Concurrency Debugging with Differential Schedule Projections

Nuno Machado
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
nuno.machado@tecnico.ulisboa.pt

Brandon Lucia *
Carnegie Mellon University, USA
blucia@ece.cmu.edu

Luís Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
ler@tecnico.ulisboa.pt

Abstract

We present Symbiosis: a concurrency debugging technique based on novel *differential schedule projections* (DSPs). A DSP shows the small set of memory operations and data-flows responsible for a failure, as well as a reordering of those elements that avoids the failure. To build a DSP, Symbiosis first generates a *full, failing, multithreaded schedule* via thread path profiling and symbolic constraint solving. Symbiosis selectively reorders events in the failing schedule to produce a *non-failing, alternate schedule*. A DSP reports the ordering and data-flow *differences* between the failing and non-failing schedules. Our evaluation on buggy real-world software and benchmarks shows that, in practical time, Symbiosis generates DSPs that both isolate the small fraction of event orders and data-flows responsible for the failure, and show which event reorderings prevent failing. In our experiments, DSPs contain 81% fewer events and 96% fewer data-flows than the full failure-inducing schedules. Moreover, by allowing developers to focus on only a few events, DSPs reduce the amount of time required to find a valid fix.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Diagnostics

General Terms Algorithms, Design, Reliability

Keywords Concurrency, Debugging, Symbolic Execution, Constraint Solving, Differential Schedule Projection

1. Introduction

Concurrent and parallel programming are the new norm and are much more difficult than sequential programming. Shared-memory multi-threading is especially wide-spread, and requires programmers to reason about multiple threads of execution that interact by reading and writing shared variables. Operations in different threads are unordered by default, unless ordered by synchronization, and different executions may non-deterministically adhere to different *schedules* of unordered operations that produce different results. A key challenge is that most schedules are correct, but some may permit a multi-threaded sequence of shared-memory accesses that leads to undesirable behavior, like a crash. Such a schedule causes a *failure* and is the result of a *concurrency bug* (*i.e.*, a mistake in the code that incorrectly permits a failing schedule).

Eliminating concurrency bugs is extremely difficult. Failing schedules may manifest rarely and reproducing them is often difficult. Prior work has addressed reproducibility with a number of different strategies, including *deterministic record and replay* (R&R) (both order-based [20, 23, 49] and search-based [35, 39, 53]) and *deterministic execution* [4, 10, 37]. These techniques help produce an execution that fails, but simply reproducing a failure may provide no insight into its cause. The key to debugging is understanding a failure's *root cause*, *i.e.*, the set of event orderings that are *necessary* for failure. The number of events that comprise a root cause is typically small [6], but it is often unclear which events in a full schedule to focus on. Any operation in any thread may have led to the failure and blindly analyzing a full schedule is a metaphorical search for a *needle in a haystack*. Even if the programmer *finds* the root cause, they still must understand how to change the code in such a way the problematic events do not execute in the failure-inducing order, which is also difficult.

We present Symbiosis, a system which helps finding and understanding a failure's root cause, as well as fixing the underlying bug. Figure 1 presents a schematic view of our system. Symbiosis first collects single-threaded path profiles from a concrete, failing execution. The profiles guide a symbolic execution, yielding per-thread symbolic event traces compatible with the failure. These are then used to generate a Satisfiability Modulo Theory (SMT) formulation, the solution to which represents a multi-threaded failing schedule. To prune irrelevant events from the failing schedule, Symbiosis generates an *unsatisfiable* SMT formulation encoding the failing schedule, but the absence of the failure. As a result, the SMT solver reports a subset of constraints that conflict in the unsatisfiable SMT formulation; their corresponding event orderings are necessary for the failure, and form the pruned *root cause schedule*. The root cause schedule is used in another SMT formulation to compute *non-failing, alternative schedules* that comprise reorderings of the root cause schedule's events. Symbiosis enhances the debugging utility of the root cause schedule by reporting *only the important ordering and data-flow differences* between failing and non-failing schedules. We call the output of our novel debugging approach a *differential schedule projection* (DSP).

DSPs simplify debugging for two main reasons. First, by showing only what differs between a failing and non-failing schedule, the programmer sees only a very small number of relevant operations, rather than a full schedule. Second, DSPs illustrate, not only the failing schedule, but also the way execution *should* behave, if not to fail. Seeing the different event orders side-by-side helps understand the failure and, often, how to fix the bug. Crit-

* This work was done in part while the author was a Researcher at Microsoft Research.

PLDI'15, June 13–17, 2015, Portland, OR, USA
Copyright 2015 ACM 978-1-4503-3468-6/15/06...$15.00
http://dx.doi.org/10.1145/2737924.2737973

ically, Symbiosis produces a DSP from a *single* failing schedule, enabling its use for failures observed rarely (*i.e.*, in production) and does not require repeated program executions like prior work [31]. Our evaluation in Section 5 shows that DSPs have, on average, 81% fewer events than full schedules and shows qualitatively, with case studies, that DSPs help understand failures and fix bugs.

To summarize, our contributions are: *i)* An SMT constraint formulation, based on the computed failing schedule, that identifies the sub-schedule that is a failure's root cause. *ii)* An SMT constraint formulation that systematically varies the order of root cause events to find alternative non-failing schedules similar to the original failing schedule. *iii)* A novel *differential schedule projection* methodology that isolates important control and data-flow changes between failing and non-failing schedules computed by Symbiosis. *iv)* An implementation of Symbiosis for C/C++ and Java and an evaluation, showing the debugging efficacy of Symbiosis and its applicability to failure avoidance and failure reproduction.

2. Background

Symbiosis helps with concurrency debugging by leveraging prior work on symbolic execution and Satisfiability Modulo Theory (SMT) solving. This section briefly reviews these topics.

Concurrency Bugs. Concurrency bugs are errors in code that permit multi-threaded schedules that lead to a failure. Concurrency bugs have been studied extensively in the literature [13, 16, 18, 28, 31–34, 41, 51, 52]. Data-races [13, 16, 41], atomicity violations [18, 28, 32], and ordering violations [29, 30, 38, 51, 52] are different types of concurrency bugs studied by prior work. These bugs vary in their mechanism and result. For example, while data-races may lead to violations of sequential consistency [27], atomicity violations may lead to unserializable behavior of atomic regions. We defer to the literature for a detailed discussion of these bug types and their failure modes. Instead, we just emphasize that they share the following important characteristic: they lead to a failure when they permit operations in different threads to execute in an order that should be forbidden. Symbiosis attacks the debugging problem by identifying such incorrect operation orderings that constitute the root cause of a failure.

Symbolic Execution. *Symbolic execution* [25] explores the space of possible executions of a program by emulating or directly executing its statements. During symbolic execution, some variables have concrete values (*e.g.*, 12), and other variables, like unknown inputs, have *symbolic* values. A symbolic value represents a set of possible concrete values. Assignments to and from symbolic variables and operations involving symbolic variables produce results that are also symbolic. When an execution reaches a branch dependent on a symbolic variable, it spawns two identical copies of the execution state – one in which the branch is taken, one in which the branch is not taken. Spawned copies continue independently along these different *paths* and the process repeats for every new symbolic branch. Each path has a *path constraint*, encoding all branch outcomes on that path. Thus, the path constraint determines possible concrete values for symbolic variables that lead execution down a particular path. As we describe in Section 3.3, Symbiosis uses symbolic execution to find a path in each thread that leads to a failure. Like unknown inputs, Symbiosis treats *shared variables* as symbolic, because they might be modified non-deterministically by any thread during a multi-threaded execution.

SMT Solvers. A *Satisfiability Modulo Theories* (SMT) solver is a tool that, given a formula over variables, finds a satisfying assignment of the variables or reports that it is unsatisfiable. SMT is based on boolean satisfiability (SAT). However, SMT is more expressive than SAT, for example, handling arithmetic. SAT and SMT are NP-complete, but decades of research have produced solvers (*e.g.*, Z3 [9]) that practically solve large problems. Practical SMT has found use in many areas: hardware [12] and software verification [40], program analysis [26], and test generation [45].

