Karlsruhe Institute of Technology

Predicting Errors in Concurrent Systems

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Luis M. Carril Rodríguez

aus A Coruña, Spanien

Tag der mündlichen Prüfung:	04-05-2017
Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Dr. Andreas Oberweis

Abstract

The unstoppable popularity of multicore chips has made concurrent programming ubiquitous. Parallel programming is difficult; multiple control flows and non-determinism make the development error-prone. Additionally, it introduces a new set of errors such as data races, deadlocks, or order violations. These errors are difficult to find due to the large number of possible interleavings in a parallel program.

Dynamic analysis techniques execute a program and perform some checks on the observed execution. Dynamic analysis is precise, as it relates to an actual execution with real values and states. These approaches generate false negatives due to non-explored paths and interleavings. The different interleavings are manifested with varying frequency because of external factors such as compilers, processors, or workloads.

In this work we present an approach that predicts errors from a single trace of a parallel program. We compute alternative interleavings off-line to reduce the timing effects of the observed execution. Using a process algebra, we build a model that generalizes the ordering of the trace, extrapolating possible interleavings. The model is explored for different concurrency errors. A predicted error is accompanied by a schedule, which is enforced in the program to manifest the error.

The approach has been evaluated for deadlock and data race detection, and compared with other dynamic approaches. While maintaining or increasing precision, between 50% and 86% fewer false warnings were produced. The presented tool and its model are also customizable to support other kinds of ordering failures. These failures require a specification that defines the relevant events in the program and describes their valid or invalid orderings. This feature is demonstrated in seven use cases.

Kurzfassung

Die Verbreitung von Multikernprozessoren hat die parallele Programmierung allgegenwärtig gemacht. Parallele Programmierung ist schwierig, da sie die Entwicklung auf Grund vielfacher Kontrollflüsse und Nicht-Determinismus fehleranfällig macht. Zusätzlich gibt es neue Arten von Fehlern, wie z.B. Wettläufe, Verklemmungen oder Reihenfolgeverletzung. Diese Fehler sind wegen der großen Anzahl von möglichen Verschränkungen schwer zu finden.

Dynamische Analyseansätze führen das Programm aus und untersuchen die beobachtete Ausführung. Dynamische Analyse ist präzise, da sie eine tatsächliche Ausführung mit realen Werten und Zuständen beobachtet. Diese Ansätze erzeugen falsch Negative, wegen nicht erforschter Wege und Verschränkungen. Die unterschiedlichen Verschränkungen treten mit unterschiedlicher Häufigkeit auf, wegen externer Faktoren, wie z.B. Compiler, Prozessor oder Arbeitslast.

In dieser Arbeit präsentieren wir einen Ansatz, der Fehler aus einer einzigen Spur eines parallelen Programms vorhersagt. Wir berechnen off-line alternative Verschränkungen, um den Zeitmessungseffekt der beobachteten Ausführung zu reduzieren. Wir bauen ein Modell mit Hilfe einer Prozessalgebra, das die Reihenfolge der Spur verallgemeinert; es extrapoliert mögliche Verschränkungen. Im Modell wird nach verschiedenen Nebenläufigkeitsfehlern gesucht. Ein vorhergesagter Fehler wird von einer Verschränkung begleitet, die in dem Programm erzwungen wird, um den Fehler zu reproduzieren.

Der Ansatz wurde für Verklemmungs- und Wettlauferkennung im Vergleich zu anderen dynamischen Ansätzen evaluiert. Die Verringerung von falschen Warnungen bewegt sich im Bereich von 50% bis 86%, mit gleicher oder höherer Präzision. Das vorgestellte Werkzeug und sein Modell sind anpassbar für andere Reihenfolgeverletzungen. Diese Fehler erfordern eine Spezifikation, die die relevanten Ereignisse im Programm definiert und die gültigen oder ungültigen Ordnungen beschreibt. Wir zeigen dieses Vorgehen an sieben Anwendungsbeispielen.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

In the past programmers relied on the ever increasing speed of new hardware to make their sequential programs run faster without effort. Around 2005 several issues started to limit that trend: dissipation issues due to power consumption growth, limitations in instruction level parallelism, and mismatches between processor speed and memory speed, as described in Sutter's article [Sut05]. The industry switched from packaging a single powerful processor on a chip to putting together multiple but slower processors.

To benefit from the new architecture, the programs need to execute their tasks in parallel. But concurrent programs suffer from concurrency failures[1], a new class of errors, such as data races, deadlocks, atomicity violations, and order violations. Program correctness means that a given program matches its specification. Some specifications are universal, e.g. no data races or no deadlocks, and all programs should comply with them. These failures have a general description that can be used to detect them. However, there are failures such as some atomicity and order violations, which are specific to a program or domain. To detect them, the intended semantics of the program are needed. In this case the developer provides a description of the desired behavior or of the failure, which is checked against the program.

In general correctness is undecidable, and property checking tools rely on approximations, sacrificing precision or completeness for the sake of practicality.

[1] A failure or fault is the deviation of the behavior of the software from the specification. A defect or bug is a deficiency in the source code that can lead to a failure. A mistake is a human action that causes a defect. An error can refer to any of these three terms.

A sequential program requires the appropriate input, so in testing techniques the responsibility to provide a set of relevant data rests with the developer. A concurrent program also needs an interleaving that orders the tasks executed in parallel under an input to observe a specific property. Concurrent programs are naturally non-deterministic to make the most of a multicore processor, and the number of interleavings grows exponentially with the number of instructions. For the developer it is very difficult to find or enforce an erroneous interleaving.

Figure 1.1 presents an example of non-determinism and a data race in a program.

```
int x = 0, y = 0, z = 0;
void main() {
    fork(worker);
    x = x + 2;
    y = x;
    join(worker);
    assert(x == 3);
}

void worker() {
    x = x + 1;
    z = x;
}
```

Figure 1.1.: Non-deterministic and racy program

Two threads, `main` and `worker`, concurrently update variable `x`. Depending on the executed interleaving, the result varies drastically. If we assume that the thread `main` is executed completely (reaching the `join`) before the thread `worker` starts, then `y = 2` and `z = 3`. But if `worker` is faster, the end values are `y = 3` and `z = 1`. Other results are possible; if `main` is preempted before executing the instruction `y = x`, and then `worker` is completed, we obtain `y = z = 3` at the end. In any of these cases the final value of `x` in memory is always going to be 3. Here the developer expects that the values of `y` and `z` are going to differ because of non-determinism, but the developer also expects that the value of `x` is the same, independently of the interleaving.

A more intricate interleaving is the following: `worker` loads the value of `x` from memory to execute the increment `x = x + 1`. But before the increment stores the result, which is 1 in memory, `main` completely executes the increment `x = x + 2`, effectively storing a value of 2 in memory. Then `worker` overwrites `x` with its own result of 1, thanks to the store operation of `x = x + 1`. At the end the value of `y`, `z` and `x` is 1. A similar interleaving, but reversing the order between the threads, would result in `x = y = z = 2`. These outputs are completely unexpected and could be the cause of other failures in the program.

This failure – a data race – is produced by a misunderstanding of the developer, namely that the increments are executed atomically. The programmer must explicitly ensure that they are executed without interruption by other threads. A mechanism to solve this problem is to use *locks* to establish a mutually exclusive area, as shown in Figure 1.2.

```
int x = 0, y = 0, z = 0;
mutex m;
void main() {
    fork(worker);

    lock(m);
    x = x + 2;
    y = x;
    unlock(m);

    join(worker);
    assert(x == 3);
}

void worker() {
    lock(m);
    x = x + 1;
    z = x;
    unlock(m);
}
```

Figure 1.2.: Protected non-deterministic increment example

Both threads use the synchronization primitives `lock` and `unlock` on a variable `m`. These primitives ensure that the instructions executed between a call to `lock` and a call to `unlock` by one thread cannot be preempted by other instructions in a similar block on another thread. In this case the final result of `x` is always 3. Thanks to the use of the mutex, no data race happens. But the non-determinism in thread execution stands, as the values of `y` and `z` can still be 2 and 3, or 3 and 1. In fact the synchronization reduces the total number of possible interleavings in the program to two: `main` reaches the `join` first or `worker` completes first.

Figure 1.3 shows a program with a domain-specific concurrency error.

```
file f;
void main() {
    fork(worker);
    f = open_file("foo");

    some_work();

    join(worker);
    close_file(f);
}

void worker() {
    char[] data = compute();
    write_file(f, data);
    clear(data);
}
```

Figure 1.3.: Concurrent use of file API

The thread `main` has the task of opening and closing a file, with the functions `open_file` and `close_file`. Meanwhile at some point the thread `worker` writes to the file with the function `write_file`. The expected behavior by the programmer is that `write_file` happens between the opening and the closing of the file. The programmer relies on undefined time periods between the three calls for a correct order. The correct order only occurs when `worker` spends enough time in the function `compute` so `main` has time to open the file. But no mechanism ensures that `worker` executes `write_file` in-between. The thread `worker` can be faster than the thread `main` and perform `write_file` before the file is open. This situation could produce an immediate failure or be the root of subsequent failures, e.g. data loss.

This failure is an order violation. The programmer assumes that due to the amount of work done by both threads, the call to `write_file` always occurs in the middle. This error is also domain specific: the file-related API in this example requires the file to be open and closed explicitly. Another API could perform or check the opening and closing implicitly in `write_file`. Thus, the API developer needs to define the valid or invalid orders.

The associated defect is solved defining an explicit order between the threads, with a $signal - wait$ mechanism, as in Figure 1.4.

```
file f;
cond v;
void main() {
    fork(worker);
    f = open_file("foo");
    signal(v);
    some_work();
    join(worker);
    close_file(f);
}

void worker() {
    char[] data = compute();
    wait(v);
    write_file(f, data);
    clear(data);
}
```

Figure 1.4.: Ordered use of file API

The thread executing a `wait` call is blocked until the other thread executes a corresponding call to `signal` on the same variable `v`. This synchronization forces the thread `worker` to execute the function `write_file` at some point after the file has been opened. In general all concurrency failures imply an interleaving not expected by the developer.

1.2. Problem Statement

There are two fundamental techniques to detect concurrency failures. Static techniques analyze the source code and perform diverse analyses to find defects, such as control flow, data-flow, or problem specific algorithms, e.g. *lockset* for data races. These techniques produce many false positives; they have to make approximations, because the discovery of the required properties in a non-running program is undecidable.

Dynamic analyses however execute a program and perform some checks on the observed execution. They limit themselves to the observed execution, which generates false negatives due to non-explored paths and interleavings, but they are less noisy as they relate to an actual execution with real values and states. These analyses use a monitor to check the fulfillment or violation of a desired property; data-race-specific algorithms are *lockset* and *happens-before* algorithms. Lockset assumes that all shared accesses must be consistently protected by the same lock, while happens-before establishes a logical order between instructions based on synchronization primitives.

For example, the interleaving shown in Figure 1.5 is a specific observed execution of a program. Two threads access the two variables `x` and `y`, but only the accesses to the second one are correctly protected by a lock.

thread 1	thread 2
`x = 1`	
`lock``(m)`	
`y = y + 2`	
`unlock``(m)`	
	`lock``(m)`
	`y = y + 1`
	`unlock``(m)`
	`x = 2`

Figure 1.5.: Interleaving with no happens-before detectable race

In the described interleaving a happens-before detector does not find the possible data race between both accesses to `x`. This happens because the critical sections protected by the lock `m` are happens-before-ordered, and they extend the order to the accesses on `x`, because the happens-before relationship is transitive. A happens-before detector finds the race if the observed interleaving is the one in Figure 1.6.

thread 1	thread 2
	`lock``(m)`
	`y = y + 1`
	`unlock``(m)`
	`x = 2`
`x = 1`	
`lock``(m)`	
`y = y + 2`	
`unlock``(m)`	

Figure 1.6.: Interleaving with happens-before detectable race

In this case the critical sections do not propagate the happens-before order to the accesses on `x`, and the race is detected.

