Underspecified Harnesses and Interleaved Bugs

Saurabh Joshi  
IIT Kanpur, Kanpur, India  
sbjoshi@cse.iitk.ac.in

Shuvendu K. Lahiri  
Microsoft Research, Redmond, WA, USA  
shuvendu@microsoft.com

Akash Lal  
Microsoft Research, Bangalore, India  
akashi@microsoft.com

Abstract

Static assertion checking of open programs requires setting up a precise harness to capture the environment assumptions. For instance, a library may require a file handle to be properly initialized before it is passed into it. A harness is used to set up or specify the appropriate preconditions before invoking methods from the program. In the absence of a precise harness, even the most precise automated static checkers are bound to report numerous false alarms. This often limits the adoption of static assertion checking in the hands of a user.

In this work, we explore the possibility of automatically filtering away (or prioritizing) warnings that result from imprecision in the harness. We limit our attention to the scenario when one is interested in finding bugs due to concurrency. We define a warning to be an interleaved bug when it manifests on an input for which no sequential interleaving produces a warning. As we argue in the paper, limiting a static analysis to only consider interleaved bugs greatly reduces false positives during static concurrency analysis in the presence of an imprecise harness.

We formalize interleaved bugs as a differential analysis between the original program and its sequential version and provide various techniques for finding them. Our implementation CBUGS demonstrates that the scheme of finding interleaved bugs can alleviate the need to construct precise harnesses while checking real-life concurrent programs.

Categories and Subject Descriptors  D.2.4 [Software Engineering]: Software/Program Verification

General Terms  Verification, Reliability

Keywords  Concurrency verification, differential analysis, static analysis, false alarms

1. Introduction

Static analysis is concerned with the problem of finding bugs (or proving their absence) in code without actually running the code. In this paper, we apply static analysis to open programs or libraries (i.e., programs that do not have a main procedure, but instead expose a set of API methods). In such a setting, the user of static analysis has to provide a harness that invokes appropriate methods from the library.

Often times, the exercise of writing a correct harness is difficult and error prone when done manually. One reason for this difficulty is that the libraries usually have undocumented preconditions that must be satisfied before invoking particular methods. For instance, before invoking a read operation on a file, the library may assume that file handle is properly initialized. If one applies static analysis to such a library without the precondition, the read operation can fail, and the static analysis will report a bug. However, this does not reveal any bug in the implementation of the library; instead, it only reveals a problem with the harness that was used to invoke the library. Thus, no matter how precise the static analysis is, it is bound to report false alarms while using underspecified harnesses. In our experience, the problem of false alarms (including those due to underspecified harness) severely limits the adoption of static analysis tools that aim at checking user-defined assertions in programs.

The problem of spurious warnings due to underspecified harness does not get better while checking concurrent programs. However, there is a very natural option for prioritizing false alarms in order to look for the more interesting warnings:

Find violations to assertions assuming that the sequential executions of the concurrent program (under a correct harness) do not violate any assertion.

This assumption about the sequential executions being "safe" may be justified because a program is more likely to be thoroughly tested for interesting inputs rather than for different interleavings (as it is beyond the control of a tester). This assumption immediately leads to pruning of the set of inputs to the program: if an input i fails an assertion in the program in a sequential interleaving, then it has to be an illegal input (that should have been filtered by a missing precondition in the harness). Hence, we are left to search for over the space of inputs for which no sequential execution violates an assertion. If an input in this space violates an assertion, it has to manifest in a complex interleaving of threads. We term bugs that manifest on such inputs as interleaved bugs. The static analysis problem then becomes that of finding interleaved bugs instead of finding all bugs.

Formally, a concurrent program P has an interleaved bug if some assertion in the program fails for an input i for which no sequential interleaving of threads in P result in an assertion failure.\footnote{We formally define the notion of sequential interleavings in §4 as one in which threads do not interfere with each other.}

Interleaved bugs might still be spurious. However, we make the following conjecture based on looking at a large number of spurious warnings:

If an illegal input (due to an underspecified harness) violates an assertion, then it does so in a fairly simple execution.

More precisely, whenever an illegal input leads to an assertion failure, there would be a sequential interleaving witnessing this failure.
We formalize the problem as a differential analysis of two programs. For two programs $P_1$ and $P_2$, DIFFERROR($P_1$, $P_2$) is the problem of finding an input $i$ such that $P_2$ can fail when started with $i$ but $P_1$ cannot. In some sense, $P_1$ acts as a filter for $P_2$: run $P_2$ on an input only when $P_1$ doesn’t fail on it.

Let $P$ be the concurrent program under test. Let $P_2$ be the same as $P$ but executions of $P_2$ are restricted to be sequential, i.e., $P_2$ can have multiple threads, but it does not interleave them. We formally describe how to capture $P_2$ as a program in §4. Then, finding interleaved bugs is the same as solving DIFFERROR($P$, $P_2$). Although solving all instances of DIFFERROR is infeasible (it is undecidable), we provide techniques (§3) that allow us to find interleaved bugs in many real-world programs.

We also show that one can choose various interesting underapproximations of $P_2$ for the purpose of proving the absence of interleaved bugs; the various choices have impact on the performance of the analysis.

One technical complication is that DIFFERROR($P_1$, $P_2$) is harder when the programs are non-deterministic, i.e., when they have multiple possible executions for a given input. As with any static-analysis tool, non-determinism is unavoidable — it comes from modeling of external calls as well as the thread scheduler. We give techniques to handle this difficulty.

We have implemented our algorithms for finding interleaved bugs in a tool called CBUGS that uses PIOROT [17], an SMT-based bug finder, as the static analysis tool. We evaluated CBUGS on (real and concurrent) Windows device drivers. Examples of false alarms and real bugs in these drivers can be found in §2.

This paper makes the following contributions:

1. We define interleaved bugs for assertion checking of concurrent programs (§4), and illustrate its role in reducing false alarms due to underspecified harnesses.
2. We describe the problem of finding interleaved bugs as an instance of the DIFFERROR problem for comparing two non-deterministic programs with respect to a set of assertions, and various techniques for solving it (§3).
3. Our experiments (§5) show that CBugs is able to remove all of the false alarms due to missing preconditions, and retain true concurrency bugs.

The rest of the paper is organized as follows. In §2, we present real-world examples to justify our observation on underspecified harnesses for concurrent programs. In §3, we formalize the more abstract problem DIFFERROR and present algorithms for solving it. In §4, we apply DIFFERROR for finding interleaved bugs, and discuss specific filters and optimizations. In §5, we evaluate our approach experimentally. In §6, we mention related work.

