, variables, and enabled timers. To evaluate it, Table 2 show the percentage reduction in CPU time when prediction is used compared to when it is not used. For the smallest 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 cuts the exploration time by 95%, i.e., exploring without prediction is slower by a factor of 20.

5.4 Comparison with Alternatives

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, 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 7 shows the percentage of additional, invalid states produced by untimed model checking,[1] when 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

[1]This comparison based on invalid states alone hides one additional 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).

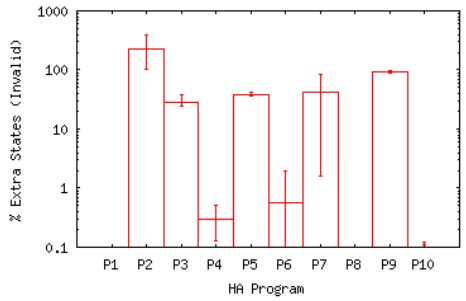

Figure 7: Invalid states generated by untimed exploration.

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.

MoDist Unlike untimed exploration, MoDist maintains temporal consistency during exploration, but at the expense of incomplete exploration (§2.1). To illustrate this, we implement MoDist's algorithm for exploring timers in DeLorean and compare it with our exploration. We compare two metrics–state coverage and code coverage. State coverage measures the number of unique program states explored and code coverage measures the number of lines of code exercised during exploration. Figures 8 and 9 show code and state coverage, respectively, for MoDist and DeLorean averaged over 24 trials, each exploring one hour. Programs omitted in Figure 8 have equivalent coverage in MoDist and DeLorean.

5.5 Unintended Behaviors

To informally gauge DeLorean's ability to find such behaviors, we inspect comments in two of the programs (P9, P10) and turn them into invariants for which DeLorean should report violations. We find four violations.

P9-1 A comment indicated the lights in the back of the house should turn on if motion is detected in the evening (i.e., sunset to 11:35PM). But DeLorean found that the

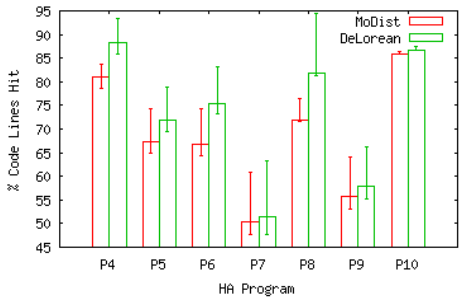

Figure 8: Code coverage.

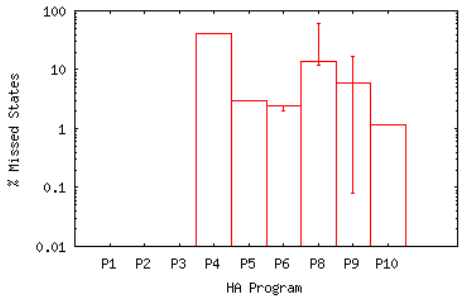

Figure 9: States missed by MoDist.

lights could be on even if there was no motion. A rule 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 the front porch light should stay on from a half hour after sunset until 2AM. There were two rules to implement this invariant: one turning the light on at a half hour after sunset and one turning it off at 2AM. But DeLorean found cases where the light was off in that time window. Inspection revealed another rule to turn off the light at 7:45PM. Thus, the invariant is violated if sunset occurs after 7:15PM, which can happen where the user of P9 resides. Exploring sunset values higher than 7:15PM uncovered the violation.

P10-1 A comment indicated the user wanted to turn on a 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 the user wanted to treat three devices identically (i.e., 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 a device.

	SLoC	#VCs	GCD (s)	#trans 1 VC	#trans 2 VCs	#trans 4 VCs
PySwitch	234	13	1	6210	49k	8.8M
LoadBalancer	2063	14	2	351k	512k	3.8M
EnergyTE	434	10	5	442k	1.7M	21M

Table 3: The OpenFlow programs we study.