The problem here with a happens-before detector is its dependency on the timing of the observed interleaving. Different interleavings manifest with varying frequency due to the influence of compilers, processor architectures, operating systems, or workload in the machine. The developer can consistently observe the same behavior across dozens of executions, but any change of any factor or in the source code could lead the program to exhibit another set of interleavings more frequently. Gait [Gai86] named this problem the *probe effect*. The program during testing, or under a race detector, could exhibit the interleaving in Figure 1.5 regularly; but in production the interleaving in Figure 1.6 could be more frequent. Or the race detector executes the interleaving where the race is easy to find, but when the developer tries to observe the race with debugging techniques, the debugger influences the scheduling, making it difficult to reproduce the race situation.

Two issues arise: one is the number of alternative interleavings observed by the algorithm, the other is the reproducibility of the reported failure. For a happens-before detector the reproducibility can be solved by storing the whole trace, or at least the synchronization events, of each execution, so the order observed by the detector can be replayed. If the detector explores the same set of interleavings, the same traces are stored each time. Each analysis of an already explored interleaving now suffers additional execution and storage overheads due to the trace logging. More time spent per execution implies less time to perform additional executions, which could increase the interleaving coverage. The number of interleavings is still limited, and the race in Figure 1.5 would not be found. The application of the lockset algorithm (alone or combined with happens-before) in this case detects the race in both interleavings, but does not build an order of events that can be used to reproduce the race.

Other failures, such as the order violation shown in Figure 1.3, require a specific monitor that tracks the desired events and describes the valid orderings. Happens-before can be used to check if the observed event order is one of the valid orders, but cannot reason about completely different interleavings. But lockset offers a more limited utility, as the algorithm can only report if a pair of events takes place without the protection of the same lock. The lockset algorithm cannot be used to increase the number of explored interleavings on general properties as in race detection.

1.3. Thesis Objectives

The objective of this work is to present a dynamic automatic approach to detect concurrency failures that:

- Covers non-observed interleavings, exploring reorderings of a trace and maximizing the number of detected failures in a single recording.
- Enables reproduction of erroneous interleavings, so the developer can re-execute the program and pinpoint causes.
- Does not produce false positives, relieving developers from chasing false alarms.
- Minimizes the size of the trace, as size directly impacts the overhead. A smaller trace also reduces the probe effect.
- Is extensible, allowing the definition of different failure patterns, so not only generic failures can be detected, but also domain-specific ones.

With these requirements, we present our research hypothesis:

> *Model-based analysis can predict failures from a single observed trace and generate reproducible witnesses without false positives and few false negatives.*

The key idea of this work is to execute a program under a defined input, such as a test case, and record the minimum necessary events for the target failure type, e.g. only the synchronization instructions for a deadlock. Then we build a formal model using a process algebra to represent the partial orders in the threads. A process algebra is naturally suited to describe a concurrent system by its transitions, which are determined by the events performed in the trace. The model represents all possible reorderings of the events in the trace while maintaining the order of the synchronization instructions, such as mutual exclusion for locks or a complete happens-before order for signal-wait. The failures are also defined in terms of the process algebra; they are checked against the model extracted from the program through model checking. We provide the definitions of data races and deadlocks. For other types of failures that depend on program semantics, the developer must provide a description of the property to be checked – the erroneous or the correct behavior. The events will be extracted automatically from the program. Model checking provides examples of how the property was violated. The example is used to force the program to follow the same scheduling.

The idea of this thesis has been introduced in our publications [CT15a] and [CT15b]. We implemented the technique in our tool called RaceQuest. Race-Quest performs the steps mentioned above automatically: trace capture, model generation, checking, and reproduction. RaceQuest works on parallel programs written in C/C++ using the POSIX threads library.

1.4. Structure of the Thesis

The rest of this thesis is structured in the following way: Chapter 2 presents some concepts and algorithms, and introduces the CSP process algebra. Chapter 3 describes prior work on concurrency error detection. Chapter 4 describes the different steps of RaceQuest in detail: trace, model, checking, and reproduction; it also uses deadlock detection as a motivating example. Chapter 5 extends the trace and model used for deadlock detection to data race detection. Chapter 6 facilitates the extension of the trace, model, and check by the user, to detect domain-specific failures. Finally, Chapter 7 concludes the thesis along with ideas for future work.

2. Basic Concepts

In this chapter we introduce some concepts, e.g. synchronization constructs and concurrency errors. We also describe several algorithms and technologies: data race detection algorithms, CSP process algebra, Aspect-Oriented Programming, and LLVM.

2.1. Concurrent Systems

A concurrent system is one that performs multiple tasks simultaneously. In a sequential system, only one task is performed at a given point, and it must end before starting the next task. The most popular concurrency model is multithreading with shared memory. Languages such as C/C++ or Java use this model. In multithreading a program is composed of several instruction flows, threads, that access the same data, the shared memory. Lee [Lee06] states that it is difficult to reason about a multithreading program, hence developers easily make mistakes. Multithreading suffers from new types of failures, which are not present in sequential programs, such as data races and deadlocks. There are alternative concurrency models, such as message-passing or data parallelism. Adopting any of these other models prevents some of these errors.

2.1.1. Trace

A trace is a record of the operations performed by an executed program. In a sequential program a trace is a sequence of operations with a total order. But in a parallel program a trace only provides a partial order of operations. For each individual thread the order is total, but operations by different threads can take place simultaneously. Synchronization primitives create an order among operations in different threads.

Thread segment

A thread segment is the set of operations by a single thread in a trace between two synchronization operations. Thread segments do not include any synchronization operations; thread segments are the frontiers between segments. The thread in the trace consists of a sequence of ordered thread segments, where each segment is a sequence of ordered instructions. The synchronization operations define the order among thread segments in two different threads.

Thread segments are useful to group instructions with the same logical ordering in respect to instructions in other threads. If two thread segments are concurrent, then all the instructions in both thread segments are concurrent.

2.1.2. Synchronization Mechanisms

Threads run concurrently and non-deterministically, and they work on the same shared data. Threads can conflict in using the same data by overwriting the work of other threads. To avoid such failures, the programmer must regulate the thread accesses to the shared memory, using synchronization mechanisms. A synchronization mechanism creates a logical order between source code blocks. This order makes a set of instructions non-concurrent. Synchronization reduces the non-determinism in the program.

Mutex

A critical section is a block of code that accesses a shared resource and must not be concurrently accessed by more than one thread.

A *mutex* or *lock* is a synchronization mechanism that enforces mutual exclusion on critical sections. A mutex starts in a free state. Any thread can acquire, *lock*, the mutex, which goes into a locked state. While locked no other thread is able to acquire it. Other threads trying to acquire the mutex are blocked. A mutex becomes free again after a release operation, *unlock*. Only the thread that has acquired the lock can release it. Once the mutex is free, another thread can acquire it.

The source code located between the acquire-release calls for the same mutex in a multithreaded program will never be executed in parallel. Mutexes are unaware of the operations done while they are being held, and they do not check or enforce any behavior on the protected operations.

Figure 2.1 describes a program with two parallel functions and a mutex. The calls to `lock` and `unlock` with the mutex `m` ensure that the lines 5 and 11 are never executed in parallel. The first thread reaching a `lock` acquires the mutex. The other thread cannot complete its own call to `lock` until the mutex is released by the other thread.

```
 1  int x = 0;
 2  mutex m;
 3  void thread_1() {
 4      lock(m);
 5      x++;
 6      unlock(m);
 7  }
 8
 9  void thread_2() {
10      lock(m);
11      x = 3;
12      unlock(m);
13  }
```

Figure 2.1.: Mutex example

There are variations of mutexes and their primitives. A *try_lock* primitive is a variation of the acquire operation; if the mutex is already in the locked state, the calling thread is notified and aborts the operation. A *timed_lock* is another acquire operation variant; the blocked thread will abort the acquire operation after waiting for the specified time. A read-write mutex is a variation of the mutex with the additional state *read-locked*. Threads can *read-acquire* or *acquire* the read-write mutex. A normal acquire still transitions the mutex to the locked state. But a read-acquire goes from the free state to the read-locked state. In the read-locked state multiple threads can read-acquire the mutex. It is not possible to go from read-locked state to normal lock without going through the free state. The use case behind read-write mutexes is to allow multiple threads to read shared data concurrently but to write them exclusively. The read-write mutex avoids reading and writing happening in parallel. It is the responsibility of the developer to ensure that only read operations are executed while doing a read-acquire.

Ordering threads

A mutex does not specify which source code block must be executed first, it allows any order. Another use case is when a specific operation must always take place following another operation, such as the use of a variable after its initialization. Several mechanisms impose such an order.

Thread creation and destruction operations impose an order between the involved threads. A thread spawns a new thread, its child thread, with access to the same memory with a *fork* operation. The operations of the parent thread after the fork are parallel to all the instructions of the child thread. Parent thread operations prior to the fork cannot be parallel to the child thread operations, as the thread did not exist before that point. Matching the fork operation there is the *join* operation. A parent thread waits in a join call until the specified child thread ends. All instructions in the parent thread after the join happen after the operations in the child thread, as that no longer exists.

Signal-wait

Arbitrary orderings between two threads, similar to those provided by a join, can be specified with the *signal-wait* combination. A thread executing a *wait* is blocked on a condition variable c until another thread wakes it up with a *signal* on c. The operations in the signaler thread before the signal take place before the operations in the waiter thread after the wait.

In practice the wait operation depends on a predicate; the signaling thread must ensure that the predicate is satisfied. The wait call is preceded by a conditional branch with the predicate as condition. If the predicate is already met before reaching the wait call, then the wait call is skipped. Otherwise the wait occurs. To achieve efficient implementation, the wait allows sporadic wake-ups of blocked threads. The preceding conditional check must be reevaluated and, if the check is not passed, the thread must wait again. Multiple threads could be blocked on the same condition variable c with the same or different predicates.

Figure 2.2 contains a program with two parallel functions using the signal-wait mechanism. We want to ensure that the function `second_task` is always executed after the function `first_task`. The predicate is represented by the variable `flag`. The thread `waiter` checks the variable `flag` and waits until `flag` turns true. Concurrently to the wait, the thread `signaler` executes the function `first_task`, sets `flag` to true, and signals on `c`. Only then will `waiter` continue and perform the function `second_task`. Note that the variable `flag` is accessed in parallel by both threads; it is protected by the mutex m. The function `wait` takes the mutex m as an argument and internally unlocks and locks m before and after being blocked.

```
boolean flag = false;
mutex m;
cond_var c;
void waiter() {
    lock(m);
    while(flag == false) {
      wait(c, m);
    }
    unlock(m);
    second_task();
}

void signaler() {
    first_task();
    lock(m);
    flag = true;
    signal(c);
    unlock(m);
}
```

Figure 2.2.: Condition variable example

A signal call wakes up a single thread waiting on *c*. A call to *broadcast* wakes up all threads waiting on *c*. Signal and broadcast are not buffered; they only wake up threads already waiting on *c*. A variant of the wait call is the *timed_wait*, where the waiting thread is automatically woken up after a specific period of time.

Detecting signal-wait orders is not easy; the order between the threads exists even if the wait call is not executed at all, i.e. the predicate is already met. We use additional instrumentation to detect the signal-wait orderings, as described in Section 4.3.2.