CLAP [21] and our work link concurrency, SMT, and symbolic execution. A symbolic path constraint corresponds to an SMT formula that constrains variables at each point in a *sequential* execution [7, 45]. Thus, a *concurrent* symbolic execution corresponds to *i)* a combination of the SMT formulae for each thread's symbolic execution, and *ii)* additional constraints encoding inter-thread data-flow and synchronization. CLAP's goal was to reproduce failed executions, so it added constraints corresponding to a failure's manifestation. Symbiosis adds these constraints as well.

When an SMT formula is unsatisfiable, some SMT solvers [9] are able to *explain why* by reporting which constraints conflict in an unsatisfiability core, or *UNSAT Core*. BugAssist [24] pioneered the use of the UNSAT core to help isolate errors in sequential programs, but Symbiosis makes novel use of this feature to debug concurrency errors and reduce the information it must analyze when building differential schedule projections.

3. Symbiosis

Symbiosis is a technique for concisely reporting the root cause of a failing multi-threaded execution, alongside a set of non-failing, alternate executions of the events that make up the root cause. Symbiosis produces *differential schedule projections*, which reveal bugs' root causes and aid in debugging. Symbiosis has five phases, as depicted in Figure 1:

1) Symbolic trace collection. In a concrete, failing program execution, Symbiosis traces the basic blocks executed in each thread independently. The per-thread path profiles are used to guide symbolic execution, producing a set of per-thread traces with symbolic information (*e.g.* path conditions and read-write accesses to shared variables).

2) Failing schedule generation. Symbiosis produces an SMT formula that corresponds to the symbolic execution trace. The formula includes constraints that represent each thread's path, as well as the failure's manifestation, memory access orderings, and synchronization orderings. The solution to the SMT formula corresponds to a complete, failing, multi-threaded execution. In other words, this solution specifies the ordering of events that triggers the error.

3) Root cause sub-schedule generation. Symbiosis produces an SMT formula corresponding to the symbolic trace, but specifies that the execution should not fail, by negating the failure condition. Combined with the constraints representing the order of events in the full, failing schedule, the SMT instance is unsatisfiable. The SMT solver produces an UNSAT core that contains the constraints representing the execution event orders that conflict with the absence of the failure. Those event orders are necessary for the failure to occur, *i.e.*, the failure's root cause sub-schedule.

4) Alternate sub-schedule generation. Symbiosis examines each pair of events from different threads in the root cause. For each pair, Symbiosis produces a new SMT formula, identical to the one used to find the root cause, but with constraints implying the ordering of the events in the pair reversed. When Symbiosis finds an instance that is satisfiable, the corresponding schedule is very similar to the failing schedule, but does not fail. Symbiosis reports the alternate, non-failing schedule that is identical to the failing schedule, but with the pair of events reordered.

5) Differential schedule projections and failure avoidance. Symbiosis produces a differential schedule projection by comparing the failing schedule and the alternate, non-failing sub-schedule. The DSP shows how the two schedules differ in the order of their events and in their data-flow behavior. Additionally, as the reordered pair from the alternate non-failing schedule eliminates an

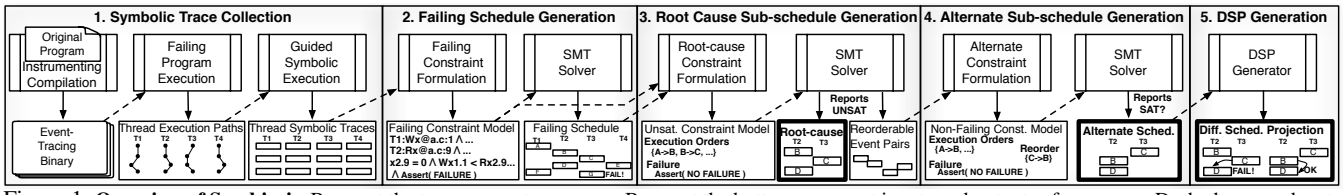

Figure 1: **Overview of Symbiosis.** Boxes at the top represent processes. Boxes at the bottom represent inputs and outputs of processes. Dashed arrows denote an input relationship and solid arrows denote and output relationship. **Boldface** boxes represent the final outputs of Symbiosis.

event order necessary for the failure to occur, it can be leveraged by a dynamic failure avoidance system [31] to prevent future failures.

3.1 A Running Example

To illustrate the main concepts of Symbiosis, we use a running example that consists of the modified version of `pfscan` file scanner studied in prior work [11]. A slightly simplified snippet of the program's code is depicted in Figure 2a. The program uses three threads. The first thread enqueues elements into a shared queue. The two other threads attempt to dequeue elements, if they exist. A shared variable, named *filled*, records the number of elements in the queue. The code in the *get* function checks that the queue is non-empty (reading *filled* at line 10), decreases the count of elements in the queue (updating *filled* at line 20), then dequeues the element.

The code has a concurrency bug because it does not ensure that the check and update of *filled* execute atomically. The lack of atomicity permits some unfavorable execution schedules in which the two consumer threads both attempt to dequeue the queue's last element. In that problematic case, both consumers read that the value of *filled* is 1, passing the test at line 10. One of the threads proceeds to decrement *filled* and dequeue the element. The other reaches the assertion at line 19, reads the value 0 for filled, and fails, terminating the execution. Figure 2b shows the interleaving of operations that leads to the failure in a concrete execution.

The next sections show how Symbiosis starts from a concrete failing execution (like the one in Figure 2b), produces a focused root cause, and reports its significant differences from alternate non-failing schedules to aid in debugging.

3.2 Symbolic Trace Collection

Like CLAP [21], Symbiosis avoids the overhead of directly recording the exact read-write linkages between shared variables that lead to a failure. Instead, Symbiosis collects only per-thread path profiles from a failing, concrete execution. As in prior work [21], Symbiosis's path profile for a thread consists of the sequence of executed basic blocks for that thread in the failing execution.

Symbiosis uses the per-thread path profiles to guide a symbolic execution of each thread and to produce each thread's separate symbolic execution trace. Symbolic execution normally explores all paths, following the path along both branch outcomes. Symbiosis, in contrast, guides the symbolic execution to correspond to the per-thread path profiles by considering only paths that are compatible with the basic block sequence in the profile. As symbolic execution proceeds, Symbiosis records information about control-flow, failure manifestation, synchronization, and shared memory accesses in each per-thread symbolic execution trace. Together, the traces are compatible with the original, failing, multi-threaded execution.

Each per-thread, symbolic, execution trace contains four kinds of information. First, each trace includes a path condition that permits the failure to occur. A trace's path condition is the sequence of control-flow decisions made during the trace's respective execution. Second, the trace for the thread that experienced the failure must include the event that failed (*e.g.*, the failing assertion). Third, the trace must record synchronization operations, noting their type (*e.g.*, lock, unlock, wait, notify, fork, join, *etc.*), and the synchronization variable involved (*e.g.*, the lock address) if applicable. Fourth, the trace must record loads from and stores to shared memory locations. A key aspect of the shared memory access traces is that these are *symbolic*: loads always read fresh symbolic values and stores may write either symbolic or concrete values. Recall from Section 2 that a symbolic value holds the last operation that manipulated a value. Also, a symbolic value may, itself, be an expression that refers to other symbolic or concrete values.

Note that any technique for collecting concrete path profiles and generating symbolic traces is adequate. In our implementation of Symbiosis that targets C/C++, we use a technique very similar to the front-end of CLAP [21]: Symbiosis records a basic block trace and uses KLEE to generate per-thread symbolic traces conformant with the block sequence. Symbiosis for Java uses Soot [46] to collect path profiles and JPF [48] for symbolic execution. With some additional engineering effort, Symbiosis could also use Pex [45] for C#, or general R&R techniques [20, 53].