6 Case Study: SDN

To further demonstrate the value of TA-based exploration, we model and test SDN programs in DeLorean. Similar to NICE [6], we create a model of the NOX platform in C#, including the controller, switches, and hosts. We discover relevant packet headers during pre-exploration, using Pex to symbolically execute the event handlers that make up the OpenFlow program. Since OpenFlow switches have complex internal behavior (e.g., flow tables) that we cannot observe externally, we manually define models of OpenFlow switches and hosts. OpenFlow programs have no dependency on absolute time, therefore we use no wall clock time.

6.1 Dataset

We evaluate DeLorean using three real programs—a MAC-learning switch (PySwitch), a web server load balancer[21], and energy efficient traffic engineering (REsPoNse) [20]. We manually translate the programs from Python into C# for testing in DeLorean. Table 3 shows the source lines of code (SLoC), the total number of VCs that can be dynamically created during an exploration, and the GCD of the clock constraints. As in NICE, we bound the state space by limiting certain events, such as the number of times a host sends a packet.

Each program has dependencies on relative time. PySwitch, for example, uses a timer to periodically check entries in a cache of MAC address-port mappings and expire entries older than a specific time. In this case, a VC is needed to express the timer scheduling the periodic check, and another VC for each entry in the cache.

6.2 Comparison with Alternatives

We compare DeLorean to NICE, a model checker for OpenFlow programs. Similar to untimed model checking, NICE does not systematically model time. Instead, application-specific heuristics are used to trigger timers in each of the SDN applications tested. We construct a model of the NOX platform for DeLorean and simulate NICE's exploration by running DeLorean with no VCs. We also implement NICE's heuristics for exploring timer behavior. To informally gauge DeLorean's ability to find unintended behaviors, we create invariants from the 11 bugs discovered by NICE. We find DeLorean can reproduce violations for all 11 bugs.

We now compare DeLorean's coverage of a program's

	% Missing			% Incorrect		
	1 VC	2 VCs	4 VCs	1 VC	2 VCs	4 VCs
Pyswitch	0%	34%	84%	0%	0%	0%
Loadbalancer	95%	95%	95%	117%	123%	123%
EnergyTE	69%	87%	97%	26%	12%	46%

Table 4: Missing and invalid states generated by NICE, compared to explorations in DeLorean using 1, 2, and 4 VCs.

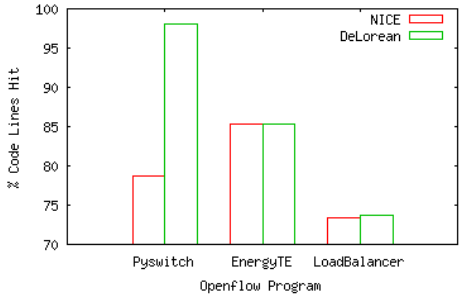

Figure 10: Code coverage in DeLorean and NICE.

state space to that of NICE. In Figure 10, we show the code coverage of DeLorean and NICE for the three programs. In PySwitch, NICE does not explore the timer for periodically checking cache entries, therefore missing an entire function.

In Table 4, we show the number of missed and incorrect states generated by NICE compared to explorations in DeLorean using 1, 2, and 4 VCs. Because NICE does not trigger any timers in PySwitch, it misses potential behavior, but does not introduce any invalid states that would be generated from timers firing at incorrect times from incorrect states The heuristic for trigger timers in the EnergyTE application, however, fires timers from every possible state, resulting in invalid states. Similarly, in LoadBalancer, timers can also fire from invalid states.

Further, NICE's heuristics do not test the expiration of flows. Correctly exploring this behavior requires, and verifying correctness properties related to flow expirations, requires more systematic treatment of time. This results in missed states in both the EnergyTE and LoadBalancer applications.

We see that with non-systematic treatment of time, program exploration can introduce false behaviors or miss potential behaviors. In programs dependent on absolute and relative time, such as HA programs, we find untimed exploration can produce too many invalid states to be useful. Even in programs with dependencies only on relative time, such as SDN programs, we see non-systematic treatment of time can also produce as many invalid states as valid states.