Barrier

A *barrier* is a multithread synchronization mechanism. A barrier allows multiple threads to synchronize together at a specific point, a call to *barrier_wait*. The threads are blocked in the *barrier_wait* call. A barrier starts closed and must be initialized with a number. The number indicates how many threads must reach the barrier, call *barrier_wait*, to open it. When the barrier opens the threads return from the *barrier_wait* primitive and resume their execution. All operations of the threads before the barrier take place before all operations after the barrier. After a barrier is used, it is automatically reset and can be reused.

Figure 2.3 displays a program with two parallel functions and a barrier. Both `pre_task` functions are executed concurrently. When both threads have reached the function `barrier_wait`, the threads resume their execution. The `post_task` functions also run in parallel, but no `pre_task` runs at the same time as any `post_task`.

```
barrier b = barrier_init(2);
void thread_1() {
    pre_task();
    barrier_wait(b);
    post_task();
}

void thread_2() {
    pre_task();
    barrier_wait(b);
    post_task();
}
```

Figure 2.3.: Barrier example

Semaphore

A *semaphore* is a counter with two operations, *sem_post* and *sem_wait*. If the value of the semaphore is greater than zero, a thread executing a sem_wait will decrease the counter by one and go forward. But if the value is zero, the thread will block on the sem_wait call until the counter is increased. The counter is only increased with sem_post. A semaphore has an initial value for the counter.

Figure 2.4 depicts a program with two parallel functions and a semaphore `s`. The semaphore counter starts at zero. The thread `waiter` will be blocked until the thread `poster` increases the value of the semaphore. The functions `pre_task` and `post_task` never run in parallel.

The primitive *sem_try_wait* is a variation of the sem_wait operation; if the semaphore counter is zero, the calling thread does not wait, otherwise it be-

```
semaphore s = sem_init(0,10);
void poster() {
    pre_task();
    sem_post(s);
}

void waiter() {
    sem_wait(s);
    post_task();
}
```

Figure 2.4.: Semaphore example

haves as sem_wait. Another sem_wait operation variant is *sem_timed_wait*: the blocked thread will abort the wait operation after waiting the specified time. A mutex is a special semaphore: it is initialized to zero, the maximum value of the counter is one, and there is ownership associated with the counter.

Atomic operations

Atomic operations are indivisible operations that appear to the rest of the threads to occur instantaneously. Atomic operations can be enforced in software with the use of mutexes or in hardware with specific instructions, such as test-and-set or fetch-and-add. Concurrent atomic operations on the same memory address do not produce any data race.

2.1.3. Ad-Hoc Synchronization

Ad-hoc synchronization is a synchronization mechanism constructed by the developers. This synchronization is not part of any standard library or language, but usually imitates some standard synchronization mechanism.

Ad-hoc synchronization commonly consists of a tight loop and a shared variable that works as condition, as the example in Figure 2.5. The threads `waiter` and `signaler` run in parallel. We define `x` as an atomic variable to avoid races on `x`, as well as compiler and processor reorderings. The thread `waiter` cannot leave the while-loop until the thread `signaler` modifies the value of `x`. The ad-hoc synchronization works as a signal-wait, and the function `first_task` is always executed before the function `second_task`.

Most concurrency error detection tools rely on standard synchronization primitives to detect inter-thread communication. The use of ad-hoc synchronization

```
atomic x = 0;
void waiter() {
    while(x == 0) {
      yield();
    }
    second_task();
}

void signaler() {
    first_task();
    x = 1;
}
```

Figure 2.5.: Ad-hoc synchronization loop

is a challenge for these tools, and becomes a source of false warnings. Our work does not detect ad-hoc synchronization. There are techniques to add support of ad-hoc synchronization. Xiong et al. [XPZ$^+$10] presented an automatic static technique to annotate ad-hoc synchronization, together with a study about its harmful effects. A runtime approach to detect ad-hoc synchronization was presented by Janeesari et al. [JT10].

2.2. Concurrency Failures

2.2.1. Deadlock

A deadlock is a situation that occurs when two or more competing actions of a program require the exclusive use of two or more resources. Each action acquires a different resource, but they cannot acquire another because it is already held by the other action. In this situation the actions cannot complete, and the program cannot progress further. Coffman et al.[CES71] defined four necessary and sufficient conditions for a deadlock:

1. Mutual exclusion: the resources involved cannot be shared; only one action can use the same resource at the same time.
2. Hold and wait of resources: an action retains a requested resource while requesting additional resources.
3. No preemption: a resource cannot be taken away from the current holder.
4. Circular wait: there is a circular chain of actions holding resources and requesting other resources.

Prevention of any of these four conditions is enough to avoid the deadlock situation. In multithreaded systems the actions are usually, but not necessarily, threads. A single thread can deadlock itself, e.g. request the same non-recursive mutex twice.

A communication deadlock occurs where the shared resource is a communication channel: two or more threads are waiting to receive a message from another thread. Communication deadlocks can occur while waiting for a lock release or the opening of a semaphore. Our work detects this kind of deadlocks.

Figure 2.6 shows an example of a program that can exhibit a deadlock. The two threads want to acquire the mutexes m and p. The thread main can be preempted just after acquiring the mutex m by the thread worker. In this case worker acquires the mutex p, but it is unable to acquire the mutex m. Likewise, main wants to acquire the mutex p, which is already held by worker. Both threads are deadlocked waiting for the release of the other mutex. This situation does not always manifest itself in the program, only under the described interleaving and a symmetric interleaving where the thread worker acquires the mutex p first.

```
mutex m, p;
void main() {
    fork(worker);

    lock(m);
    lock(p);
    unlock(p);
    unlock(m);

    join(worker);
}

void worker() {
    lock(p);
    lock(m);
    unlock(m);
    unlock(p);
}
```

Figure 2.6.: Deadlock example

2.2.2. Data Race

A data race is a concurrent access by two threads to the same memory location where at least one access is a write.

Figure 2.7 displays an example of a data race on variable x. Depending on the interleaving, the final value of x can be 3. If the read and write on x by both threads takes place concurrently, these instructions can overwrite each other's results, so x could be 1 or 2.

```
int x = 0;
void main() {
    fork(worker);
    x = x + 2;
    join(worker);
}

void worker() {
    x = x + 1;
}
```

Figure 2.7.: Data race example

Narayanasamy et al. [NWT+07] differentiate between benign and harmful data races, based on the influence on program correctness. Boehm [Boe11] argues against 'benign' races and considers that all data races are harmful at source code level. Memory models of mainstream imperative languages, such as C and Java, describe data races explicitly as undefined behavior. In this work we look for data races originated by source code defects, so we side with Boehm and aim for race-free programs.

A data race can be solved by transforming the racy accesses into atomic operations or protecting them with a common lock. These solutions do not guarantee that there is no higher-level failure, such as an order violation or an atomicity violation. A data race can be a symptom of any of these failures.

2.2.3. Atomicity Violation

An atomicity violation happens when a critical region is not executed atomically, and another thread concurrently executes a conflicting memory access. This is caused by the developer overlooking the need of explicitly enforcing atomicity for a set of instructions, e.g. with a lock.

A data race can be seen as an atomicity violation, i.e. the developer assumed atomicity of all memory instructions. Data races are dependent on the memory model of the language. In this work atomicity violations are only related to the intended semantics of the program, not the language. A race-freedom in a program does not imply absence of atomicity violations.

Figure 2.8 shows an example of a program with an atomicity violation. The intended operation is that both threads increment the value on `x`, but the thread `main` uses an intermediate variable. If the thread `worker` executes its critical section between the critical sections of `main`, its increment at line 22 will be overwritten by the assignment `x=temp` at line 14. Note that each access to the shared variable `x` is correctly protected, so the program is race-free. In this case the whole execution of the thread `main` needs to be atomic. The intended operation is not explicit in the program. Without user assistance it is difficult to define where an atomic region should start and end.

```
 1  int x = 0;
 2  mutex m;
 3  void main() {
 4      fork(worker);
 5      int temp;
 6
 7      lock(m);
 8      temp = x;
 9      unlock(m);
10
11      temp++;
12
13      lock(m);
14      x = temp;
15      unlock(m);
16
17      join(worker);
18  }
19
20  void worker() {
21      lock(m);
22      x++;
23      unlock(m);
24  }
```

Figure 2.8.: Atomicity violation example

2.2.4. Order Violation

An order violation is the execution of two sets of instructions in an unexpected order. Similar to atomicity violations, the developer overlooks the need to explicitly define an order through synchronization primitives, such as a signal-wait. Order violations are also dependent on the intended semantics on the program.

Figure 2.9 shows an example of an order violation. The thread `main` initializes a pointer `x` to a custom structure. The thread `worker` uses some content of the structure referenced by `x` as an argument for the function `compute`. If `worker` tries to dereference the pointer before its initialization, it will generate a runtime fault, e.g. a segmentation fault. The developer assumed that the structure would be initialized by the time `worker` uses the pointer. But there is nothing to ensure this behavior. The solution would be a signal-wait pair ordering both the initialization and the dereference.

```
my_struct *x;
void main() {
    fork(tid, worker);

    x = create_my_struct();

    join(tid);
}

void worker() {
    some_work();
    compute(x->value);
}
```

Figure 2.9.: Order violation example

2.3. Race Detection Algorithms

In this section we describe the two main algorithms for race detection, the *happens-before* algorithm and the *lockset* algorithm. We use both algorithms as filters to reduce the number of candidates during race detection in Section 5.5.2.

2.3.1. Happens-Before Algorithm

We define a race as two parallel accesses to the same variable, and two events occurring in parallel if the program does not define a specific order between them. Knowing the logical order of the events in a trace is useful for race detection.

Lamport's *happens-before* relationship [Lam78] establishes a partial order between events in a concurrent system. The events executed by a single thread are naturally happens-before ordered by their execution order. In a message-passing system, a message sending event a and its corresponding receiving event b by another thread are happens-before ordered; the message cannot arrive before being sent. In shared memory systems the ordering is defined by the causal relationships of the thread synchronization constructs, such as fork-start, end-join, and unlock-lock pairs. We define that event a happens before event b as $a \xrightarrow{hb} b$.

The happens-before relation has the following properties:

- Transitivity - $\forall\, a, b, c$ if $a \xrightarrow{hb} b$ and $b \xrightarrow{hb} c$, then $a \xrightarrow{hb} c$. The relationship determines that if for three events a, b, c, a happens before b and b before c, then a must happen before c.
- Irreflexivity - $\forall\, a, a \overset{hb}{\not\rightarrow} a$. An event cannot happen before itself.
- Antisymmetry - $\forall\, a, b$ $a \neq b$, if $a \xrightarrow{hb} b$ then $b \overset{hb}{\not\rightarrow} a$. If event a happens before another event b, then b does not happen before a.

Two events occur in parallel, $a \parallel b$, if they are not ordered, then based on the happens-before relationship:

$$a \parallel b \Leftrightarrow a \overset{hb}{\not\rightarrow} b \land b \overset{hb}{\not\rightarrow} a$$

Two events, such as two memory accesses, can happen concurrently if none of the two happens-before relationships between the two events is true, i.e. one relationship in each direction.

Vector clocks

The happens-before relationship can be computed using vector clocks. A vector clock is a logical timestamp of an event, similar to Lamport timestamps. The vector clocks of two events can be compared to determine the happens-before relation between the events.

A vector clock VC is a vector of length N, where N is the number of threads in the system. $VC(x)$ denotes the vector clock for event x. Each of the N positions of the vector is a clock, i.e. a counter that describes the last known state of a particular thread. We reference the position z of a particular vector clock as $VC(x)_z$. Each thread t maintains an internal vector clock VC^t which is updated with the following rules:

- Initially all positions of all thread vectors are initialized to zero: $VC^t_z = 0 \,\forall\, t, z$.
- Each time a thread t performs an event, it increases the counter of its own position on the clock array of its vector clock by one: $VC^t_t = VC^t_t + 1$.
- Each time a thread t sends a message, it sends its current vector clock along with the message m: $VC(m) = VC^t$.
- Each time a thread t receives a message m, it updates each position of its own vector clock with the maximum between its vector clock and the vector clock sent along with the message for that position:
  $VC^t_z = max(VC^t_z, VC(m)_z) \,\forall\, z$.