Trace Collection Example. Figure 2c illustrates a symbolic trace collection for our running example: it shows the execution path followed by each thread for the failing schedule in Figure 2b and the corresponding symbolic trace produced by Symbiosis. Each path condition in the trace represents a control-flow outcome in the original execution (*e.g. filled@2.10* > 0 denotes that thread T2's should read a value greater than zero from *filled* at line 10). Thread T2's trace includes the assertion that leads to the failure. Each trace includes both symbolic and concrete values in their memory access traces, as well as synchronization operations from the execution. Note that we slightly simplified the threads' traces to keep the figure uncluttered. *enqueue* and *dequeue* also access shared data but we only show operations that manipulate *filled* and perform synchronization because they are sufficient to illustrate the failure.

3.3 Failing Schedule Generation

The symbolic, per-thread traces do not explicitly encode the multi-threaded schedule that led to the failure. Symbiosis uses the information in the symbolic traces to construct a system of SMT constraints that encode information about the execution. The solution to those SMT constraints corresponds to a multi-threaded schedule that ends in failure and is compatible with each per-thread symbolic trace. This section describes how the constraints are computed.

The SMT constraints refer to two kinds of unknown variables, namely the *value variables* for the fresh symbolic symbols returned by read operations and the *order variables* that represent the position of each operation from each trace in the final, multi-threaded schedule. We notate value variables as $var_{t.l}$, meaning the value read from variable *var* by thread t at line l. We notate order variables as $Op_{t.l}$, meaning the order of instruction *Op* executed by thread t at line l, where *Op* can be a read (R), write (W), lock (L), unlock (U), or other synchronization operations such as wait/signal (our notation differs slightly from that in [21] for clarity).

Figure 2d shows part of the system of SMT constraints generated by Symbiosis from the symbolic traces presented in Figure 2c. The system, denoted Φ_{fail}, can be decomposed into five sets of constraints:

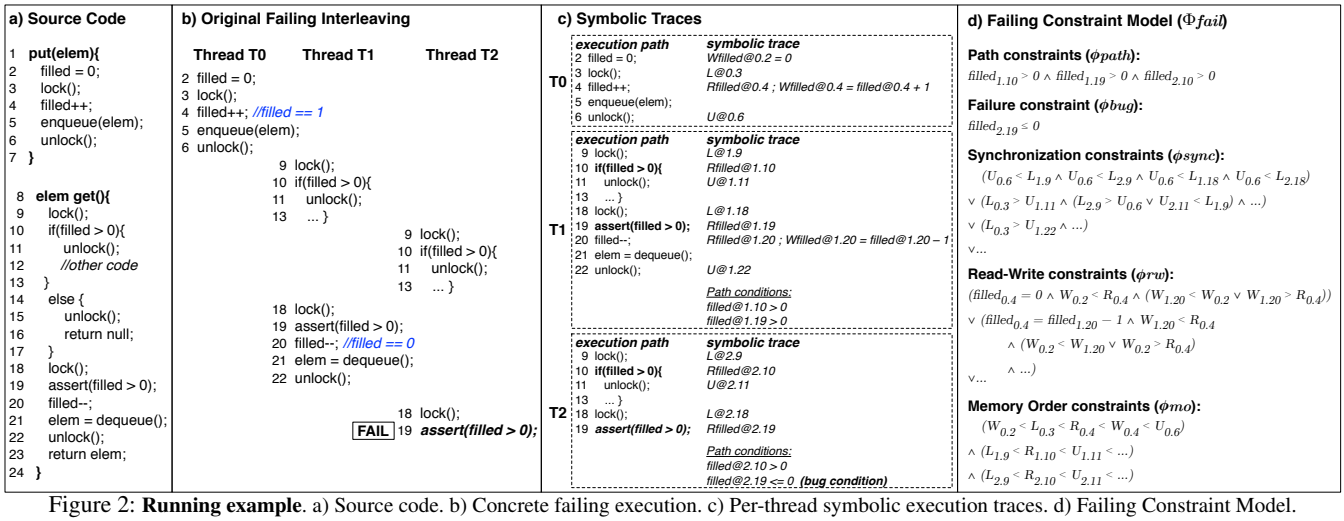

Figure 2: **Running example**. a) Source code. b) Concrete failing execution. c) Per-thread symbolic execution traces. d) Failing Constraint Model.

$$\Phi_{fail} = \phi_{path} \wedge \phi_{bug} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo}$$

where ϕ_{path} encodes the control-flow path executed by each thread, ϕ_{bug} encodes the occurrence of the failure, ϕ_{sync} encodes possible inter-thread interactions via synchronization, ϕ_{rw} encodes possible inter-thread interactions via shared memory, and ϕ_{mo} encodes possible operation reorderings permitted by the memory consistency model. The following paragraphs explain how Symbiosis derives each set of constraints from the symbolic execution traces.

Path Constraint ***(ϕ_{path})*** The path SMT constraint encodes branch outcomes during symbolic execution. Symbiosis gathers path conditions by recording the branch outcomes along the basic block trace from the concrete path profile. A thread's path constraint is the conjunction of the path conditions for the execution of the thread in the symbolic trace. The ϕ_{path} constraint is the conjunction of all threads' path constraints. Each conjunct represents a single control-flow decision by constraining the value variables for one or more symbolic operands. In our running example, the shared variable *filled* is symbolic, resulting a ϕ_{path} with three conjuncts. The three conjuncts express that the value of *filled* should be greater than 0 when thread T1 executes lines 10 and 19, as well as when thread T2 executes line 10. Figure 2d shows ϕ_{path} for our example.

Failure Constraint ***(ϕ_{bug})*** The failure SMT constraint expresses the failure's necessary conditions. The constraint is an expression over value variables for symbolic values returned by some subset of read operations (*e.g.*, those in the body of an *assert* statement). Figure 2d shows ϕ_{bug} for the running example, representing the sufficient condition for the assertion in thread T2 to fail.

Synchronization Constraints ***(ϕ_{sync})*** There are two types of synchronization SMT constraints: *partial order constraints* and *locking constraints*. Partial order constraints represent the partial order of different threads' events resulting from *fork/join/wait/signal* operations. For instance, a constraint for *fork* states that the first event of a child thread must occur after the *fork* operation in the parent thread. Locking constraints represent the mutual exclusion effects of *lock* and *unlock* operations. We define a locking constraint for two threads, t and t', stating that each thread executes a critical section protected by a *lock* operation and an *unlock* operation – namely L/U and L'/U'. The constraint is a disjunction of SMT expressions representing two possible cases. In the first case, t acquires the lock first and U happens before L'. In the second case, t' acquires the lock first and U' happens before L. For each such pair, the disjunction that becomes part of ϕ_{sync} is composed of two SMT constraints, defined over order variables, representing the alternation of the two cases. Figure 2d shows a subset of the locking constraints for our running example that involve the lock/unlock pair of thread T0 $(L_{0:3}, U_{0:6})$.

Read-Write Constraints ***(ϕ_{rw})*** Read-write SMT constraints encode the matching between read and write operations that leads to a particular read operation reading the value written by a particular write operation. Read-write constraints model the possible inter-thread interactions via shared memory. A read-write constraint encodes that, for every read operation r on a variable v, if r is matched to a write w of the same variable, then the order variable (and hence execution order) for all other writes on v are either less than that of w or greater than that of r. The constraint also implies that r's value variable takes on the symbolic value written by w. Note that read-write SMT constraints are special in that they affect order variables and value variables.

In the running example, thread T0 reads *filled* at line 4. If T0 reads 0 at that point, then the most recent write to *filled* must be the one at line 2. The matching of that read and write implies that the order of the write must precede read operation (*i.e.* $W_{0.2} < R_{0.4}$), and that all the other writes to *filled* (*e.g.* $W_{1.20}$) either occur before $W_{0.2}$ or after $R_{0.4}$. The same reasoning is also applied for the remaining reads of the program on symbolic variables.