2. Motivation

We motivate our work with a few real-world examples of checking properties of concurrent programs, where (a) missing environment assumptions result in spurious warnings, (b) the spurious warnings often manifest in sequential traces, and (c) our technique only reports true concurrency bugs without requiring the user to specify environment assumptions.

2.1 Example 1

Figure 1 illustrates a simplified version of a bug found by the STORM [15] tool in a Microsoft Windows device driver usbamp [22]. This particular bug is an instance of "use-after-free" class of bugs where an object is accessed after it has been destroyed. The method UsbSampEvtIoRead is a dispatch routine that handles read requests sent to this driver — denoted by the Request parameter. Among other things, the method makes a call to WdfRequestMarkCancelable with the request and a cancel method UsbSampEvtRequestCancel; one of the side-effects of the call is to set a field Request->cancelRoutine to UsbSampEvtRequestCancel. After this call, the cancel method may be called asynchronously on Request by the device driver to cancel the request by invoking the Cancel routine. Similarly, the method WdfRequestUnmarkCancelable sets the Request->cancelRoutine to NULL, disabling the cancel routine from being invoked. The request is completed (or destroyed) by a call to WdfRequestComplete method.

The cancel method UsbSampEvtRequestCancel accesses fields in the request by a call to GetRequestContext. It is the responsibility of the driver developer to ensure that a request is not accessed after being completed. Several guidelines such as the following are provided for driver writers — "If a driver has called WdfRequestMarkCancelable, and if the driver's EvtRequestCancel callback function has not executed and called WdfRequestComplete, the driver must call WdfRequestUnmarkCancelable before it calls WdfRequestComplete outside of the EvtRequestCancel callback function."

The use-after-free property can be modeled by (a) introducing a ghost field completed for each request; it is set to true by a call to WdfRequestComplete, and (b) guarding every access to a request by the assertion assert(!!completed(Request)). These assertions are implicitly present in this program before the calls to the methods WdfRequestMarkCancelable, WdfRequestComplete and GetRequestContext that access fields in Request.

The routine Test is the harness (or the test driver); it invokes a set of procedure calls (in BODY:).

Some of these procedures may
be executed asynchronously by spawning new threads (using the async keyword). In addition to the method calls, the user has to additionally constrain the inputs to the methods. This can be done by either constructing objects that are passed on to the methods, or specifying some constraints on the inputs. The (commented out) "assume" statement in the test is one such constraint that has to be manually specified to model the environment assumptions for the open program. Let us understand the need for these assume statements by looking at the warnings reported by a precise analysis for this program.

We describe an interleaved trace by a sequence of events \(\langle (1, 2), (1, 7)\rangle\), where \(tsd_i \in \{1, 2, \ldots\}\) is a thread identifier and \(l_i\) is the line number of event. We sometimes avoid specifying all the events in the trace and specify only the end points of an uninterrupted context in each thread.

1. The trace \(\langle (1, 2), (1, 7)\rangle\) would cause an assertion failure for the access to Request in WdfRequestMarkCancelable. However, this is a spurious warning because the read routine is called with incomplete requests in any legal execution.

2. The trace \(\langle (2, 13), (2, 15), (2, 7)\rangle\) is another spurious warning that may happen if Request->cancelRoutine is set to UabSampEvtRequestCancel, and Request is completed in the input to Test.

3. The trace \(\langle (1, 2), (1, 7), (2, 13), (2, 15), (1, 11), (1, 14), (2, 7)\rangle\) denotes a feasible trace that may cause the device driver to access a completed request in line 7 in the second thread. This was the bug reported by STORM [15] and has been fixed in future versions of the device driver. Note that the bug manifests in a small window — the instruction (2, 13) has to be executed between (1, 7) and (1, 11).

It is easy to observe the only the third warning is an interleaved bug, i.e., it cannot manifest on any sequential interleaving of the threads. On the other hand, the first two manifest in a sequential interleaving. Our method only reports the third alarm (without the need for environment assumption in line 7 in Test that rules out the first two warnings), thereby reducing false alarms automatically. On the other hand, if we use a strategy to suppress any error location that can fail sequentially, we will be unable to discover this bug.

### 2.2 Example 2

Figure 2 shows another example where failure to specify environment assumptions may cause large set of false alarms. The example is a simplified version of the serial device driver [22].

The example consists of two threads running two dispatch routines for write (SerialWrite) and read (SerialRead) respectively. The PIRP structure denotes a pointer to an interrupt request packet (IRP) or a request, and we check a similar use-after-free property. The main differences in this example are (1) the device explicitly manipulates nested pointer fields in the IRP (e.g. Tail.Overlay.CurrStackLoc in line 21 of SerialWrite), (2) the completion routine SerialCompleteRequest may destroy all pointers reachable from a PIRP pointer that it completes. Unlike the uabSamp driver in §2.1, the serial driver (an older driver) does not use much data abstraction mechanisms and often manipulates deeply nested fields — making it more challenging for analysis. We have explicitly added the various assertions to model the use-after-free property.

Ideally, a tool that analyzes the Test harness should report that there are no assertion violations in this example. This is because for any concrete execution, the device driver ensures the following precondition to Test: (a) the pointers reachable from each request are not completed, and (b) the two requests Irp1 and Irp2 are disjoint, i.e., the set of pointers reachable from Irp1 and Irp2 do not overlap. It is not hard to see that in the absence of such assumptions, any tool will yield false alarms. For example, the sequential warning traces \(\langle (1, 9), (1, 14)\rangle\) and \(\langle (2, 9), \ldots, (2, 23)\rangle\) require a precondition that pointers reachable from Irp1 and Irp2 are not completed. Similarly, the trace \(\langle (1, 9), (1, 29), (2, 9), (2, 23)\rangle\) requires that two requests have disjoint set of reachable pointers.

One approach taken in practice is to create fresh objects by invoking constructors; this ensures that the set of pointers in two requests are disjoint and not completed. However, this approach is not always desirable for several reasons. First, a freshly created request packet may not represent the most general request packet and thereby provide poor coverage of the code it exercises. Secondly, in our case the routines that create a request resides in the kernel and may not be invoked from the driver. Therefore, a user of a tool is forced to write the environment assumptions as a non-trivial set of precondition constraints in the Init block (not shown here). Among the various preconditions, constraining the two requests to be disjoint is quite cumbersome to express.

Instead, our tool is able to automatically rule out all warnings because UabSamEvtIoRead and UabSamEvtIoWrite do not have any interleaved bugs. An interesting fact about this example is that the false alarms manifest in both sequential orderings of threads: SerialWrite|SerialRead as well as SerialRead|SerialWrite. This observation motivated our definition of interleaved bugs given in §4.
The reader might argue that we are only looking for buggy traces that require an “interleaving” of actions from different threads. However, such a scheme will report numerous warnings for this example. For instance, the assertion in line 23 may be interpreted as “the write may execute in between the reads.”

