Con2colic Testing

Azadeh Farzan
University of Toronto, Canada

Andreas Holzer
TU Wien, Austria

Niloofar Razavi
University of Toronto, Canada

Helmut Veith
TU Wien, Austria

ABSTRACT

In this paper, we describe con2colic testing — a systematic testing approach for concurrent software. Based on *conc*rete and symb*olic* executions of a *con*current program, con2colic testing derives inputs and schedules such that the execution space of the program under investigation is systematically explored. We introduce *interference scenarios* as key concept in con2colic testing. Interference scenarios capture the flow of data among different threads and enable a unified representation of path and interference constraints. We have implemented a con2colic testing engine and demonstrate the effectiveness of our approach by experiments.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Verification, Security

Keywords

Testing, Concurrency, Concolic, Interference

1. INTRODUCTION

Testing and bug finding for concurrent programs has been a very active area of research in recent years. There is an array of tools that successfully discover errors in concurrent programs by using a wide range of *heuristic* techniques that have been developed to alleviate the interleaving explosion problem that is inherent in analysis of concurrent programs. From a classical point of view, testing (sequential program testing, to be more exact) techniques are often coupled with a notion of *coverage* that the technique guarantees. Various *coverage* criteria have been introduced for sequential program testing over the years. Existing concurrency testing techniques can be divided into three categories of *coverage* guarantees (on program runs, inputs, code, assertions, etc.) that they provide:

(i) No specific coverage guarantees: heuristic-based techniques [25, 24, 22, 12, 4, 2, 17] are based on the philosophy of using heuristics to target interleavings that are more likely to contain bugs, and testing as many of those as possible (under the given time and space limitations). These techniques have been extremely successful in finding bugs, but cannot provide any useful information to the tester about what they have or have not missed during testing, that is, they cannot provide any coverage guarantees.

(ii) Coverage guarantees over the space of interleavings: search prioritization techniques [11, 5, 10, 14, 3] take a more coverage-oriented approach than the techniques in category (i) by using ideas like preemption-bounding [10], context-bounding [14, 11], depth-bounding [5], and delay-bounding [3] to prioritize the search in the space of all program interleavings. These prioritizations enable us to quantify (in a meaningful way) how much of the interleaving space is tested, a property that the heuristic-based search techniques of category (i) do not provide. Search prioritization techniques have been very successful in discovering bugs. For example, CHESS [11] is a successful testing tool based on the context-bounding search prioritization. Nevertheless, these techniques all work based on a fixed (pre-determined) set of inputs and, therefore, if a bug cannot be discovered on the given set of inputs, it will be missed (see Section 3 for an example). Moreover, coverage guarantees are only over the space of program interleavings.

(iii) Coverage guarantees over the space of program inputs and interleavings: There are also *sequentialization* techniques [8, 23, 15, 13, 16], that, based on context-bounding (or more recently other types of search prioritization), translate a concurrent program to a sequential program that has the same behaviour (up to a certain context bound), and then analyze the sequential program (statically) for the property of interest. To check if (for example) an assertion fails, the space of both inputs and interleavings (but up to the bound) are searched. These techniques are excellent at discovering bugs (they are incomplete, due to the bound, for proving properties). However, the sequential programs generated are not appropriate models to be used for testing (cf. Section 9). More recently, there has been some focus on exploring input and interleaving spaces of the program [19, 21] in a more systematic way. However, these techniques only explore a subset of the program behaviours, and therefore, cannot aim for maximum coverage.

We propose a concurrency testing technique that belongs in category (iii). Our goal is to systematically test concurrent code such that meaningful coverage guarantees can be provided. Concolic (*conc*rete and symb*olic*) testing [6, 1] is the gold standard of sequential testing with exactly the same point of view for sequential programs. There, the set of possible inputs (the only parameter that can change for sequential programs) is systematically explored to provide standard code coverage guarantees, such as branch coverage, for the program and existing program errors are discovered meanwhile. Our proposed approach can be viewed as the general-

ESEC/FSE '13, August 18Ű26, 2013, Saint Petersburg, Russia

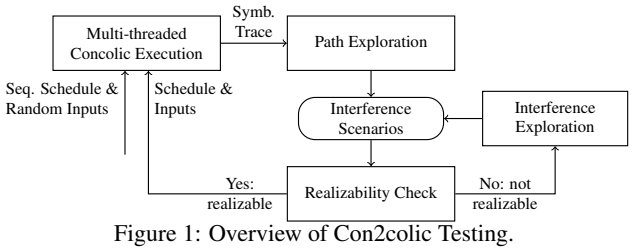


Figure 1: Overview of Con2colic Testing.

ization of concolic testing to concurrent programs, and hence we call it *concurrent concolic testing*, or con2colic testing in short. In Section 3, we introduce the running example of this paper. Targeting code coverage is a reasonable goal for a testing routine, since it has been shown that most dynamic program errors can be encoded as program assertions [7] which can easily be tested/verified through branch coverage. In con2colic testing, we aim to achieve maximal code coverage for the program (time and space allowing).

It is important to note that almost all existing concurrency testing techniques today use an *under-approximation* of the set of concurrent executions as a model for testing. Most techniques in category (i), mentioned before, use one or several program runs as a basis for predicting buggy runs. The techniques in category (ii), use the set of program runs prioritized by one of the bounding techniques discussed above, as the under-approximation of their choice. A common limitation among most of these techniques is that their under-approximate model is fixed a priori. Once they start testing, they are limited to performing the tests within their bounded model[1]. Specifically, all techniques that we list in category (iii) (which is our focus) also have this limitation. Con2colic testing does not have this limitation. Similar to concolic testing, we have the program at our disposal and we can keep expanding the set of behaviours that we consider for testing. Naturally, our approach is also limited by the same constraints that hold concolic testing back, namely the unavailability of external library functions or undecidable logics. This aspect of con2colic testing has a very important consequence. We can theoretically guarantee *completeness* in the limit; by this, we mean that if the testing algorithm runs for long enough, then we can cover every program branch or declare it unreachable (again within the limitations of concolic testing).

We implemented the con2colic testing approach as a tool for testing multi-threaded C code. We use a set of benchmarks found in concurrency research literature to demonstrate the practical efficiency of our approach in providing code coverage and finding bugs in these benchmarks. In the next section, we provide an extended overview of con2colic testing.

2. CON2COLIC TESTING

We start by a short overview of concolic testing, in order to highlight what needs to be done to generalize it to concurrent programs. Concolic testing has three main components: concolic execution engine, path exploration, and realizability checker. The concolic execution engine executes the program on a given input vector concolically (i.e. using both concrete and symbolic values for inputs) and as a result, generates an execution trace that contains a sequence of path constraints on symbolic inputs (i.e. branch conditions encountered during execution). The goal is to try to diverge from the just observed execution by taking a different side of an encountered branch. The path exploration component, hence, selects some of these branch conditions and negates them in order to guide the execution along a different path. In the realizability checker, SMT solvers are used to generate an input vector (if possible) that would satisfy the new path constraints, with the understanding that such an input vector is likely to force the program to execute a different path. For a deterministic sequential program, each input vector specifies a single path in the program.

The behaviour of a concurrent program, however, is not only influenced by the input vector, but also by the interleaving of execution of threads. Each interleaving of execution of threads determines a pattern of interferences among the threads. An *interference* occurs when a thread reads a value that is generated by another thread. An interference could substantially change the course of the local computation of a thread, because the value generated by another thread may not be producible locally. Therefore, the local behaviour (e.g. an assertion violation) followed by this interference may never surface by pure sequential testing.

We propose con2colic testing (see Figure 1) as a concolic testing approach tailored for concurrent programs. We introduce the concept of *interference scenario* (formally defined in Section 5) as a representation of a set of interferences among the threads. Conceptually, interference scenarios are the essential information that define the important scheduling constraints for a concurrent program run; in other words, all runs with the same interference scenario are *behaviourally* equivalent under the same input values. The constraint system for sequential concolic execution is also modified to model interferences as well. The con2colic testing engine is then able to produce schedules, in addition to input vectors, for concurrent programs. To accommodate this change, con2colic testing has one more component (compared to sequential concolic testing) called *interference exploration*, that navigates the space of all interference scenarios in a systematic way.

