Stateful Dynamic Partial Order Reduction for Model Checking Event-Driven Applications that Do Not Terminate

Rahmadi Trimananda[1(✉)], Weiyu Luo[1], Brian Demsky[1], and Guoqing Harry Xu[2]

[1] University of California, Irvine, USA
{rtrimana,weiyul7,bdemsky}@uci.edu
[2] University of California, Los Angeles, USA
harryxu@g.ucla.edu

Abstract. Event-driven architectures are broadly used for systems that must respond to events in the real world. Event-driven applications are prone to concurrency bugs that involve subtle errors in reasoning about the ordering of events. Unfortunately, there are several challenges in using existing model-checking techniques on these systems. Event-driven applications often loop indefinitely and thus pose a challenge for stateless model checking techniques. On the other hand, deploying purely stateful model checking can explore large sets of equivalent executions.

In this work, we explore a new technique that combines dynamic partial order reduction with stateful model checking to support non-terminating applications. Our work is (1) the first dynamic partial order reduction algorithm for stateful model checking that is sound for non-terminating applications and (2) the first dynamic partial reduction algorithm for stateful model checking of event-driven applications. We experimented with the IoTCheck dataset—a study of interactions in smart home app pairs. This dataset consists of app pairs originated from 198 real-world smart home apps. Overall, our DPOR algorithm successfully reduced the search space for the app pairs, enabling 69 pairs of apps that did not finish without DPOR to finish and providing a 7× average speedup.

1 Introduction

Event-driven architectures are broadly used to build systems that react to events in the real world. They include smart home systems, GUIs, mobile applications, and servers. For example, in the context of smart home systems, event-driven systems include Samsung SmartThings [46], Android Things [16], OpenHAB [35], and If This Then That (IFTTT) [21].

B. Finkbeiner and T. Wies (Eds.): VMCAI 2022, LNCS 13182, pp. 400–424, 2022.
https://doi.org/10.1007/978-3-030-94583-1_20

Event-driven architectures can have analogs of the concurrency bugs that are known to be problematic in multithreaded programming. Subtle programming errors involving the ordering of events can easily cause event-driven programs to fail. These failures can be challenging to find during testing as exposing these failures may require a specific set of events to occur in a specific order. Model-checking tools can be helpful for finding subtle concurrency bugs or understanding complex interactions between different applications [49]. In recent years, significant work has been expended on developing model checkers for multithreaded concurrency [2,19,22,25,57,60,62,64], but event-driven systems have received much less attention [22,30].

Event-driven systems pose several challenges for existing *stateless* and *stateful* model-checking tools. Stateless model checking of concurrent applications explores all execution schedules without checking whether these schedules visit the same program states. Stateless model checking often uses dynamic partial order reduction (DPOR) to eliminate equivalent schedules. While there has been much work on DPOR for stateless model checking of multithreaded programs [2,12,19,25,64], stateless model checking requires that the program under test terminates for fair schedules. Event-driven systems are often intended to run continuously and may not terminate. To handle non-termination, stateless model checkers require hacks such as bounding the length of executions to verify event-driven systems.

Stateful model checking keeps track of an application's states and avoids revisiting the same application states. It is less common for stateful model checkers to use dynamic partial order reduction to eliminate equivalent executions. Researchers have done much work on stateful model checking [17,18,32,56]. While stateful model checking can handle non-terminating programs, they miss an opportunity to efficiently reason about conflicting transitions to scale to large programs. In particular, typical event-driven programs such as smart home applications have several event handlers that are completely independent of each other. Stateful model checking enumerates different orderings of these event handlers, overlooking the fact that these handlers are independent of each other and hence the orderings are equivalent.

Stateful model checking and dynamic partial order reduction discover different types of redundancy, and therefore it is beneficial to combine them to further improve model-checking scalability and efficiency. For example, we have observed that some smart home systems have several independent event handlers in our experiments, and stateful model checkers can waste an enormous amount of time exploring different orderings of these independent transitions. DPOR can substantially reduce the number of states and transitions that must be explored. Although work has been done to combine DPOR algorithms with stateful model checking [61,63] in the context of multithreaded programs, this line of work requires that the application has an *acyclic state space*, *i.e.*, it terminates under all schedules. In particular, the approach of Yang *et al.* [61] is designed explicitly for programs with acyclic state space and thus cannot check programs that do not terminate. Yi *et al.* [63] presents a DPOR algorithm for

stateful model checking, which is, however, incorrect for cyclic state spaces. For instance, their algorithm fails to produce the asserting execution in the example we will discuss shortly in Fig. 1. As a result, prior DPOR techniques all fall short for checking event-driven programs such as smart home apps, that, in general, do not terminate.

Our Contributions. In this work, we present a stateful model checking technique for event-driven programs that may not terminate. Such programs have cyclic state spaces, and existing algorithms can prematurely terminate an execution and thus fail to set the necessary backtracking points to fully explore a program's state space. Our **first** technical contribution is the *formulation of a sufficient condition to complete an execution of the application that ensures that our algorithm fully explores the application's state space.*

In addition to the early termination issue, for programs with cyclic state spaces, a model checker can discover multiple paths to a state s before it explores the entire state space that is reachable from state s. In this case, the backtracking algorithms used by traditional DPOR techniques including Yang *et al.* [61] can fail to set the necessary backtracking points. Our **second** technical contribution is *a graph-traversal-based algorithm to appropriately set backtracking points on all paths that can reach the current state.*

Prior work on stateful DPOR only considers the multithreaded case and assumes algorithms know the effects of the next transitions of all threads before setting backtracking points. For multithreaded programs, this assumption is *not* a serious limitation as transitions model low-level memory operations (*i.e.*, reads, writes, and RMW operations), and each transition involves a *single* memory operation. However, in the context of event-driven programs, events can involve many memory operations that access multiple memory locations, and knowing the effects of a transition requires actually executing the event. While it is conceptually possible to execute events and then rollback to discover their effects, this approach is likely to incur large overheads as model checkers need to know the effects of enabled events at each program state. As our **third** contribution, *our algorithm avoids this extra rollback overhead by waiting until an event is actually executed to set backtracking points and incorporates a modified backtracking algorithm to appropriately handle events.*

We have implemented the proposed algorithm in the Java Pathfinder model checker [56] and evaluated it on hundreds of real-world smart home apps. We have made our DPOR implementation publicly available [50].

Paper Structure. The remainder of this paper is structured as follows: Sect. 2 presents the event-driven concurrency model that we use in this work. Section 3 presents the definitions we use to describe our stateful DPOR algorithm. Section 4 presents problems when using the classic DPOR algorithm to model check event-driven programs and the basic ideas behind how our algorithm solves these problems. Section 5 presents our stateful DPOR algorithm for event-driven programs. Section 6 presents the evaluation of our algorithm implementation on hundreds of smart home apps. Section 7 presents the related work; we conclude in Sect. 8.

2 Event-Driven Concurrency Model

In this section, we first present the concurrency model of our event-driven system and then discuss the key elements of this system formulated as an event-driven concurrency model. Our event-driven system is inspired by—and distilled from—smart home IoT devices and applications deployed widely in the real world. Modern smart home platforms support developers writing apps that implement useful functionality on smart devices. Significant efforts have been made to create integration platforms such as Android Things from Google [16], SmartThings from Samsung [46], and the open-source openHAB platform [35]. All of these platforms allow users to create *smart home apps* that integrate multiple devices and perform complex routines, such as implementing a home security system.