This example may seem trivial from the perspective of concurrency analysis, but that is because we have omitted the synchronization protocol that guards a request among different threads. We use this example only to show the difficulty of understanding false alarms and writing preconditions to rule them out manually.

### 2.3 Non-deterministic filters

As noted earlier, we find interleaved bugs using sequential interleave filters. However, the program that captures these interleave filters may be non-deterministic, i.e., it may have multiple behaviors for the same input state. There are two main reasons for non-determinism in the filter programs:

- **Data non-determinism**: The concurrent program may have calls to external libraries that are modeled non-deterministically. For example, in order to model `GetTimeOfDay`, one would write a model (or stub) that returns any non-deterministically chosen time (possibly, in an increasing sequence). The sequential program inherits these sources of non-determinism.

- **Control non-determinism**: As we illustrate in §4, the program representing the sequential interleave filters may have a limited amount of non-determinism in scheduling. For example, the filter used in §2.2 has to capture scheduling either of the two threads first.

One of the main technical challenges in this work is to deal with these non-deterministic filters. Therefore, we start by considering the problem of comparing two non-deterministic programs in the next section (§3).

### 3. Differential error checking

In this section, we study the more abstract problem `DIFFERROR` of comparing two (possibly non-deterministic) programs with respect to a set of assertions. In §4, we apply the results of this section towards finding interleaved bugs.

#### 3.1 Programs

We consider a simplified syntax for imperative programs with shared-memory concurrency. We assume that there is a single global variable `g` of type `T`. We intentionally leave `T` undefined.\(^3\) We only require the presence of certain predicates over `T`. Let `Failed` be a predicate of type `T → Bool`.

A program is a list of procedure declarations. Each procedure takes a single variable of type `T` as input, returns a single variable of type `T` as output, and has a single statement. A program statement `st` has the syntax:

```
| st ::= st; st | (Sequence) |
| if (e) st else st | (if-then-else) |
| while (e) do st | (Loop) |
| x := e | (Assignment) |
| assume e | (Assume) |
| assert e | (Assert) |
| havoc x | (Non-deterministic assignment) |
| call x := foo(e) | (Procedure call) |
| return x | (Procedure return) |
| async call foo(e) | (Thread spawn) |
```

\(^3\)One can encode programs with multiple global variables into our syntax by simply considering them to be a vector of types.
Here \( e \) is an expression over variables in scope, using some operators that we leave undefined. Semantics of our language is standard. The havoc \( x \) statement assigns a non-deterministically-chosen value to a variable \( x \). An assume \( e \) statement blocks in a state when \( e \) does not hold, and has no effect otherwise. An assert \( e \) statement fails in a state when \( e \) does not hold, in which case the control jumps to the end of the program and the global variable \( g \) is modified such that \( \text{Failed}(g) \) holds. (There is another way for \( \text{Failed}(g) \) to hold.) An async call \( \text{foo}(e) \) statement spawns a new thread that executes procedure \( \text{foo} \) with argument \( e \).

Even though we have defined a compact syntax, we will still use C-like syntax for easy writing of example programs. We will sometimes write \( x = * \) in place of havoc \( x \).

Non-determinism in this language arises from two sources: the havoc statement induces data non-determinism, whereas concurrency (via threads spawned by async statements) induces control non-determinism. For the rest of this section, we do not distinguish between these two sources of non-determinism.

We identify a program with the name of its main procedure. At any point in a program's execution, we refer to the value of variables in scope as the program's state. In particular, the input and output state of a program is the value of global variable \( g \) at beginning and end of the main procedure of the program, respectively. Given a program \( P \), let \( F_p \) be its input-output relation: We say that \( (s,t) \in F_P \) if there is some execution of the program from input \( s \) that ends in a state \( t \). A program has a buggy execution on input \( s \) if \( (s,t) \in F_P \) for some \( t \) and \( \text{Failed}(t) \) holds.

### 3.2 Problem formulation

In this section, we describe the problem of differential error (DiffError) in more detail, along with different algorithms to solve it. We aim to solve DiffError over two programs that expect the same input. Also, we assume that we are always given programs with assertions that capture the property of interest. The DiffError problem can be formally defined as follows.

**Definition 3.1 (DiffError).** Given two programs \( P_1 \) and \( P_2 \), DiffError \((P_1, P_2)\) holds if there exists an input state \( s \) such that (1) \( P_1 \) has some execution of \( P_2 \) starting at \( s \) that violates an assertion and (2) no execution of \( P_1 \) on \( s \) can violate an assertion. We say that \( P_1 \) acts as an input-filter for \( P_2 \).

Note that DiffError is harder than standard verification. Let \( \phi(x,y) \) be a formula in a decidable fragment of logic, say quantifier-free first-order logic with equality. Then we can reduce the satisfiability check of \( \exists x \forall y \phi(x,y) \) to the DiffError problem as follows. Construct two programs \( P_1 \) and \( P_2 \) with a global variable \( x \) and local variable \( y \) in \( P_1 \):

\[
P_1() \; \{ \text{havoc } y; \; \text{assert } \phi(x,y); \}
\]

\[
P_2() \; \{ \text{assert false}; \}
\]

Then DiffError \((P_1, P_2)\) holds if and only if \( \exists x \forall y \phi(x,y) \) is true. Thus, even though verifying \( P_1 \) and \( P_2 \) individually is decidable, DiffError \((P_1, P_2)\) is not (because checking the satisfiability of quantified first-order logic with function symbols and equality is undecidable [3]).

The next few subsections describe a few algorithms for solving the DiffError problem. In §3.3, we consider the case when the filter program is deterministic and terminating. In §3.4, we consider a method for non-deterministic filter programs; the method may not terminate for all programs. Both these approaches (§3.3 and §3.4) use quantifier-free reasoning of theorem provers.

### 3.3 Deterministic Filters

When the program \( P_1 \) is deterministic (i.e., given a fixed input, it has exactly one execution) and terminating (i.e., given any input, the program either terminates or fails an assertion in finite amount of time) then we have an easy way of solving DiffError by reducing it to standard verification. Let \( \text{P1Assume} \) be a program obtained from \( P_1 \) by replacing all assert \( e \) statements with assume \( e \) statements. Consider the program \( P \), shown in Fig. 3, that executes \( \text{P1Assume} \) and \( P_2 \) on the same input.

```cpp
var g : T;
Program P() {
  var g0 := g;
  call P1Assume();
  g := g0;
  call P2();
}
```

**Figure 3.** A program \( P \) constructed from two programs \( P_1 \) and \( P_2 \).

Because we have replaced asserts by assumes in \( P_1 \), the program \( \text{P1Assume} \) will block on all inputs that cause \( P_1 \) to fail. Consequently, an execution of \( P \) on that input will not even reach the call to \( P_2 \). Thus, \( P \) can only fail on some input \( i \) provided \( P_1 \) does not fail on \( i \) and \( P_2 \) fails on \( i \), which exactly solves DiffError.