In con2colic testing, the execution engine is leveraged to execute a concurrent program based on a given schedule. The important part of the observed execution is stored in a *forest* data structure (formally defined in Section 5) that keeps track of various interference scenarios that have been explored already. The path exploration component then decides what *new* scenario to try next, aiming at covering previously uncovered parts of the program, based on the set of interferences that have already been explored. For each interference scenario, the realizability checker investigates whether there exist a set of inputs and a *feasible* schedule such that a program execution based on these inputs and this schedule results in the same set of interferences. If the answer is yes, then the input and the schedule are used in the next round of concolic execution. Otherwise, the interference exploration component generates new candidates by introducing new interferences.

Next, we briefly explain con2colic testing components and then we present an exploration strategy that we implemented.

Concolic Execution. There are two input parameters for the concolic execution engine in con2colic testing: (1) an input vector and (2) a schedule. The engine executes the concurrent program with the given input vector and according to the given schedule. The program is instrumented such that, during the execution, all important events are recorded and a *symbolic trace* is produced as a result. These important events include synchronization operations, accesses to shared variables, and path constraints. The symbolic trace contains all the necessary information for the con2colic engine to make progress. However, it excludes extra information, such as details of local computations of threads, that can safely be ignored in our approach to gain scalability and efficiency.

Path Exploration. The role of the path exploration routine is to explore the input space for a new set of input values that may cover the yet uncovered parts of the program. This module does not modify the set of interferences observed in the previous execution, and

[1]Category (ii) techniques do not have this limitation theoretically, but in practice, only very small values for bounds are feasible.

only guides the execution down a different program path by negating the conditions of branches (observed along the last execution). Therefore, the changes made in this phase are only changes of the input values, and the essential structure of the schedule (i.e. interference scenario) remains the same.

Realizability Checker. The role of the realizability checker is to determine if there is a set of inputs and a feasible schedule that realize the given interference scenario. Each interference scenario imposes two constraint systems: (1) constraints on input values that are satisfied by all executions having the same interference scenario and (2) constraints that define the temporal order of the events of different threads (e.g., a read event of one thread has to occur after the corresponding write event of another thread). We formally define the resulting constraint systems in Section 6. If both constraint systems have a solution, then an input vector and a schedule can be inferred which give rise to a real program execution with exactly the same set of interferences as defined in the interference scenario. If at least one of the constraint systems does not have a solution then the interference scenario has to change. To that end the current interference scenario is passed to the interference exploration module (described below), such that a new one will be derived from it. In Section 7.5, we will discuss how to prune the exploration space (of interference scenarios) based on the reason for the unsatisfiability of the constraint system.

Interference Exploration. Interference exploration produces new interference scenarios from previously explored interference scenarios, essentially by introducing a new interference. This is done by picking a read from the given interference scenario that is not interfered by other threads, and an appropriate write from the forest, and adding an interference from the write to the read to generate a new interference scenario. Note that the occurrence of the write event itself may be conditional on the existence of other interferences. Therefore, to preserve soundness, all of those interferences should be included in the produced interference scenario as well. We discuss the function of this module in detail in Section 7.

Exploration Strategy and Completeness. With the above components, con2colic testing can exploit different search strategies and heuristics to explore the interference scenario space. We have implemented a search strategy that targets branch coverage (Section 7). The search strategy then explores interference scenarios with an increasing number of interferences. That is, all interference scenarios with one interference are explored first, and then interference scenarios with two interferences are explored, and so on. A nice feature of this exploration strategy is that it is complete (Theorem 7.3) modulo the exploration bound (and of course concolic testing limitations).

3. RUNNING EXAMPLE

We use the buggy implementation of function `addAll` in Fig. 2 as a running example in this paper. The example is a variation of the `addAll` method of the built-in class `Vector` in Java. The error in it is a real error. We have rewritten it in C (our tool's input language) and simplified it a bit to make it suitable as a running example. `addAll` has two input parameters which are pointers to vector structures. It appends all elements of the second vector to the end of the first vector. Each `vector` has three fields: `data` which is an array holding vector elements, `size` which represents the size of `data`, and `cnt` which keeps track of the number of elements in `data`. Function `addAll` uses a lock `lk` to synchronize the calls to this function. It first checks whether there is enough space to insert all elements of `u->data` into `v->data`, i.e. `v->cnt` + `u->cnt` $\leq$ `v->size` (cf. line 4). If not, it increases the size of `v->data`

```
typedef struct {int cnt, int size, int* data} vector;
pthread_mutex lk;

1  void addAll(vector* v, vector* u) {
2    int numElem = v->cnt + u->cnt;
3    pthread_mutex_lock(&lk);
4    if(numElem > v->size) {
5      v->data = realloc(numElem * 2);
6      v->size = numElem * 2;
7    }
8    assert(v->size ≥ u->cnt + v->cnt);
9    ...  //copy data from u to v
10   v->cnt = v->cnt + u->cnt;
11   pthread_mutex_unlock(&lk);
12 }
```

Figure 2: Function `addAll`: a buggy concurrent implementation of `vector` concatenation.

accordingly. The invariant `v->size` $\geq$ `u->cnt` + `v->cnt` is stated as an assertion in line 8. Finally, it copies the elements and updates `v->cnt`. The bug in `addAll` corresponds to the fact that the value of `v->cnt` is being read (at line 2) outside the lock block and hence `v->cnt` can be changed by other threads before the lock block is executed, leading to an inconsistent state.

For simplicity, we just refer to `v->cnt` and `u->cnt`, while in the real implementation, these accesses are protected by locks. Therefore, there is no data race on these accesses and the error that we discuss here is independent of that data race and exists in the original race-free code.

Imagine a concurrent program with two threads T and T' each of them calling `addAll` with `v` and `u` as arguments, where `v` is shared between the threads and `u` is an input of the program. Therefore, each individual field of `v` is treated as a shared variable and each individual field of `u` is treated as an input. Also, suppose that initially `v->cnt` is 10 and `v->size` is 20. Consider the situation where `u->cnt`=7 and the program is executed as follows:

(i) The first thread T executes line 2, reading 10 from `v->cnt`, 7 from `u->cnt` and storing value 17 in `numElem`.

(ii) The second thread T' is executed completely. It reads values 10 and 7 from `v->cnt` and `u->cnt`, respectively, at line 2 and assigns 17 to `numElem`. Then, it enters the lock block. Since `v->size` is greater than 17 it skips lines 5 and 6 and assigns 17 to the shared variable `v->cnt` before exiting the lock block.

(iii) Then, T continues execution: It skips lines 5 and 6 since (`numElem`=17) $<$ (`v->size`=20). However, when T gets to the assertion, `v->cnt` has value 17 written by T'. Therefore, we have (`v->size`=20) $<$ (17 + 7), and hence the assertion is violated.

This particular error occurs because the naive locking of individual `vector`s is not the correct way of copying from one vector (that can change meanwhile) to another. The error is interesting because it requires a combination of a particular concurrent schedule combined a with particular (relative) values for the input vectors to manifest. If the threads are executed sequentially back to back, nothing goes wrong. On the other hand, if we execute the same interleaving (as described above), but start with `u->cnt` having the value 3 (instead of 7), then nothing goes wrong; the first thread assigns 13 to `numElem`, the second thread skips lines 5 and 6 and assigns 13 to `u->cnt`. Then, the first thread skips lines 5 and 6 since (`numElem`=13) $<$ (`v->size`=20). This means that triggering this concurrency bug does not solely depend on the *concurrent schedule*, nor does it solely depend on the chosen *input* values; it depends on finding the right combination of input values and the choice of concurrent schedule. Any testing technique that does not explore the combination space systematically has the potential of missing on this bug.

4. BASIC DEFINITIONS AND NOTATIONS

We will now introduce some notions from concolic testing adjusted to our application. Classical sequential concolic testing [6, 1] logs a set of path constraints over *input variables* during concolic execution which describes the conditions on the values of the inputs that have to be true to drive the execution of the program along the same *path*. However, doing the same for concolic execution of multi-threaded programs would result in a set of constraints that are closely tied to the specific schedule performed during program execution. To solve this problem, we proceed as follows: Instead of explicitly tracking scheduling decisions, we introduce symbolic variables which enable us to track the information flow between threads. More precisely, we introduce an additional symbolic variable each time a shared variable is read and for each shared variable write, we store the symbolic value (based on symbolic inputs and symbolic read variables). This will enable us to build constraints which capture the essence of a concurrent execution path.