Memory Order Constraints ***(ϕ_{mo})*** The memory-order constraints specify the order in which instructions are executed in a specific thread. Although is possible to express different memory consistency models [21], in this paper we opted not to focus on relaxed memory ordering, instead focusing on sub-schedule generation and differential schedule projections. Therefore, here we consider sequential consistency only, meaning statements in a thread execute in program order. For the running example in Figure 2b, the memory order constraint requires that operations in thread T0 respect the constraint $W_{0.2} < L_{0.3} < R_{0.4} < W_{0.4} < U_{0.6}$.

3.4 Root Cause Sub-schedule Generation

Each order variable referred to by an SMT constraint represents the ordering of two program events from the separate single-threaded symbolic traces. A binding of truth values to the SMT order variables corresponds directly to an ordering of operations in the otherwise unordered, separate, per-thread traces. Solving the constraint system binds truth values to variables, producing a multi-threaded schedule. The constraint system includes a constraint represent-

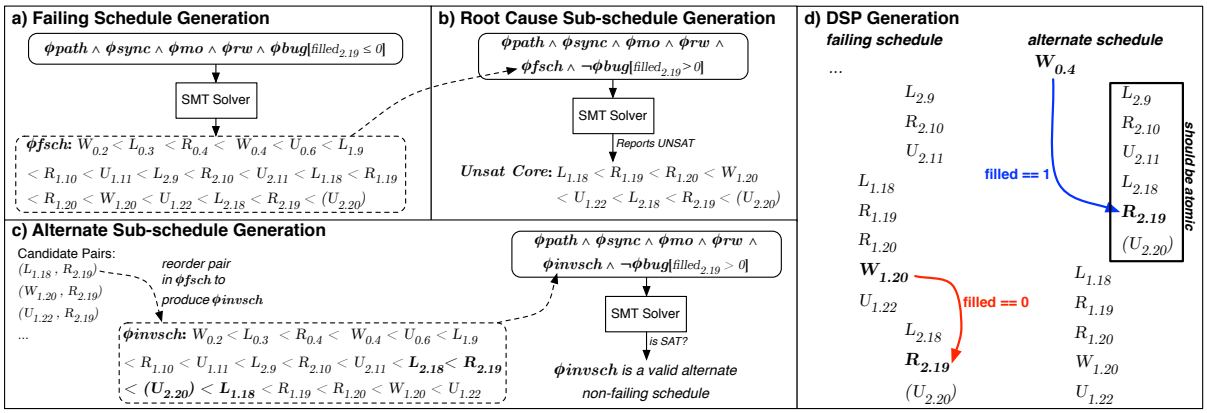

Figure 3: **Root cause and Alternate schedule generation**. a) Possible failing schedule produced by the SMT solver for the constraint system in Figure 2d ($(U_{2.20})$ represents a synthetic unlock event). b) Root cause sub-schedule, which corresponds to the UNSAT core produced by the solver. c) Candidate pair reordering and respective alternate schedule. d) Differential Schedule Projection generated by Symbiosis.

ing the occurrence of the failure, so the produced multi-threaded schedule manifests the failure (ϕ_{bug}). Solving the generated SMT formulae, Symbiosis produces a full, failing, multi-threaded schedule ϕ_{fsch}. The entire multi-threaded schedule may be long, complex, and may contain information that is irrelevant to the root cause of the failure. Symbiosis uses a special SMT formulation to produce a *root cause sub-schedule* that prunes some operations in the full schedule, but preserves event orderings that are *necessary* for the failure to occur. To compute the root cause sub-schedule, Symbiosis generates a new constraint system, denoted Φ_{root}, that is *designed to be unsatisfiable* in a way that reveals the necessary orderings. Symbiosis leverages the ability of the SMT solver to produce an explanation, of why a formula was unsatisfiable, to report only those necessary orderings.

To build the root cause sub-schedule SMT formula, Symbiosis logically inverts the *failure constraint*, effectively requiring the failure not to occur (*i.e.* $\neg\phi_{bug}$). Symbiosis adds constraints to the formula that directly encode the event orders in ϕ_{fsch} (*i.e.* the full, failing schedule that was previously computed). The complete root cause sub-schedule formula is then written as follows:

$$\Phi_{root} = \phi_{path} \wedge \neg\phi_{bug} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo} \wedge \phi_{fsch}$$

The original SMT formula that Symbiosis used to find the full failing schedule considers *all* possible multi-threaded schedules that are consistent with the symbolic, per-thread schedules. In contrast, the root cause sub-schedule SMT formula adds the failing schedule ϕ_{fsch} constraint, accommodating *only* the full, failing schedule. Combining the inverted failure constraint and the ordering constraints for the full, failing schedule make an unsatisfiable constraint formula: the inverted failure constraint requires the failure not to occur and the failing schedule's ordering constraints require the failure to occur.

When an SMT solver, like Z3, attempts to solve the unsatisfiable formula, it produces an unsatisfiable (UNSAT) core which is a subset of constraint clauses that conflict, leaving the formula unsatisfiable. The UNSAT core for Φ_{root} encodes the subset of clauses that conflict because the ϕ_{fsch} requires the failure to occur and $\neg\phi_{bug}$ requires the failure not to occur. The event orderings that correspond to those conflicted constraints are the ones in ϕ_{fsch} that imply ϕ_{bug}. Those orderings are a necessary condition for the failure; their corresponding constraints, together with $\neg\phi_{bug}$ are responsible for the unsatisfiability of Φ_{root}. Reporting the sub-schedule corresponding to the UNSAT core yields fewer total events than are in the full, failing schedule, yet includes event orderings necessary for the failure.

Figure 3a shows a possible failing schedule produced by the constraint system corresponding to the execution depicted in Figure 2d. The failure constraint ϕ_{bug} requires the corresponding execution to manifest the failure. The generated path and memory access constraints are compatible with the failure and the system is satisfiable, producing the failing execution trace shown (ϕ_{fsch}). Note that Symbiosis inserts a *synthetic unlock* event $(U_{2.20})$ in the model, in order to preserve the correct semantics of synchronization constraints (see § 4).

In Figure 3b, the failure constraint is *negated*, requiring the corresponding execution not to manifest the failure (*i.e.*, $filled_{2.19} > 0$ and the assertion at line 19 does not fail). The UNSAT core shows why Φ_{root} is unsatisfiable: the negated failure constraint conflicts with the subset of ordering constraints from ϕ_{fsch} that cause *filled* to be less than 0 when thread 2 executes line 19 (note that $R_{2.19}$ defines the value of $filled_{2.19}$ read by the assertion).

In our experience, the UNSAT core produced by Z3 is typically not minimal. As a result, while helpful, an UNSAT core alone is not sufficient for debugging and necessitates a Differential Schedule Projection to isolate a bug's root cause.

3.5 Alternate Sub-schedule Generation

In addition to reporting the bug's root cause, Symbiosis also produces *alternate, non-failing sub-schedules*. These alternate sub-schedules are *non-failing* variants of the root cause sub-schedule, with the order of a single pair of events reversed. Alternate sub-schedules are the key to building *differential schedule projections* (§ 3.6). Symbiosis generates alternate, non-failing sub-schedules after it identifies the root cause. To generate an alternate sub-schedule, Symbiosis selects a pair of events from different threads that were included in the bug's root cause. Symbiosis then generates a new constraint model, like the one used to identify the root cause. We call this model Φ_{alt}. The Φ_{alt} model includes the inverted failure constraint. The model also includes a set of constraints, denoted ϕ_{invsch}, that encode the original full, failing schedule, *except the constraint representing the order of the selected pair of events is inverted*. Inverting the order constraint for the pair of events yields the following new constraint model.

$$\Phi_{alt} = \phi_{path} \wedge \neg\phi_{bug} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo} \wedge \phi_{invsch}$$

The new Φ_{alt} model corresponds to a different, full execution schedule in which the events in the pair occur in the order opposite to that in the full, failing schedule. If this new model is satisfiable, then reordering the pair of events in the full, failing schedule pro-

duces a new alternate schedule in which the failure does not manifest, as shown in Figure 3c.

