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
akashl@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.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

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.[1] 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.

[1] We formally define the notion of sequential interleavings in §4 as one in which threads do not *interfere* with each other.

(It may be possible that a concurrent execution on an illegal input fails; we are only claiming that in such a case, there would also be some sequential execution that fails.) Under this conjecture, most interleaved bugs would correspond to real errors. Our experiments (§5) support this conjecture: all interleaved bugs that we found were actual bugs in the program. Although this result might not hold for all programs, it gives us greater confidence that our approach retains actual concurrency bugs. Note that our technique does not provide help when one is looking for sequential bugs. We are only addressing the scenario when one is interested in finding concurrency bugs, but static analysis results in many false alarms.

Let us contrast the approach of looking for interleaved bugs against two other incomparable approaches for filtering warnings. One possibility is to find (a) the set S of warning locations (failing asserts) that manifest in sequential executions, and (b) the set C of warning locations that manifest in concurrent executions, and only report the locations in the difference of C and S. This approach does not provide any guarantee on the absence of bugs when the program has been sequentially verified (unlike our approach). This is because the presence of a single illegal input that can fail an assertion sequentially suppresses that assert, even when a legal input could fail the same assert.

Another approach is to suppress all the sequential traces that violate an assertion, and only report those traces that have interleaving between threads. This may still report too many false alarms — it may report *trivial* interleavings that still transform the shared state in manner similar to some sequential interleaving. An example of such a program is given in §2.2. We believe that interleaved bugs achieves a good trade-off for suppressing false alarms due to underspecified environment assumptions while not supressing true concurrency bugs.

In this paper, we devise algorithms for finding interleaved bugs. We formalize the problem as a *differential* analysis of two programs. For two programs P_1 and P_2, $\textsc{DiffError}(P_1, P_2)$ 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_s be the same as P but executions of P_s are restricted to be sequential, i.e., P_s can have multiple threads, but it does not interleave them. We formally describe how to capture P_s as a program in §4. Then, finding interleaved bugs is the same as solving $\textsc{DiffError}(P_s, P)$. 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_s 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 $\textsc{DiffError}(P_1, P_2)$ 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 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 POIROT [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 `usbsamp` [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 `UsbSamp_EvtIoRead` 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 `UsbSamp_EvtRequestCancel`; one of the side-effects of the call is to set a field `Request->cancelRoutine` to `UsbSamp_EvtRequestCancel`. 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 `UsbSamp_EvtRequestCancel` 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."[2]

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 `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

[2] http://msdn.microsoft.com/en-us/library/ff549983(v=VS.85).aspx

```
1 // Thread T1
2 VOID UsbSamp_EvtIoRead(
3     ...,
4     WDFREQUEST Request
5   ) {
6   ...
7   WdfRequestMarkCancelable(
8     Request,
9     UsbSamp_EvtRequestCancel);
10  ...
11  WdfRequestUnmarkCancelable(
12    Request);
13  ...
14  WdfRequestComplete(Request,
15                     status);
16  ...
17 }
```

```
1 // Thread T2
2 VOID UsbSamp_EvtRequestCancel(
3     WDFREQUEST Request
4   ) {
5   PREQUEST_CONTEXT rwContext;
6   ...
7   rwContext =
8     GetRequestContext(Request);
9   ...
10 }
11
12 VOID Cancel(WDFREQUEST Request) {
13   f = Request->cancelRoutine;
14   Request->cancelRoutine = NULL;
15   if (f) f(Request);
16 }
```

```
1 //Harness
2 VOID Test(
3     ...,
4     WDFREQUEST Request
5   ) {
6 INIT:
7   //assume(!completed(Request));
8 BODY:
9   async UsbSamp_EvtIoRead(..., Request);
10  async Cancel(Request);
11 }
```

Figure 1. Stripped down version of a bug found by STORM [15] in the `usbsamp` device driver. The three columns show the two threads and the test driver (harness).