From the vector clocks, we can derive if there is a happens-before relationship between two events, a condition needed to avoid a data race. In general $a \xrightarrow{hb} b \Leftrightarrow VC(a) < VC(b)$, and by extension two events are parallel:

$$a \parallel b \Leftrightarrow VC(a) \not< VC(b) \wedge VC(b) \not< VC(a)$$

A vector clock is strictly smaller $<$ than another if:

$$VC(a) < VC(b) \Leftrightarrow VC(a)_z \leq VC(b)_z \,\forall\, z \wedge \exists\, y | VC(a)_y < VC(b)_y$$

A vector $VC(a)$ is strictly smaller than another vector $VC(b)$, if all the values in $VC(a)$ are equal or smaller than the corresponding in $VC(b)$ and at least one value is strictly smaller. Intuitively this means that, for an event a to happen before another event b, the known states at the execution of a cannot be bigger that any known state at the execution of b, and at least the state of one thread is smaller.

The interleaving in Figure 2.10 contains four write memory accesses on two distinct variables, `k` and `m`. The graph in Figure 2.11 represents the internal and inter-thread transitions in the interleaving with the corresponding vector clocks. In this example the only send-receiving message equivalents in multi-threading are fork-start and end-join synchronizations. We see the evolution of the different values of the vector clocks for the different internal and inter-thread events.

With the calculated vector clocks, we can check the happens-before relationship of the accesses to `k` and `m`. For `k` we have $VC(k = 1) = (1, 0)$ and $VC(k = 2) = (2, 1)$, where `k=1` was performed by `thread 1` and `k=2` by `thread 2`. As $VC(k = 1) < VC(k = 2)$, then by the previous definition `k=1` $\xrightarrow{hb}$ `k=2`, so both accesses are happens-before ordered and no race is possible. For `m` we have $VC(m = 1) = (3, 0)$ and $VC(m = 2) = (2, 2)$, where `m=1` was performed by `thread 1` and `m=2` by `thread 2`. In this case $VC(m = 1) \not< VC(m = 2)$ and also $VC(m = 2) \not< VC(m = 1)$, so there is no happens-before order in any direction. Without any happens-before ordering both events take place in parallel, which can lead to a data race.

thread 1	thread 2
`k = 1`	
`fork``(thread 2)`	
	`start`
	`k = 2`
	`m = 2`
	`end`
`m = 1`	
`join``(thread 2)`	

Figure 2.10.: Interleaving with a race

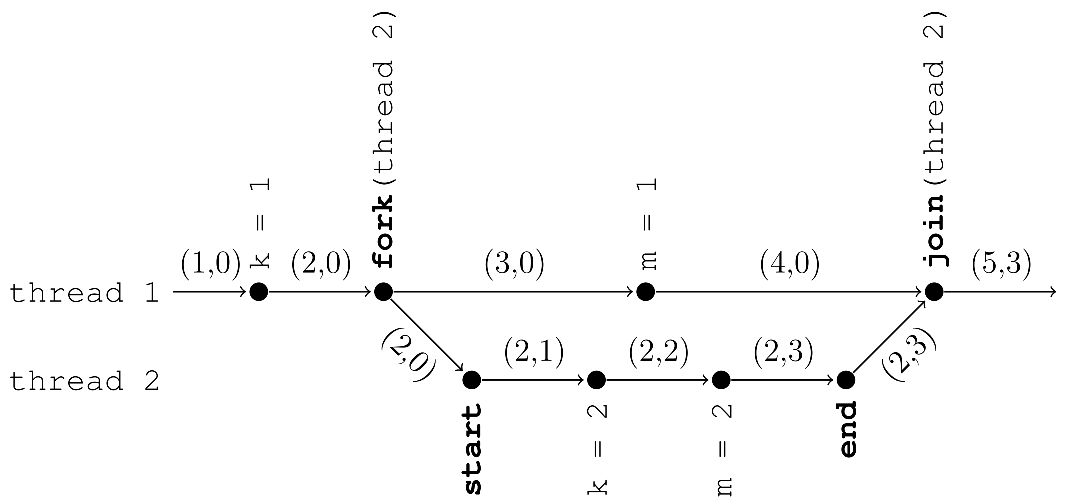


Figure 2.11.: Graph with vector clocks

2.3.2. Lockset Algorithm

The *lockset* algorithm, presented by Savage [SBN+97], assumes that all accesses to a shared variable v must always be protected by a common lock. This algorithm only pays attention to lock/unlock synchronizations and does not consider any other construct, which is a source of false positives in the algorithm. The lockset algorithm can be applied dynamically or statically without changes in the algorithm itself. The algorithm is show in Figure 2.12

```
Let $locks\_held_t$ be the set of locks held by thread $t$
for each shared variable $v$ do
    initialize $C(v)$ to the set of all locks
end for
for each access to variable $v$ by thread $t$ do
    $C(v) := C(v) \cap locks\_held_t$
    if $C(v) = \emptyset$ then
        issue a warning
    end if
end for
```

Figure 2.12.: Basic lockset algorithm

The algorithm maintains a set $C(v)$ of candidate common locks for each shared variable v. At the beginning it is unknown which the common lock should be, so $C(v)$ is initialized with all locks. Each thread t maintains a set $locks\_held_t$ with the locks that it holds during a specific instruction. For each access to v, the current thread updates $C(v)$, intersecting it with the current $locks\_held_t$. The intersection only reduces the candidate set, it cannot increase it. As long as the resulting $C(v)$ has at least one element, then all accesses seen until the current instruction share at least this lock. If the set $C(v)$ is empty, then the last access does not share a common lock with the previous accesses, and the algorithm emits a warning.

An example of the algorithm for the interleaving in Figure 2.13 is displayed in Figure 2.14. At the start $C(x)$ is composed of both locks `m` and `n`. The first to access `x` is `thread 1` with lock `m`. The update $C(x) := C(x) \cap \{m\}$ leaves the lock `m` as the content of $C(x)$. When `thread 2` accesses `x` with lock `n`, then $locks\_held(t_2) = \{n\}$; it is using a lock that is no longer part of $C(x)$. So $C(x) := C(x) \cap \{n\} = \{m\} \cap \{n\} = \emptyset$. With an empty $C(x)$, a warning is issued about the inconsistent access by `thread 2`.

There are further improvements of the algorithm to reduce the number of false positives. One improvement removes false positives, adding a state machine for each variable to differentiate the case of unprotected variable initialization. Another adds support to read-write locks, using an additional candidate set per variable to track the accesses under read-locks.

	thread 1	thread 2
1	**lock**(m)	
2	x++	
3	**unlock**(m)	
4		**lock**(n)
5		x++
6		**unlock**(n)

Figure 2.13.: Interleaving with inconsistent lockset

	Access to x	$locks\_held_{thread\ 1}$	$locks\_held_{thread\ 2}$	$C(x)$
0	-	$\emptyset$	$\emptyset$	$\{m, n\}$
2	read	$\{m\}$		$\{m\}$
2	write	$\{m\}$		$\{m\}$
5	read		$\{n\}$	$\emptyset$
5	write		$\{n\}$	$\emptyset$

Figure 2.14.: Update of candidate set $C(x)$

2.4. Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a formal system to describe concurrent systems and reason about them. CSP was first presented by Hoare in 1978 [Hoa78]; since then it has been expanded and studied. CSP has influenced hardware architectures, e.g. the Transputer T9000, and programming languages such as Go and Erlang. CSP is a process algebra, a mathematical theory that describes a system by the interactions performed, without enumerating its internal states. CSP is used to refine a system iteratively. The system is described with the same syntax in all the iterations, independently of the abstraction level. The formal semantics of CSP enable automatic checks on whether a detailed description follows the behavior of a more abstract description, i.e. if the detailed behavior refines the abstract behavior.

The following presents a brief introduction to the *blackboard* CSP syntax used in this work. Figure 2.15 depicts the relevant CSP grammar. The books by

$$
\begin{array}{lcl}
\langle \mathit{Process} \rangle & ::= & \mathit{STOP} \\
 & | & \mathit{SKIP} \\
 & | & \langle \mathit{event} \rangle \rightarrow \langle \mathit{Process} \rangle \\
 & | & \langle \mathit{Process} \rangle \; \Box \; \langle \mathit{Process} \rangle \\
 & | & \langle \mathit{Process} \rangle \setminus \langle \mathit{event\text{-}set} \rangle \\
 & | & \langle \mathit{Process} \rangle \upharpoonright \langle \mathit{event\text{-}set} \rangle \\
 & | & \langle \mathit{Process} \rangle \; ; \langle \mathit{Process} \rangle \\
 & | & \langle \mathit{Process} \rangle \; \Theta_{\langle event-set \rangle} \; \langle \mathit{Process} \rangle \\
 & | & \langle \mathit{Process} \rangle \; ||| \; \langle \mathit{Process} \rangle \\
 & | & \langle \mathit{Process} \rangle \underset{\langle event-set \rangle}{\parallel} \langle \mathit{Process} \rangle \\
 & | & \langle \mathit{Process} \rangle \; \triangle \; \langle \mathit{Process} \rangle \\
 & & \\
\langle \mathit{event} \rangle & ::= & \text{identifier } \langle \mathit{field} \rangle^* \\
 & & \\
\langle \mathit{field} \rangle & ::= & . \text{ identifier} \\
 & | & ! \text{ identifier} \\
 & | & ? \text{ identifier } [:\langle \mathit{event\text{-}set} \rangle] \\
 & & \\
\langle \mathit{event\text{-}set} \rangle & ::= & \{ \; \langle \mathit{event} \rangle^* \; \} \\
 & | & \{|\langle \mathit{event} \rangle|\}
\end{array}
$$

Figure 2.15.: Summary of CSP grammar

Roscoe [Ros10] and Schneider [Sch99] contain more in-depth descriptions of modern CSP.

In CSP the system and each subcomponent is described as a *process*. CSP is compositional; processes can be combined with different operators to generate more complex processes. Each process performs *events*, which are atomic and instantaneous. When a process emits or performs an event, we say that the process communicates with the environment, i.e. the event can be observed externally. Some operators enable communication between two processes. The communication is done by a synchronous share of an event (rendezvous), i.e. both processes must agree to execute the same event simultaneously. Processes are denoted in uppercase and events in lowercase.

For example, we want to define a CSP description of a vending machine. The process will be called $VENDING$. This machine will only provide a chocolate bar, the event *choc*, after receiving a coin, the event *coin*, afterwards the machine will stop working. It only provides a single bar during its lifetime.

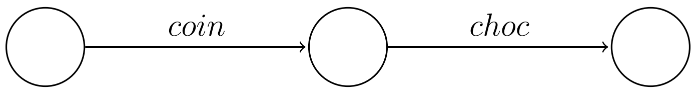
The description of this vending machine in CSP is as follows:

$$VENDING = coin \rightarrow choc \rightarrow STOP$$

At the beginning only the event *coin* can happen. Afterwards only the event *choc* is possible. After delivering the chocolate bar the process behaves like the process *STOP*.

The process *STOP* is a process that does nothing. In CSP terms it is *deadlocked*. It is the most basic process in CSP, the fixed point in the algebra.