If there are many event pairs in the root cause, then Symbiosis must generate and attempt to solve many constraint formulae. Symbiosis systematically evaluates a set of candidate pairs in a fixed order, choosing the pair separated by the fewest events in the original schedule first. The reasoning behind this design choice is that events in a pair that are further apart are less likely to be meaningfully related and, thus, less likely to change the failure behavior when their order is inverted. By default, we configured Symbiosis to stop after finding a single alternate, non-failing schedule. However, the programmer can instruct Symbiosis to continue generating alternate schedules, given that studying sets of schedules may reveal useful invariants [34].

Arbitrary operation reorderings may yield infeasible schedules. Reordering may change inter-thread data flow, producing values that are inconsistent with a prior branch dependent on those values. The inconsistency between the data and the execution path makes the execution infeasible. Symbiosis produces only feasible schedules by including path constraints in its SMT model. If a reordering leads to inconsistency, the SMT path constraints become unsatisfiable and Symbiosis produces no schedule.

3.6 Differential Schedule Projection

Differential schedule projection (DSP) is a novel debugging methodology that uses root cause sub-schedules and non-failing alternate sub-schedules. The key idea behind debugging with a DSP is to show the programmer the salient differences between failing, root cause schedules and non-failing, alternate schedules. Examining those differences helps the programmer understand how to *fix* the bug, rather than helping them understand the failure only, like techniques that solely report failing schedules.

Concretely, a DSP consists of a pair of sub-schedules decorated with several pieces of additional information. The first sub-schedule is the root cause sub-schedule, which is the *source* of the projection. The second sub-schedule is an excerpt from the alternate, non-failing schedule, which is the *target* of the projection.

The order of memory operations differs between the schedules and, as a result, the outcome of some memory operations may differ. A read may observe a different write's result in one schedule than it observed in another, or two writes may update memory in a different order in one schedule than in another, leaving memory in a different final state. These differences are precisely the changes in data-flow that contribute to the failure's occurrence. Symbiosis highlights the differences by reporting *data-flow variations*: data-flow between operations in the source sub-schedule that do not occur in the target sub-schedule and *vice versa*.

To simplify its output, Symbiosis reports only a subset of operations in the source and target sub-schedules. An operation is included if it makes up a data-flow variation or if it is one of a pair of operations that occur in a different order in one sub-schedule than in the other. Alternate, non-failing schedules vary in the order of a single pair of operations, so all operations that precede both operations in the pair occur in the same order in the source and target sub-schedules. Symbiosis does not report operations in the common prefix, unless they are involved in a data-flow variation. By selectively including only operations related to data-flow and ordering differences, a DSP focuses programmer attention on the changes to a failing execution that lead to a non-failing execution. Understanding those changes are the key to changing the program's code to fix the bug. For instance, the DSP in Figure 3d shows that the data-flow $W_{1.20} \rightarrow R_{2.19}$ (in ϕ_{fsch}) changes to $W_{0.4} \rightarrow R_{2.19}$ (in ϕ_{invsch}). This data-flow variation is the actual bug's root cause. In addition, note that, by reordering the events, the DSP also suggests that the block of operations $L_{2.9}$–$(U_{2.20})$ should execute atomically, which indeed fixes the bug.

4. Implementation

4.1 Instrumenting Compiler and Runtime

Our Symbiosis prototype implements trace collection for both C/C++ and Java programs. C/C++ programs are instrumented via an LLVM function pass. Java programs are instrumented using Soot [46], which injects path logging calls into the program's bytecode. Like CLAP, we assign every basic block with a static identifier and, at the beginning of each block, we insert a call to a function that updates the executing thread's path. The function logs each block as the tuple *(thread Id, basic block Id)* whenever the block executes. The path logging function is implemented in a custom library that we link into the program. Although our prototype is fully functional, it has not been fully optimized yet. For instance, lightweight software approaches (*e.g.*, Ball-Larus [2]) or a hardware accelerated approaches (*e.g.*, Vaswani *et al* [47]) could also be used to improve the efficiency of path logging. The Symbiosis prototype is publicly available at `https://github.com/nunomachado/symbiosis`.

4.2 Symbolic Execution and Constraint Generation

Symbiosis' guided symbolic execution for C/C++ programs has been implemented on top of KLEE [7]. Since KLEE does not support multithreaded executions, similarly to CLAP, we fork a new instance of KLEE's execution to handle each new thread created. We also disabled the part of KLEE that solves path conditions to produce test inputs because Symbiosis does not use them. For Java programs, we have used Java PathFinder (JPF) [48]. In this case, we have disabled the handlers for *join* and *wait* operations to allow threads to proceed their symbolic execution independently, regardless of the interleaving. Otherwise, we would have to explore different possible thread interleavings when accessing these operations, in order to find one conforming with the original execution.

Additionally, we made the following changes to both symbolic execution engines. First, we ignore states that do not conform with the threads' path profiles traced at runtime, guiding the symbolic execution along the original paths only. Second, we generate and output a per-thread symbolic trace containing read/write accesses to shared variables, synchronization operations, and path conditions observed across each execution path.

Consistent Thread Identification. Symbiosis must ensure that threads are consistently named between the original failing execution and the symbolic executions. We use a technique previously used in jRapture [44] that relies on the observation that each thread spawns its children threads in the same order, regardless of the global order among all threads. Symbiosis instruments thread creation points, replacing the original PThreads/Java thread identifiers with new identifiers based on the parent-children order relationship. For instance, if a thread t_i forks its jth child thread, the child thread's identifier will be $t_{i:j}$.

Marking Shared Variables As Symbolic. Precisely identifying accesses to shared data, in order to mark shared variables as symbolic, is a difficult program analysis problem. Although it is possible to conservatively mark all variables as symbolic, varying the number of symbolic variables varies the size and complexity of the constraint system. For C/C++ programs we manually marked shared variables as symbolic, like prior work [21]. We also marked variables symbolic if their values were the result of calls to external libraries not supported by KLEE. For Java programs, we use Soot's *thread-local objects* (TLO) static escape analysis strategy [19], which soundly over-approximates the set of shared variables in a program (*i.e.*, some non-shared variables might be marked shared). At instrumentation time, Symbiosis logs the code point of each shared variable access. During the symbolic execution, whenever JPF attempts to read or write a variable, it consults

Table 1: **Benchmarks and performance.** Column 2 shows lines of code; Column 3, the number of threads; Column 4, the number of shared program variables; Column 5, the number of accesses to shared variables; Column 6, the overhead of path profiling; Column 7, the size of the profile in bytes; Column 8, the symbolic execution time; Column 9, the number of SMT constraints; Column 10, the number of unknown SMT variables; Column 11, the time in seconds to solve the SMT system.

	App.	LOC	# Thd.	#Shrd. Var.	#Shrd. Acc.	Prof. Ovhd.	Log Size	Symb. Time	#SMT Const.	#SMT Var.	SMT Time
C/C++	crasher	70	6	4	266	25.4%	458B	0.02s	22295	400	1m2s
	sbuff	151	2	5	69	16.7%	632B	0.05s	423	102	1s
	pfscan	830	5	9	74	6.6%	3.8K	1.87s	678	131	1s
	pbz (S)	1942	9	14	176	2.5%	1.7K	11.16s	1361	289	1s
	pbz (M)				367	1.3%	2.6K	36.17s	6771	564	26s
	pbz (L)				1156	2.5%	9.4K	7m11s	514548	2866	15h15m
Java	airline	108	8	2	36	22%	262B	1.30s	2670	84	1s
	bank	125	3	3	115	12.4%	788B	1.56s	8 250	197	2s
	2stage	123	4	4	49	14.8%	196B	2.53s	264	88	1s
	c4j (S)	2344	4	7	28	7.3%	366B	1.64s	122	51	1s
	c4j (M)				1247	8.6%	17K	4.56s	303626	1810	51s
	c4j (L)				1411	9.3%	24K	4.76s	1142120	2051	1h25m

the log to check whether that variable is shared or not. If so, JPF treats the variable as symbolic.