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 $\langle \ldots, (tid_i, l_i), \ldots \rangle$, where $tid_i \in \{1, 2, \ldots\}$ 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 `UsbSamp_EvtRequestCancel`, 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 driver explicitly manipulates nested pointer fields in the IRP (e.g. `Tail.Overlay.CurrStckLoc` in line 21 of `SerialWrite`), (2) the completion routine `SerialCompleteRequest` may destroy all pointers reachable from a `PIRP` pointer that it completes. Unlike the `usbsamp` 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 `UsbSam_EvtIoRead` and `UsbSam_EvtIoWrite` 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.

```
1 // Thread T1
2 VOID
3 SerialWrite(
4     PDEVICE_OBJECT DO,
5     PIRP Irp
6     )
7 {
8
9   PSERIAL_DEVICE_EXTENSION Extn =
10          DO->DevExtn;
11  struct _IO_STACK_LOCATION *csl;
12
13  ...
14  assert(!completed(Irp));
15  Irp->IoStatus.Status =
16       STATUS_SUCCESS;
17
18  ...
19  assert(!completed(Irp));
20  csl =
21    Irp->Tail.Overlay.CurrStckLoc;
22
23  assert(!completed(csl));
24  if (csl->Parameters.Write.Length){
25    //put the write into a queue
26  } else {
27     ...
28     assert(!completed(Irp));
29     SerialCompleteRequest(.., Irp, 0);
30  }
31 }
```

```
1 // Thread T2
2 VOID
3 SerialRead(
4     PDEVICE_OBJECT DO,
5     PIRP Irp
6     )
7 {
8
9   PSERIAL_DEVICE_EXTENSION Extn =
10          DO->DevExtn;
11  struct _IO_STACK_LOCATION *csl;
12
13  ...
14  assert(!completed(Irp));
15  Irp->IoStatus.Status =
16       STATUS_SUCCESS;
17
18  ...
19  assert(!completed(Irp));
20  csl =
21    Irp->Tail.Overlay.CurrStckLoc;
22
23  assert(!completed(csl));
24  if (csl->Parameters.Read.Length){
25    //put the read into a queue
26  } else {
27     ...
28     assert(!completed(Irp));
29     SerialCompleteRequest(.., Irp, 0);
30  }
31 }
```

```
1 //Harness
2 VOID Test(
3     PDEVICE_OBJECT DO,
4     PIRP Irp1,
5     PIRP Irp2
6   ) {
7 INIT:
8
9 BODY:
10  async SerialWrite(DO,Irp1);
11  async SerialRead(DO,Irp2);
12 }
```

Figure 2. Simplified example (from `serial` device driver) illustrating the need for aliasing preconditions.

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 can fail on many interleavings between the two threads (e.g. $\langle(1,\ 9),(2,\ 9),(2,\ 14),\ldots,(1,\ 23)\rangle$).

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 interleavings as filters. However, the program that captures these interleavings 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 interleavings 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 $\mathtt{T} \rightarrow \mathit{Bool}$.

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 `T` 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 *Failed*(g) holds. (There is no other way for *Failed*(g) to hold.) An async call foo(e) statement spawns a new thread that executes procedure `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 *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) there is 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:

$$P_1()\ \ \{\text{havoc y; assert } \phi(\text{x}, \text{y});\}$$
$$P_2()\ \ \{\text{assert false;}\}$$