The presence of multiple apps that can control the same device creates undesirable interactions [49]. For example, a homeowner may install the `FireCO2Alarm` [38] app, which upon the detection of smoke, sounds alarms and unlocks the door. The same homeowner may also install the `Lock-It-When-I-Leave` [1] app to lock the door automatically when the homeowner leaves the house. However, these apps can interact in surprising ways when installed together. For instance, if smoke is detected, `FireCO2Alarm` will unlock the door. If someone leaves home, the `Lock-It-When-I-Leave` app will lock the door. This defeats the intended purpose of the `FireCO2Alarm` app. Due to the increasing popularity of IoT devices, understanding and finding such conflicting interactions has become a hot research topic [27,28,54,55,58] in the past few years. Among the many techniques developed, model checking is a popular one [49,59]. However, existing DPOR-based model checking algorithms do not support non-terminating event-handling logic (detailed in Sect. 4), which strongly motivates the need of developing new algorithms that are both sound and efficient in handling real-world event-based (*e.g.*, IoT) programs.

2.1 Event-Driven Concurrency Model

We next present our event-driven concurrency model (see an example of event-driven systems in Appendix A in [51]). We assume that the event-driven system has a finite set $\mathcal{E}$ of different event types. Each event type $e \in \mathcal{E}$ has a corresponding event handler that is executed when an instance of the event occurs. We assume that there is a potentially shared state and that event handlers have arbitrary access to read and write from this shared state.

An event handler can be an arbitrarily long finite sequence of instructions and can include an arbitrary number of accesses to shared state. We assume event-handlers are executed atomically by the event-driven runtime system. Events can be enabled by both external sources (*e.g.*, events in the physical world) or event handlers. Events can also be disabled by the execution of an event handler. We assume that the runtime system maintains an unordered set of enabled events to execute. It contains an event dispatch loop that selects an arbitrary enabled event to execute next.

This work is inspired by smart-home systems that are widely deployed in the real world. However, the proposed techniques are general enough to handle other types of event-driven systems, such as web applications, as long as the systems follow the concurrency model stated above.

2.2 Background on Stateless DPOR

Partial order reduction is based on the observation that traces of concurrent systems are equivalent if they only reorder independent operations. These equivalence classes are called Mazurkiewicz traces [31]. The classical DPOR algorithm [12] dynamically computes persistent sets for multithreaded programs and is guaranteed to explore at least one interleaving in each equivalence class.

The key idea behind the DPOR algorithm is to compute the next pending memory operation for each thread, and at each point in the execution to compute the most recent conflict for each thread's next operation. These conflicts are used to set backtracking points so that future executions will reverse the order of conflicting operations and explore an execution in a different equivalence class. Due to space constraints, we refer the interested readers to [12] for a detailed description of the original DPOR algorithm.

3 Preliminaries

We next introduce the notations and definitions we use throughout this paper.

Transition System. We consider a transition system that consists of a finite set $\mathcal{E}$ of events. Each event $e \in \mathcal{E}$ executes a sequence of instructions that change the *global* state of the system.

States. Let *States* be the set of the states of the system, where $s_0 \in States$ is the initial state. A state s captures the heap of a running program and the values of global variables.

Transitions and Transition Sequences. Let $\mathcal{T}$ be the set of all transitions for the system. Each transition $t \in \mathcal{T}$ is a partial function from *States* to *States*. The notation $t_{s,e} = next\,(s, e)$ returns the transition $t_{s,e}$ from executing event e on program state s. We assume that the transition system is deterministic, and thus the destination state $dst(t_{s,e})$ is unique for a given state s and event e. If the execution of transition t from s produces state s', then we write $s \xrightarrow{t} s'$.

We formalize the behavior of the system as a transition system $A_G = (States, \Delta, s_0)$, where $\Delta \subseteq States \times States$ is the transition relation defined by

$$(s, s') \in \Delta \text{ iff } \exists t \in \mathcal{T} : s \xrightarrow{t} s'$$

and s_0 is the initial state of the system.

A transition sequence $\mathcal{S}$ of the transition system is a finite sequence of transitions $t_1, t_2, ..., t_n$. These transitions advance the state of the system from the initial state s_0 to further states $s_1, ..., s_i$ such that

$$\mathcal{S} = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots s_{i-1} \xrightarrow{t_n} s_i.$$

Enabling and Disabling Events. Events can be enabled and disabled. We make the same assumption as Jensen *et al.* [22] regarding the mechanism for enabling and disabling events. Each event has a special memory location associated with it. When an event is enabled or disabled, that memory location is written to. Thus, the same conflict detection mechanism we used for memory operations will detect enabled/disabled conflicts between events.

Notation. We use the following notations in our presentation:

- $event(t)$ returns the event that performs the transition t.
- $first(\mathcal{S}, s)$ returns the first occurrence of state s in $\mathcal{S}$, *e.g.*, if s_4 is first visited at step 2 then $first(\mathcal{S}, s_4)$ returns 2.
- $last(\mathcal{S})$ returns the last state s in a transition sequence $\mathcal{S}$.
- $\mathcal{S}.t$ produces a new transition sequence by extending the transition sequence $\mathcal{S}$ with the transition t.
- $states(\mathcal{S})$ returns the set of states traversed by the transition sequence $\mathcal{S}$.
- $enabled(s)$ denotes the set of enabled events at s.
- $backtrack(s)$ denotes the backtrack set of state s.
- $done(s)$ denotes the set of events that have already been executed at s.
- $accesses(t)$ denotes the set of memory accesses performed by the transition t. An access consists of a memory operation, *i.e.*, a read or write, and a memory location.

State Transition Graph. In our algorithm, we construct a state transition graph $\mathcal{R}$ that is similar to the visible operation dependency graph presented in [61]. The state transition graph records all of the states that our DPOR algorithm has explored and all of the transitions it has taken. In more detail, a state transition graph $\mathcal{R} = \langle V, E \rangle$ for a transition system is a directed graph, where every node $n \in V$ is a visited state, and every edge $e \in E$ is a transition explored in some execution. We use $\rightarrow_r$ to denote that a transition is reachable from another transition in $\mathcal{R}$, *e.g.*, $t_1 \rightarrow_r t_2$ indicates that t_2 is reachable from t_1 in $\mathcal{R}$.

Independence and Persistent Sets. We define the independence relation over transitions as follows:

Definition 1 (Independence). *Let $\mathcal{T}$ be the set of transitions. An independence relation $I \subseteq \mathcal{T} \times \mathcal{T}$ is a irreflexive and symmetric relation, such that for any transitions $(t_1, t_2) \in I$ and any state s in the state space of a transition system A_G, the following conditions hold:*

1. *if $t_1 \in enabled(s)$ and $s \xrightarrow{t_1} s'$, then $t_2 \in enabled(s)$ iff $t_2 \in enabled(s')$.*
2. *if t_1 and t_2 are enabled in s, then there is a unique state s' such that $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$.*

If $(t_1, t_2) \in I$, then we say t_1 and t_2 are independent. We also say that two memory accesses to a shared location *conflict* if at least one of them is a write. Since executing the same event from different states can have different effects on the states, *i.e.*, resulting in different transitions, we also define the notion of *read-write independence* between events on top of the definition of independence relation over transitions.

Definition 2 (Read-Write Independence). *We say that two events x and y are read-write independent, if for every transition sequences τ where events x and y are executed, the transitions t_x and t_y corresponding to executing x and y are independent, and t_x and t_y do not have conflicting memory accesses.*

Definition 3 (Persistent Set). *A set of events $X \subseteq \mathcal{E}$ enabled in a state s is persistent in s if for every transition sequence from s*

$$s \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots \xrightarrow{t_{n-1}} s_{n-1} \xrightarrow{t_n} s_n$$

where $\text{event}(t_i) \notin X$ *for all* $1 \leq i \leq n$, *then* $\text{event}(t_n)$ *is read-write independent with all events in* X.

In Appendix B in [51], we prove that exploring a persistent set of events at each state is sufficient to ensure the exploration of at least one execution per Mazurkiewicz trace for a program with cyclic state spaces and finite reachable states.