The equivalent labeled transition system for $VENDING$ is as follows:



The process $VENDING$ can be composed as a series of processes:

$$VENDING = coin \rightarrow MIDDLE$$
$$MIDDLE = choc \rightarrow STOP$$

The prefix operator $\rightarrow$ composes an event a and a process P into a new process Q, where $Q = a \rightarrow P$. Q is a process that performs a and then behaves like P.

The process $VENDING\_REC$ describes a vending machine that never stops. It always serves a chocolate bar for a coin.

$$VENDING\_REC = coin \rightarrow choc \rightarrow VENDING\_REC$$

This process uses recursion. Instead of terminating with *STOP*, it returns to the beginning and behaves like $VENDING\_REC$ again, offering the event *coin*.

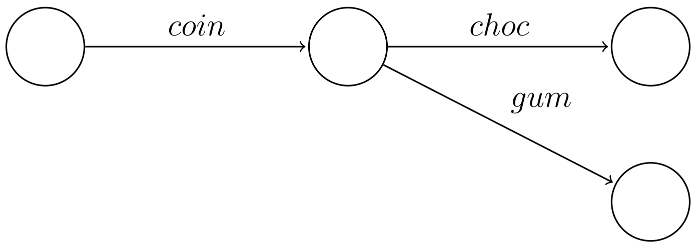
The set of events emitted by a process P is the alphabet of the process, and it is denoted as $\alpha(P)$. The set of all events in a system is denoted Σ. In our example $\Sigma = \{coin, choc\}$.

2.4.1. Sequential Operators

Branching or letting the environment choose between two processes, is done with the external choice operator $\Box$. $VENDING\_GUM$ is a new vending machine that offers a chocolate bar or a chewing gum, the event *gum*, for a coin and then stops.

$$\begin{aligned} VENDING\_GUM = coin \rightarrow (&choc \rightarrow STOP \\ &\Box\, gum \rightarrow STOP) \end{aligned}$$

The equivalent labeled transition system for $VENDING\_GUM$ is:



The external choice operator $\Box$ composes two processes P and R into a new process Q, where $Q = P \Box R$. Q is a process that behaves like P or behaves like R. The process Q does not decide which branch to take; it offers the events of P and R and lets the environment choose.

The choice between two processes can be dependent on the result of a Boolean predicate b. The process $P \lhd b \rhd R$ behaves like P if the predicate b is true, otherwise it behaves like R.

We can abstract a process, making some events non-observable with the hiding operator $\backslash$. The process $VENDING\_CHOC$ builds upon the process $VENDING$, which conserves its internal behavior, but only shows us the event *choc*.

$$VENDING\_CHOC = VENDING \setminus \{coin\}$$

The hiding operator $P \setminus X$ creates a new process from P, where the new process shows no events of the set X. The projection operator $P \upharpoonright X$ is a shortcut for hiding all events except those specified in the set X, so $P \upharpoonright X \equiv P \setminus (\Sigma - X)$

Processes can be directly concatenated with the sequential operator ; . In $Q = P \,;\, S$, Q behaves like P, and if P terminates successfully it behaves like

S. Successful termination is implemented with the pre-defined CSP process $SKIP$. $SKIP$ derives from $STOP$, and is defined as $SKIP = \checkmark \rightarrow STOP$. The event $\checkmark$ is a special event that represents success.

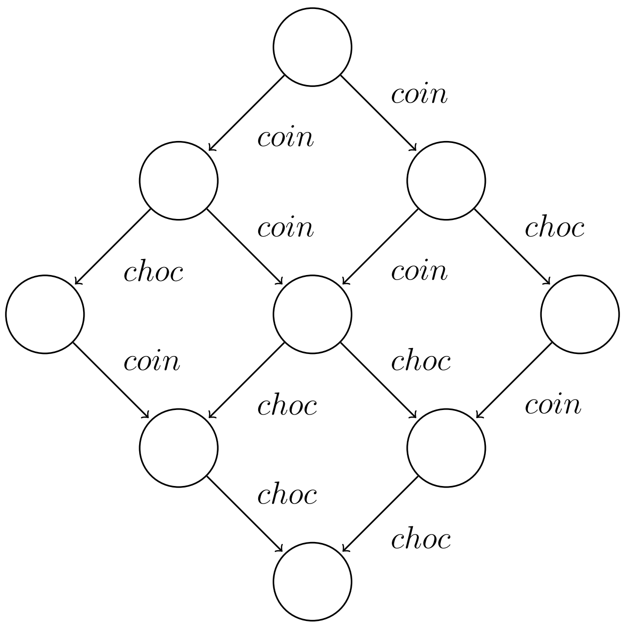
The exception operator Θ_x, changes the behavior of a process after executing an event in set x. In $Q = P \Theta_x R$ the process Q behaves initially as P. After an event in set x is performed by P, Q changes its behavior from P to R.

2.4.2. Concurrency Operators

Processes can operate concurrently and communicate with several different operators. Non-communicating concurrency is the simplest combination, and is expressed with the interleaving operator $|||$. The process $VENDING\_2$ represents the common view of two independent vending machines, each operating concurrently as in $VENDING$.

$$VENDING\_2 = VENDING \mid\mid\mid VENDING$$

The global view includes all possible combinations of events of the composing processes. Each composing process follows its internal behavior independently, no chocolate bar is offered before the correspondent coin. The equivalent labeled transition system representing all the possible interleavings of $VENDING\_2$ is:



Manual enumeration of all states can be extremely demanding. With a process algebra the user only needs to identify the transitions of the components, without worrying about individual states.

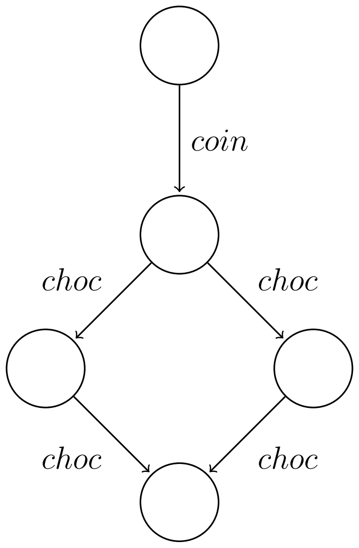
The interleaving operator $|||$ combines two processes, P and R, into a new process Q, where $Q = P \; ||| \; R$. Q is a process that executes P and R concurrently and independently.

The interrupt operator $\triangle$, as in $Q = P \triangle R$, is similar to the interleaving operator. Q behaves initially as $P \; ||| \; R$, but after any event of R is performed, P is stopped and Q continues its execution as $Q = R$. With the interrupt operator P and R are never really executed concurrently, because at the moment an event in R is executed, P stops running. The interrupt operator is also similar to the exception operator Θ.

Concurrency with communication is established with the interface parallel operator $\underset{x}{\|}$, where x is the set of events with which the involved processes must communicate. If the event set x is empty, then $\underset{\{\}}{\|}$ is the same as $|||$, the processes do not communicate but run concurrently. To execute any event of set x, both processes must be in a state willing to execute it. When the shared event is executed, both composing processes advance their internal state accordingly. The process $VENDING\_2SHARED$ represents the common view of two vending machines, each as in $VENDING$, operating concurrently, similar to $VENDING\_2$. In this case the machines share the coin slot, and both will give a chocolate after a single coin is introduced.

$$VENDING\_2SHARED = VENDING \underset{\{coin\}}{\|} VENDING$$

A single coin event is enough to trigger both chocolate events, which will happen concurrently. The equivalent labeled transition system representing all the interleavings of $VENDING\_2SHARED$ is:



The interface operator $\underset{X}{\|}$ combines two processes, P and R, into a new process Q, where $Q = P \underset{X}{\|} R$. Q is a process that concurrently executes P and R. When P or Q wants to perform any event of the set X, the other process, Q or P, must also be able to perform this event. Both processes synchronize on the event, they perform the event together, and continue their executions concurrently. The interface operator is the main building block in CSP to compose concurrent processes under communication.

Another example is the process $INTERACTION$, which combines a process $PERSON$, representing a buyer, and the vending machine $VENDING$.

$$INTERACTION = PERSON \underset{\{coin, choc\}}{\|} VENDING$$
$$PERSON = choc \rightarrow STOP$$

The buyer wants the chocolate bar, but does not give a coin for it. Both processes need to be able to perform the events *coin* and *choc* simultaneously, as denoted by the event set in the parallel operator. While $VENDING$ is willing to perform the *coin* event, $PERSON$ is not. The whole system is *deadlocked* and is equivalent to $STOP$, meaning that it does not perform any event.

2.4.3. Expanding Event and Process Definitions

Event semantics can be expanded by giving structure to the events. Events can be structured by concatenating identifiers with dots. An event, $coin.10$, is composed of two identifiers, the first one, *coin* is called the channel, whereas 10 is called field. This dotted structure can be interpreted as the communication of the value 10 on channel *coin*. Multiple fields can be concatenated with more dots, e.g. $x.y.z$.

Instead of a dot we can use an exclamation mark ! to denote an output in the channel, or a question mark ? for an input. For example, $coin!10$ sends the value 10 to the channel *coin*, while $coin?x$ accepts any value, such as 10 or 20, on the channel *coin* and binds it to the identifier x. The use of ! or ? instead of a dot does not alter the synchronous nature of event communication in CSP. An output mark has the same formal meaning as using a dot, it is only a help for the user. The input mark is the equivalent to writing an external choice with all possible values for the event. The input mark can be restricted to a set of events, e.g. $coin?x : \{10, 20\} \rightarrow P$ where the only valid bindings value for x are 10 or 20. It is equivalent to the external choice: $coin.10 \rightarrow P \,\square\, coin.20 \rightarrow P$. The use of dotted events, channels, and restricted inputs are shortcuts to have multiple individual events represented as only one. Channels are also useful to easily generate sets of events. The expansion set $\{|coin|\}$ represents all events that can take place on the channel *coin*.

A process can be parameterized as P_i or $P(i)$, and its arguments used by its events or processes:

$$P_i = count.i \rightarrow P_{i+1}$$

P_i is a process that emits the value of an ever increasing counter, represented by i, in the channel *count*. Parameterized processes can also be used with pattern matching. We can define multiple processes with the same name and different values in the arguments, and the most fitting process is chosen at runtime. The following example demonstrates this process:

$$\begin{aligned} P_i &= count.i \rightarrow P_{i+1} \\ P_{10} &= goal \rightarrow STOP \end{aligned}$$

When an iteration of P_i should be executed for i=10, the process P_{10} is executed instead, emitting the *goal* event and stopping. In this example the event *goal* can only be reached if the initial value of i is smaller than 10. If the counter is bigger than 10, the counter runs forever and the event *goal* is never reached.

2.4.4. Semantic Models

CSP has multiple formal semantics that provide meaning to a CSP expression. CSP has defined multiple denotational, operational and algebraic semantics.

In this work we use two denotational models: the trace model and the failures model.

Trace model

The trace model describes the set of sequences of events that a process is able to perform. A sequence of events is called a trace, which is represented as a sequence of CSP events enclosed by $\langle\rangle$. The set of traces for process P is $traces(P)$. For example, for the process $VENDING$:

$$traces(VENDING) = \{\langle\rangle, \langle coin\rangle, \langle coin, choc\rangle\}$$

$VENDING$ has a finite set of traces. It can exhibit the empty trace $\langle\rangle$ just at the beginning, the trace $\langle coin\rangle$ after the first event, and the trace $\langle coin, choc\rangle$ after all the events have been executed. The process $STOP$ only contains the empty trace: $traces(STOP) = \{\langle\rangle\}$. The recursive process $VENDING\_REC$ contains an infinite set of traces:

$$\begin{aligned} traces(VENDING\_REC) = \{&\langle\rangle, \langle coin\rangle, \langle coin, choc\rangle, \langle coin, choc, coin\rangle,\\ &\langle coin, choc, coin, choc\rangle, \langle coin, choc, coin, choc, coin\rangle, ...\}\end{aligned}$$

It is not possible to determine if the process is able to perform additional events or not from a concrete trace in the trace model. We take the following processes P and Q as an example:

$$\begin{aligned} P &= a \rightarrow b \rightarrow c \rightarrow STOP\\ Q &= a \rightarrow b \rightarrow STOP\end{aligned}$$

The trace $\langle a, b\rangle$ is contained in both $traces(P)$ and $traces(Q)$. After this trace Q cannot perform any other event, while P may. We cannot distinguish P from Q using this trace. To determine if a process is deadlocked, in CSP terms, after a trace we need more information, such as which events can or cannot be performed. We need a denotational semantic model that includes more information than the trace model. The failures model provides us with this additional information.