A concurrent program consists of a set $T = \{T_1, T_2, \ldots\}$ of threads T_i, a set of input variables *IN*, a set of shared variables *SV*, a set of local variables *LV*, and a set of locks L. Let *SymbIN* be a set of symbolic input variables $\{i_0, i_1, \ldots\}$ and *SymbRV* be a set of symbolic shared read variables $\{r_0, r_1, \ldots\}$. Let *Expr* represent the set of all expressions over *SymbIN* and *SymbRV*, and let *Pred(Expr)* represent the set of all predicates over *Expr*. Then, the set of actions Σ that a thread can perform on a set of shared variables *SV* and locks L is defined as:

$$\Sigma = \{rd(x,r) \mid x \in SV, r \in SymbRV\} \cup \{wt(x,val) \mid x \in SV, val \in Expr\} \cup \{tf(T_i) \mid T_i \in T\} \cup \{ac(l),\ rel(l) \mid l \in L\} \cup \{br(\psi) \mid \psi \in Pred(Expr)\}$$

Action $rd(x,r)$ corresponds to reading from a shared variable x and the symbolic value of the shared variable x becomes the symbolic variable r. Each time we observe a read from a shared variable during concolic execution, we introduce a new symbolic variable $r \in SymbRV$ that is uniquely associated with that specific read. Action $wt(x, val)$ corresponds to writing to a shared variable x a symbolic value which is represented as an expression val. To couple a read of x with a write to x, it is enough to connect the stored expression at the write to the symbolic value of the read, i.e., $r = val$. Action $tf(T_i)$ represents forking thread T_i. Actions $ac(l)$ and $rel(l)$ represent acquiring and releasing of lock l, respectively. Finally, action $br(\psi)$ denotes a branch condition which requires that predicate ψ is true. We model assertions in a program by two branches, i.e., one branch for passing the assertion and one branch for violating the assertion.

We denote the execution of an action by a thread as an *event*. Formally, an event is a tuple $(T_i, a) \in T \times \Sigma$. Let EV denote the set of all possible events. During concolic execution, we observe a sequence of events, a so-called *symbolic trace*:

DEFINITION 4.1 (SYMBOLIC TRACE). *A symbolic trace is a finite string $\pi \in EV^*$. By $\pi[n]$, we denote the n-th event of π. Given a symbolic trace π, $\pi|_{T_i}$ is the projection of π to events involving T_i. A symbolic trace π is thread-local, if $\pi = \pi|_{T_i}$ for some T_i.*

The inputs to our concolic execution engine are an input vector and a schedule which exactly specifies the resulting program run:

DEFINITION 4.2 (PROGRAM RUN). *Consider a* deterministic *concurrent program P. A (partial)* run *of P, represented by $R = P(\overline{in}, \sigma)$, is uniquely described by a valuation $\overline{in}$ of the input variables IN and a schedule σ. A* schedule *σ is defined by a sequence $(T_{i_1}, n_1)(T_{i_2}, n_2) \ldots (T_{i_{m-1}}, n_{m-1})(T_{i_m}, -)$ where $T_{i_j} \in T$, for $1 \leq j \leq m$, and $n_j > 0$, for $1 \leq j < m$, specifies the number of executed actions. A tuple $(T_{i_j}, -)$ represents the execution of thread T_{i_j} until T_{i_j} terminates. A run of program P is* feasible *if P can be executed with input vector $\overline{in}$ and according to schedule σ. Each feasible program run R yields a symbolic trace $\pi(R)$.*

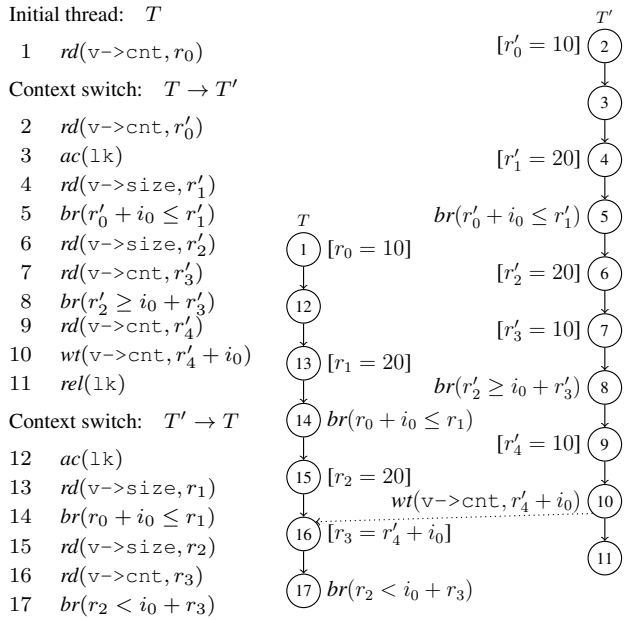


Figure 3: Symbolic trace π obtained from the assertion violating execution of the program in Fig. 2 and its corresponding interference scenario $IS(\pi)$. i_0 represents the symbolic value of input `v->cnt`. r_0, r'_0, r'_3, and r'_4 read initial value 10, and r_1, r_2, r'_1, and r'_2 read initial value 20, and r_3 reads $r'_4 + i_0$

We assume that the program is instrumented in such a way that all program actions covered in EV are actually observed by $\pi(R)$. In Fig. 3, on the left, we show a symbolic trace π obtained from the assertion violating execution of the program in Fig. 2, discussed in Section 3. Observe that the concolic execution engine only logs reads from and writes to shared variables, but no reads from or writes to local variables. Internally, the concolic execution engine keeps track of the symbolic values of local variables and is therefore able to correctly update the symbolic value of a shared variable when it gets written.

5. INTERFERENCE SCENARIOS

An interference occurs whenever a thread reads a value that is written by another thread. We introduce interference scenarios to describe a class of program executions under which certain interferences happen during concolic execution. Intuitively, an interference scenario is a set of thread-local traces extended with an interference relation between write and read events from different threads. We represent a set of interference scenarios in a data structure called interference forest. Formally, an interference forest is a finite labeled directed acyclic graph whose nodes represent events and whose edges express relations between events.

DEFINITION 5.1 (INTERFERENCE FOREST). *An* interference forest *is a tuple $I = (V, E, \ell)$ where V is a set of nodes, $\ell : V \to EV$ is a labeling function which assigns events to nodes. For $v \in V$ where $\ell(v) = (T_i, a)$, we also define $Th(v) = T_i$ and $Ac(v) = a$*

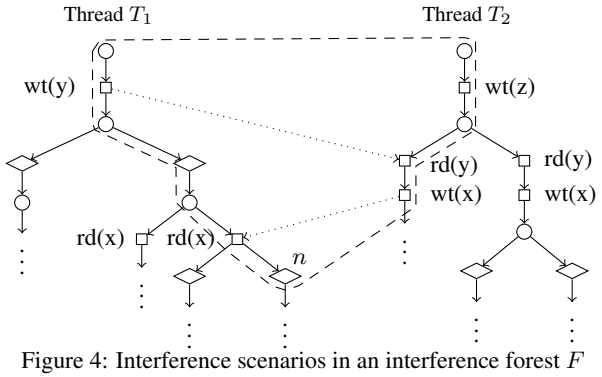


Figure 4: Interference scenarios in an interference forest F

to be the thread and the action of the corresponding event, respectively. The set of edges E is the disjoint union $E = E_L \dot\cup E_I$ of thread-local edges E_L and interference edges E_I. A thread-local edge (or, simply, a local edge) is an edge $(s,t) \in E_L$ where $Th(s) = Th(t)$. An interference edge $(s,t) \in E_I$ is an edge where $Th(s) \neq Th(t)$ and $Ac(s) = wt(x, val)$ and $Ac(t) = rd(x, r)$ for some x, val, and r. We require that E_I is an injective relation, i.e., each read is connected to at most one write by E_I. The thread-local edges can be naturally partitioned *according to their threads, i.e., $E_L = E_{T_1} \dot\cup E_{T_2} \ \ldots \ \dot\cup E_{T_n}$. Each E_{T_i} induces a subforest G_{T_i} which consists of all nodes with $Th(v) = T_i$ and edges in E_{T_i}. We require that each G_{T_i} is a rooted tree. The number of interference edges $|E_I|$ is called the* degree *of the interference forest.*

An isomorphism between two interference forests is a graph isomorphism which preserves the labeling function. Given an interference forest J, $R_I(J)$ denotes the read nodes involved in the interference edges of J, i.e., $R_I(J) = \{n_r \mid \exists n_w.(n_w, n_r) \in E_I\}$.

Figure 4 shows an interference forest. The nodes labeled with read/write and branch actions are represented by squares and diamonds, respectively. Local edges are presented by arrows and interference edges are presented by dotted arrows. The left tree represents G_{T_1} and the right tree represents G_{T_2}. The degree of the interference forest is 2.