4 Technique Overview

This section overviews our ideas. These ideas are discussed in the context of four problems that arise when existing DPOR algorithms are applied directly to event-driven programs. For each problem, we first explain the cause of the problem and then proceed to discuss our solution.

4.1 Problem 1: Premature Termination

The first problem is that the naive application of existing stateless DPOR algorithms to stateful model checking will prematurely terminate the execution of programs with cyclic state spaces, causing a model checker to miss exploring portions of the state space. This problem is known in the general POR literature [13,37,52] and various provisos (conditions) have been proposed to solve the problem. While the problem is known, all existing stateful DPOR algorithms produce incorrect results for programs with cyclic state spaces. Prior work by Yang *et al.* [61] only handles programs with acyclic state spaces. Work by Yi *et al.* [63] claims to handle cyclic state spaces, but overlooks the need for a proviso for when it is safe to stop an execution due to a state match and thus can produce incorrect results when model checking programs with cyclic state spaces.

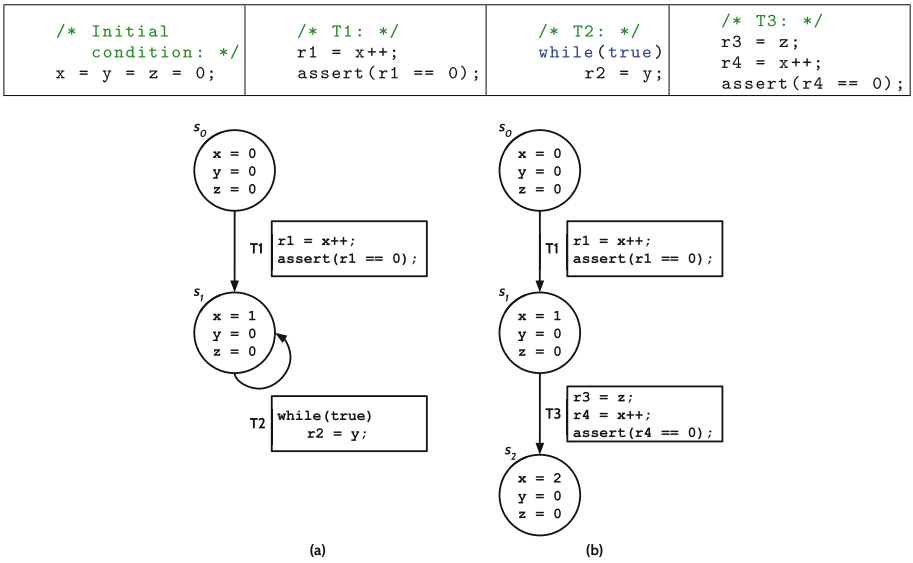

Fig. 1. Problem with existing stateful DPOR algorithms on a non-terminating multithreaded program. Execution (a) terminates at a state match without setting any backtracking points. Thus, stateful DPOR would miss exploring Execution (b) which has an assertion failure.

Figure 1 presents a simple multithreaded program that illustrates the problem of using a naive stateful adaptation of the DPOR algorithm to check programs with cyclic state spaces. Let us suppose that a stateful DPOR algorithm explores the state space from s_0, and it selects thread $\mathtt{T}_1$ to take a step: the state is advanced to state s_1. However, when it selects $\mathtt{T}_2$ to take the next step, it will revisit the same state and stop the current execution (see Fig. 1-a). Since it did not set any backtracking points, the algorithm prematurely finishes its exploration at this point. It misses the execution where both threads $\mathtt{T}_1$ and $\mathtt{T}_3$ take steps, leading to an assertion failure. Figure 1-b shows this missing execution. The underlying issue with halting an execution when it matches a state from the current execution is that the execution may not have explored a sufficient set of events to create the necessary backtracking points. In our context, event-driven applications are non-terminating. Similar to our multithreaded example, executions in event-driven applications may cause the algorithm to revisit a state and prematurely stop the exploration.

Our Idea. Since the applications we are interested in typically have cyclic state spaces, we address this challenge by changing our termination criteria for an execution to require that an execution either (1) matches a state from a previous execution or (2) matches a previously explored state from the current execution and has explored every enabled event in the cycle at least once since the first exploration of that state. The second criterion would prevent the DPOR algorithm from terminating prematurely after the exploration in Fig. 1-a.

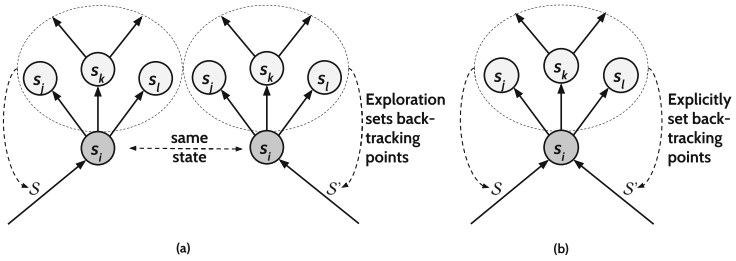

Fig. 2. (a) Stateless model checking explores s_i, s_j, s_k, and s_l twice and thus sets backtracking points for both $\mathcal{S}$ and $\mathcal{S}$'. (b) Stateful model checking matches state s_i and skips the second exploration and thus we must explicitly set backtracking points.

4.2 Problem 2: State Matching for Previously Explored States

Typically stateful model checkers can simply terminate an execution when a previously discovered state is reached. As mentioned in [61], this handling is unsound in the presence of dynamic partial order reduction. Figure 2 illustrates the issue: Fig. 2-a and b show the behavior of a classical stateless DPOR algorithm as well as the situation in a stateful DPOR algorithm, respectively. We assume that $\mathcal{S}$ was the first transition sequence to reach s_i and $\mathcal{S}$' was the second such transition sequence. The issue in Fig. 2-b is that after the state match for s_i in $\mathcal{S}$', the algorithm may *inappropriately* skip setting backtracking points for the transition sequence $\mathcal{S}$', preventing the model checker from completely exploring the state space.

Our Idea. Similar to the approach of Yang *et al.* [61], we propose to use a graph to store the set of previously explored transitions that may set backtracking points in the current transition sequence, so that the algorithm can set those backtracking points without reexploring the same state space.

4.3 Problem 3: State Matching Incompletely Explored States

Figure 3 illustrates another problem with cyclic state spaces—even if our new termination condition and the algorithm for setting backtrack points for a state match are applied to the stateful DPOR algorithm, it could still fail to explore all executions.

With our new termination criteria, the stateful DPOR algorithm will first explore the execution shown in Fig. 3-a. It starts from s_0 and executes the events $\mathtt{e}_1$, $\mathtt{e}_2$, and $\mathtt{e}_3$. While executing the three events, it puts event $\mathtt{e}_2$ in the backtrack set of s_0 and event $\mathtt{e}_3$ in the backtrack set of s_1 as it finds a conflict between the events $\mathtt{e}_1$ and $\mathtt{e}_2$, and the events $\mathtt{e}_2$ and $\mathtt{e}_3$. Then, the algorithm revisits s_1. At this point it updates the backtrack sets using the transitions that are reachable from state s_1: it puts event $\mathtt{e}_2$ in the backtrack set of state s_2 because of a conflict between $\mathtt{e}_2$ and $\mathtt{e}_3$.