**Theorem 3.2.** For a deterministic program \( P_1 \), the following statements are true: (a) if the program \( P \) in Fig. 3 fails on some input, then DiffError \((P_1, P_2)\) holds, and (b) if \( P_1 \) is also terminating and \( P \) does not fail any assertion, then DiffError \((P_1, P_2)\) does not hold.

This theorem states that checking DiffError can be reduced to checking assertions in a program and we can leverage standard verification techniques. Note that the requirement that \( P_1 \) is deterministic and terminating is important for this theorem as the next example shows.

Consider the example in Fig. 4. \( P_1 \) and \( P_2 \) are two programs that take a pointer \( p \) as input. The final assert in \( \text{foo} \) can fail because the programmer made a mistake: the operation in the else branch should be subtraction, not addition. We assume that both programs have implicit assertions for pointer dereferences, i.e., there is an assert \( p != \text{null} \) before any statement that dereferences \( p \). The intention is to find a bug in \( P_2 \) that reveals that the assertion in the last line can fail. Static analysis of \( P_2 \) can get distracted and report that the initial dereference \( p \rightarrow x \) can fail when \( p == \text{null} \). However, solving DiffError \((P_1, P_2)\) correctly guides us to the desired bug. \( P_1 \) will filter out the input \( p == \text{null} \) because it can fail on that input.

Note that Thm. 3.2 doesn’t apply for this example because \( P_1 \) is non-deterministic. If we were to construct the program \( P \) as in Fig. 3, then \( P \) can still fail on input \( p == \text{null} \). A similar argument holds when \( P_1 \) is non-terminating. For example, consider a program \( P_1 \) with a single statement assume false. This program never fails, and should not filter any input for \( P_2 \). However, for the program \( P \) of Fig. 3, it will block all input from reaching \( P_2 \).

### 3.4 Non-deterministic Filters

We now describe a technique for solving DiffError \((P_1, P_2)\) that can handle non-deterministic filter programs. However, the technique may fail to terminate on all programs, even when both \( P_1 \) and \( P_2 \) are bounded-length programs.

First, note that the construction of Fig. 3 is useful even when the filter program \( P_1 \) is not deterministic (but terminating) because then \( P \) necessarily fails less often than \( P_2 \). More specifically, \( P \) does not fail on those inputs on which \( P_1 \) deterministically fails (i.e., every execution of \( P_1 \) fails). Hence, we can always replace \( P_2 \) by \( P \).
A non-deterministic filter program.

Algorithm 1 Algorithm for solving DIFFER

Require: Programs $P_1$ and $P_2$

1: loop
2: if $r_1 := \text{FINDBUG}(P_2)$
3: if $r_2 := \text{NOBUG}()$
4: return NOBUG
5: end if
6: assert $(t, \psi)$
7: if $t$ is an SMT-based theorem
8: $\phi := \text{pre}(\text{Determinize}(t), \text{true})$
9: $P_2 := \text{assump} \sim \phi, P_2$
10: end if
11: $\phi := \text{pre}(\text{Determinize}(t'), \text{false})$
12: $P_2 := \text{assump} \sim \phi, P_2$
13: end loop

Our algorithm is shown in Alg. 1. It uses a static-analysis tool, which we call FINDBUG. We assume that FINDBUG, given a program with assertions, either returns NOBUG (meaning that the program has no bugs) or returns TRACEx(i, i) meaning that the program can fail on input i and t is the execution trace witnessing that failure. A trace consists of a sequence of program statements (in the order in which they got executed), along with values of variables at each point in the trace. FINDBUG can also be supplied with an input, in which case it only checks the program under that input. Alg. 1 returns its output in the same format as FINDBUG: it returns NOBUG if there is no input for which DIFFER(P1, P2) holds; or returns TRACEx(i, i) such that DIFFER(P1, P2) holds and t is an execution of P2 that fails on input i (but P1 does not fail on input i).

The algorithm works by iteratively filtering away more and more inputs. If it finds a bug in P2 (line 2), then it checks to see if P1 has a bug on the same input i (line 7). If P1 doesn’t, then it returns this input (line 9). If P1 does have a bug, then i needs to be filtered. For this, it uses a modification of the trace t’ itself to create a deterministic filter f (line 12) that filters out i as well as other inputs that cause P1 to fail along the trace t’. In some sense, f acts as a sub-filter that filters out some of the input that causes P1 to fail.

The procedure Determinize(t’) takes a trace t’ and performs the following modification to it — it replaces the havoc x statements with $x := c$, where c is the concrete value assigned to x in the trace. The predicate transformer pre(t, ψ) takes a trace t (a sequence of statements) and a formula ψ and returns a formula representing the path condition in terms of the inputs to the trace. It is defined inductively on the structure of the trace as follows (note that Determinize removes havoc statements from the trace):

pre(skip, ψ) = ψ
pre(assume φ, ψ) = φ ∧ ψ
pre(assump φ, ψ) = ¬φ ∧ ψ
pre(x := e, ψ) = ψ[e/x]
pre(x; t, ψ) = pre(s, pre(t, ψ))

Fig. 5 (a) shows an example of a trace that fails the assertion in $P_1$ in Fig. 4. For the trace havoc t, we indicate the value (say, 5) assigned to t in the trace. Fig. 5 (b) shows the formula φ constructed in line 12, which is equivalent to $p == \text{null}$.

Alg. 1 is not guaranteed to terminate (even for bounded-length programs), which is expected because DIFFER is undecidable in general. However, it does terminate for bounded-length programs with bounded non-determinism. Formally, non-determinism in a program is bounded if for any non-deterministic choice value v in the program, v only appears in expressions of the form $v \bowtie c$, where $\bowtie \in \{\bowtie, \leq, \geq\}$ and c is a constant. This subsumes the case when v is Boolean, i.e., $v \in \{\text{true}, \text{false}\}$. We did not come across unbounded non-determinism in our experiments.

Consider the example in Fig. 6, where the hashFunc is a procedure with complex operations to compute the hash value of an input. It is not hard to see that DIFFER(P1, P2) does not hold for this example. However, Alg. 1 will diverge by enumerating all the possible values of the non-deterministic choice of $\bowtie$ in P1. (The variable j in P1 is not necessary for the divergence.)