DEFINITION 5.2 (INTERFERENCE SCENARIO). *We define an* interference scenario (IS) *as an interference forest where each G_{T_i} is a path.*

As mentioned at the beginning of this section, an interference forest is a compact representation for a set of interference scenarios.

DEFINITION 5.3 (CAUSAL INTERFERENCE SCENARIO). *Let $I = (V, E, \ell)$ be an interference forest. The transitive closure E^* of the edge relation E is called the causality relation of I. Given a node n, the* causal interference scenario (CIS) *of n is the subforest of I induced by the causal predecessors of n, i.e., by the node set $\{v \mid (v, n) \in E^*\}$. We denote it by $C = CIS(I, n)$ and call n the* sink *of C, i.e., sink(C) = n.*

Every causal interference scenario is an interference scenario. This is also the crucial property why interference forests serve as compact representations for sets of interference scenarios. In Fig. 4, the causal interference scenario of node n is the interference scenario enclosed by dashed lines.

Construction of new interference scenarios from existing ones and merging interference scenarios into an interference forest are two central operations in our testing approach. But we cannot combine arbitrary interference forests/scenarios; They have to be compatible with each other:

DEFINITION 5.4 (COMPATIBLE INTERFERENCE FORESTS). *Two interference forests I, J are* compatible *if there is an interference forest K and interference subforests I', J' of K such that I' and J' are interference forests themselves and I is isomorphic to I' and J is isomorphic to J'.*

Definition 5.4 applies to compatible interference scenarios as well since each interference scenario is an interference forest.

REMARK 5.5. *Compatible interference forests can be merged into an interference forest by naturally taking the minimal K, i.e., K only contains nodes and edges corresponding to I' and J'. If interference scenarios I and J are not compatible, then there is at least one thread for which I and J describe different computations.*

Each symbolic trace π defines an interference scenario, denoted by $IS(\pi)$. Intuitively, each event represents a unique node in $IS(\pi)$ which is labeled with that event. For each thread T_i, thread-local edges are added between the corresponding nodes according to the order in $\pi|_{T_i}$ (where $\pi|_{T_i} = \pi_{i,1}, \pi_{i,2}, .., \pi_{i,m}$ denotes the projection of events in π on thread T_i.) An interference edge is added for each node labeled with a read event if the last write event to the same shared variable before the read event in π is performed by another thread. More formally, $IS(\pi) = (V, E, \ell)$ is defined as:

- $V = \bigcup_{T_i} \{n_{i,j} \mid n_{i,j}$ is a unique node for event $\pi_{i,j}\}$,
- $\ell(n_{i,j}) = \pi_{i,j}$ for each node $n_{i,j}$,
- $E_{T_i} = \{(n_{i,k}, n_{i,k+1}) \mid 0 \leq k \leq m-1\}$, and
- $E_I = \{(n_{i,k}, n_{j,h}) \mid Th(n_{i,k}) \neq Th(n_{j,h}), Ac(n_{i,k}) = wt(x, val), Ac(n_{j,h}) = rd(x, r)$ for some x, val, r and $\pi_{i,k}$ is the last write to x in π before $\pi_{j,h}\}$.

Figure 3 shows the interference scenario for the symbolic trace obtained from the assertion violating execution of the program in Figure 2 which is discussed in Section 3.

DEFINITION 5.6 (REALIZABLE INTERFERENCE SCENARIO). *An interference scenario I is* realizable *iff there is a feasible partial program run R with $\pi = \pi(R)$ such that $I = IS(\pi)$. We say R realizes I for such a feasible program run R.*

Realizable interference scenarios define equivalence classes on the set of program runs which represent the same flow of data among the threads. Note that interference scenarios are not monotonic wrt. realizability. Let I and I' be two interference scenarios where I is a subgraph of I'. Then, the realizability of I does not imply the realizability of I' and vice versa. We will discuss the reasons for this behaviour in Section 6.

Interference scenarios specify partial program runs and therefore unanticipated behaviour can be observed:

DEFINITION 5.7 (UNFORESEEN INTERFERENCES). *Let I be a realizable interference scenario and R be a partial program run with $\pi = \pi(R)$ such that $I = IS(\pi)$. Let R' be a run that extends R, i.e., $\pi' = \pi(R')$ and π is a prefix of π'. Then, $IS(\pi')$ is a supergraph of $IS(\pi)$. More specifically, $IS(\pi')$ might contain some additional interferences. We refer to these interferences as interferences* unforeseen *by interference scenario I in run R'.*

6. CONSTRAINT SYSTEMS FOR INTERFERENCE SCENARIOS

Each interference scenario implies constraints on both data and temporal order of the events. In this section, we describe these constraints in detail. In Section 7, we present our soundness theorem

Data Constraints $DC(I)$

$DC(I)$:	$DC_{branch}(V) \wedge DC_{interfere}(I) \wedge DC_{local}(I)$
$DC_{branch}(V)$:	$\bigwedge_{\psi \in BR(V)} \psi$ where $BR(V) = \{\psi \mid v \in V, Ac(v) = br(\psi)\}$
$DC_{interfere}(I)$:	$\bigwedge_{(v_{wt}, v_{rd}) \in E_I} DC_{match}(v_{wt}, v_{rd})$
$DC_{local}(I)$:	$\bigwedge_{(v_{wt}, v_{rd}) \in E_L} DC_{match}(v_{wt}, v_{rd})$ where E_L is the set of local write-read matches.
$DC_{match}(v_{wt}, v_{rd})$:	$(val = r)$ for $Ac(v_{wt}) = wt(x, val)$, $Ac(v_{rd}) = rd(x, r)$

Temporal-Consistency Constraints $TC(I)$

$TC(I)$:	$\bigwedge_{T_i \in T} PO_{T_i} \wedge FC \wedge LC_1 \wedge LC_2 \wedge WRC_{interfere} \wedge WRC_{local}$
PO_{T_i}:	$\bigwedge_{n_{i,j} \in G_{T_i},\, n_{i,j} \text{ is not a leaf}} (t_{n_{i,j}} < t_{n_{i,j+1}})$
FC:	$\bigwedge_{T_i \in T} (t_{n_{tf(T_i)}} < t_{n_{i,1}})$
LC_1:	$\bigwedge_{T_i \neq T_j} \bigwedge_{l \in L} \bigwedge_{\substack{[aq,rl] \in L_{T_i,l} \\ [aq',rl'] \in L_{T_j,l}}} (t_{rl} < t_{aq'} \vee t_{rl} < t_{aq})$
LC_2:	$\bigwedge_{T_i \neq T_j} \bigwedge_{l \in L} \bigwedge_{\substack{aq \in NoRel_{T_i,l} \\ [aq',rl'] \in L_{T_j,l}}} (t_{rl'} < t_{aq})$
$WRC_{interfere}$:	$\bigwedge_{(u,v) \in E_I} Coupled(v, u)$
WRC_{local}:	$\bigwedge_{v \notin intReads} Coupled(v, LocW(v))$
$Coupled(v, u)$:	$(t_u < t_v) \wedge \bigwedge_{n \in W_x \setminus \{u\}} ((t_n < t_u) \vee (t_v < t_n))$

Figure 5: Constraint systems $DC(I)$ and $TC(I)$ for an interference scenario $I = (V, E, \ell)$

(Thm. 7.2) that shows how these constraints can be used to check for the realizability of an interference scenario.

Data Constraints. Each interference scenario $I = (V, E, \ell)$ defines a data constraint $DC(I)$ as shown in Fig. 5. Any solution to $DC(I)$ (if one exists), defines an input vector $\bar{i}$ for the concurrent program. The constraint $DC(I)$ consists of three parts: (i) DC_{branch}, (ii) $DC_{interfere}$, and (iii) DC_{local}. The constraint DC_{branch} encodes all branch conditions occurring in I. The intuition behind this constraint is that the program execution should follow the control path represented by the respective branching conditions. $DC_{interfere}$ relates reads from shared variables, which should be interfered by writes from other threads, to the symbolic values of the write the read should interfere with. Finally, DC_{local} relates each read from a shared variable, which should not be interfered by any writes from other threads, to the most recent write to the same shared variable performed by the same thread. If there is no such write before the read, we constrain the symbolic value of the shared variable to the initial value of the variable.