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 `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 `P1Assume` and P_2 on the same input.

```
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 `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 `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 != 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->x` can fail when `p == null`. However, solving DIFFERROR(P_1, P_2) correctly guides us to the desired bug. P_1 will filter out the input `p == 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 == 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.

P_1	P_2

<pre>struct ST *p;
 	
void bar() {
   int t; havoc t;
   if(t) {
      p->x = 0;
   }
}</pre>
	<pre>struct ST *p;
	
	void foo() {
	  int x;
	  p->x = 0;
	  if(p->y > p->x) {
	     t = p->y - p->x;
	  } else {
	     t = p->x + p->y;
	  }
	  assert(t >= 0);
	}</pre>

Figure 4. A non-deterministic filter program.

Algorithm 1 Algorithm for solving DIFFERROR

Require: Programs P_1 and P_2

1: **loop**
2: r_1 := FINDBUG(P_2)
3: **if** r_1 = NOBUG **then**
4: **return** NOBUG
5: **end if**
6: Let $\text{TRACE}(t, i) = r_1$
7: r_2 := FINDBUG(P_1, i)
8: **if** r_2 := NOBUG **then**
9: **return** r_1
10: **end if**
11: Let $\text{TRACE}(t', i) = r_2$
12: $\phi := \mathit{pre}(\mathit{Determinize}(t'), \mathsf{true})$
13: $P_2 := \mathsf{assume}\ \neg\phi; P_2$
14: **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 $\text{TRACE}(t, i)$, 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 $\text{DIFFERROR}(P_1, P_2)$ holds; or returns $\text{TRACE}(t, i)$ such that $\text{DIFFERROR}(P_1, P_2)$ holds and t is an execution of P_2 that fails on input i (but P_1 does not fail on input i).

The algorithm works by iteratively filtering away more and more inputs. If it finds a bug in P_2 (line 2), then it checks to see if P_1 has a bug on the same input i (line 7). If P_1 doesn't, then it returns this input (line 9). If P_1 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 P_1 to fail along the trace t'. In some sense, f acts as a *sub-filter* that filters out some of the input that causes P_1 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 $\mathit{pre}(t, \psi)$ 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):

$$
\begin{array}{lcl}
\mathit{pre}(\mathsf{skip}, \psi) & = & \psi \\
\mathit{pre}(\mathsf{assume}\ \phi, \psi) & = & \phi \wedge \psi \\
\mathit{pre}(\mathsf{assert}\ \phi, \psi) & = & \neg\phi \wedge \psi \\
\mathit{pre}(\mathtt{x} := e, \psi) & = & \psi[e/\mathtt{x}] \\
\mathit{pre}(s; t, \psi) & = & \mathit{pre}(s, \mathit{pre}(t, \psi))
\end{array}
$$

Fig. 5 (a) shows an example of a trace that fails the assertion in P_1 in Fig. 4. For the statement `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 == null`.

`havoc t; // 5`
`assume t != 0;`
`assert p != null;`
	`(5 != 0 && p == null)`

(a)	(b)

Figure 5. (a) A trace through P_1 from Fig. 4 and (b) the corresponding formula.

The most expensive parts in Alg. 1's loop are the calls to FINDBUG. One can optimize these calls in cases when FINDBUG is *incremental*. For instance, the calls on line 2 always contains the same program pre-pended with an increasing list of assume statements. The calls on line 7 are to the same program but with different inputs. Thus, any information that FINDBUG infers about P_1 or P_2 can be retained across iterations.

THEOREM 3.3. *The following statements are true. (a) If Alg. 1 returns* NOBUG*, then* $\text{DIFFERROR}(P_1, P_2)$ *does not hold, and (b) if Alg. 1 returns* $\text{TRACE}(t, i)$ *then* $\text{DIFFERROR}(P_1, P_2)$ *holds and input i and trace t are the witnesses.*

Alg. 1 is not guaranteed to terminate (even for bounded-length programs), which is expected because DIFFERROR 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 $\bowtie \in \{=, \leq, <\}$ and c is a constant. This subsumes the case when v is Boolean, i.e., $v \in \{\mathsf{true}, \mathsf{false}\}$. 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 $\text{DIFFERROR}(P_1, P_2)$ 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. (The variable `j` in P_1 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 $\mathit{pre}(,)$ on the trace t' directly instead of determinizing it. We extend $\mathit{pre}(,)$ for havoc statements:

$$\mathit{pre}(\mathsf{havoc}\ \mathtt{x}, \psi) \quad = \quad \exists x.\ \psi$$

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

$$\neg(\exists i, j :: j \neq 0 \wedge \neg\mathsf{a}[i] \geq 0)$$

If FINDBUG is able to reason about such quantifiers, then we will be able to prove that $\text{DIFFERROR}(P_1, P_2)$ 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 P_1. For example, if FINDBUG is an SMT-based theorem prover, the quantified verification condition generated may not be