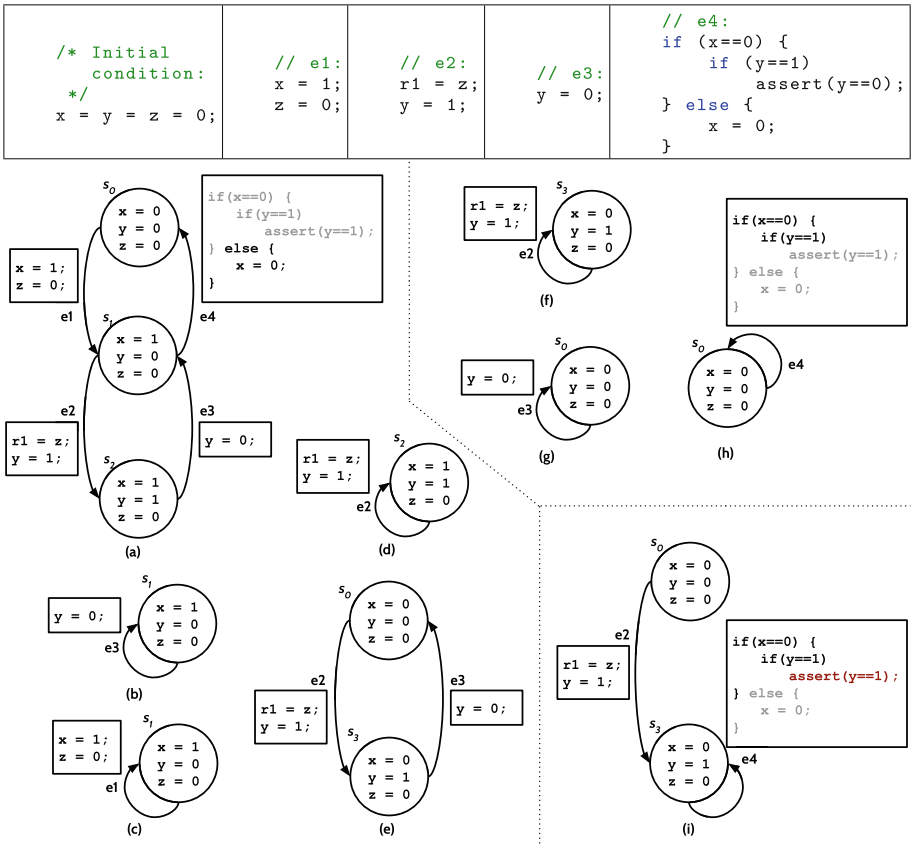

Fig. 3. Example of a event-driven program that misses an execution. We assume that `e1`, `e2`, `e3`, and `e4` are all initially enabled.

However, with the new termination criteria, it does not stop its exploration. It continues to execute event $\mathtt{e}_4$, finds a conflict between $\mathtt{e}_1$ and $\mathtt{e}_4$, and puts event `e4` into the backtrack set of s_0. The algorithm now revisits state s_0 and updates the backtrack sets using the transitions reachable from state s_0: it puts event $\mathtt{e}_1$ in the backtrack set of s_1 because of the conflict between $\mathtt{e}_1$ and $\mathtt{e}_4$. Figures 3-b, c, and d show the executions explored by the stateful DPOR algorithm from the events $\mathtt{e}_1$ and $\mathtt{e}_3$ in the backtrack set of s_1, and event $\mathtt{e}_2$ in the backtrack set of s_2, respectively.

Next, the algorithm explores the execution from event $\mathtt{e}_2$ in the backtrack set of s_0 shown in Fig. 3-e. The algorithm finds a conflict between the events $\mathtt{e}_2$ and $\mathtt{e}_3$, and it puts event $\mathtt{e}_2$ in the backtrack set of s_3 and event $\mathtt{e}_3$ in the backtrack set of s_0 whose executions are shown in Figs. 3-f and g, respectively. Finally, the algorithm explores the execution from event $\mathtt{e}_4$ in the backtrack set of s_0 shown

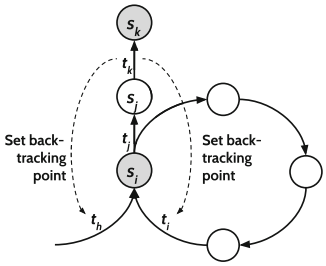

Fig. 4. Stateful model checking needs to handle loops caused by cyclic state spaces.

in Fig. 3-h. Then the algorithm stops, failing to explore the asserting execution shown in Fig. 3-i.

The key issue in the above example is that the stateful DPOR algorithm by Yang *et al.* [61] does not consider all possible transition sequences that can reach the current state but merely considers the current transition sequence when setting backtracking points. It thus does not add event $\mathtt{e}_4$ from the execution in Fig. 3-h to the backtrack set of state s_3.

Our Idea. Figure 4 shows the core issue behind the problem. When the algorithm sets backtracking points after executing the transition t_k, the algorithm must consider both the transition sequence that includes t_h and the transition sequence that includes t_i. The classical backtracking algorithm would only consider the current transition sequence when setting backtracking points.

We propose a new algorithm that uses a backwards depth first search on the state transition graph combined with summaries to set backtracking points on previously discovered paths to the currently executing transition. Yi *et al.* [63] uses a different approach for updating summary information to address this issue.

4.4 Problem 4: Events as Transitions

The fourth problem, also identified in Jensen *et al.* [22], is that existing stateful DPOR algorithms and most DPOR algorithms assume that each transition only executes a single memory operation, whereas an event in our context can consist of many different memory operations. For example, the $\mathtt{e}_4$ handler in Fig. 3 reads `x` and `y`.

A related issue is that many DPOR algorithms assume that they know, ahead of time, the effects of the next step for each thread. In our setting, however, since events contain many different memory operations, we must execute an event to know its effects. Figure 5 illustrates this problem. In this example, we assume that each event can only execute once.

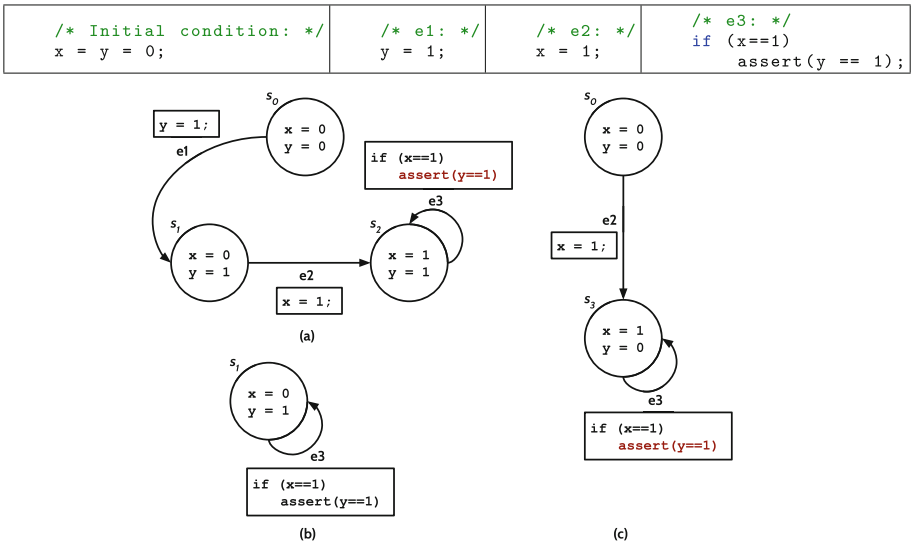

Fig. 5. Example of an event-driven program for which a naive application of the standard DPOR algorithm fails to construct the correct persistent set at state s_0. We assume that `e1`, `e2`, and `e3` are all initially enabled.

Figure 5-a shows the first execution of these 3 events. The stateful DPOR algorithm finds a conflict between the events $\mathtt{e}_2$ and $\mathtt{e}_3$, adds event $\mathtt{e}_3$ to the backtrack set for state s_1, and then schedules the second execution shown in Fig. 5-b. At this point, the exploration stops prematurely, missing the assertion violating execution shown in Fig. 5-c.