Temporal-Consistency Constraints. An interference scenario I also defines a temporal consistency constraint $TC(I)$. This constraint is over symbolic traces and any solution to it defines a schedule for the concurrent program. The constraints in $TC(I)$, as defined in Fig. 5, are divided into the following four categories: (i) thread-local program-order consistency (PO_{T_i}), (ii) thread-fork consistency (FC), (iii) lock consistency (LC_1&LC_2), and (iv) write-read consistency (WRC_1&WRC_2). For each node n in I, an integer variable t_n (timestamp) is considered to encode the *index* of the event of the node in a symbolic trace π. In the constraints, let $n_{i,j}$ represent the j^{th} node in G_{T_i}, and let $n_{tf(T_i)}$ represent the node n where $Ac(n) = tf(T_i)$. The constraints of $TC(I)$ are:

PO_{T_i}: Ensures that for thread T_i, the thread-local program order is respected in the schedule.

FC: Ensures that no thread can be scheduled before it is forked.

LC_1 & LC_2: Each lock acquire node aq with $Ac(aq) = ac(l)$ and $Th(aq) = T_i$ and its corresponding lock release node rl in T_i define a lock block, represented by $[aq, rl]$. Let $L_{T_i,l}$ be the set of lock blocks in thread T_i regarding lock l. LC_1 ensures that no two threads can be inside lock blocks of the same lock l, simultaneously. LC_2 ensures that the acquire of lock l by a thread that never releases it in I must occur after all releases of lock l in other threads. In this formula, $NoRel_{T_i,l}$ stands for lock acquire nodes in T_i with no corresponding lock release nodes.

$WRC_{interfere}$ & WRC_{local}: Let W_x represent the set of all nodes u with $Ac(u) = wt(x, val)$, *intReads* be all nodes v with $Ac(n) = rd(x, r)$ such that v is involved in an interference edge in E_I, and *LocW* be a function that for each node v with $Ac(v) = rd(x, r)$ and $Th(v) = T_i$ returns a node u with $Ac(u) = wt(x, val)$ and $Th(u) = T_i$ in I such that u is the closest such node to v before v in G_{T_i}. For each read node v and write node u, the formula $Coupled(v, u)$ ensures that the read event of v is coupled with the write event of u in π by forcing all events that write to the corresponding variable to happen either before the event of u or after the event of v in π.

Non-Monotonicity of Realizability. We can now explain the non-monotonic behaviour of interference scenarios wrt. realizability that was mentioned in the discussion following Def. 5.6. Let I and I' be two interference scenarios where I is a subgraph of I'. Then, according to the data constraints, all constraints in $DC_{branch}(I)$ and $DC_{interfere}(I)$ appear in $DC_{branch}(I')$ and $DC_{interfere}(I')$, respectively. However, the constraints in $DC_{local}(I)$ and $DC_{local}(I')$ are incomparable. The same phenomenon exists in the temporal-consistency constraints, i.e., WRC_{local} in I and I' are incomparable. This implies that, by extending an interference scenario, the resulting constraint systems do not change in a monotonic way.

7. TESTING ALGORITHM

We now present our con2colic testing algorithm. The algorithm tries to increase branch coverage in concurrent programs. Recall that we model each assertion in the program by two branches and, therefore, implicitly target at finding assertion violations. Since we aim for branch coverage, we are specifically interested in interference scenarios related to nodes labeled with branch actions:

DEFINITION 7.1 (INTERFERENCE SCENARIO CANDIDATE). *Let n be a branch node, i.e., $Ac(n) = br(\psi)$, for some ψ. A causal interference scenario C is an* interference scenario candidate (ISC) *for n if $sink(C) = n$.*

Note that each ISC C with $sink(C) = n$ (if realizable) defines a set of partial program runs where $Ac(n)$ is the last action in the run. Our algorithm enumerates all ISCs of degree k (starting at $k = 0$), checks their realizability and moves on to ISCs of degree $(k + 1)$.

To keep the exposition simple, we will make the following simplifying assumptions: (i) There are no unforeseen interferences for an ISC C, i.e., each program run R' extending a partial run R, with $C = IS(R)$, results in an interference scenario $IS(R')$ which has exactly the same interferences as C. (ii) There are no barriers in a program P. (iii) There are no locks in a program P. In Section 7.3 we will address assumptions (i) and (ii) and in Section 7.4 we will address assumption (iii). Note that we state all these assumptions for ease of presentation and that our approach is not limited to settings where these assumptions are true, especially, all benchmark programs in Section 8 contain locks.

7.1 Testing Algorithm

Alg. 1 shows our con2colic testing algorithm. Given a concurrent program P and a threshold k_{max}, the algorithm investigates all ISCs whose degree is $\leq k_{max}$. For each such ISC the algorithm tries to compute a corresponding test.

Algorithm 1: Test(program P, bound k_{max})

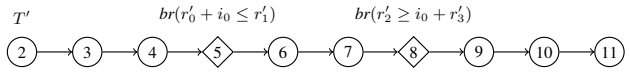
```
1  IForest forest ← ∅
2  ISC-Set W^0, ..., W^{k_max}, UN^0, ..., UN^{k_max}
3  for k = 0 to k_max do
4      W^k ← ∅
5      UN^k ← ∅
6  ī ← random inputs
7  foreach thread T_j do
8      π ← ConcolicExecution(P, (ī, (T_j, −)))
9      W^0 ← W^0 ∪ ExtractISCs(π)
10 for k = 0 to k_max do
11     while W^k ≠ ∅ do
12         pick and remove C from W^k
13         (result, ī, σ) ← RealizabilityCheck(C)
14         ISC-Set iscs ← ∅
15         if result ≠ realizable then
16             UN^k ← UN^k ∪ {C}
17             iscs ← ExploreISCs(C, write-nodes(forest))
18         else
19             π ← ConcolicExecution(P, (ī, σ))
20             W^k ← W^k ∪ ExtractISCs(π)
21             Wrts ← new-write-nodes(forest, π)
22             forall the C' ∈ UN^i, 0 ≤ i ≤ k − 1 do
23                 iscs ← iscs ∪ ExploreISCs(C', Wrts)
24         foreach C' ∈ iscs do
25             k' ← Degree(C')
26             if k' ≤ k_max then
27                 W^{k'} ← W^{k'} ∪ {C'}
```

1-5: Data Structures. We have three central data structures: (i) a global interference forest *forest* that stores all interference scenarios explored by concolic execution, (ii) a list of sets $W^0, \ldots, W^{k_{max}}$, where each W^k, for $0 \leq k \leq k_{max}$, serves as a worklist for ISCs of degree k, and (iii) a list of sets $UN^0, \ldots, UN^{k_{max}}$, where each UN^k, for $0 \leq k \leq k_{max}$, stores all processed but unrealizable ISCs of degree k. All these data structures are initially empty (cf. lines 1 to 5). During execution of Alg. 1, each generated ISC C of degree k is initially inserted into W^k and later on, if C is not realizable, it is moved to UN^k for further exploration.

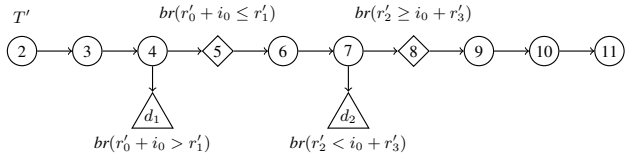
6-9: Initial Path Exploration. We initialize W^0 by executing a test $(\bar{i}, (T_j, -))$ for each thread T_j (line 8), where $\bar{i}$ is a random input vector (we use the same $\bar{i}$ for each thread T_j) and the schedule $(T_j, -)$ allows only a sequential execution of thread T_j without any interruption from other threads. When the concolic execution engine reaches the end of thread T_j, the program execution is aborted without executing any other thread (our approach can be generalized to handle barriers, but for simplicity of presentation we ignore them here). As the result of the concolic execution, a symbolic trace π is returned. Then, at line 9, ExtractISCs takes the symbolic trace π and derives new ISCs with the same degree as $IS(\pi)$ for further exploration of program behaviour, i.e., during initialization it generates ISCs of degree 0. We describe algorithm ExtractISCs in the next paragraph. We insert the returned ISCs into worklist W^0. Now, after the initialization phase in lines 3 to 5 in Alg. 1, W^0 contains for each thread in P a set of ISCs for further (initially thread-local) exploration.