Failures model

The failures model is an extension of the trace model. This model includes for each trace the set of events that the process cannot perform, called the refusal set. The set $failures(P)$ for the process P is a set of tuples, where each tuple is a pair composed of a trace of P and the corresponding refusal set. All the traces in the set $traces(P)$ are also present in $failures(P)$. For example, the $VENDING$ process has the following failures set:

$$\begin{aligned} failures(VENDING) = \{&(\langle\rangle, \{choc\}), (\langle coin\rangle, \{coin\}), \\ &(\langle coin, choc\rangle, \{coin, choc\})\} \end{aligned}$$

At the beginning, $VENDING$ cannot perform *choc*, only *coin* is allowed. After $\langle coin\rangle$, only *choc* is possible. Finally, after $\langle coin, choc\rangle$ the process cannot perform any other event, so the refusal set contains both *coin* and *choc*. For the recursive process $VENDING\_REC$ the infinite failure set is as follows:

$$\begin{aligned} failures(VENDING\_REC) = \{&(\langle\rangle, \{choc\}), (\langle coin\rangle, \{coin\}), \\ &(\langle coin, choc\rangle, \{choc\}), (\langle coin, choc, coin\rangle, \{coin\})...\} \end{aligned}$$

The previous processes P and Q were indistinguishable after the trace $\langle a, b\rangle$. Now with the failures model, that is with $failures(P)$ and $failures(Q)$, we can see that both processes are different after this trace. For P the refusal set is $\{a, b\}$ and for Q is $\{a, b, c\}$. We see that after the trace $\langle a, b\rangle$ the process P is willing to perform the event c, but Q is not, because c is not in the refusal set of P after this trace. And then P and Q are not equivalent after the trace $\langle a, b\rangle$.

We can use the failures model to detect whether a process is deadlocked after a concrete trace. If the corresponding refusal set contains all events possible in the systems, then the process is deadlocked after that trace. In other words, if the refusal set is equal to Σ, the set of all possible events in the system, then the process is not able to perform any kind of event, which is exactly the definition of deadlock in CSP. For example, after the trace $\langle a, b\rangle$ the process P is not deadlocked, because the corresponding refusal set does not contain all the events in the system, that is $\{a, b\} \neq \Sigma$ while here $\Sigma = \{a, b, c\}$. On the other hand the process Q is deadlocked because its refusal set for $\langle a, b\rangle$ contains all events in the system, $\{a, b, c\} = \Sigma$.

2.4.5. Refinement

CSP is used to describe systems with different levels of abstraction, using the same syntax. It is possible to describe a high-level process representing the specification of the system, and iteratively create a more detailed version of it until the desired implementation level is reached. This process is called refinement. CSP allows automatic checking whether a process is a refinement of another process, i.e. whether the detailed process exhibits the same behavior as the abstract one. A refinement relationship $S \sqsubseteq I$, between a specification process S and a more detailed implementation process I, holds if and only if $\mathit{behavior}(I) \subseteq \mathit{behavior}(S)$. All observable behaviors of I are possible behaviors in the more abstract process S. The behavior of a process can be any semantic model, such as the trace or failures model. For example, a trace-refinement, using the trace model, is defined as follows:

$$S \sqsubseteq_{\mathrm{T}} I \Leftrightarrow \mathit{traces}(I) \subseteq \mathit{traces}(S)$$

A concrete example of two processes and a refinement relationship is the following:

$$\begin{aligned} VENDING &= \mathit{coin} \rightarrow \mathit{choc} \rightarrow STOP \\ VENDING\_REC &= \mathit{coin} \rightarrow \mathit{choc} \rightarrow VENDING\_REC \\ VENDING\_REC &\sqsubseteq_{\mathrm{T}} VENDING \end{aligned}$$

We can say that $VENDING\_REC$ is refined by $VENDING$. The three traces contained in the set $\mathit{traces}(VENDING)$ are also present in the set $\mathit{traces}(VENDING\_REC)$:

$$\mathit{traces}(VENDING) \subset \mathit{traces}(VENDING\_REC)$$

However, the opposite refinement does not hold. $VENDING\_REC$ contains many traces not possible in $VENDING$, such as $\langle \mathit{coin}, \mathit{choc}, \mathit{coin} \rangle$.

$$VENDING \not\sqsubseteq_{\mathrm{T}} VENDING\_REC$$

If we change the semantic model, i.e. the criteria to compare behaviors, the refinement result can change. As the failures mode is an extension of the trace model, failures-refinement implies trace-refinement, but not the opposite. For example, where $VENDING$ trace-refines the process $VENDING\_REC$, for a failures-refinement, it is no longer true:

$$\begin{aligned} VENDING\_REC &\not\sqsubseteq_{\mathrm{F}} VENDING \\ \mathit{failures}(VENDING) &\not\subseteq \mathit{failures}(VENDING\_REC) \end{aligned}$$

The trace-refusal pair $(\langle \mathit{coin}, \mathit{choc} \rangle, \{\mathit{coin}, \mathit{choc}\})$ of the process $VENDING$ is not contained in $\mathit{failures}(VENDING\_REC)$. After the trace $\langle \mathit{coin}, \mathit{choc} \rangle$, $VENDING\_REC$ cannot perform the event coin.

The refinement relationship is the main instrument in CSP to compare two processes. The semantic model chosen defines which information is compared.

2.4.6. Practicalities in Model Checking

The Failures-Divergences Refinement Checker FDR3 from Oxford University [GRABR14] is the main tool to check refinement relationships in CSP. FDR3 takes a description of CSP processes in the machine-readable format CSP_M along a set of refinement assertions. [1] Each assertion is a refinement relationship between two processes under a semantic model, such as the trace or failures models. FDR3 explores the involved processes and checks if the refinement relationship holds. If the relationship does not hold, then it provides a counterexample. The counterexample is the trace performed by the involved processes and the refusal set in the failures model. For composed processes it is possible to observe the internal behavior of the component processes.

One of the main issues in model checking is the number of implicit states in the system. As the number of states grows, more time and resources are needed. This is a general issue in model checking called the *state explosion problem*. To alleviate this problem we have to design our CSP models so they have as few states as possible. We follow several general strategies in this work:

- Minimize the number of events in the model. We do not insert superfluous events in the model if they are not needed for the current refinement check. For example, in data races the memory accesses to y are not needed when checking races on variable x.
- Limit the number of alternatives in choice operators or input fields. For example, if a value of an input field can only be a 0 or a 1 in the whole system, it is better to define the valid set of values than to allow any integer value.
- The hiding operator makes events invisible for a new abstract level. These events cannot cause new interactions and are confined to the internal behavior of the process. FDR can apply partial order reduction algorithms to these internal events, thus reducing the state space.

2.5. Aspect-Oriented Programming

In Chapter 6 we extend our tool so it can verify properties defined by the user. The extension requires the user defining custom events in the target program. To facilitate and automate the task of capturing the relevant events,

[1] A brief introduction to CSP_M, along with the CSP_M version of the examples in this work is available in Appendix A.

we developed an Aspect-Oriented Programming framework. In this section we introduce Aspect-Oriented Programming and its associated concepts.

Aspect-Oriented Programming (AOP) is a programming paradigm that addresses the problem of *cross-cutting concerns* and how they affect the modularity of a system. A cross-cutting concern is code that is repeated and scattered through diverse modules and cannot be isolated in a single module, such as logging and authentication systems. These modules interact and introduce dependencies in multiple parts of the code and 'pollute' the *core concern*, i.e. the code of the main functionality of the program.

AOP solves this problem by defining these cross-cutting modules as an independent set of functions, and describing at which points of the main program these functions are inserted. The code from the cross-cutting concern is isolated from the code of the core concern. For example, for a logging module, the logging code includes the functions that emit a message to the log file. These logging functions are inserted automatically into other functions of the program.

An AOP system requires a join-point model which defines three components:

- *Join-points* - the points in the target language where an *advice* can be inserted, such as function calls or accesses to global variables.
- *Pointcuts* - a set of concrete *join-points* in the target program. For example, the concrete name of a function to be logged.
- *Advice* - pieces of code that run at a *join-point.* In the logging module, this would be the actual code that emits the messages to a file.

An advice can be inserted at a pointcut in two ways: weaving it automatically into the main program during compilation, or dynamically intercepting the pointcut at runtime with a call to the correspondent advice. The last option is usually done in languages executed by an interpreter.

2.6. LLVM

The LLVM project [LA04] is an infrastructure that provides reusable modules to build compilers and toolchains. LLVM works as a common backend for multiple compilers, such as C/C++, Java bytecode, or Haskell. The compilers translate the source code to the LLVM intermediate representation (IR), which is language agnostic.

The IR is a RISC like assembler language with a strong type system, and each instruction is in static single assignment form (SSA). SSA is a refinement of a three-address code, a combination of assignment and binary operation, where each variable is assigned exactly once and afterwards becomes immutable. SSA simplifies dependency analyses, e.g. use-def chains become explicit as they have a single element. An example of the IR in human-readable format is shown in Figure 2.16; the program is the translation of a ”Hello world” program originally written in C.

```
@.str = private constant [14 x i8] c"hello,␣world\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main(i32 %argc, i8** %argv) {
entry:
    %1 = getelementptr [14 x i8]* @.str, i32 0, i32 0
    %2 = call i32 (i8*, ...)* @printf( i8* %1 )
    ret i32 0
}
```

Figure 2.16.: ”Hello world” in LLVM IR

Note that external calls, like `printf` in the example, are dependent on the libraries used by the original program. Any analysis based on external calls is language-dependent.

Code optimizations, such as loop normalization, hoisting, or vectorization need to be implemented only once for the IR. All languages supported by LLVM can benefit from these optimizations. Each IR optimization is a LLVM Pass. Passes can be independent or depend on the result of previous passes.

In this work we target C/C++ programs using the POSIX threads library. We perform program static analyses and instrumentation in LLVM IR as LLVM Passes. As synchronization primitives are implemented through the external POSIX library, additional support is needed to apply the current work to other languages/libraries.

3. Related Work

In this chapter we review related work. The reviewed approaches make different compromises between precision (false positives), recall (false negatives), scalability, reproducibility, and input requirements such as annotated source code. We describe the main tools and algorithms related to data race and deadlock detection. Furthermore, we also explain approaches that enable the definition of error patterns that are tied to the domain of the program. The characteristics of each method are also discussed, as well as how they relate to our approach.

3.1. Race Detection

3.1.1. Static Analysis

Race detection can be performed at source code level using static analysis. Static analysis is fundamentally limited, finding all data races is undecidable; Miller describes it as an instance of the halting problem[NM92]. Race detection requires knowing if at least two concurrent instructions operate on the same address and whether one is a write. Not all conditions can be decided with total precision without scalability limits. Aliasing, if two instructions operate on the same address, is a classical and general problem in compiler theory. Aliasing is undecidable, as stated by Ramalingam [Ram94], and is a source of false positives.