The key issue is that the set $\{\mathtt{e}_1\}$ is not a persistent set for state s_0. Traditional DPOR algorithms fail to construct the correct persistent set at state s_0 because the backtracking algorithm finds that the transition for event $\mathtt{e}_3$ conflicts with the transition for event $\mathtt{e}_2$ and stops setting backtracking points. This occurs since these algorithms do not separately track conflicts from different memory operations in an event when adding backtracking points—they simply assume transitions are comprised of single memory operations. Separately tracking different operations would allow these algorithms to find a conflict relation between the events $\mathtt{e}_1$ and $\mathtt{e}_3$ (as both access the variable `y`) in the first execution, put event $\mathtt{e}_2$ into the backtrack set of s_0, and explore the missing execution shown in Fig. 5-c.

Our Idea. In the classical DPOR algorithm, transitions correspond to single instructions whose effects can be determined ahead of time without executing the instructions [12]. Thus, the DPOR algorithm assumes that the effects of each thread's next transition are known. Our events on the other hand include many instructions, and thus, as Jensen *et al.* [22] observes, determining the effects of an event requires executing the event. Our algorithm therefore determines the effects of a transition when the transition is actually executed.

Algorithm 1: Top-level exploration algorithm.

```
1 ExploreAll()
2 |  H := ∅
3 |  R := ∅
4 |  S := ∅
5 |  Explore(s_0)
6 |  while ∃s, backtrack(s) ≠ done(s) do
7 |  |  Explore(s)
8 |  end
9 end
```

A second consequence of having events as transitions is that transitions can access multiple different memory locations. Thus, as the example in Fig. 5 shows, it does not suffice to simply set a backtracking point at the last conflicting transition. To address this issue, our idea is to compute conflicts on a *per-memory-location* basis.

5 Stateful Dynamic Partial Order Reduction

This section presents our algorithm, which extends DPOR to support stateful model checking of event-driven applications with cyclic state spaces. We first present the states that our algorithm maintains:

1. **The transition sequence** $\mathcal{S}$ contains the new transitions that the current execution explores. Our algorithm explores a given transition in at most one execution.
2. **The state history** $\mathcal{H}$ is a set of program states that have been visited in completed executions.
3. **The state transition graph** $\mathcal{R}$ records the states our algorithm has explored thus far. Nodes in this graph correspond to program states and edges to transitions between program states.

Recall that for each reachable state $s \in \mathit{States}$, our algorithm maintains the $\mathit{backtrack}(s)$ set that contains the events to be explored at s, the $\mathit{done}(s)$ set that contains the events that have already been explored at s, and the $\mathit{enabled}(s)$ set that contains all events that are enabled at s.

Algorithm 1 presents the top-level ExploreAll procedure. This procedure first invokes the Explore procedure to start model checking from the initial state. However, the presence of cycles in the state space means that our backtracking-based search algorithm may occasionally set new backtracking points for states in completed executions. The ExploreAll procedure thus loops over all states that have unexplored items in their backtrack sets and invokes the Explore procedure to explore those transitions.

Algorithm 2 describes the logic of the Explore procedure. The `if` statement in line 2 checks if the current state s's *backtrack* set is the same as the current

Algorithm 2: Stateful DPOR algorithm for event-driven applications.

```
1  EXPLORE(s)
2    if backtrack(s) = done(s) then
3      if done(s) = enabled(s) then
4        if enabled(s) is not empty then
5          select e ∈ enabled(s)
6          remove e from done(s)
7        else
8          add states(S) to H
9          S := ∅
10         return
11       end
12     else
13       select e ∈ enabled(s) \ done(s)
14       add e to backtrack(s)
15     end
16   end
17   while ∃b ∈ backtrack(s) \ done(s) do
18     add b to s.done
19     t := next(s, b)
20     s' := dst(t)
21     add transition t to R
22     foreach e ∈ enabled(s) \ enabled(s') do
23       add e to backtrack(s)
24     end
25     UPDATEBACKTRACKSET(t)
26     if s' ∈ H ∨ ISFULLCYCLE(t) then
27       UPDATEBACKTRACKSETSFROMGRAPH(t)
28       add states(S) to H
29       S := ∅
30     else
31       if s' ∈ states(S) then
32         UPDATEBACKTRACKSETSFROMGRAPH(t)
33       end
34       S := S.t
35       EXPLORE(s')
36     end
37   end
38 end
```

state s's *done* set. If so, the algorithm selects an event to execute in the next transition. If some enabled events are not yet explored, it selects an unexplored event to add to the current state's *backtrack* set. Otherwise, if the *enabled* set is not empty, it selects an enabled event to remove from the *done* set. Note that this scenario occurs only if the execution is continuing past a state match to satisfy the termination condition.

Then the `while` loop in line 17 selects an event b to execute on the current state s and executes the event b to generate the transition t that leads to a new state s'. At this point, the algorithm knows the memory accesses performed by the transition t and thus can add the event b to the backtrack sets of the previous states. This is done via the procedure UpdateBacktrackSet.

Traditional DPOR algorithms continue an execution until it terminates. Since our programs may have cyclic state spaces, this would cause the model checker to potentially not terminate. Our algorithm instead checks the conditions in line 26 to decide whether to terminate the execution. These checks see whether the new state s' matches a state from a previous execution, or if the current execution revisits a state the current execution previously explored and meets other criteria that are checked in the IsFullCycle procedure. If so, the algorithm calls the UpdateBacktrackSetsFromGraph procedure to set backtracking points, from transitions reachable from t, to states that can reach t. An execution will also terminate if it reaches a state in which no event is enabled (line 4). It then adds the states from the current transition sequence to the set of previously visited states $\mathcal{H}$, resets the current execution transition sequence $\mathcal{S}$, and backtracks to start a new execution.

If the algorithm has reached a state s' that was previously discovered in this execution, it sets backtracking points by calling the UpdateBacktrackSetsFromGraph procedure. Finally, it updates the transition sequence $\mathcal{S}$ and calls Explore.

Algorithm 3: Procedure that updates the backtrack sets of states in previous executions.

```
1 UpdateBacktrackSetsFromGraph(t_s)
2 |  R_t := {t ∈ R | t_s →_r t}
3 |  foreach t ∈ R_t do
4 |  |  UpdateBacktrackSet(t)
5 |  end
6 end
```

Algorithm 3 shows the UpdateBacktrackSetsFromGraph procedure. This procedure takes a transition t that connects the current execution to a previously discovered state in the transition graph $\mathcal{R}$. Since our algorithm does *not* explore all of the transitions reachable from the previously discovered state, we need to set the backtracking points that would have been set by these skipped transitions. This procedure therefore computes the set of transitions reachable from the destination state of t and invokes UpdateBacktrackSet on each of those transitions to set backtracking points.

Algorithm 4: Procedure that checks the looping termination condition: a cycle that contains every event enabled in the cycle.

```
1 IsFullCycle(t)
2   if ¬dst(t) ∈ states(S) then
3     return false
4   end
5   S^fc := {t_j ∈ S | i = first(S, dst(t)), and i < j} ∪ {t}
6   E_fc := {event(t') | ∀t' ∈ S^fc }
7   E_enabled := {enabled(dst(t')) | ∀t' ∈ S^fc }
8   return E_fc = E_enabled
9 end
```

Algorithm 4 presents the IsFullCycle procedure. This procedure first checks if there is a cycle that contains the transition t in the state space explored by the current execution. The example from Fig. 1 shows that such a state match is not sufficient to terminate the execution as the execution may not have set the necessary backtracking points. Our algorithm stops the exploration of an execution when there is a cycle that has explored *every event that is enabled in that cycle*. This ensures that for every transition t in the execution, there is a future transition t_e for each enabled event e in the cycle that can set a backtracking point if t and t_e conflict.

Algorithm 5 presents the UpdateBacktrackSet procedure, which sets backtracking points. There are two differences between our algorithm and traditional DPOR algorithms. First, since our algorithm supports programs with cyclic state spaces, it is possible that the algorithm has discovered multiple paths from the start state s_0 to the current transition t. Thus, the algorithm must potentially set backtracking points on multiple different paths. We address this issue using a backwards depth first search traversal of the $\mathcal{R}$ graph. Second, since our transitions correspond to events, they may potentially access multiple different memory locations and thus the backtracking algorithm potentially needs to set separate backtracking points for each of these memory locations.