Algorithm 2: ExtractISCs (Symbolic Trace π) : ISC-Set

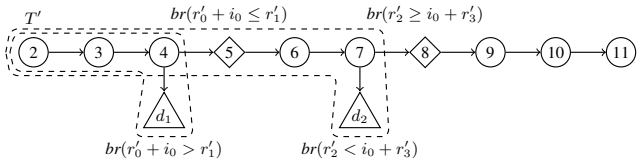
```
1 F ← addDanglingNodes(IS(π))
2 MergeInterferenceForests(forest, F)
3 ISC-Set iscs ← ∅
4 foreach dangling node n newly added to forest do
5     iscs ← iscs ∪ {CIS(forest, n)}
6 return iscs
```



(a) Interference scenario $IS(\pi_{T'})$ for a symbolic trace $\pi_{T'}$ obtained by a sequential execution of thread T' (cf. Fig. 3).



(b) Interference scenario $IS(\pi_{T'})$ extended with dangling nodes d_1 and d_2.



(c) Interference scenario candidates $CIS(forest, d_1)$ and $CIS(forest, d_2)$.

Figure 6: Example showing initialization for thread T' (cf. Fig. 3).

Algorithm ExtractISCs. ExtractISCs, shown in Alg. 2, gets a symbolic trace π as input. Each branch event in π, has a corresponding dual branch event where its symbolic constraint is negated. Alg. 2 first obtains $IS(\pi)$. For example, Fig. 6a shows $IS(\pi)$ where π is the symbolic trace returned by the initial sequential execution of thread T' (introduced in Fig. 3). In line 1 in Alg. 2, $IS(\pi)$ is extended to an interference forest F by introducing for each dual branch event a so called *dangling node*, e.g., nodes d_1 and d_2 in Fig. 6b. Then, F is merged into *forest* (cf. line 2) as described in Remark 5.5. For each dangling node which was not merged with an existing node, we create an ISC (cf. lines 4 and 5), e.g., $CIS(forest, d_1)$ and $CIS(forest, d_2)$ in Fig. 6c. These ISCs are returned to the main algorithm. Note that all generated ISCs will have the same degree as $IS(\pi)$. This is due to the fact that the dangling nodes which are not already present in *forest* occur after the sink of the ISC which was used when generating the test for π. Since *forest* is initially empty, during the initialization phase an ISC is generated for each dangling node in F.

10-27: Main Loop. The testing algorithm processes worklists $W^0, \ldots, W^{k_{max}}$ in ascending order (cf. main loop at line 10). While processing W^k (the loop at line 11), each ISC $C \in W^k$ is removed from W^k and its realizability is checked (see Section 7.2, algorithm RealizabilityCheck). Given an ISC C, RealizabilityCheck returns a triple $(\text{result}, \bar{i}, \sigma)$ where result indicates whether C is realizable or not. If C is realizable, then $(\bar{i}, \sigma)$ forms a test that realizes C.

15-17: ISC Exploration. If C is not realizable, then we store C into UN^k for later processing. Since the realizability of ISCs is not monotonic (as discussed in Section 5), C still has a chance to become realizable if we introduce some more interferences in it. Therefore, we collect all write nodes stored in *forest* (cf. line 17) and then use ExploreISCs (Alg. 3) to extend C to a set of ISCs

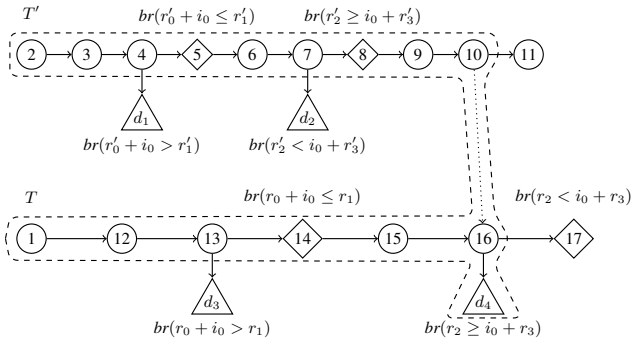


Figure 7: Interference scenario $IS(\pi)$ from Fig. 3 extended by dangling nodes d_1, d_2, d_3, and d_4. The loosely dashed lines enclose the interference scenario candidate $CIS(\mathit{forest}, d_4)$.

for target branch *sink*(C) by introducing a new interference from a write in $Wrts$ to a read in C. Each of the generated ISCs has a degree $i > k$ and is added to W^i in lines 24 to 27. Since $i > k$, the newly generated ISCs will be processed after W^k is processed completely. We discuss ExploreISCs at the end of Section 7.1.

19-20: Path Exploration. If C is realizable, then the program is concolically executed with input vector $\bar{i}$ and according to schedule σ (cf. line 19). The moment *sink*(C) is executed, the schedule σ enforces an exclusive execution of thread *Th*(*sink*(C)) without any interruption from other threads. As in the sequential case, the moment the end of thread *Th*(*sink*(C)) is encountered, the concolic execution engine aborts program execution. The concolic execution returns a symbolic trace π which is fed to ExtractISCs to derive ISCs from π similar to the $k = 0$ case described earlier (see Fig. 7 for an example with $k > 0$). All generated ISCs are added to W^k.

21-23: ISC Re-Exploration. During concolic execution we might observe a write event we have not observed so far (this might be due to the fact that we covered a new code location, or that we observed a write to a shared variable combined with a symbolic value which was not observed before). Note that, at line 17, we extended an unrealizable ISC by interferences resulting from writes stored in *forest*. When we observe a write event which was not in *forest* back then, we have to reconsider all previously unrealizable ISCs and have to extend them by interferences from this new write event to reads in these ISCs. This happens in lines 21 and 23. There, each previously unrealizable ISC C with degree smaller than k is re-explored with the newly observed writes. Each such write event must occur after *sink*(C) was executed and, therefore, has degree k. Then, the ISCs generated in line 23 have degree greater than k and are added to the according worklists in lines 24–27.

Algorithm ExploreISCs. Algorithm ExploreISCs explores ISCs by extending existing ISCs with new interferences. Let n_r be a read node in a given ISC C. Let n_w be a write node in *forest* and let I_w be the causal interference scenario of n_w, i.e., $I_w = CIS(\mathit{forest}, n_w)$. To create an ISC C'' which extends C by the interference (n_w, n_r), the algorithm checks the following conditions:

(i) n_r and n_w are in different threads, i.e., *Th*(n_r) $\neq$ *Th*(n_w).

(ii) n_r reads from the variable to which n_w writes to, i.e., if $Ac(n_r) = rd(x, r)$, for some symbolic variable r, then $Ac(n_w)$ is of the form $wt(x, val)$, for some expression val.

(iii) n_r is not involved in any interference in C or I_w, i.e., $n_r \notin R_I(C)$ and $n_r \notin R_I(I_w)$ (cf. Section 5).

(iv) C and I_w are compatible (cf. Def. 5.4 and Remark 5.5).

Algorithm 3: ExploreISCs(ISC C, write nodes $Wrts$) : ISC-Set

```
1  ISC-Set iscs ← ∅
2  foreach n_r ∈ read-nodes(C) \ R_I(C) do
3      let Ac(n_r) = rd(x, r) for some x
4      foreach n_w ∈ Wrts do
5          if Ac(n_w) = wt(x, val) for some val then
6              if Th(n_w) ≠ Th(n_r) then
7                  I_w ← CIS(forest, n_w)
8                  if n_r ∉ R_I(I_w) and compatible(C, I_w) then
9                      C' ← merge(C, I_w)
10                     C'' ← extend C' by interference (n_w, n_r)
11                     iscs ← iscs ∪ {C''}
12 return iscs
```

If all conditions are satisfied, then, in line 9, C and I_w are merged as described in Remark 5.5 and form an ISC C'. Then, in line 10, C' is extended to ISC C'' by introducing the interference edge (n_w, n_r). Alg. 3 collects all generated ISCs (cf. line 11) and finally returns them in line 12. Note that each generated ISC C'' has the same sink as C, i.e., *sink*(C'') = *sink*(C), and has at least one more interference than C, i.e., Degree(C'') $\geq$ Degree(C)+1. The degree of C'' might increase by more than one interference, because I_w might contain interferences which are not present in C, but, due to the merge, they show up in C' and C'' as well.

7.2 Soundness and Completeness

In Section 6, we discussed data constraints *DC*(I) and temporal constraints *TC*(I) corresponding to an interference scenario I. The following theorem shows how these constraints can be used to figure out whether an ISC is realizable or not.