Locks Held at Failure Points. If a thread holds a lock when it fails, a reordering of operations in the critical region protected by the lock may lead to a deadlocking schedule. Other threads will wait indefinitely attempting to acquire the failing thread's held lock because the failing thread's execution trace includes no release. We skirt this problem by adding a *synthetic* lock release for each lock held by the failing thread at the failure point. The synthetic releases allow the failing thread's code to be reordered without deadlocks.

4.3 Schedule Generation and DSPs

We implemented failing and non-failing, alternate schedule generation, as well as differential schedule projections from scratch in around 4000 lines of C++ code. After building its SMT formula, Symbiosis solves it using Z3 [9]. Symbiosis then parses Z3's output to obtain the solution of the model, or the UNSAT core, when generating the root cause sub-schedule. Finally, to pretty-print its output, Symbiosis generates graphical output showing the differences between the failing and the alternate schedules.

5. Evaluation

We evaluated Symbiosis with three main goals. First, we show that Symbiosis efficiently collects path profiles and symbolic path traces. Second, we show that Symbiosis formulates and solves its SMT formulae in a practical, useful amount of time. Third, we show how differential schedule projections help to understand the failure behavior and points to the code requiring a bug fix. We substantiate our results with characterization data and several case studies, using buggy, multithreaded C/C++ and Java applications, including both real-world and benchmark programs. We used four C/C++ test cases: *crasher*, a toy program with an atomicity violation; *stringbuffer*, a C++ implementation of a bug in the Java JDK1.4 `StringBuffer` library, developed in prior work [17]; *pfscan*, a real-world parallel file scanner adapted for research by [11]; and *pbzip2*, a real-world, parallel bzip2 compressor. We used four Java programs: *cache4j*, a real-world Java object cache, driven externally by concurrent update requests; and three tests from the IBM ConTest benchmarks [14]: *airline*, *bank*, and *2stage*. Columns 1-4 of Table 1 describe the test cases.

We evaluated the scalability of Symbiosis for *pbzip2* and *cache4j* by varying the size of their workload. For *pbzip2*, we compressed input files of different sizes: 80KB (small), 2.6MB (medium), and 16MB (large). For *cache4j*, we re-ran its test driver, for update loop iteration counts of 1 (small), 5 (medium), and 10 (large). In some cases, we inserted calls to the `sleep` function, changing event timing and increasing the failure rate. Our work is not targeting the orthogonal failure reproduction problem [21], so this change does not taint our results. We ran our C/C++ experiments on an 8-core, 3.5Ghz machine with 32GB of memory, running Ubuntu 10.04.4. For Java we used a dual-core i5, 2.8Ghz CPU with 8GB of memory, running OS X.

Table 2: **Differential schedule projections.** Column 2 is the number of event pairs reordered to find a satisfiable alternate schedule (*#Alt Pairs*). Column 3 shows the number of events in the failing schedule (*#Evts FS*) and Column 4 shows the number of events in the corresponding differential schedule projection (*#Evts DSP*). Column 5 shows the number of data-flow edges in the failing schedule (*#D-F in FS*) and Column 6 shows the number of data-flow variations in the differential schedule projection (*#D-F in DSP*). Columns 4 and 6 show the percent change compared to the full schedule. Column 7 shows the number of operations involved in the data-flow variations (*#Ops to Grok*). Columns 8-9 show whether the differential schedule projection explains the failure, and whether it directly points to a fix of the underlying bug in the code.

App.	#Alt. Pairs	#Evts FS	#Evts DSP (Δ%)	# D-F in FS	# D-F in DSP (Δ%)	Ops to Grok	Explanatory?	Finds Fix?
crasher	27	287	8 (↓**97**)	107	1 (↓**99**)	3	Y	Y
sbuff	9	73	16 (↓**78**)	28	1 (↓**96**)	3	Y	Y
pfscan	5	93	21 (↓**77**)	32	1 (↓**96**)	3	Y	Y
pbz (S)	1	206	20 (↓**90**)	29	1 (↓**96**)	3	Y	N
pbz (M)	1	397	36 (↓**91**)	82	1 (↓**98**)	3		
pbz (L)	2	1223	168 (↓**86**)	264	2 (↓>**99**)	6		
airline	1	58	7 (↓**88**)	25	1 (↓**92**)	3	Y	Y
bank	181	124	56 (↓**55**)	72	2 (↓**97**)	6	Y	Y
2stage	14	60	12 (↓**80**)	27	1 (↓**96**)	3	Y	Y
c4j (S)	1	39	28 (↓**28**)	11	2 (↓**82**)	6	Y	N
c4j (M)	1	1257	11 (↓>**99**)	552	1 (↓>**99**)	3		
c4j (L)	1	1422	5 (↓>**99**)	628	1 (↓>**99**)	3		

5.1 Trace Collection Efficiency

We measured the time and storage overhead of path profiling relative to native execution and the time cost of symbolic trace collection. Columns 6-10 of Table 1 report the results, averaged over five trials. Symbiosis imposes a tolerable path profiling overhead, ranging from 1.3% in *pbzip2 (medium)* to 25.4% in *crasher*. Curiously, the runtime slowdown is smaller for real-world applications (*pfscan, pbzip2, and cache4j*) than for benchmarks. The reason is that the latter programs have more basic blocks with very few operations, making block instrumentation frequent. The space overhead of path profiling is also low, ranging from 196B (*2stage*) to 24K (*cache4j*). CLAP [21] showed that recording threads' path profiles only reduces storage overheads considerably (up to 97%!) compared to R&R (*e.g.* LEAP [20]). Symbiosis enjoys this reduction as well. Symbiosis collects symbolic traces in just seconds for most test cases. The only exception is *pbzip2 (large)*, which took KLEE around seven minutes. JPF quickly produced the symbolic traces for all programs.

5.2 Constraint System Efficiency

The last three columns of Table 1 describe the SMT formulae Symbiosis built for each test case. The Table also reports the amount of time Symbiosis takes to solve its SMT constraints with Z3, yielding a failing schedule. The data show that solver time is very low (*i.e.*, seconds) in most cases. Solver time often grows with constraint count, but not always. *cache4j (large)* has more than double the constraints of *pbzip2 (large)*, but was around 11 times faster. Figure 4 helps explain the discrepancy by showing the composition of the SMT formulations by constraint type. *pbzip2* has many *locking* and *read-write* constraints, while *cache4j* has no *locking*, although many *read-write* constraints. The solution to locking constraints determines the execution's lock order, constraining the solution to read-write constraints. The formulation's

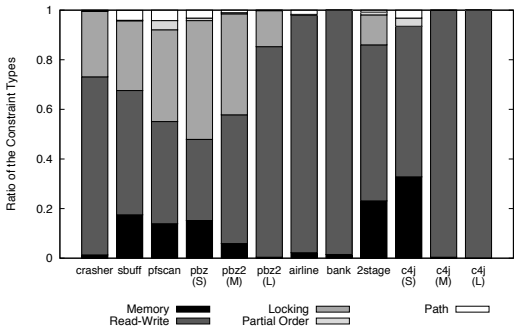

Figure 4: **Breakdown of the SMT constraint types.**

complexity grows not only with the count, but mainly with the interaction of these constraint types.