The UpdateBacktrackSetDFS procedure implements a backwards depth first traversal to set backtracking points. The procedure takes the following parameters: t_{curr} is the current transition in the DFS, t_{conf} is the transition that we are currently setting a backtracking point for, $\mathcal{A}$ is the set of accesses that the algorithm searches for conflicts for, and $\mathcal{T}_{\text{exp}}$ is the set of transitions that the algorithm has explored down this search path. Recall that accesses consist of both an operation, *i.e.*, a read or write, and a memory location. Conflicts are defined in the usual way—writes to a memory location conflict with reads or writes to the same location.

Algorithm 5: Procedure that updates the backtrack sets of states for previously executed transitions that conflict with the current transition in the search stack.

```
1  UpdateBacktrackSet(t)
2  |  UpdateBacktrackSetDFS(t, t, accesses(t), {t})
3  end
4  UpdateBacktrackSetDFS(t_curr, t_conf, A, T_exp)
5  |  foreach t_b ∈ pred_R(t_curr) \ T_exp do
6  |  |  A_b := accesses(t_b)
7  |  |  t_conf' := t_conf
8  |  |  if ∃a ∈ A, ∃a_b ∈ A_b, conflicts(a, a_b) then
9  |  |  |  if event(t_conf) ∈ enabled(src(t_b)) then
10 |  |  |  |  add event(t_conf) to backtrack(src(t_b))
11 |  |  |  else
12 |  |  |  |  add enabled(src(t_b)) to backtrack(src(t_b))
13 |  |  |  end
14 |  |  |  t_conf' := t_b
15 |  |  end
16 |  |  A_r := {a ∈ A | ¬∃a_b ∈ A_b, conflicts(a, a_b)}
17 |  |  UpdateBacktrackSetDFS(t_b, t_conf', A_r, T_exp ∪ {t_b})
18 |  end
19 end
```

Line 5 loops over each transition t_b that immediately precedes transition t_{curr} in the state transition graph and has not been explored. Line 8 checks for conflicts between the accesses of t_b and the access set $\mathcal{A}$ for the DFS. If a conflict is detected, the algorithm adds the event for transition t_{conf} to the backtrack set. Line 16 removes the accesses that conflicted with transition t_b. The search procedure then recursively calls itself. If the current transition t_b conflicts with the transition t_{conf} for which we are setting a backtracking point, then it is possible that the behavior we are interested in for t_{conf} requires that t_b be executed first. Thus, if there is a conflict between t_b and t_{conf}, we pass t_b as the conflict transition parameter to the recursive calls to UpdateBacktrackSetDFS.

Appendix B in [51] proves correctness properties for our DPOR algorithm. Appendix C in [51] revisits the example shown in Fig. 3. It describes how our DPOR algorithm explores all executions in Fig. 3, including Fig. 3-i.

6 Implementation and Evaluation

In this section, we present the implementation of our DPOR algorithm (Sect. 6.1) and its evaluation results (Sect. 6.2).

6.1 Implementation

We have implemented the algorithm by extending IoTCheck [49], a tool that model-checks pairs of Samsung's SmartThings smart home apps and reports

conflicting updates to the same device or global variables from different apps. IoTCheck extends Java Pathfinder, an explicit stateful model checker [56]. In the implementation, we optimized our DPOR algorithm by caching the results of the graph search when UpdateBacktrackSetsFromGraph is called. The results are cached for each state as a summary of the potentially conflicting transitions that are reachable from the given state (see Appendix D in [51]).

We selected the SmartThings platform because it has an extensive collection of event-driven apps. The SmartThings official GitHub [45] has an active user community—the repository has been forked more than 84,000 times as of August 2021.

We did not compare our implementation against other systems, *e.g.*, event-driven systems [22,30]. Not only that these systems do not perform stateful model checking and handle cyclic state spaces, but also they implemented their algorithms in different domains: web [22] and Android applications [30]—it will not be straightforward to adapt and compare these with our implementation on smart home apps.

6.2 Evaluation

Dataset. Our SmartThings app corpus consists of 198 official and third-party apps that are taken from the IoTCheck smart home apps dataset [48,49]. These apps were collected from different sources, including the official SmartThings GitHub [45]. In this dataset, the authors of IoTCheck formed pairs of apps to study the interactions between the apps [49].

We selected the 1,438 pairs of apps in the Device Interaction category as our benchmarks set. It contains a diverse set of apps and app pairs that are further categorized into 11 subgroups based on various device handlers [44] used in each app. For example, the `FireCO2Alarm` [38] and the `Lock-It-When-I-Leave` [1] apps both control and may interact through a door lock (see Sect. 1). Hence, they are both categorized as a pair in the `Locks` group. As the authors of IoTCheck noted, these pairs are challenging to model check—IoTCheck did not finish for 412 pairs.

Pair Selection. In the IoTCheck evaluation, the authors had to exclude 175 problematic pairs. In our evaluation, we further excluded pairs. First, we excluded pairs that were reported to finish their executions in 10 s or less—these typically will generate a small number of states (*i.e.*, less than 100) when model checked. Next, we further removed redundant pairs across the different 11 subgroups. An app may control different devices, and thus they may use various device handlers in its code. For example, the apps `FireCO2Alarm` [38] and `groveStreams` [39] both control door locks and thermostats in their code. Thus, the two apps are categorized as a pair both in the `Locks` and `Thermostats` subgroups—we need to only include this pair once in our evaluation. These steps reduced our benchmarks set to 535 pairs.

Experimental Setup. Each pair was model checked on an Ubuntu-based server with Intel Xeon quad-core CPU of 3.5 GHz and 32 GB of memory—we allocated

Table 1. Sample model-checked pairs that finished with or without DPOR. **Evt.** is number of events and **Time** is in seconds. The complete list of results for 229 pairs that finished with or without DPOR is included in Table A.2 in Appendices in [51].

No.	App	Evt.	Without DPOR			With DPOR		
			States	Trans.	Time	States	Trans.	Time
1	smart-nightlight–ecobeeAwayFromHome	14	16,441	76,720	5,059	11,743	46,196	5,498
2	step-notifier–ecobeeAwayFromHome	11	14,401	52,800	4,885	11,490	35,142	5,079
3	smart-security–ecobeeAwayFromHome	11	14,301	47,608	4,385	8,187	21,269	2,980
4	keep-me-cozy–whole-house-fan	17	8,793	149,464	4,736	8,776	95,084	6,043
5	keep-me-cozy-ii–thermostat-window-check	13	8,764	113,919	4,070	7,884	59,342	4,515
6	step-notifier–mini-hue-controller	6	7,967	47,796	2,063	7,907	40,045	3,582
7	keep-me-cozy–thermostat-mode-director	12	7,633	91,584	3,259	6,913	49,850	3,652
8	lighting-director–step-notifier	14	7,611	106,540	5,278	2,723	25,295	2,552
9	smart-alarm–DeviceTamperAlarm	15	5,665	84,960	3,559	3,437	40,906	4,441
10	forgiving-security–smart-alarm	13	5,545	72,072	3,134	4,903	52,205	5,728

28 GB of heap space for JVM. In our experiments, we ran the model checker for every pair for at most 2 h. We found that the model checker usually ran out of memory for pairs that had to be model checked longer. Further investigation indicates that these pairs generate too many states even when run with the DPOR algorithm. We observed that many smart home apps generate substantial numbers of *read-write* and *write-write* conflicts when model checked—this is challenging for any DPOR algorithms. In our benchmarks set, 300 pairs finished for DPOR and/or no DPOR.