Another source of false positives is the algorithms that detect if instructions may be concurrent. May-Happen-in-Parallel (MHP) analyses are a family of algorithms that search pairs of concurrent instructions. Barik [Bar06] presents a context and flow sensitive MHP algorithm to check if two Java threads may happen in parallel, synchronization constructs are not considered. Other MHP algorithms are restricted to some forms of parallelism, such as Sankar [SN16] for async-finish parallelism in X10 languages. Bouajjani[BMT$^+$05] proposed dynamic pushdown networks (DPNs), an abstract model of multithreaded programs based on pushdown automata. DPNs are used to compute reachability in a parallel program, as a race would be two conflicting memory accesses. These algorithms have rarely been used for race detection, as they either have limited scalability, or do not support the programming model (or synchronization constructs) of mainstream imperative languages.

A commonly used heuristic is the assumption that instructions without a shared lock can be executed concurrently, i.e. have disjoint locksets. RacerX [EA03] uses a lockset analysis with an inter-procedural flow analysis. Relay [VJL07] computes a relative lockset for each function independently, using data-flow analysis. Afterwards the relative locksets are aggregated. The approach enables the parallelization of the analysis and scale to 4.5 million lines. Locksmith [PFH11] uses a similar technique of summarization as Relay, but also performs a global data-flow analysis. The global analysis improves the precision of the tool, but limits its scalability to 20000 lines of code. Lockset approaches ignore any kind of synchronization that is not based on mutexes.

Another approach to avoid or reduce the previous problems is the use of code annotations, such as in Warlock [Ste93]. The developer must provide additional information about expected locks, concurrent sets, or alias information. Tools that rely heavily on specific annotations are disregarded by developers, as they impose high adoption costs and the user can commit mistakes in the annotations.

Static analysis tools are conservative, in that they produce many false warnings without missing any errors. A high number of false positives causes developers to lose interest, as they have to spend too much time triaging false warnings. In our work, we want to avoid the generation of false positives and also obtain a scheduling that leads to the error, so we focus on a dynamic analysis of the program to avoid the ambiguities of static analysis. However, static analysis can be used as a filter to remove race candidates before running a more precise detector, as initially all memory accesses are candidates.

3.1.2. Dynamic Analysis

A dynamic analysis tool executes a given program under a specific workload and analyzes the executed instruction stream. Observing the real behavior of the program allows the analysis to overcome the aliasing problem of static approaches, as real memory addresses and values are observed. Dynamic tools instrument the program under test to observe the relevant events. It can be done at source code level (program to program transformation), binary level (with a binary level instrumentation framework like Valgrind [NS07]) or with the help of specialized hardware, as in HARD [ZT07]. The analysis can be executed on-line, simultaneously with program execution, or off-line, with subsequent analysis of a trace. Both approaches impose different overheads on program execution: CPU, memory, IO or storage.

On-Line Approaches

Eraser [SBN+97] originally presented the *lockset* algorithm, and implemented it in a dynamic fashion. It monitors all shared memory accesses on-line and lock acquisitions and releases. The algorithm checks the locking discipline; a memory location must always be protected by the same lock. Eraser generates false warnings as it is only aware of locks; other mechanisms such as fork-join and signal-wait are completely ignored.

Using Lamport's *happens-before* relationship [Lam78], race detectors, such as DJIT [ISZBM99], Helgrind [Val07], and ThreadSanitizer [SI09], build a partial order graph of the memory accesses on-line. If two memory accesses are not ordered in the graph, then they could take place in parallel (in the observed execution or a similar one) and produce a data race. The happens-before algorithm allows the inclusion of non-lock primitives; fork-join, signal-wait or barriers can produce happens-before edges in the graph. The algorithm is implemented with the tracking of the vector clock at each memory access. Implementations concerned with scalability and performance have specific optimizations; FastTrack [FF09] introduces fewer vector clocks and uses additional timestamps, while RaceTrack [YRC05] dynamically changes the granularity of the detection at the cost of less precise reports. Although the on-line computation of the happens-before order produces precise reports, it costs more than the use of the lockset algorithm, because vector clocks require more space and take more time than locksets.

The happens-before algorithm is sensitive to the order of locking primitives. Critical sections can often commute, which is not considered by the algorithm. The happens-before ordering is propagated between two critical sections, and false negatives arise as seen in Figure 1.6. The algorithm can be weakened to not track the happens-before edges of lock and unlocking operations, trading the false negatives for false positives. Smaragdakis et al. [SES+12] presented another variation of happens-before: the *causally-precedes* relationship. It ignores the happens-before edges of two critical sections when they are conflict free, i.e. do not share a write on the same variable. Causally-precedes does not produce false positives, analyzes more interleavings than happens-before, and can be evaluated in polynomial time. Another variant is hybrid approaches, such as MultiRace [PS07] or Helgrind$^+$ [JT08]. They combine the happens-before relation with the tracking of the locking discipline. They explore more schedulings because of the lockset agnosticism to critical sections, but similar to what happens with the lockset, they cannot reconstruct an interleaving that leads to the erroneous state.

Dynamic approaches are sensitive to execution order. Multiple re-executions can generate different warnings or no warnings at all. This sensitivity also increases the difficulty in reproducing the failures, due to the non-deterministic decisions of the scheduler. The developer may not be able to reproduce a similar state to understand which events led to a race. With the more relaxed models, such as lockset or causally-precedes, the order in which the interleaving happens is lost and the trace cannot be reproduced.

In RaceQuest we explore multiple different interleavings, i.e. fewer false negatives, and conserve the order of synchronization events. We also use a weakened happens-before and lockset algorithms as filters to discard some error candidates from the trace before generating our CSP model. This filtering reduces the number of events in the trace and generates a smaller CSP model.

Off-Line Approaches

Off-line, or post-mortem approaches generate a trace of a single execution of the target program, usually perform some analysis on the trace, and enable the replay of the program. The tools provide different levels of granularity. They may only store and enforce the order of synchronization operations, or ensure that all memory accesses of all threads occur in exactly the same order. Fine granularity is more precise at the cost of higher logging overhead and trace size, as in InstantReplay [LMC87]. With the reproduction the developer is able to use a debugger or other common tools to observe the program behavior, with higher consistency between executions than with dynamic on-line tools. RecPlay [RB99] is an example with coarse granularity. During the recording it only stores the synchronization primitives; later it detects races using a happens-before analysis.

Plain record and replay systems only observe a single program interleaving. Predictive trace analyses take a recorded execution and build an abstract causal model that encompasses not only the observed but also alternative interleavings, including reorderings of synchronization events not considered by happens-before. The presented work belongs to this family of tools.

The approach of Said et al. [SWYS11] takes a trace with all synchronization and shared memory accesses, including the read and written values, and encodes the trace as a satisfiability problem. Each event is considered a single variable in the formulas, with additional constraints to represent dependencies between the operations, including semantics of synchronization operations. Read-write consistency is also included; the formulas encode that a read operation must happen after a write operation on the same variable with the same value. Then a data race between two concrete events is an additional formula where two events must happen one after the other, without any other event in-between. This formula is added to the system, and the whole set of satisfiability equations is fed to an SMT solver. If the equation system is satisfiable, then a total order of the events that produce the data race exists. In this case it can be retrieved and used to realize a fine-grained reproduction of the trace (although this has not been implemented). The model presented is conservative and does not generate false positives, so warnings are real and the counterexamples are feasible.

Huang et al. [HMR14] extend the work of Said in a tool called RVPredict. RVPredict takes branch instructions into account in the trace and the model. The read-write consistency is reduced to only instructions that affect branch conditions. For each branch a backward slice is computed. These extensions generalize the Said et al. model, by containing a higher number of alternative interleavings. Huang et al. prove that their extended model is maximal – it encompasses all feasible interleavings from the trace for their criteria.

Said and Huang's models pursue a sound analysis, and obtain no false positives. To accomplish this objective, the trace must contain all shared memory accesses. This limits the application of static analysis, in order to reduce the amount of instrumentation and the number of traced events. The models also need the read and written values for all memory operations, further increasing the size of the trace. Both models are designed for Java programs and are more difficult to implement for C/C++ programs. In Java each element allocated in memory cannot be accessed without a known reference and is accessed as a whole, but in C/C++ each byte in program memory can be accessed at any time with pointer arithmetic. The number of constraints in the model increases as memory locations need to be split into individual bytes. A single constraint for an integer would be transformed into four different constraints with the corresponding part of the data for each byte, as each byte can be addressed afterwards by different memory operations. The additional number of constraints increases the workload of the solver and reduces scalability. Also, C has a weaker type system than Java, limiting the effectiveness of alias and static analysis, and more memory accesses need to be instrumented and stored. Said and Huang's evaluations do not take the costs of generating the trace into consideration, but only the detection costs. The present work minimizes the number of memory events logged with static and dynamic algorithms. Our modeling step can generate false positives, so we execute the program under test with the resulting interleavings to prune false positives. Additionally, Said and Huang's models only consider mutexes and signal-wait constructs, and do not model the semantics of other constructs. We explicitly model barriers, semaphores, and read-write mutexes to further reduce the number of false negatives.

3.1.3. Influencing the Scheduler

As dynamic tools depend on the observed interleaving, some techniques build upon enforcing different interleavings. They execute the program multiple times modifying the decisions taken by the scheduler in each execution, and simultaneously perform error detection with an available algorithm.

CalFuzzer [JNPS09] is a framework to find schedules in Java programs. It uses the output of another static analysis tool about an error, such as a data race or deadlock. It instruments the program accordingly to the type of the error, and executes it under a biased random scheduler. The scheduler takes random decisions, but tries to direct the program to a state where the error would occur, for example by enforcing thread scheduling preemption points just before a possibly racy access.

Systematic concurrency testing tools, such as CHESS [MQB07] and Inspect [YCG08], go one step further and do not rely on a random scheduler. They re-execute the program multiple times, each with the same workload but a different thread scheduling. They systematically explore all possible interleavings without repeating any. But as the number of interleavings grows exponentially, they use several techniques to reduce the combinatorial explosion. One technique is to not allow preemptions on each instruction, but only after a determined number of instructions; this number is increased after all possible interleavings have been explored. Another technique is iterative schedule bounding: the number of actual executed preemptions in a single execution is limited. This limit is also increased iteratively after exhausting the current search space. Dynamic partial order reduction techniques look for concurrent and independent instructions during runtime and avoid exploring interleavings that only commute these instructions, as the program state is the same for all these interleavings.

Tools that influence the scheduler can achieve a better coverage than multiple executions of other dynamic on-line tools. These tools can also explore more interleavings than off-line approaches, as they can see program paths not available in a trace. In practice, these tools do not scale well with the size of a program. All potential interleavings cannot be explored, as the number grows exponentially, and the tool has to remember which interleavings have already been explored. The exploration of an interleaving implies the execution of the whole program, so the costs of sequential parts of code are paid each time. Dynamic algorithms after one execution analyze an abstract model, which is faster to explore than re-execution of the program for each interleaving.

3.2. Deadlock Detection

Deadlock detection due to lock acquisition is a field intertwined with data race detection. Most race detection tools that track the lockset also perform deadlock detection: RacerX [EA03] does it statically, while Eraser [SBN$^+$97], CHESS [MQB07], or ThreadSanitizer [SI09] do it dynamically. In addition to locksets, they track the order in which locks are acquired. If two locks are ever acquired in different orders, they report that as a warning of a possible deadlock. This approach generates false positives when the locks are acquired in different orders, e.g. in parts of the program that are not concurrent.