Symbiosis's SMT solving times are practical for debugging use. To produce a DSP, Symbiosis requires only a trace from a single, failing execution and does not require any changes to the code or input. Our experiments are realistic because a programmer, when debugging, often has a bug report with a small test case that yields a short, simple execution. The data suggest that Symbiosis handles such executions very quickly (*e.g.*, *pbzip2 (small)*, *cache4j (medium)*). Debugging is a relatively rare development task, unlike compilation, which happens frequently. Giving Symbiosis minutes or hours to help solve hard bugs (like *pbzip2 (large)*) is reasonable. Additionally, Symbiosis could use parallel SMT solving, like CLAP or incorporate lock ordering information, like [5], to decrease solver time.

5.3 Differential Schedule Projection Efficacy

Symbiosis produces a Graphviz visualization of its differential schedule projections (DSPs) as a graph with specific identifying information on nodes and edges that reflects source code lines and variables. This information includes both schedule variations and data-flow variations as well.

To assess the efficacy of DSPs, we first measured the number of program events and data-flow edges in the full, failing execution that Symbiosis computes. We then compared that measurement to the number of events and data-flow edges in the differential schedule projection.

Table 2 summarizes our results. The most important result is that Symbiosis's differential schedule projections are simpler and clearer than looking at full, failing schedules. Symbiosis reports a small fraction of the full schedule's data-flows and program events in its output – on average, 80.8% fewer events and 96.2% fewer data-flows. By highlighting only the operations involved in the data-flow variations, Symbiosis focuses the programmer on just a few events (3 to 6 in our tests). Furthermore, all events Symbiosis reports are part of data-flow or event orders that dictate the presence or absence of the failure. DSPs depict those events only, simplifying debugging.

Symbiosis finds an alternate, non-failing schedule after reordering few event pairs – just 1 in many cases (*e.g.*, *cache4j*, *pbzip2*). Symbiosis reorders one pair at a time, starting from those closer in the schedule to failure, and the data show that this usually works well. *bank* is an outlier – Symbiosis reordered 181 different pairs before finding an alternate, non-failing schedule. The bug in this case is an atomicity violation that breaks a program invariant that is not checked until later in the execution. As a result, Symbiosis must search many pairs, starting from the failure point, to eventually reorder the operations that cause the atomicity violation.

Note that, even if a failure occurs only in the presence of a particular chain of event orderings, it suffices to reorder any pair

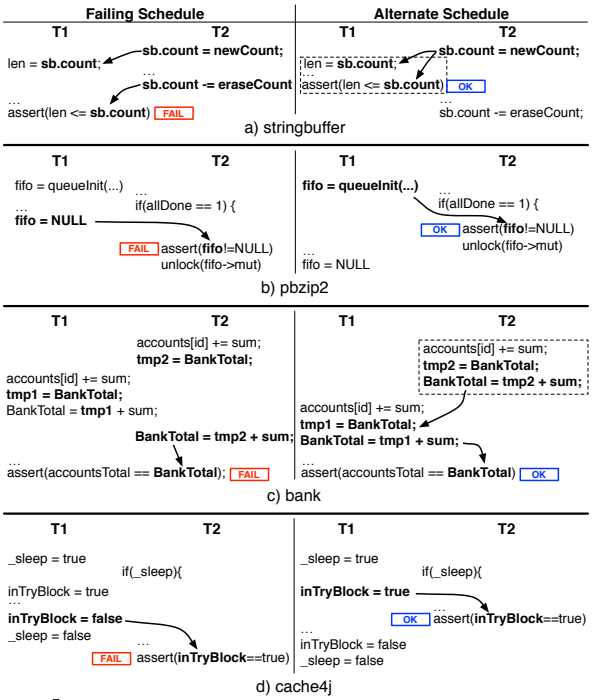

Figure 5: **Summary of Symbiosis' output for some of the test cases.** Arrows depict data-flows and dashed boxes depict regions that Symbiosis suggests to be executed atomically.

in the chain to prevent that failure. This phenomenon is called the *Avoidance-Testing Duality*, and is detailed in previous work [31].

We now use some case studies to illustrate how differential schedule projections focus on relevant operations and help understand each failure.

stringbuffer is an atomicity violation first studied in [17] and its DSP is depicted in Figure 5. *T1* reads the length of the string buffer, *sb*, while *T2* modifies it. When *T2* erases characters, the value *T1* read becomes stale and *T1*'s assertion fails . The DSP shows that the cause of the failure is *T2*'s second write, interleaving *T1*'s accesses to *sb.count*. Moreover, Symbiosis' alternate schedule suggests that, for *T1*, the write on value *len* and the verification of the assertion should execute atomically in order to avoid the failure. For this case, this is actually a valid bug fix.

pbzip2 is an order violation studied in [22]. Figure 5b shows Symbiosis's DSP that illustrates the failure's cause. *T1*, the producer thread, communicates with *T2* the consumer thread via the shared queue, *fifo*. If *T1* sets the *fifo* pointer to null while the consumer thread is still using it, *T2*'s assertion fails. The alternate schedule in Figure 5b, explains the failure because reordering the assignment of *null* to *fifo* after the assertion prevents the failure. The DSP is, thus, useful for understanding the failure. However, to fix the code, the programmer must order the assertion with the null assignment using a `join` statement. The DSP does not provide this suggestion, so, despite helping explain the *failure*, it does not completely reveal how to fix the bug.

bank is a benchmark in which multiple threads update a shared bank balance. It has an atomicity violation that leads to a lost update. Figure 5c shows the DSP for the failure: *T1* and *T2* read the same initial value of *BankTotal* and subsequently write the same updated value, rather than either seeing the result of the other's update. The final assertion fails, because *accountsTotal*, the sum of per-account balances, is not equal to *BankTotal*. The Figure shows

that Symbiosis's DSP correctly explains the failure and shows that eliminating the interleaving of updates to *BankTotal* prevents the failure. It is noteworthy that in this example the atomicity violation is not *fail-stop* and happens in the middle of the trace. Scanning the trace to uncover the root cause would be difficult, but the DSP pinpoints the failure's cause precisely.

cache4j has a data-race that leads to an uncaught exception when one thread is in a `try` block and another interrupts it with the library `interrupt()` function [42]. JPF doesn't support exception replay, so we slightly modified its code, preserving the original behavior, by replacing the exception with an assertion about a shared variable. Figure 5d shows that in our version of the code, *inTryBlock* indicates whether a thread is inside a `try-catch` block or not, the assertion `inTryBlock == true` replaces the `interrupt()` call. The program fails when *T1* is interrupted outside a `try` block as in the original code. The schedule variations reported in the DSP explain the cause of failure – if the entry to the `try` block (*i.e.*, `inTryBlock = true`) precedes the assertion, execution succeeds; if not, the assertion fails. The involvement of exceptions makes the fix for this bug somewhat more complicated than simply adding atomicity, but the understanding that the DSP provides points to the right part of the code and illustrates the correct behavior.

6. Related Work

The work that is most closely related to ours is CLAP [21], a technique for reproducing concurrency failures, via symbolic constraints. Symbiosis builds on the following CLAP features: independent track of thread control flow, use of guided symbolic execution, and resorting to SMT constraints. Like CLAP, Symbiosis can *also* reproduce bugs (§ 3.4). Finally, Symbiosis's C/C++ implementation uses KLEE, as well. Despite these similarities, Symbiosis differs fundamentally from CLAP in its purpose and mechanisms. Unlike CLAP, Symbiosis produces precise root cause sub-schedules and alternate, non-failing sub-schedules. Symbiosis's sub-schedule reports include many fewer operations than full schedules and do not require examining the whole schedule to find the few operations involved in the failure, as is the case with CLAP. Symbiosis's sub-schedule reports are the foundation of differential schedule projections, a new debugging technique not explored by CLAP. Symbiosis's mechanism for computing sub-schedules based on the SMT solver's UNSAT core is novel, as is Symbiosis's technique for reordering event pairs to compute alternate, non-failing sub-schedules. Note that CLAP does not compute alternate, non-failing schedules. Unlike CLAP, we do not limit the context switch count, because Symbiosis precisely isolates the root cause, regardless of the number of context switches in the entire execution. Finally, Symbiosis is more broadly applicable than CLAP and we have demonstrated prototypes for Java *and* C/C++.