There are several ways to extend our approach to deal with such cases, at the cost of predictability. A natural extension is to set a bound k on the number of times a source of non-deterministic values participates in Determinize during the execution of Alg. 1. After this threshold is exceeded, we perform pre( ) on the trace t’ directly instead of determining it. We extend pre( ) for havoc statements:

pre(havoc x, ψ) = $\exists x. \psi$

This results in quantified filters in line 13 of Alg. 1. For the above example with $k = 0$, we will generate the filter:

$\neg(\exists x, j : j \neq 0 \land \neg a[i] \geq 0)$

If FINDBUG is able to reason about such quantifiers, then we will be able to prove that DIFFER(P1, P2) does not hold for this example. However, if FINDBUG is unable to reason precisely about these quantifiers, then Alg. 1 may diverge enumerating the same path in P1. For example, if FINDBUG is an SMT-based theorem prover, the quantified verification condition generated may not be
amenable to the trigger-based schemes for instantiating quantifiers in most SMT solvers [7] — there is no good trigger for the bound variable \( j \) in the formula above (this is the reason why \( j \) is present in \( P_1 \)). The above formula would need some simplifications (e.g., quantifier elimination) in conjunction with quantifier instantiations. We also present a variant of this idea in Appendix A that instead produces a quantified formula to precisely describe whether DIFFERROR \((P_1, P_2)\) holds for bounded programs, and then hands it off to a theorem prover. The main difference is that it pushes the divergence from within Alg. 1 to within the theorem prover.

4. Interleaved bugs

We now return to the topic of finding interleaved bugs for concurrent programs in the presence of underspecified harnesses. We start by defining the problem formally, show how it can be cast as a DIFFERROR problem, and then describe a few optimizations specific to our setting.

Let \( P \) be a concurrent program with dynamic thread creation (using async statements). We define a non-interleaved program execution to be one that has a single-thread of execution. Formally, a non-interleaved execution, while executing thread \( T_1 \), follows one of two possibilities at an assert statement that spawns thread \( T_2 \):

- \( T_1 \) waits for \( T_2 \): The spawned thread \( T_2 \) executes immediately and \( T_1 \) waits until \( T_2 \) and any thread spawned by \( T_2 \) completes. Additionally, while \( T_2 \) is executing, any async call must follow this same option. In some sense, the async call acts like a synchronous procedure call.
- \( T_2 \) waits for \( T_1 \): The spawned thread \( T_2 \) does not execute until \( T_1 \) finishes. When \( T_1 \) finishes, any of the threads spawned by it can start executing.

Let \( F_{P_1}^{seq} \) be a subset of \( F_P \) such that \( (s, t) \in F_{P_1}^{seq} \) if and only if \( (s, t) \in F_P \) and there is some non-interleaved execution of \( P \) from input \( s \) that ends in state \( t \). Our intuition is that assertion violations resulting from illegal inputs will often manifest in non-interleaved executions. Thus, they will be captured in \( F_{P_1}^{seq} \).

**Definition 4.1** (Interleaved bug). A program \( P \) has an interleaved bug if there is a pair of states \( (s, t) \in F_P \) such that \( \text{Failed}(t) \) holds and for all \( (s, t') \in F_{P_1}^{seq}, \text{Failed}(t') \) does not hold.

We find interleaved bugs using DIFFERROR.

**Theorem 4.2.** Given two concurrent programs \( P \) and \( Q \) the following statements are true. (i) If \( F_Q \subseteq F_{P_1}^{seq} \) and DIFFERROR \((Q, P)\) does not hold, then \( P \) has no interleaved bugs. (ii) If \( F_{P_1}^{seq} \subseteq F_Q \) and DIFFERROR \((Q, P)\) holds, then \( P \) has an interleaved bug.

Thm. 4.2 suggests that it suffices to work with underapproximations of \( F_{P_1}^{seq} \) while proving the absence of interleaved bugs and overapproximations of \( F_{P_1}^{seq} \) while proving the presence of interleaved bugs. In §4.1 and §4.2, we define under-approximate filters as programs. In §4.3, we define the program that captures \( F_{P_1}^{seq} \) precisely. In each case, we define the filter using a syntactic program transformation, and the resultant filter is a sequential program.

4.1 Filter: AsyncAsSync

The filter program AsyncAsSync always chooses the behavior where the async statement is treated as a synchronous procedure call that executes immediately. This corresponds to the first option in the definition of non-interleaved executions. The use of this filter is desirable as it can lead to a deterministic filter if the program does not make use of data non-determinism (or the non-determinism does not influence the assertions).

This filter can be obtained from \( P \) simply by replacing all async calls with normal procedure calls. The program shown in Fig. 1 uses this filter to remove all non-interleaved bugs.

4.2 Filter: AsyncAsEvents

The filter program AsyncAsEvents explores all behaviors in which spawned threads are delayed until the parent thread finishes. This corresponds to the second option in the definition of non-interleaved executions.

We capture this filter as an event-driven program. Note that an async call, in this case, is like posting an event that has to be processed when the current event finishes. This behavior is typical of event-driven programs.

Let \( \text{main} \) be the entry procedure of \( P \). Let \( \text{EventSet} \) be a multiset of events, where each event is a function pointer along with its arguments. The filter has the entry procedure shown in Fig. 7. It initializes the set of events with the \( \text{main} \) procedure and then executes an arbitrary event from \( \text{EventSet} \) in a loop. Events are added to the set by executing a async statement; the following transformation is applied to each async statement:

\[
\text{async foo(e)} \rightarrow \text{EventSet.Add(new Event(foo, e))}
\]

It is easy to see that a spawned thread does not execute until the parent thread finishes.

There are several existing analyses of event-driven programs. The work by Sen and Viswanathan [18] discusses the complexity of analyzing such programs. Jhala and Majumdar [13] present a software-model-checking approach, and Emmi et al. [10] present an underapproximate SMT-based analysis. A real-world example where this filter is required is described in §5.1.2. In our experiments, we use a tool based on the approach of Emmi et al. [10].

**Remark.** When the concurrent program \( P \) has synchronization (e.g., thread join operations) then it is possible that AsyncAsEvents may deadlock. This is acceptable in our setting because we do not consider deadlocks to be bugs.

4.3 Filter: AsyncGeneral

The AsyncGeneral filter program explores all non-interleaved executions. Such a program can have strictly more behaviors than both previous options put together. This is because it allows an execution to follow the first option (in the definition of non-interleaved executions) in some places and to follow the second option in other places.