THEOREM 7.2 (SOUNDNESS). *Let C be an ISC in W^k where $0 \leq k \leq k_{max}$. Then, C is realizable if and only if DC(C) and TC(C) are satisfiable.*

Algorithm RealizabilityCheck. Given an ISC C with *sink*(C) = n, the realizability of C is checked by determining whether *DC*(C) and *TC*(C) are both satisfiable. Assume that C is realizable and σ' and $\bar{i}$ are solutions for *TC*(C) and *DC*(C), respectively. Then RealizabilityCheck algorithm (called in Alg. 1 at line 13) returns a triple (result, $\bar{i}, \sigma$) where result determines that C is realizable and $\sigma = \sigma'(Th(n), -)$ that forces the sequential execution of thread *Th*(n) after σ'. According to Thm. 7.2, our test generation approach is sound, i.e., if the program is executed with input vector $\bar{i}$, and according to schedule σ', then the branch node n will be covered.

THEOREM 7.3 (COMPLETENESS). *Given a deterministic program P and a bound k_{max}, Alg. 1 covers all branches of P that require at most k_{max} many interferences to be covered.*

Like all completeness results in concolic testing Thm. 7.3 relies on several idealizing assumptions. The theorem states that for deterministic programs without non-linear arithmetics and calls to external library functions, our con2colic testing algorithm covers all branches of P that require at most k_{max} many interferences to be covered. In practice, concolic execution falls back upon concrete values observed during execution to handle non-linear computations or calls to external library functions, for which no good

symbolic representation is available. For that reason, it always underapproximates program behaviours.

Note that if we do not stop Alg. 1 after it performed k_{max}-many iterations of the main loop or after full branch coverage is achieved, then Alg. 1 actually achieves, in the limit, a stronger coverage than branch coverage. We leave an exact characterization of the coverage achievable by Alg. 1 for future work.

7.3 Unforeseen Interferences

In order to drop assumption (i) stated at the beginning of Section 7, we need to make the following changes: (1) The concolic execution engine stops as soon as an unforeseen interference is observed and returns a symbolic trace π that ends with the read event of the unforeseen interference. (2) Alg. 2 is extended as follows: When building forest F in line 1, we add a distinguished *dangling node* which is labeled with the read event of the unforeseen interference. However, we do not add the unforeseen *interference* to F. As an effect, Alg. 2 will then, in line 5, create a causal interference scenario for this special dangling node. Consequently, Alg. 1 will then try to realize this interference scenario, first without introducing an interference. If this is not possible then it will introduce some interferences later. This enables us to explore the interference space and build the interference forest in a systematic way, i.e., while processing W^k, the interference forest is updated with interference scenarios of degree k. Observe that we have to extend the notion of CIS to allow sink nodes labeled with read events. Since our algorithms never make use of the fact that the sink of a CIS is a branch node, the overall testing algorithm is not changed. Barriers can be handled in a similar way; due to space restrictions, we omit details.

7.4 Lock-Protected Writes

Consider an ISC C with $sink(C) = n$. It might be the case that for a thread $T_i \neq Th(sink(C))$, the last node in G_{T_i} is labeled with a write event that happened while T_i was holding some locks. This may cause the following problems: (i) C might never become realizable, e.g., if n is also protected by the same locks, then T_i does not have any chance to release the locks. (ii) C might be realizable but the test generated for C may lead to a deadlock, e.g. thread $Th(sink(C))$ acquires any of these locks later. To solve these problems, whenever we create a new ISC C, we extend all thread-local sub-scenarios of C according to *forest*, with the exception of thread $Th(sink(C))$, such that for each thread T_i the last node in G_{T_i} is not protected by any lock. As an example consider the ISC shown in Fig. 7. There, T' holds a lock at node 10. Therefore, the ISC is extended to include node 11 where T' releases the lock. Note that this extension might not be unique if the release of a lock can happen in different code branches. To preserve completeness, we consider all possible extensions, i.e., we actually generate a set of ISCs. Furthermore, like in the case of re-exploration of unrealizable ISCs due to newly observed write nodes, we have to re-explore ISCs whenever we observe new lock-release events. In our benchmarks, the extensions until lock-free nodes were unique.

7.5 Optimizations for ISC Exploration

Unsat-Core Guidance. In Alg. 1, ISC exploration is performed by adding new interferences to ISCs. In case an ISC is not realizable, it might be the case that no extension of the ISC by interferences will ever get realizable. From the unsatisfying core of the constraint systems defined in Section 6, we can identify such situations. Let $C = (V, E, \ell)$ be an ISC. Data constraint $DC(C)$ is then equal to $DC_{branch}(V) \wedge DC_{interfere}(C) \wedge DC_{local}(C)$. Extending C to a new interference scenario C' by adding an interference to C removes some predicates in $DC_{local}(C)$ from $DC(C')$ but the predicates in $DC_{branch}(V)$ and $DC_{interfere}(C)$ remain as part of $DC(C')$. Therefore, if the unsatisfying core of $DC(C)$ does not involve predicates from $DC_{local}(C)$, we can conclude that $DC(C')$ or any other extension of C will not be realizable as well and, therefore, we can exclude C from further exploration. Analogously, if $TC(C)$ is not feasible and no constraints from WRC_{local} are involved in the unsatisfying core then, again, we can conclude that C will not become realizable by adding new interferences and we can exclude C from further exploration. Furthermore, in both cases, the unsat core can be used to guide the exploration by introducing an interference for a so far local read whose constraints are involved in the unsat core.

Duplication Freedom. Alg. 1 allows multiple instantiations of the same ISC. For example, suppose that an ISC C becomes realizable by introducing interferences for two reads. The algorithm can first select any of these reads and generate two ISCs in which one of these reads is interfered. Then, in the future, these two ISCs can be extended such that the other read is also interfered, generating two instances of the same ISC. To avoid duplication of ISCs, we use a caching mechanism. In this way, an ISC will be processed only if it is not already in the cache.

Prioritized Exploration. While processing each worklist W^k, we can choose to prioritize the ISCs in W^k. For example, in our implementation, we assign higher priorities to ISCs which would cover some yet uncovered part of program code (in case they are realizable). Based on this exploration strategy, Alg. 1, at line 11 first processes ISCs with higher priorities.

8. EXPERIMENTS

Our con2colic testing approach is implemented in a tool called CONCREST as an extension to CREST [1]. We use a standard collection of concurrency benchmarks to evaluate its effectiveness. We ran our experiments on a dual-core 64-bit Linux machine with 3.2GHz and 16GB RAM.

Benchmarks. `bluetooth` is a simplified version of the Bluetooth driver from [15]. `sor` is from Java Grande multi-threaded benchmarks (which we translated to C). `ctrace-a` and `ctrace-b` are two test drivers for the `ctrace` library. `apache-a` and `apache-b` are test drivers for APACHE FTP server from BugBench [9]. `splay` and `rbTree` are test drivers for a C library implementing several types of trees. `aget` is a multi-threaded download accelerator. `pfscan` is a multi-threaded file scanning program. Finally, `art` is an example designed by us to evaluate the scalability of our approach when the number of threads increase. It has the property that there is a new assertion in it every time we increase the number of threads by one.

Experimental Results. In our experiments we set $k_{max} = 100$ (at most 100 interferences) and a timeout of 2 hours. The results are presented in Table 1. We learned the following important facts: **(i)** CONCREST is effective at finding bugs. All the known bugs were discovered. **(ii)** All bugs discovered by CONCREST in benchmarks were the result of a branch which would not be covered sequentially. **(iii)** All bugs were discovered under a relatively small number of interferences (maximum 4). **(iv)** On average, a substantial number of branches were not sequentially coverable and were only covered after interferences were introduced, e.g., for `rbTree` which has fixed input, branch coverage increases from 67 to 95 (maximum number of coverable branches). In the lack of a bug found, reaching this maximum provides guarantees to the tester that, e.g., no assertions in the code can be violated. **(v)** We set the maximum number of interferences to be 100, but the actual bound explored by CONCREST is much smaller. This is because in most cases (with the exception of 2 timeout cases), we either achieved