7 Related Work

Our work builds on progress the research community has made towards verifying the behavior of real systems.

Model checking programs One class of techniques is model checking, where programs are modeled as FSAs and their behavior is comprehensively explored [12, 15, 18]. Recent work, like us, also combines model checking with symbolic execution [5, 8, 22]. However, most model checking work ignores time. 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 [6]. However, its treatment of time is not systematic and instead relies on heuristics to explore timer behavior.

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, 24] as well as transformations that speed explorations [7, 13]. 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, is complementary to other program debugging approaches. Record and replay [17] 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 [11]. These focus on computing the outcomes of configuration changes; in contract, we study the dynamic behavior of more general programs.

8 Conclusions

Mistakes in control programs can impact the safety and efficiency of their system. We develop a technique using timed automata to systematically explore program behavior and verify temporal properties. We implement our approach in a tool named DeLorean and apply it to two domains where timing in control programs is important—home automation and software-defined networks. We show it results in higher fidelity analysis, including better state and code coverage, than existing techniques that do not systematically model time.

References

[1] ALIMI, R., WANG, Y., AND YANG, Y. Shadow configuration as a network management primitive. *ACM SIGCOMM* (August

2008).

[2] ALUR, R., AND DILL, D. L. A theory of timed automata. *Theoretical Computer Science* (1994).

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

[4] BENGTSSON, J., LARSEN, K., LARSSON, F., PETTERSSON, P., AND YI, W. UPPAAL: A tool suite for automatic verification of real-time systems. *Hybrid Systems III* (1996).

[5] CANINI, M., JOVANOVIC, V., VENZANO, D., SPASOJEVIC, B., CRAMERI, O., AND KOSTIC, D. Toward online testing of federated and heterogeneous distributed systems. In *USENIX ATC* (2011).

[6] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., AND REXFORD, J. A NICE way to test OpenFlow applications. In *NSDI* (2012).

[7] DAWS, C., AND YOVINE, S. Reducing the number of clock variables of timed automata. In *Real-Time Systems Symposium* (1996).

[8] DOBRESCU, M., AND ARGYRAKI, K. Software dataplane verification. In *NSDI* (2014).

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

[10] EVANS, D. The Internet of things. `http://blogs.cisco.com/news/the-internet-of-things-infographic/`, 2011.

[11] FEAMSTER, N., AND REXFORD, J. Network-wide prediction of BGP routes. *IEEE/ACM Trans. Networking* (April 2007).

[12] GODEFROID, P. Model checking for programming languages using verisoft. In *POPL* (1997).

[13] HENDRIKS, M., AND LARSEN, K. G. Exact acceleration of real-time model checking. In *Electronic Notes in Theoretical Computer Science* (2002).

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

[15] KILLIAN, C. E., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI* (2007).

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

[17] LEBLANC, T., AND MELLOR-CRUMMEY, J. Debugging parallel programs with instant replay. *IEEE Transactions on Computers 36* (1987).

[18] MUSUVATHI, M., PARK, D. Y. W., CHOU, A., ENGLER, D. R., AND DILL, D. L. CMC: A programatic approach to model checking real code. In *OSDI* (2002).

[19] TILLMANN, N., AND DE HALLEUX, J. Pex: White box test generation for .NET. In *Tests and Proofs (TAP)* (April 2008).

[20] VASIĆ, N., NOVAKOVIĆ, D., SHEKNAR., S., BHURAT, P., CANINI, M., AND KOSTIĆ, D. Identifying and using energy-critical paths. In *CoNEXT* (2011).

[21] WANG, R., BUTNARIU, D., AND REXFORD, J. OpenFlow-based server load balancing gone wild. In *Hot-ICE* (2011).

[22] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI* (2009).

[23] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MoDist: Transparent model checking of unmodified distributed systems. In *NSDI* (2009).

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