### Figure 6. Example where Alg. 1 diverges.

```c
void P1(int a[], int b[])
{
    int i = *;
    int j = *;
    if (j != 0)
        assert a[1] >= 0;
    assert a[1] == 0;
}

void P2(int a[], int b[])
{
    int i = hashFunc(b);
    assert a[1] >= 0;
}
```

### Figure 7. Entry procedure for AsyncAsEvents and AsyncGeneral.
AsyncAsSync

determinism in the order in which they pick events. The filter
AsyncAsEvents
themselves. Both
did not modify them.

These stubs were created for an earlier study on static analysis; we
device object or fail and return one of a fixed number of error codes.

As for any static analysis tool, one has to close the pro-
filters). The main source of non-determinism is from the environ-
non-determinism (which justifies our interest in non-deterministic

The filters defined in previous sections have multiple sources of
non-determinism. For instance, in order to model
system call in Windows
is inherently non-deterministic. For instance, in order to model
First
thread must also be executed immediately. When
spawned thread immediately. In this case, any recursively spawned

The variable First is true when we have decided to execute a
spawned thread immediately. In this case, any recursively spawned
thread must also be executed immediately. When First is false,
we non-deterministically decide between two options: execute the
thread immediately or delay it until the current thread finishes.

4.4 Non-determinism

The filters defined in previous sections have multiple sources of
non-determinism (which justifies our interest in non-deterministic
filters). The main source of non-determinism is from the environ-
ment. As for any static analysis tool, one has to close the pro-
gram by writing stubs for the environment (such as the opera-
tion system). These stubs over-approximate the environment and
are inherently non-deterministic. For instance, in order to model
IoCreateDevice system call in Windows\(^4\); we used the stub
shown in Fig. 8. It can non-deterministically choose to allocate the
device object or fail and return one of a fixed number of error codes.
These stubs were created for an earlier study on static analysis; we
did not modify them.

Another source of non-determinism is in modeling the filters
themselves. Both AsyncAsEvents and AsyncGeneral have non-
determinism in the order in which they pick events. The filter
AsyncAsAsync does not add any extra non-determinism.

The stub in Fig. 8 and the filter structure both induce a
bounded amount of non-determinism, which is suitable for our


**Figure 8.** Stub for IoCreateDevice.

Let main be the entry procedure of \(P\). Let EventSet be as de-
defined in §4.2. The AsyncGeneral filter has the same entry pro-
cedure as AsyncAsEvents. The difference is in the transforma-
tion of async statements. We add a Boolean variable First to
AsyncGeneral, initialize it to false, and then carry out the fol-
lowing transformation:

async foo(e) →
  if(First) {
    foo(e);
  } else if(nondet()) {
    First = true; foo(e); First = false;
  } else {
    EventSet.Add(new Event(foo, e));
  }

The variable First is true when we have decided to execute a
spawned thread immediately. In this case, any recursively spawned
thread must also be executed immediately. When First is false,
we non-deterministically decide between two options: execute the
thread immediately or delay it until the current thread finishes.

4.5 Optimizations

For the purpose of finding interleaved bugs, we have a special
instance of DIFFERROR(\(F_1, F_2\)), namely one in which \(F_1 \subseteq F_2\). In this setting, we can optimize Alg. 1 by avoiding a few calls to FINDBUG.

More concretely, let \((t, i)\) be as defined on line 6 of Alg. 1.
Then \(t\) is a concurrent execution, possibly with many threads. We
can permute statements in \(t\) such that the resulting trace conforms
to the filter in use, and then check the feasibility of the resulting
trace. For example, let \(t\) have two threads \(T_1\) and \(T_2\), and consists
of statements executed in the following order:

\[
t = (a_1; a_2; a_3; b_1; b_2; a_4; a_5; b_3; b_4; a_6;)
\]

where the \(a_i\)s refer to statements fired by \(T_1\), and the \(b_i\)s refer to
statements fired by \(T_2\). Further, let \(a_3\) be the statement that
spawns thread \(T_2\). Let \(t_1\) and \(t_2\) be the following permutations
of statements in \(t\):

\[
t_1 = (a_1; a_2; a_3; b_1; b_2; b_3; a_4; a_5; a_6;)
\]

\[
t_2 = (a_1; a_2; a_3; a_4; a_5; a_6; b_1; b_2; b_3;)
\]

Then \(t_1\) conforms to AsyncAsAsync, \(t_2\) conforms to
AsyncAsEvents, and both conform to AsyncGeneral. Suppose
we are using AsyncAsAsync as the filter, and \(t_1\) happens to be
feasible on input \(i\) (i.e., \(i\) satisfies \(\text{pre}(t_1, \text{true})\)), then we can let \(t_2\)
be \(\text{TRACE}(t', i)\) and jump to line 11, thereby avoiding the call to
FINDBUG on line 7. In general, there are many ways to permute
a given concurrent trace to make it correspond to a filter. In our
implementation we try a few permutations. If any of these work,
then we can avoid a call to FINDBUG.

Note that this optimization does not work the other way. Con-
sider the example shown in Fig. 9, where every dereference is
implicitly protected by a null-pointer assertion. It has an execution
that fails on line L2 that requires four context switches and no se-
quential permutation of this execution is feasible. However, we can-
not conclude that the execution is an interleaved bug because there
is a sequential execution on the same input (but takes a different path
due to the non-determinism at L1) that fails at label L3.

5. Evaluation

We implemented Alg. 1 (mentioned in §3) on concurrent programs
in a tool called CBUGS. It first uses the AsyncAsAsync filter. If it does
not find bugs in the presence of this filter, then it stops. Otherwise,
it uses the AsyncGeneral filter and reports resulting bugs as inter-
leaved bugs. The use of AsyncAsAsync is an optimization because it
is easier to analyze than AsyncGeneral. CBUGS uses POIROT as

```c
int x;
void main(int *p) {
  void bar(int *p) {
    x = 0;
    if(nondet()) {
      async foo();
      if(x == 1) x = 2;
      async bar(p);
      if(x == 3)
    }
  }
}

void foo() {
  if(x == 0) x = 1;
  if(x == 2) x = 3;
}

L1: *p = 10;
L2: *p = 5;
L3: *p = 5;
}
```
the tool for finding bugs in concurrent programs, i.e., POIROT acts as FIndBUG in Alg. 1. POIROT is one of the bug-finding tools for concurrent programs that uses iterative context-bounding to look for bugs [6, 15, 17].

We conducted experiments to evaluate CBUGS on two goals. First, can CBUGS rule out bugs caused by illegal input, while retaining the true bugs? Second, we compare CBUGS against a different filtering strategy, namely one that filters based on asserts. In this strategy, if a sequential trace leads to an assertion violation, then we remove that assertion and repeat until no more warnings are produced.