Table 1: Experimental Results

Benchmark	♯Threads	♯Inputs	♯Branches (total)	♯Branches k=0/1/2/3/4/...	Max k reached (reason)	♯Branches k=0 → Max k	Bug found (k)	♯ ISC (total)	time (total)
bluetooth	3	2	24	14/8/2	2 (Full Cov.)	14→24	yes(2)	282	0.5
sor	3	-	48	37/8/0/0/3	4 (Full Cov.)	37→48	yes(3)	145	0.6
ctrace-a	3	-	94	54/3	5 (Max Cov.)	54→57	yes(1)	28	0.7
apache-a	3	3	72	41/0/1	11 (Max Cov.)	41→42	yes(2)	392	1.0
splay	3	-	112	46/14/4	15 (Max Cov.)	46→64	no	3501	6.2
apache-b	3	3	48	35/3	11 (Max Cov.)	35→38	yes(1)	22150	15.4
aget	3	-	88	56/0/1	21 (Max Cov.)	56→57	yes(2)	23197	170.4
rbTree	3	-	146	67/22/4/2	24 (Max Cov.)	67→95	no	77037	296.3
pfscan	3	2	130	92/0/0/0/1	4 (Timeout)	92→93	yes(4)	3012548	7200.0
ctrace-b	3	-	128	75/5	2 (Timeout)	76→81	yes(1)	315639	7200.1
art2	3	2	8	7/1	1 (Full Cov.)	7→8	yes(1)	80	0.3
art3	4	3	12	10/1/1	2 (Full Cov.)	10→12	yes(2)	17942	21.8
art4	5	4	16	13/1/1/1	3 (Full Cov.)	13→16	yes(3)	2842066	197.1
art5	6	5	20	16/1/1/1/1	4 (Full Cov.)	16→20	yes(4)	10851573	741.1

♯**Branches**: number of *static* branches, i.e. number of basic code blocks. k: number of interferences. **Full Cov.**: all branches are covered. **Max Cov.**: all possible interference scenario candidates are explored. ♯**ISC**: number of explored interference scenario candidates. "14/8/2" means 14 branches covered at $k = 0$, 8 (new) branches at $k = 1$, and 2 (new) branches at $k = 2$. $14 \rightarrow 24$ indcates the difference between the number of branches covered sequentially (14) and the total number of branches covered (24).

full branch coverage or explored all possible ISCs (i.e. no more branches are coverable). **(vi)** Our approach scales well as the number of threads increase; see `art`.

There are cases where maximum branch coverage is achieved, but the number does not coincide with the total number of static branches. These are due to (sanity-check type) assertions in the code which were never meant to be violated.

Table 2 presents the effect of the optimizations discussed in Section 7.5 for `pfscan` (as an example). In this benchmark, the bug (i.e. assertion violation) is discovered at $k = 4$. When there is no optimization enabled, it runs out of memory with $k = 2$. The efficacy of unsat-core guidance is clear because without this optimization k cannot go higher than 2. In fact, to move to $k = 4$ and catch the assertion violations, both unsat-core guidance and duplication-freedom have to be enabled. The effect of prioritized exploration can be observed by comparing rows 1 and 3: when prioritization is enabled the assertion violation is found earlier.

We compared our tool with Poirot [13] on some of the benchmarks. Poirot exploits context-bounding sequentialization of concurrent programs and performs a static analysis to check for safety properties. A side by side comparison with Poirot (when it does not aim for coverage in the same sense as CONCREST is not meaningful). Our experiments showed that Poirot did not scale well for the programs that we checked. For example, the bug in `sor` was found within a second by our tool, but Poirot was not able to find it for context-bound of 2, 3, and 4 within 1.5hrs (for each bound).

Table 2: Effects of optimizations on benchmark `pfscan`.

Row	U	P	D	Assertion Coverage t[s]	Max k	Total t[s]
1	+	+	+	4554	4	7200 (timeout)
2	-	+	+	-	2	7200 (timeout)
3	+	-	+	6701	4	7200 (timeout)
4	+	+	-	-	3	7200 (timeout)
5	-	-	+	-	2	7200 (timeout)
6	-	+	-	-	2	out of memory
7	+	-	-	-	3	7200 (timeout)
8	-	-	-	-	2	out of memory

U = unsat core guidance, **P** = prioritized exploration, **D** = duplication freedom. Symbols + and - represent optimizations being on and off, respectively. Last three columns correspond to the time it took for the assertion to be covered, the maximum k explored, and the total time for testing.

9. RELATED WORK

In the introduction, we surveyed a number of techniques for testing concurrent programs. Here, we focus on a subset of them which are closer to con2colic testing and discuss how con2colic testing distinguishes from them in more detail.

Extensions of concolic testing to concurrent programs have been proposed before. In jCute [18], the program is executed concolically and data-races in the observed execution are identified. Then, either the schedule is fixed and new input values are generated for the same concurrent schedule, or a new schedule is produced by keeping the inputs fixed and simply re-ordering the events involved in a data-race. In contrast to con2colic testing, jCute is incomplete due to its race-based schedule selection heuristic. Moreover, if a timeout occurs, it is impossible to quantify the partial work done as a meaningful coverage measure for the program.

Similar to con2colic testing, a recent related work [17], generates tests with the aim of increasing code coverage of concurrent programs. However, it uses an under-approximation of the program (i.e. a set of program runs), as opposed to the actual program. Therefore, it is incomplete. Furthermore, test generation is done by solving a constraint system that encodes the scheduling constraints and the data-flow constraints together while considering the *whole* computation in the runs. However, con2colic testing generates separate constraint systems for schedule generation and input generation which are based on only shared variable accesses and synchronization events; This reduces the complexity of the constraint systems drastically and increases scalability.

Some other recent work [21, 20] build a framework based on over and under-approximations of interferences of the programs to check for safety properties. Like [17], they build a constraint system which includes local computation as well as global computation. Therefore, to reduce scalability issues, they focus only on program slices obtained from program executions.

Finally, several sequentialization techniques [8, 23, 15, 13, 16] have been proposed to reduce the problem of verifying concurrent programs to verification of sequential programs. However, one cannot apply traditional sequential testing techniques on the sequentialized programs obtained by many of them as they are highly non-deterministic. Furthermore, to execute sequential programs obtained by [8, 13, 16], we should guess the values of shared variables at the beginning of each context. However, wrong guesses might result in reaching invalid program states.

10. ACKNOWLEDGEMENTS

This work was supported by the Canadian NSERC Discovery Grant, the Vienna Science and Technology Fund (WWTF) grant PROSEED, and the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF).

11. REFERENCES

[1] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. In *ASE*, pages 443–446, 2008.

[2] F. Chen, T. Serbanuta, and G. Roşu. JPredictor: A Predictive Runtime Analysis Tool for Java. In *ICSE*, pages 221–230, 2008.

[3] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-Bunded Scheduling. *SIGPLAN Not.*, 46(1):411–422, 2011.

[4] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting Null-Pointer Dereferences in Concurrent Programs. In *FSE*, pages 47:1–47:11, 2012.

[5] P. Godefroid. Model Checking for Programming Languages Using VeriSoft. In *POPL*, pages 174–186, New York, NY, USA, 1997. ACM.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.

[7] P. Godefroid, M. Y. Levin, and D. A. Molnar. Active Property Checking. In *EMSOFT*, pages 207–216, 2008.

[8] A. Lal and T. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. *Form. Methods Syst. Des.*, 35:73–97, 2009.

[9] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[10] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. *SIGPLAN Not.*, 42(6):446–455, 2007.

[11] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A Systematic Testing Tool for Concurrent Software, 2007.

[12] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *ASPLOS*, pages 25–36, 2009.

[13] S. Qadeer. Poirot: A Concurrency Sleuth. In *ICFEM*, pages 15–15, 2011.

[14] S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. In *In TACAS*, pages 93–107. Springer, 2005.

[15] S. Qadeer and D. Wu. KISS: Keep It Simple and Sequential. *SIGPLAN Not.*, pages 14–24, 2004.

[16] Z. Rakamarić. STORM: Static Unit Checking of Concurrent Programs. In *ICSE*, pages 519–520, 2010.

[17] N. Razavi, F. Ivancic, V. Kahlon, and A. Gupta. Concurrent Test Generation Using Concolic Multi-Trace Analysis. In *APLAS*, 2012.

[18] K. Sen. *Scalable Automated Methods for Dynamic Program Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 2006.

[19] K. Sen and G. Agha. Concolic Testing of Multithreaded Programs and Its Application to Testing Security Protocols. Technical Report UIUCDCS-R-2006-2676, University of Illinois at Urbana Champaign, 2006.

[20] N. Sinha and C. Wang. Staged Concurrent Program Analysis. In *FSE*, FSE'10, pages 47–56, 2010.

[21] N. Sinha and C. Wang. On Interference Abstractions. In *POPL*, pages 423–434, 2011.

[22] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *FSE*, pages 37–46, 2010.

[23] S. Torre, P. Madhusudan, and G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In *CAV*, pages 477–492, 2009.

[24] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic Predictive Analysis for Concurrent Programs. In *FM*, pages 256–272. Springer, 2009.

[25] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *ASPLOS*, pages 251–264, 2011.