```
void P1(int a[], int b){          void P2(int a[], int b){
    int i = *;                        int i = hashFunc(b);
    int j = *;                        assert a[i] >= 0;
    if (j != 0)                   }
       assert a[i] >= 0;
}
```

Figure 6. Example where Alg. 1 diverges.

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 $\text{DIFFERROR}(P_1, P_2)$ 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 async 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^{seq} be a subset of F_P such that $(s, t) \in F_P^{seq}$ 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^{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 Failed(t) holds and for all $(s, t') \in F_P^{seq}$, 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 $F_Q \subseteq F_P^{seq}$ and $\text{DIFFERROR}(Q, P)$ does not hold, then P has no interleaved bugs. (ii) If $F_P^{seq} \subseteq F_Q$ and $\text{DIFFERROR}(Q, P)$ holds, then P has an interleaved bug.*

Thm. 4.2 suggests that it suffices to work with *underapproximations* of F_P^{seq} while proving the absence of interleaved bugs and *overapproximations* of F_P^{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^{seq} precisely. In each case, we define the filter using a syntactic program transformation, and the resultant filter is a sequential program.

```
void AsyncAsEventsOrGeneral(args) {
  EventSet = new MultiSet<Event>();
  EventSet.Add(new Event(main, args));
  while(!EventSet.empty()) {
    let (f,e) = EventSet.GetAny();
    f(e);
  }
}
```

Figure 7. Entry procedure for $AsyncAsEvents$ and $AsyncGeneral$.

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 main be the entry procedure of P. Let 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 main procedure and then executes an arbitrary event from EventSet in a loop. Events are added to the set by executing a async statement; the following transformation is applied to each async statement:

async foo(e) $\mapsto$ 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.

```
NTSTATUS
IoCreateDevice(...,
               OUT PDEVICE_OBJECT *DeviceObject)
{
  PDEVICE_OBJECT deviceObject;

  int x = nondet();
  if (x == 0) {
    // Allocate device
    deviceObject = (PDEVICE_OBJECT)
                   malloc(sizeof(DEVICE_OBJECT));
    ...
    *DeviceObject = deviceObject;
    return STATUS_SUCCESS;
  } else if (x == 1) {
    // Fail
    return STATUS_INSUFFICIENT_RESOURCES;
  } else if (x == 2) {
    // Fail
    return STATUS_OBJECT_NAME_EXISTS;
  } else  ...
}
```

Figure 8. Stub for `IoCreateDevice`.

Let `main` be the entry procedure of P. Let `EventSet` be as defined in §4.2. The $AsyncGeneral$ filter has the same entry procedure as $AsyncAsEvents$. The difference is in the transformation of `async` statements. We add a Boolean variable `First` to $AsyncGeneral$, initialize it to `false`, and then carry out the following 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.

4.4 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 environment. As for any static analysis tool, one has to close the program by writing *stubs* for the environment (such as the operation 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 $AsyncAsSync$ 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 lazy algorithm (§3.4). However, some stubs have unbounded-nondeterminism. For instance, the stub that we use for `malloc` is one that can return any address which has not been previously allocated. However, such unboundedness has never been a problem in our experiments. One reason could be that programs (and, consequently, bugs) do not rely on the actual address values.

[4] http://msdn.microsoft.com/en-us/library/ff548397(v=vs.85).aspx

```
int x;                         void bar(int *p) {
void main(int *p) {            L1:
  x = 0;                         if(nondet()) {
  async foo();                     if(x == 1) x = 2;
  async bar(p);                    if(x == 3)
}                              L2:    *p = 10;
void foo() {                     } else {
  if(x == 0) x = 1;            L3:    *p = 5;
  if(x == 2) x = 3;              }
}                              }
```

Figure 9. Example to show incompleteness of the optimization.

4.5 Optimizations

For the purpose of finding interleaved bugs, we have a special instance of $\text{DIFFERROR}(P_1, P_2)$, namely one in which $F_{P_1} \subseteq F_{P_2}$. 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_is refer to statements fired by T_1, and the b_is 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:
$t_1 = (a_1; a_2; a_3; b_1; b_2; b_3; b_4; a_4; a_5; a_6;)$
$t_2 = (a_1; a_2; a_3; a_4; a_5; a_6; b_1; b_2; b_3; b_4;)$
Then t_1 conforms to $AsyncAsSync$, t_2 conforms to $AsyncAsEvents$, and both conform to $AsyncGeneral$. Suppose we are using $AsyncAsSync$ as the filter, and t_1 happens to be feasible on input i (i.e., i satisfies $pre(t_1, \texttt{true})$), then we can let r_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. Consider 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 sequential permutation of this execution is feasible. However, we cannot 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 *AsyncAsSync* 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 interleaved bugs. The use of *AsyncAsSync* is an optimization because it is easier to analyze than *AsyncGeneral*. CBUGS uses POIROT as 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 WinDDK 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 ("NullDeref"). 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 `mouclass_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 `ndisprot`

The existing harness for the `ndisprot` 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.

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

void CancelFn(PIRP irp) {
   irp->cancel = true;
}

void Test(PIRP irp) {
   async ReadFn(irp);
   async CancelFn(irp);
}
```