5.1 Results

We chose a collection of Windows device drivers from the WinDK suite [22] for conducting experiments. Some of these drivers were manually seeded with bugs by others in an independent study. To the best of our knowledge, these are the only bugs present in the code. We suffix the driver name with “bug” when it has a (single) seeded bug. This allows us to conduct a controlled study. None of the drivers have a precise harness.

We checked the “Cancel” property (mentioned in §2) as well as for null-pointer dereferences (“NullDereFn”). The results are reported in Fig. 10. Each row of this table has: the name of the driver (Name); the number of non-empty non-assert lines of code exercised by tool (LOC) along with the number of assertions shown in parenthesis; the property being checked (Prop.); and the type of bug present in the code (either “none” or “interleaved” or “sequential”). The next column (#LF) is the number of iterations of Alg. 1, i.e., the number of lazy filters generated by CBUGS. The rest of the columns show the number of false positives (#FP) and false negatives (#FN) of the two approaches and the total time taken to generate all the warnings (barring the manual effort of classifying a warning as a true or false positive). Whenever CBUGS used AsyncGeneral for generating a lazy filter, we mention it in the (#LF) column with “AG”.

For instance, the first row is for the daytona driver with the cancel property. The driver does not have any actual bugs, however, when we run static analysis (FIndBUG), it reports a warning (false positive) because of the imprecise harness. CBUGS, on the other hand, suppresses this warning automatically (it generates one lazy filter).

Our experiments show: (1) CBUGS does not report any false positives, whereas the assert suppression technique reported 63 false positives; (2) CBUGS did not miss any of the interleaved bugs, whereas assert suppression missed one in noclasse_bug2; (3) the use of AsyncAsSync is a useful optimization and is enough to rule out false warnings in most cases. As expected, CBUGS also ends up suppressing true sequential bugs, however, it demonstrates good results for catching concurrency bugs.

We now explain some of the results in more detail, to illustrate the need for AsyncGeneral and the kinds of preconditions that these drivers required.