Record and Replay (R&R) techniques are also relevant to our work. These techniques fall into three categories: *i) Order-based* techniques record the order of certain events during an execution and then replay them in the same order [20, 23, 49]. *ii) Search-based* techniques only trace partial information at runtime (*e.g.* record solely the order of write operations [53]) and, then, search the space of executions for one that conforms with the observed events [1, 39, 50]; *iii) Execution-based* techniques restrict all executions of a program so that, for a given input, the program's behavior is constrained to be deterministic from one run to the next [3, 4, 10, 37]. Symbiosis is mostly orthogonal to the techniques above, but shares some important characteristics. Like R&R techniques, given a concrete trace, Symbiosis can produce a failing schedule that conforms to those events, reproducing the failure. Symbiosis's precise differential schedule projections and broader applicability to debugging and failure avoidance make it novel in contrast to R&R techniques. Unlike deterministic execution systems, Symbiosis does not aim to perturb production runs, obviating the risk in doing so.

Techniques such as *interleaving pattern-matching* or *sub-schedule search* also bear similarity to Symbiosis. In particular, they aim to identify the root cause of a concurrency bug to show the programmer how to fix it, or to avoid it in future executions. *Interleaving pattern-matching* [28, 33, 38] techniques search an execution, dynamically or by reviewing a log, for problematic patterns of memory accesses. Although often effective, these solutions have the drawback of missing bugs that not fit the known patterns. Unlike these techniques, Symbiosis is not limited to searching for known patterns. Sub-schedule search, in turn, is general and not limited to specific patterns [31, 43, 52]. Unfortunately, the space of all of an execution's possible sub-schedules can be large. For some prior techniques, considering different sub-schedules requires multiple additional program executions, combined with statistical analysis to make search feasible. Symbiosis requires only a single, failing execution, does not rely on statistical reasoning, and produces precise results. Mechanically, these techniques differ in that none uses SMT to search and none produces a differential view of its result, like DSPs.

Finally, other techniques systematically explore the space of possible program executions to generate test cases. Java Path Finder [48], KLEE [7], Pex [45], and Mimic [54] use symbolic program execution to search for an *input* that induces a failing path constraint. Chess [36], PCT [6], and Concurrit [11] run a program for a particular input and rely on an augmented scheduler to push the execution to a potential failure. On the other hand, con2colic testing [15] employs *concolic execution* and uses heuristics to explore the space of possible thread interleavings and generate tests for multiple execution paths.

These techniques reveal only full, failing executions or buggy inputs, and do not provide root cause information, nor differential schedule projections. In turn, [8] also shares our goal of narrowing down the difference between successful and failing schedules to pinpoint a bug. This technique relies on random jitter and requires re-executing the program, whereas Symbiosis only operates on SMT formulations, which are sound and complete and, thus, provide a formal guarantee that alternate schedules are non-failing.

7. Conclusions & Future Work

This paper described Symbiosis, a new technique that gets to the bottom of concurrency bugs. Symbiosis reports focused *sub-schedules*, eliminating the need for a programmer or automated debugging tool to search through an entire execution for the bug's root cause. Symbiosis also reports novel *alternate, non-failing schedules*, which help illustrate *why* the root cause is the root cause and how to avoid failures. Our novel *differential schedule projection* approach links the root cause and alternate sub-schedules to data-flow information, giving the programmer deeper insight into the bug's cause than path information alone. An essential part of Symbiosis's mechanism is the use of an SMT solver and, in particular, its ability to report the part of a formula that makes it unsatisfiable. Symbiosis carefully constructs a deliberately unsatisfiable formula so that the conflicting part of that formula is the bug's root cause. We built two Symbiosis prototypes, one for C/C++ and one for Java. We used them to show that for a variety of real-world and benchmark programs from the debugging literature that Symbiosis isolates bugs' root causes and providing differential schedule projections that show how to fix those root causes.

Acknowledgements

We would like to thank the anonymous reviewers for their invaluable feedback. This work was partially supported by Fundação para a Ciência e a Tecnologia (FCT), under project UID/CEC/50021/2013.

References

[1] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multi-core debugging. In *SOSP '09*, 2009.

[2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4), July 1994. ISSN 0164-0925.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS XV*, 2010.

[4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for c/c++. In *OOPSLA '09*, 2009.

[5] M. Bravo, N. Machado, P. Romano, and L. Rodrigues. Towards effective and efficient search-based deterministic replay. In *HotDep '13*, 2013.

[6] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS XV*, 2010.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, 2008.

[8] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *ISSTA '02*, 2002.

[9] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08/ETAPS'08*, 2008.

[10] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: Deterministic shared memory multiprocessing. In *ASPLOS XIV*, 2009.

[11] T. Elmas, J. Burnim, G. Necula, and K. Sen. Concurrit: A domain specific language for reproducing concurrency bugs. In *PLDI '13*, 2013.

[12] M. Emmer, Z. Khasidashvili, K. Korovin, and A. Voronkov. Encoding industrial hardware verification problems into effectively propositional logic. In *FMCAD '10*, 2010.

[13] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP '03*, 2003.

[14] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS'03*, 2003.

[15] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *ESEC/FSE 2013*, 2013.

[16] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *PLDI '09*, 2009.

[17] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '03*, 2003.

[18] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI '08*, 2008.

[19] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT'07*, 2007.

[20] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent java programs. In *FSE '10*, 2010.

[21] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *PLDI '13*, 2013.

[22] N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *FSE '10*, 2010.

[23] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu. CARE: Cache guided deterministic replay for concurrent java programs. In *ICSE 2014*, 2014.

[24] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *PLDI '11*, 2011.

[25] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), July 1976. ISSN 0001-0782.

[26] S. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *POPL '08*, 2008.

[27] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), Sept. 1979. ISSN 0018-9340.

[28] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS XII*, 2006.

[29] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII*, 2008.

[30] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO 42*, 2009.

[31] B. Lucia and L. Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *ASPLOS '13*, 2013.

[32] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *ISCA '08*, 2008.

[33] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ISCA '10*, 2010.

[34] B. Lucia, B. P. Wood, and L. Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *PLDI '11*, 2011.

[35] N. Machado, P. Romano, and L. Rodrigues. Lightweight cooperative logging for fault replication in concurrent programs. In *DSN'12*, 2012.

[36] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI '08*, 2008.

[37] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS XIV*, 2009.

[38] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE'08*, 2008.

[39] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *SOSP '09*, 2009.

[40] S. Qadeer. Algorithmic verification of systems software using SMT solvers. In *SAS '09*, 2009.

[41] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4), Nov. 1997. ISSN 0734-2071.

[42] K. Sen. Race directed random testing of concurrent programs. In *PLDI '08*, 2008.

[43] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition?: Defuse: Definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA '10*, 2010.

[44] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *ISSTA '00*, 2000.

[45] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *TAP'08*, 2008.

[46] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99*, 1999.

[47] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A programmable hardware path profiler. In *CGO '05*, 2005.

[48] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA '04*, 2004.

[49] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. ORDER: Object centric deterministic replay for java. In *USENIX ATC '11*, 2011.

[50] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys '10*, 2010.

[51] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS XV*, 2010.

[52] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. Conseq: Detecting concurrency bugs through sequential errors. In *ASPLOS XVI*, 2011.

[53] J. Zhou, X. Xiao, and C. Zhang. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *ICSE '12*, 2012.

[54] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso. Mimic: Locating and understanding bugs by analyzing mimicked executions. In *ASE '14*, 2014.