Figure 11. Example of null-dereference checking on `daytona` driver. Every dereference of a pointer has an implicit non-null check preceding it.

However, by limiting the input to contain an empty list, most of the interesting logic related to list traversal in the code does not get exercised.

There are three versions of this driver, depending on which dispatch routine is chosen (`read`, `write`, or `ioctl`). In the absence of preconditions, we get false alarms in each of the three versions, whereas CBUGS does not report any false alarms. For `read`, CBUGS requires the use of *AsyncGeneral* filter. With *AsyncAsSync* alone, it reports a false alarm where the input list is not well-formed. This example shows that in the presence of complicated preconditions, one needs *AsyncGeneral*, although at the expense of more runtime.

5.1.2 Null-Dereference Checking

We also checked several drivers for the absence of null-dereferences. Each pointer dereference was preceded with a non-null assertion. As shown in Fig. 10, this property introduced a large number of assertions in the program.

Fig. 11 shows an example (motivated by `daytona`) that requires the $AsyncGeneral$ filter to remove the false alarms. The harness `Test` needs the preconditions that both `irp` and `irp->ptr` are non-null, to ensure that the concurrent program does not fail any assertions. It is easy to observe that the illegal inputs where `irp` is null fails by executing `ReadFn(irp)` sequentially, and is, thus, filtered away by $AsyncAsSync$.

Consider the non-null check for the dereference `*irp->ptr`. If we restrict the executions of `Test` to only consider $AsyncAsSync$, the `*irp->ptr` is dereferenced only when either `irp->b` or `irp->cancel` is set in the input. This implies that using the $AsyncAsSync$ 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 $AsyncGeneral$ 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

Name	LOC	Prop.	Buggy?	CBUGS				Assert Supression		
	(Asserts)			#LF	#FP	#FN	Time	#FP	#FN	Time
daytona	485 (10)	Cancel	No	1	0	0	122	1	0	17
daytona_bug1	484 (10)	Cancel	Seq.	2	0	1	136	1	0	21
daytona_bug2	485 (10)	Cancel	Int.	1	0	0	110	1	0	34
daytona_bug3	484 (10)	Cancel	Seq.	2	0	1	161	3	0	41
daytona_bug4	485 (10)	Cancel	Seq.	2	0	1	162	1	0	19
daytona_bug5	485 (10)	Cancel	Int.	1	0	0	126	1	0	29
ndisprot_read	588 (35)	Cancel	No	10 (AG)	0	0	2894	3	0	38
ndisprot_write	588 (35)	Cancel	No	4	0	0	217	1	0	15
ndisprot_ioctl	588 (35)	Cancel	No	7	0	0	424	1	0	32
mouclass_bug1	582 (19)	Cancel	Seq.	10	0	1	905	4	0	131
mouclass_bug2	582 (19)	Cancel	Int.	10	0	0	944	3	1	186
daytona	485 (223)	NullDeref	No	13 (AG)	0	0	459	3	0	43
kbdclass_read	695 (346)	NullDeref	No	1	0	0	227	11	0	846
kbdclass_ioctl	695 (346)	NullDeref	No	1	0	0	229	10	0	737
mouclass_read	582 (271)	NullDeref	No	1	0	0	98	10	0	965
mouclass_ioctl	582 (271)	NullDeref	No	1	0	0	91	9	0	918

Figure 10. Results obtained from running CBUGS on Windows device drivers. Times are in seconds.

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 assumptions 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.