Results. Our DPOR algorithm substantially reduced the search space for many pairs. There are 69 pairs that were *unfinished* (*i.e.*,"Unf") without DPOR. These pairs did not finish because their executions exceeded the 2-h limit, or generated too many states quickly and consumed too much memory, causing the model checker to run out of memory within the first hour of their execution. When run with our DPOR algorithm, these pairs successfully finished—mostly in 1 h or less. Table A.1 in Appendices in [51] shows the results for pairs that finished with DPOR but did not finish without DPOR. Most notably, even for the pair `initial-state-event-streamer`—`thermostat-auto-off` that has the most number of states, our DPOR algorithm successfully finished model checking it within 1 h.

Next, we discovered that 229 pairs finished when model checked with and without DPOR. Table 1 shows 10 pairs with the most numbers of states (see the complete results in Table A.2 in Appendices in [51]). These pairs consist of apps that generate substantial numbers of *read-write* and *write-write* conflicts when model checked with our DPOR algorithm. Thus, our DPOR algorithm did not significantly reduce the states, transitions, and runtimes for these pairs.

Finally, we found 2 pairs that finished when run without our DPOR algorithm, but did not finish when run with it. These pairs consist of apps that are exceptionally challenging for our DPOR algorithm in terms of numbers of

read-write and *write-write* conflicts. Nevertheless, these are corner cases—please note that our DPOR algorithm is effective in many pairs.

Overall, our DPOR algorithm achieved a **2**× state reduction and a **3**× transition reduction for the 229 pairs that finished for both DPOR and no DPOR (geometric mean). Assuming that "Unf" is equal to 7,200 s (*i.e.*, 2 h) of runtime, we achieved an overall speedup of **7**× for the 300 pairs (geometric mean). This is a lower bound runtime for the "Unf" cases, in which executions exceeded the 2-h limit—these pairs could have taken more time to finish.

7 Related Work

There has been much work on model checking. Stateless model checking techniques do not explicitly track which program states have been visited and instead focus on enumerating schedules [13–15,33].

To make model checking more efficient, researchers have also looked into various partial order reduction techniques. The original partial order reduction techniques (*e.g.*, persistent/stubborn sets [13,53] and sleep sets [13]) can also be used in the context of cyclic state spaces when combined with a proviso that ensures that executions are not prematurely terminated [13], and ample sets [7,8] that are basically persistent sets with additional conditions. However, the persistent/stubborn set techniques "suffer from severe fundamental limitation" [12]: the operations and their communication objects in future process executions are difficult or impossible to compute precisely through static analysis, while sleep sets alone only reduce the number of transitions (not states). Work on *collapses* by Katz and Peled also suffers from the same requirement for a statically known independence relation [23].

The first DPOR technique was proposed by Flanagan and Godefroid [12] to address those issues. The authors introduced a technique that combats the state space explosion by detecting *read-write* and *write-write* conflicts on shared variable on the fly. Since then, a significant effort has been made to further improve dynamic partial order reduction [26,41–43,47]. Unfortunately, a lot of DPOR algorithms assume the context of shared-memory concurrency in that each transition consists of a single memory operation. In the context of event-driven applications, each transition is an event that can consist of different memory operations. Thus, we have to execute the event to know its effects and analyze it dynamically on the fly in our DPOR algorithm (see Sect. 4.4).

Optimal DPOR [2] seeks to make stateless model checking more efficient by skipping equivalent executions. Maximal causality reduction [19] further refines the technique with the insight that it is only necessary to explore executions in which threads read different values. Value-centric DPOR [6] has the insight that executions are equivalent if all of their loads read from the same writes. Unfolding [40] is an alternative approach to POR for reducing the number of executions to be explored. The unfolding algorithm involves solving an NP-complete problem to add events to the unfolding.

Recent work has extended these algorithms to handle the TSO and PSO memory models [3,20,64] and the release acquire fragment of C/C++ [4]. The

RCMC tool implements a DPOR tool that operates on execution graphs for the RC11 memory model [24]. SAT solvers have been used to avoid explicitly enumerating all executions. SATCheck extends partial order reduction with the insight that it is only necessary to explore executions that exhibit new behaviors [9]. CheckFence checks code by translating it into SAT [5]. Other work has also presented techniques orthogonal to DPOR, either in a more general context [10] or platform specific (*e.g.*, Android [36] and Node.js [29]).

Recent work on dynamic partial order reduction for event-driven programs has developed dynamic partial order reduction algorithms for stateless model checking of event-driven applications [22,30]. Jensen *et al.* [22] consider a model similar to ours in which an event is treated as a single transition, while Maiya *et al.* [30] consider a model in which event execution interleaves concurrently with threads. Neither of these approaches handle cyclic state spaces nor consider challenges that arise from stateful model checking.

Recent work on DPOR algorithms reduces the number of executions for programs with critical sections by considering whether critical sections contain conflicting operations [25]. This work considers stateless model checking of multi-threaded programs, but like our work it must consider code blocks that perform multiple memory operations.

CHESS [33] is designed to find and reproduce concurrency bugs in C, C++, and C#. It systematically explores thread interleavings using a preemption bounded strategy. The Inspect tool combines stateless model checking and stateful model checking to model check C and C++ code [57,60,62].

In stateful model checking, there has also been substantial work such as SPIN [18], Bogor [11], and JPF [56]. In addition to these model checkers, other researchers have proposed different techniques to capture program states [17,32].

Versions of JPF include a partial order reduction algorithm. The design of this algorithm is not well documented, but some publications have reverse engineered the pseudocode [34]. The algorithm is naive compared to modern DPOR algorithms—this algorithm simply identifies accesses to shared variables and adds backtracking points for all threads at any shared variable access.

8 Conclusion

In this paper, we have presented a new technique that combines dynamic partial order reduction with stateful model checking to model check event-driven applications with cyclic state spaces. To achieve this, we introduce two techniques: a new termination condition for looping executions and a new algorithm for setting backtracking points. Our technique is the first stateful DPOR algorithm that can model check event-driven applications with cyclic state spaces. We have evaluated this work on a benchmark set of smart home apps. Our results show that our techniques effectively reduce the search space for these apps. An extended version of this paper, including appendices, can be found in [51].

Acknowledgment. We would like to thank our anonymous reviewers for their thorough comments and feedback. This project was supported partly by the National Science Foundation under grants CCF-2006948, CCF-2102940, CNS-1703598, CNS-1763172, CNS-1907352, CNS-2006437, CNS-2007737, CNS-2106838, CNS-2128653, OAC-1740210 and by the Office of Naval Research under grant N00014-18-1-2037.

References

1. Lock it when i leave (2015). https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/lock-it-when-i-leave.src/lock-it-when-i-leave.groovy
2. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: Proceedings of the 2014 Symposium on Principles of Programming Languages, pp. 373–384 (2014). http://doi.acm.org/10.1145/2535838.2535845
3. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 353–367 (2015). http://link.springer.com/chapter/10.1007
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. **2**(OOPSLA) (2018). https://doi.org/10.1145/3276505
5. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: checking consistency of concurrent data types on relaxed memory models. In: Proceedings of the 2007 Conference on Programming Language Design and Implementation, pp. 12–21 (2007). http://doi.acm.org/10.1145/1250734.1250737
6. Chatterjee, K., Pavlogiannis, A., Toman, V.: Value-centric dynamic partial order reduction. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360550
7. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. Int. J. Softw. Tools Technol. Transf. **2**(3), 279–287 (1999)
8. Clarke Jr, E.M., Grumberg, O., Peled, D.: Model Checking. MIT press, Cambridge (1999)
9. Demsky, B., Lam, P.: SATCheck: SAT-directed stateless model checking for SC and TSO. In: Proceedings of the 2015 Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 20–36 (October 2015). http://doi.acm.org/10.1145/2814270.2814297
10. Desai, A., Qadeer, S., Seshia, S.A.: Systematic testing of asynchronous reactive systems. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 73–83 (2015)
11. Dwyer, M.B., Hatcliff, J.: Bogor: an extensible and highly-modular software model checking framework. ACM SIGSOFT Softw. Eng. Notes **28**(5), 267–276 (2003)
12. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. ACM Sigplan Not. **40**(1), 110–121 (2005)
13. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag, Berlin, Heidelberg (1996)
14. Godefroid, P.: Model checking for programming languages using verisoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 174–186 (1997)