5.1.1 Cancel Property on ndisiprot

The existing harness for the ndisiprot device driver (for the Cancel property) added two preconditions: (1) a particular lock must be initialized, and (2) a doubly-linked list (of requests) is empty. Although the first is a valid precondition, the second one is the simplest way to establish the actual preconditions: (a) the incoming request does not belong to the list, and (b) the list is well-formed doubly-linked list that respects the relationships between the forward and the backwards links. Unless the latter is enforced, the pointer manipulations performed to insert or delete an element of the list does not have the desired effects. Both these preconditions are very hard to express for low-level C programs, even for bounded lists.

```c
void ReadFn(PIRP irp) {
  if(irp->b) {
    *irp->ptr = 10;
  }
  async ReadFn(irp);
}
```

```c
// AsyncAsSync filter will generate failure traces where the body of CancelFn is executed before the test on irp->cancel inside ReadFn. However, using the AsyncGeneral filter, we can defer ReadFn to execute after CancelFn has executed, thereby failing the assertion on any input. A similar situation requires the use of AsyncAsSync for the example in §2.2.

6. Related work

In this paper, we presented an approach for reducing false alarms due to underspecified environments during static assertion checking in concurrent programs, by using the sequential behaviors as an oracle. We position our work in the context of previous work in each of the italicized areas in the next few subsections.

6.1 Filtering static analysis alarms

Engler et al. [11] propose a method of discovering bugs by observing inconsistent behavior in source code. Dillig et al. [9] provide a semantic basis for finding such inconsistent behaviors by posing the problem as a type-inference problem. Both these approaches have been applied to find null dereference errors in large sequential
codebases. These work can also be seen as filtering false alarms if usage is consistent with some (unknown) protocol; e.g., if all dereferences of a variable x is not protected by a null-check, the accesses to the variable is most likely safe due to some invariant in the program. One can think of interleaved bugs as discovering inconsistencies (with respect to a given set of assertions), where the sequential behaviors describe an implicit protocol. On the other hand, unlike these approaches, we do not require a separate analysis for different patterns and provide a formal guarantee of relative correctness if there are no interleaved bugs.

Approaches for suppressing or ranking warnings have ranged from statistical techniques (such as Z-ranking [14]) to more domain specific methods (such as suppressing data-race warnings [21]). Although these approaches are applied to the results of static analyses that scale to large modules, they do not provide any formal guarantees of the bugs that are suppressed. Besides, most of these approaches are aimed at combating the imprecision of static analysis, as opposed to the environment problem.

### 6.2 Sequential filters

The idea of using sequential behaviors as oracle for concurrent implementations has been studied in the context of checking linearizability [12], which provides a natural specification in many settings for checking concurrent programs. Various static and dynamic tools have been built to check concurrent behaviors against sequential ones. LINEUP is a dynamic analysis tool that flags a concurrent behavior when it outputs a value that no sequential execution produced [4]. Burnim et al. [5] provide runtime techniques to check parallel implementations against non-deterministic sequential specifications. Siegel et al. [19] employ symbolic execution along with enumeration of interleavings to check against the sequential behaviors for numerical programs. Unlike these approaches, we consider the dual problem when specifications are present in the program but the environment is imprecise. Because we check for user-specified assertions, we can apply the technique to any concurrent program even if it is not linearizable.

### 6.3 Environment synthesis

Generating environments for model checking of open systems is a well-studied problem. Tkachuk et al. [20] generate environment models from user-specified assertions and by analyzing environment implementations. Alur et al. [1] address the problem of synthesizing the most liberal environment using a combination of predicate abstraction and automata learning. One can view our approach as inferring the most liberal environment that does not induce failures on sequential executions, and using it to check the concurrent program. As demonstrated in §5, inferring legal environment preconditions may involve inferring complex aliasing relationships on the input data that may not be amenable to finite-state approaches. Nonetheless, it would be interesting to combine synthesis techniques with our algorithm.

### 7. Conclusion

In this paper, we highlight the problem of false alarms in static analysis due to missing environment assumptions and present a solution when checking assertions in concurrent programs. We define a class of warnings as interleaved bugs when they are retained by a filter that attributes all warnings in the sequential interleavings to the underspecified harness (or to missing preconditions). We believe this can be an effective way to prioritize high quality warnings when looking at warnings generated by a static analyzer for concurrent programs. Our preliminary experience with a simple implementation to find interleaved bugs is encouraging, although we expect more work to deal with unbounded non-determinism in the filter programs.

More generally, the paper makes one of the first contributions to the area of using semantic techniques for prioritizing alarms from static analysis. Sequential filters are natural filters when prioritizing alarms for concurrency analysis. We believe that one can extend this idea to use artifacts other than sequential executions in the source code to automatically define filter programs for other domains as well.

### Acknowledgments

We thank Tom Ball, Sriram Rajamani and the anonymous reviewers for their feedback on an earlier draft. We thank Madan Musuvathi for pointing us to similarities with work on using sequential executions as oracles for concurrent modules. Finally, we thank Zvonimir Rakamaric and Shaz Qadeer who wrote many of the concurrent harnesses for the device drivers, which served as a starting point for our experiments.
A. Quantified VC generation

In this section, we describe a method for generating a (quantified) verification condition (for bounded-length programs) in the presence of non-deterministic and possibly non-terminating filter programs.

Recall that the goal of DIFFERORP($P_1$, $P_2$) is to filter any input $i$, such that there is some choice of the non-deterministic values in $P_1$ that fails $P_2$. This implies that the non-determinism in $P_1$ is angelic (in contrast to the demonic non-determinism in $P_2$). One option is to extend our language (in §3.1) to introduce a statement choose $x$ (in addition to havoc $x$), to model the angelic non-determinism. And then create a program similar to one described in Fig. 3, where we replace the havoc $x$ statements in $P_1$ with choose $x$ statements, in addition to the transformation of the assert $e$ statements.

However, we are not aware of any efficient verification condition (VC) generation algorithm in the presence of such non-determinism. Let us highlight one of the main difficulties of extending VC generation algorithms with choose statements. Most efficient VC generation algorithms (see [2]) generate a formula whose size is at most quadratic in the size of the program. This is achieved by using a variant of static single assignment, where auxiliary variables are introduced to hold different incarnations of a program variable after an assignment, a havoc statement, or at merge points. These variables are implicitly universally quantified in the resultant VC (when checking for validity). The presence of choose statements introduces an existential quantifier (e.g., \( wlp(\text{choose } x, \phi) = 2x.\phi \)).

We generate a VC for \( P_2 \) using any off-the-shelf VC generation technique after blocking all these auxiliary variables in the VC, as determining quantification \( (\forall vs. \exists) \) of a particular variable and the nesting can be challenging.

Instead, we present (in Alg. 2) a VC generation mechanism for DIFFERORP that leverages off-the-shelf VC generation along with symbolic execution along error paths, to lazily add the quantifier alternation. The idea is simple: we enumerate all the program paths in $P_1$ that lead to an assertion violation, and create an expression (purely in terms of the inputs to $P_2$) that characterizes all the conditions under which $P_1$ fails. We generate a VC for $P_2$ using any off-the-shelf VC generation technique after blocking all these inputs.

The formula $\phi$ represents the set of inputs for which there is a choice of the non-deterministic values such that an assertion in $P_1$ fails. Similarly, the set $A$ represents a set of paths in $P_1$ that lead to an assertion violation. These variables are initialized to false and \{\} respectively (line 1 and line 2). The loop from line 3 to line 11 enumerates different paths in $P_1$ that fail an assertion. We use an oracle FINDBUG (in line 4) that takes as arguments (i) a program and (ii) a set of paths and finds an error trace that avoids the set of paths specified, or NOBUG (otherwise (indicating that there are no bugs). FINDBUG can be implemented by augmenting VC generation to avoid a set of paths while checking assertions [16]. Once all the error paths have been explored, the algorithm computes the VC for $P_2$ after “blocking” all the inputs in $\phi$ using assume $\neg \phi$ (line 6). Otherwise, for a given error $r$, we extract the error trace $t$ and the input $i$ and update $A$ and $\phi$ respectively (line 9 and line 10).

The predicate transformer $wlp(\ldots)$ is extended for havoc statements:

$$\text{pre}(\text{havoc } x, \psi) = \exists x. \psi$$

Observe that $\text{pre}(t, \text{true})$ is a formula whose free variables are inputs to $P_1$, and can be massaged to a formula of the form $\exists x_1, \ldots, x_i. \phi'$, where $\phi'$ is a ground formula, after renaming the bound variables introduced due to havoc to avoid variable capture.

References

Algorithm 2 Algorithm for generating a VC for DIFFERROR

**Require:** Programs $P_1$ and $P_2$

**Ensure:** A formula representing the VC for DIFFERROR($P_1$, $P_2$)

1: $\phi := \text{false}$
2: $A := \emptyset$
3: loop
4: $r := \text{FINDBUG}(P_1, A)$
5: if $r = \text{NOBUG}$ then
6: return VC(assume $\neg\phi$; $P_2$. body)
7: end if
8: Let $\text{TRACE}(t, i) = r$
9: $A := A \cup \{t\}$
10: $\phi := \phi \lor \text{pre}(t, \text{true})$
11: end loop

**Theorem A.1.** Given programs $P_1$ and $P_2$, if $\psi$ be the formula returned by Alg. 2, then DIFFERROR($P_1$, $P_2$) holds if and only if $\psi$ is satisfiable.

It is interesting to note that the theorem above holds even when the filter program $P_1$ does not terminate on some inputs — this is because we only enumerate paths in $P_1$ that lead to an assertion violation. For instance, if we consider the non-terminating example from §3.3, where $P_1$ was simply assume false, then we will not find any bugs in $P_1$, and Alg. 2 will simply return the VC of $P_2$.

Consider the example in Fig. 6, where the hashFunc is a procedure with complex operations to compute the hash value of an input. It is not hard to see that DIFFERROR($P_1$, $P_2$) does not hold for this example. However, Alg. 1 will diverge by enumerating all the possible values of the non-deterministic choice of $i$ in $P_1$. Instead, Alg. 2 generates the following precondition to $P_2$ by capturing all the inputs that fail $P_1$:

$$\neg(\exists i, j :: j \neq 0 \land \neg a[i] \geq 0)$$

which is equivalent to $\forall i :: a[i] \geq 0$, and sufficient to prove the assertion in $P_2$ for any implementation of hashFunc.

Although the algorithm presented here generates a precise VC for the DIFFERROR problem, the undecidable nature of the DIFFERROR problem precludes any algorithm to solve all instances of the problem. The algorithm presented here may not be complete due to the incompleteness of automatic theorem provers to deal with quantifiers in the resulting VC. For example, the quantified VC generated may not be amenable to the trigger-based schemes for instantiating quantifiers in most SMT solvers [7] — there is no good trigger for the bound variable $j$ above. The above formula would need some simplifications (e.g. quantifier elimination) in conjunction with quantifier instantiations. Nevertheless, by translating the DIFFERROR($P_1$, $P_2$) to a logical formula, we can hope to leverage advances in automated theorem provers to obtain more precise and efficient solution for DIFFERROR($P_1$, $P_2$).