References

[1] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Principles of Programming Languages (POPL '05)*, pages 98–109, 2005.

[2] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87, 2005.

[3] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.

[4] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *Programming Language Design and Implementation (PLDI '10)*, pages 330–340, 2010.

[5] J. Burnim, T. Elmas, G. C. Necula, and K. Sen. Ndseq: runtime checking for nondeterministic sequential specifications of parallel correctness. In *Programming Language Design and Implementation (PLDI '11)*, pages 401–414, 2011.

[6] L. Cordeiro and B. Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *International Conference on Software Engineering (ICSE '11)*, pages 331–340, 2011.

[7] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[8] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[9] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Programming Language Design and Implementation (PLDI '07)*, pages 435–445, 2007.

[10] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *Principles of Programming Languages*, pages 411–422, 2011.

[11] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, 2001.

[12] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[13] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *Principles of Programming Languages*, pages 339–350, 2007.

[14] T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Static Analysis Symposium (SAS '03)*, LNCS 2694, pages 295–315, 2003.

[15] S. Lahiri, S. Qadeer, and Z. Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Computer Aided Verification*, 2009.

[16] K. R. M. Leino, T. D. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1-3):209–226, 2005.

[17] Poirot: The Concurrency Sleuth. `http://research.microsoft.com/en-us/projects/poirot/`.

[18] K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *Computer Aided Verification*, pages 300–314, 2006.

[19] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *International Symposium on Software Testing and Analysis (ISSTA '06)*, pages 157–168, 2006.

[20] O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu. Automated environment generation for software model checking. In *Automated Software Engineering (ASE '03)*, pages 116–129, 2003.

[21] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE '07)*, pages 205–214, 2007.

[22] Microsoft windows driver kit (WDK). `http://www.microsoft.com/whdc/devtools/ddk/default.mspx`.

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 DIFFERROR(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_1. This implies that the non-determinism in P_1 is *angelic* (in contrast to the *demonic* nondeterminism 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. $wlp(\mathsf{choose}\ \mathtt{x}, \phi) = \exists x.\phi$, whereas $wlp(\mathsf{havoc}\ \mathtt{x}, \phi) = \forall x.\phi$, where $wlp(.,.)$ refers to *weakest liberal precondition* predicate transformer [8]). This interacts badly with the use of 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 DIFFERROR 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 ϕ 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 a (ii) set of paths and finds an error trace that avoids the set of paths specified, or NOBUG therwise (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 ϕ 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 ϕ respecively (line 9 and line 10).

The predicate transformer $wlp(.,.)$ is extended for havoc statements:

$$pre(\mathsf{havoc}\ \mathtt{x}, \psi) \quad = \quad \exists x.\ \psi$$

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

Algorithm 2 Algorithm for generating a VC for DIFFERROR

Require: Programs P_1 and P_2
Ensure: A formula representing the VC for $\text{DIFFERROR}(P_1, P_2)$

ϕ := false
A := $\{\}$
loop
  r := FINDBUG(P_1, A)
  if r = NOBUG **then**
    return VC(assume $\neg\phi$; P_2.*body*)
  end if
  Let $\text{TRACE}(t, i) = r$
  A := $A \cup \{t\}$
  ϕ := $\phi \vee$ *pre*(t, true)
end loop

THEOREM A.1. *Given programs P_1 and P_2, if ψ be the formula returned by Alg. 2, then $\text{DIFFERROR}(P_1, P_2)$ holds if and only if ψ 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 $\text{DIFFERROR}(P_1, P_2)$ 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 $\texttt{P}_1$. Instead, Alg. 2 generates the following precondition to P_2 by capturing all the inputs that fail $\texttt{P}_1$:

$$\neg(\exists i, j :: j \neq 0 \wedge \neg \texttt{a}[i] \geq 0)$$

which is equivalent to $\forall i :: \texttt{a}[i] \geq 0$, and sufficient to prove the assertion in $\texttt{P}_2$ 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 $\text{DIFFERROR}(P_1, P_2)$ to a logical formula, we can hope to leverage advances in automated theorem provers to obtain more precise and efficient solution for $\text{DIFFERROR}(P_1, P_2)$.