15. Godefroid, P.: Software model checking: the verisoft approach. Form. Methods Syst. Des. **26**(2), 77–101 (2005)
16. Google: Android things website (2015). https://developer.android.com/things/
17. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73370-6_8
18. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual, vol. 1003 (2003)
19. Huang, J.: Stateless model checking concurrent programs with maximal causality reduction. In: Proceedings of the 2015 Conference on Programming Language Design and Implementation, pp. 165–174 (2015). http://doi.acm.org/10.1145/2813885.2737975
20. Huang, S., Huang, J.: Maximal causality reduction for TSO and PSO. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 447–461 (2016). http://doi.acm.org/10.1145/2983990.2984025
21. IFTTT: IFTTT (September 2011). https://www.ifttt.com/
22. Jensen, C.S., Møller, A., Raychev, V., Dimitrov, D., Vechev, M.: Stateless model checking of event-driven applications. ACM SIGPLAN Not. **50**(10), 57–73 (2015)
23. Katz, S., Peled, D.: Defining conditional independence using collapses. Theor. Comput. Sci. **101**(2), 337–359 (1992)
24. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. **2**(POPL) (2017). https://doi.org/10.1145/3158105
25. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Effective lock handling in stateless model checking. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360599
26. Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Evaluating ordering heuristics for dynamic partial-order reduction techniques. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 308–322. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12029-9_22
27. Li, X., Zhang, L., Shen, X.: IA-graph based inter-app conflicts detection in open IoT systems. In: Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 135–147 (2019)
28. Li, X., Zhang, L., Shen, X., Qi, Y.: A systematic examination of inter-app conflicts detections in open IoT systems. Technical report TR-2017-1, North Carolina State University, Dept. of Computer Science (2017)
29. Loring, M.C., Marron, M., Leijen, D.: Semantics of asynchronous Javascript. In: Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, pp. 51–62 (2017)
30. Maiya, P., Gupta, R., Kanade, A., Majumdar, R.: Partial order reduction for event-driven multi-threaded programs. In: Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 16) (2016)
31. Mazurkiewicz, A.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) ACPN 1986. LNCS, vol. 255, pp. 278–324. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-17906-2_30
32. Musuvathi, M., Park, D.Y., Chou, A., Engler, D.R., Dill, D.L.: CMC: a pragmatic approach to model checking real code. ACM SIGOPS Oper. Syst. Rev. **36**(SI), 75–88 (2002)

33. Musuvathi, M., Qadeer, S., Ball, T.: Chess: a systematic testing tool for concurrent software (2007)
34. Noonan, E., Mercer, E., Rungta, N.: Vector-clock based partial order reduction for JPF. SIGSOFT Softw. Eng. Notes **39**(1), 1–5 (2014)
35. openHAB: openhab website (2010). https://www.openhab.org/
36. Ozkan, B.K., Emmi, M., Tasiran, S.: Systematic asynchrony bug exploration for android apps. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 455–461. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_28
37. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Proceedings of the International Conference on Computer Aided Verification, pp. 377–390 (1994)
38. Racine, Y.: Fireco2alarm smartapp (2014). https://github.com/yracine/device-type.myecobee/blob/master/smartapps/FireCO2Alarm.src/FireCO2Alarm.groovy
39. Racine, Y.: grovestreams smartapp (2014). https://github.com/uci-plrg/iotcheck/blob/master/smartapps/groveStreams.groovy
40. Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based partial order reduction. In: CONCUR (2015)
41. Saarikivi, O., Kähkönen, K., Heljanko, K.: Improving dynamic partial order reductions for concolic testing. In: 2012 12th International Conference on Application of Concurrency to System Design, pp. 132–141. IEEE (2012)
42. Sen, K., Agha, G.: Automated systematic testing of open distributed programs. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 339–356. Springer, Heidelberg (2006). https://doi.org/10.1007/11693017_25
43. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 166–182. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70889-6_13
44. SmartThings: Device handlers (2018). https://docs.smartthings.com/en/latest/device-type-developers-guide/
45. SmartThings: Smartthings public github repo (2018). https://github.com/SmartThingsCommunity/SmartThingsPublic
46. SmartThings, S.: Samsung smartthings website (2012). http://www.smartthings.com
47. Tasharofi, S., Karmani, R.K., Lauterburg, S., Legay, A., Marinov, D., Agha, G.: TransDPOR: a novel dynamic partial-order reduction technique for testing actor programs. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE -2012. LNCS, vol. 7273, pp. 219–234. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30793-5_14
48. Trimananda, R., Aqajari, S.A.H., Chuang, J., Demsky, B., Xu, G.H., Lu, S.: Iotcheck supporting materials (2020). https://github.com/uci-plrg/iotcheck-data/tree/master/Device
49. Trimananda, R., Aqajari, S.A.H., Chuang, J., Demsky, B., Xu, G.H., Lu, S.: Understanding and automatically detecting conflicting interactions between smart home IoT applications. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (November 2020)
50. Trimananda, R., Luo, W., Demsky, B., Xu, G.H.: Iotcheck dpor (2021). https://github.com/uci-plrg/iotcheck-dpor, https://doi.org/10.5281/zenodo.5168843, https://zenodo.org/record/5168843#.YQ8KjVNKh6c

51. Trimananda, R., Luo, W., Demsky, B., Xu, G.H.: Stateful dynamic partial order reduction for model checking event-driven applications that do not terminate. arXiv preprint arXiv:2111.05290 (2021)
52. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0023729
53. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) ICATPN 1989. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53863-1_36
54. Vicaire, P.A., Hoque, E., Xie, Z., Stankovic, J.A.: Bundle: a group-based programming abstraction for cyber-physical systems. IEEE Trans. Ind. Inf. **8**(2), 379–392 (2012)
55. Vicaire, P.A., Xie, Z., Hoque, E., Stankovic, J.A.: Physicalnet: a generic framework for managing and programming across pervasive computing networks. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE, pp. 269–278. IEEE (2010)
56. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs **10**, 203–232 (2003)
57. Wang, C., Yang, Yu., Gupta, A., Gopalakrishnan, G.: Dynamic model checking with property driven pruning to detect race conditions. In: Cha, S.S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 126–140. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88387-6_11
58. Wood, A.D., et al.: Context-aware wireless sensor networks for assisted living and residential monitoring. IEEE Netw. **22**(4) (2008)
59. Yagita, M., Ishikawa, F., Honiden, S.: An application conflict detection and resolution system for smart homes. In: Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems, pp. 33–39. SEsCPS 2015, IEEE Press, Piscataway, NJ, USA (2015). http://dl.acm.org/citation.cfm?id=2821404.2821413
60. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Distributed dynamic partial order reduction based verification of threaded software. In: Proceedings of the 14th International SPIN Conference on Model Checking Software, pp. 58–75 (2007)
61. Yang, Yu., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient stateful dynamic partial order reduction. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 288–305. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85114-1_20
62. Yang, Y., Chen, X., Gopalakrishnan, G., Wang, C.: Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In: Proceedings of the 16th International SPIN Workshop on Model Checking Software, pp. 279–295 (2009)
63. Yi, X., Wang, J., Yang, X.: Stateful dynamic partial-order reduction. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 149–167. Springer, Heidelberg (2006). https://doi.org/10.1007/11901433_9
64. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 250–259 (2015). http://doi.acm.org/10.1145/2737924.2737956