Most deadlock detectors, such as the work of Bensalem and Havelund [BH02], or Dreadlocks [KH08], monitor the program and detect if the application deadlocks at runtime. These tools build a wait-for graph; they annotate for each lock which thread holds it, and which thread waits for it; if there is a cycle in the graph then a deadlock exists. The cycle indicates in which order the locks were acquired. Some tools use the cycle to guide the reproduction of the deadlock.

There are other static approaches. Boyapati [BLR02] presents a type system that guarantees deadlock freedom. Naik et al. [NPSG09] define six conditions for deadlock freedom, and check each condition with a different set of static analyses.

The most similar approach to our work is ConLock [CWC14], which performs a constraint-based abstraction of a program trace to predict lock cycles. Afterwards ConLock guides the scheduler towards the suspicious deadlock. ConLock also focuses on minimizing the number of scheduling points during reproduction. In contrast to our work, ConLock only searches for lock cycles and does not support deadlock prediction for other synchronization constructs, such as barriers or semaphores.

3.3. Runtime Verification

There are specific algorithms to detect clearly defined concurrency failures, such as data races and deadlocks. In dynamic analysis these algorithms work as monitors that respond with an error report. When the failure is domain-specific, no general monitor exists. Runtime Verification is a field that studies the development and implementation of custom runtime monitors, first introduced by Lee [LKK$^+$99], Kim [KVBA$^+$99], and Havelund [Hav00]. It is used for multiple purposes: testing, verification, validation, fault protection, profiling, policy monitoring, and for adaptive systems.

The monitor definition is composed of the events to observe and the property to validate or refute. The events link the point during the program execution and the property. The events are usually defined and extracted through instrumentation with Aspect-Oriented Programming. A *pointcut* specification names the different events and ties them to points in the program, such as all functions with a name that starts with `print`. The program is automatically instrumented to generate the desired events and feed them to the monitor. The property itself in the monitor is usually defined as a combination of defined events through formalisms, such as state machines, linear temporal logic, or regular expressions.

Most runtime verification tools examine only the actual execution or trace. With additional information, such as vector clocks to establish happens-before orderings, some tools predict failures in alternative reorderings, such as the work by Sen et al. [SOA08] for general assertions in SystemC. Gpredict [HQR15] is a tool based on the predictive trace analysis race detection by Huang et al. [HMR14]. They extend their work with the definition of monitors for Java programs, and they model the properties as regular expressions and custom operators. Regular expressions are chosen, as the underlying model – a set of satisfiability equations – is too awkward for user interaction. As our predictive model is based on a process algebra, we can use the algebra to describe the properties directly. We do not need to modify the algebra and it enables more complex properties than regular expressions[1], for example, counting the number of occurrences of an event and reacting to them.

[1]CSP is Turing complete.

3.4. Summary

We reviewed the literature on the detection of three error types: data races, deadlocks, and custom ordering errors through runtime verification. We showed that static tools generate false positives due to their conservative approach, which leads to wasted time and loss of interest by developers. These tools do not produce a schedule to help identify the associated defect. Furthermore, dynamic tools, such as RaceQuest, sacrifice coverage in order to avoid false positives. These tools can reproduce the detected failure more easily. In RaceQuest, we compensate the loss of coverage with off-line inference of alternative interleavings. Off-line exploration of an abstract model takes less time than re-executing the whole program under test multiple times. In difference to other predictive approaches, our model needs fewer events, models more synchronization constructs, and produces more interleavings. RaceQuest also handles the three kinds of errors with a common approach: our CSP abstract model.

4. RaceQuest and Deadlock Detection

In this chapter we present the main idea of this work as the RaceQuest tool, introduced in our publications [CT15a] and [CT15b]. The core of the approach is the interleaving generalization from a single trace by using the CSP process algebra. Along the description of the approach, we use deadlock detection as an example. Deadlock detection only requires the synchronization operations, so the CSP model is the minimal model generated by RaceQuest for any kind of concurrency failure. The chapter closes with an evaluation of deadlock detection. The following chapters extend the model with more captured events and with complimentary steps to detect other kinds of failures.

4.1. Overview

RaceQuest is an automatic tool that uses a trace of a parallel program to find concurrency errors in alternative hypothetical interleavings. Each error, or property, requires a description of which events are relevant and a description of the erroneous or correct behavior. Deadlock and data race descriptions are built-in, but the user can create a custom description, which is covered in Chapter 6.

Figure 4.1 shows the workflow of RaceQuest. RaceQuest takes the source code of a parallel program written in C/C++ using the POSIX threads library. The source code is instrumented accordingly to capture the relevant events for the error. We define the instrumentation specification as the set of events that are relevant and how are they extracted from the source code. Synchronization operations are always instrumented. The instrumentation is performed in LLVM IR. The instrumented program is executed only once, under input provided by the user. The instrumentation generates a trace of the running program, with the operations executed by all threads. In the off-line prediction step, the trace is decomposed and modeled using the process algebra CSP. The model optimistically represents alternative interleavings of the trace along the same control path. The erroneous or correct behavior is also described in CSP; we call this description the property specification. The model is explored using the FDR3 refinement checker. If the erroneous behavior is found, a counterexample is obtained. The instrumented program is re-executed with the same input, but following the scheduling dictated by the counterexample. As the model is optimistic, the counterexample can be feasible, enforceable in the program, or infeasible. The reproduction removes infeasible counterexamples. The user can also replay the program with the counterexample and observe the error under the same scheduling. RaceQuest is a dynamic off-line approach. It is only aware of the program events observed in the executed path, the trace. It cannot infer errors in non-observed program paths.

4.2. Motivational Deadlock Example

A deadlock is a situation where two actions wait for the other to finish but none of them do, as they cyclically compete for the same resources. In this work we target deadlocks caused by competing actions on communication channels, such as locking multiple mutexes. Most deadlock detection tools only detect whether the running program is actually deadlocked, and differentiate between a non-responsive state and one where the operation takes a long time.

Figure 4.2 shows a program with a possible deadlock. In the program two threads compete for simultaneously acquiring two mutexes: `m` and `p`. Three possible interleavings of this program appear in Figure 4.3. In the interleavings 4.3a and 4.3b, the program ended successfully, one of the threads was able to acquire both mutexes first. In the third interleaving 4.3c, each thread acquired one of the mutexes. As they try to acquire the second one, they cannot succeed because the other thread is holding it. Both threads remain blocked on the mutex and the program deadlocks.

With RaceQuest our target is not to detect a deadlock at runtime, but to find a potential alternative interleaving where a deadlock happens. RaceQuest takes one of the non-deadlocked interleavings and searches for a reordering of the synchronization events that can lead to a deadlock. The deadlocked

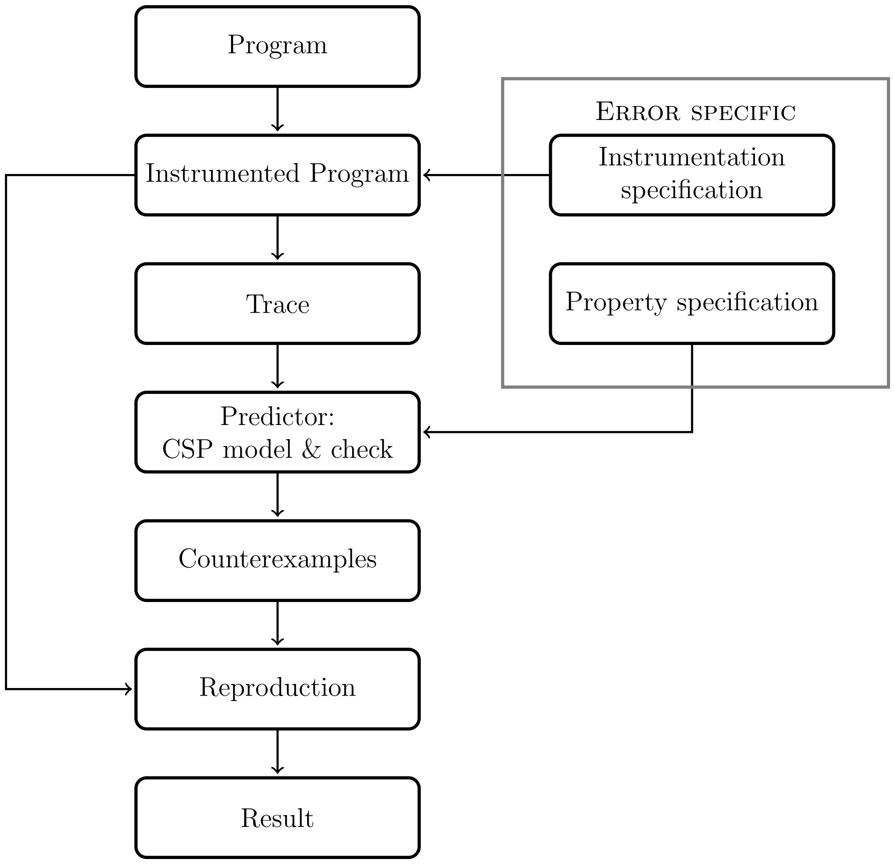


Figure 4.1.: RaceQuest workflow

interleaving consists of all the steps needed to guide the program to a deadlock. The steps are the calls to synchronization primitives.

4.3. Trace Model

First, we present our trace model: the events it contains and its internal rules. The trace model works as an interface between the execution of a program and the predictive step. The order of operations that the program must follow during the replay step is also described by using the trace model.

A multithreaded program has a set of threads, each with a unique identifier t. Each thread performs operations on different shared elements, such as a mutex m, a barrier b, or a semaphore s. A trace α is a sequence of the events performed by the different threads of a multithreaded program. The trace model is only composed of synchronization events, which are depicted in Table 4.1.

Table 4.1.: Synchronization events

Event	Description
start(t)	thread t began its execution
end(t)	thread t ended its execution
fork(t, t')	thread t spawned a child thread t'
join(t, t')	thread t was blocked until t' ended its execution
lock(t, m)	thread t acquired mutex m; if the mutex is a read-write mutex then the acquisition is in write mode
flock(t, m)	thread t executed an unsuccessful lock operation on mutex m, due a failed lock acquisition through a conditional lock (a lock busy during a try_lock) or a timeout in a timed_lock
rdlock(t, m)	thread t acquired the read-write mutex m in read mode
frdlock(t, m)	thread t executed an unsuccessful read lock operation on read-write mutex m, due a failed lock acquisition through a conditional lock (a lock busy during a try_lock) or a timeout in a timed_lock
unlock(t, m)	thread t released mutex m
signal(t, c_i)	thread t woke up a thread waiting on condition variable instance c_i
broadcast(t, c_i)	thread t woke up all threads waiting on condition variable instance c_i
wait(t, c_i)	thread t blocked until another thread executed a signal or broadcast on condition variable instance c_i
fwait(t, c)	thread t performed an unsuccessful wait operation, due a timeout on a timed_wait
barrier_init(b, i)	barrier b is initialized with value i
barrier_enter(t, b)	thread t entered and blocked at the barrier b
barrier_exit(t, b)	thread t left the barrier b
sem_init(s, i)	semaphore s is initialized with value i
sem_wait(t, s)	thread t decreased the value of semaphore s or blocked until the value is greater than zero
sem_fwait(t, s)	thread t performed an unsuccessful semaphore wait operation, due a timeout on a timed_wait or a conditional wait (semaphore counter equal to zero with a try_wait)
sem_post(t, s)	thread t increased the value of semaphore s by one
atom(t)	thread t performed an atomic operation

```
mutex m, p;
void main() {
    fork(worker);
    lock(p);
    lock(m);
    unlock(m);
    unlock(p);
    join(worker);
}

void worker() {
    lock(m);
    lock(p);
    unlock(p);
    unlock(m);
}
```

Figure 4.2.: Program with possible deadlock

Each event represents a completed operation done by a thread t. All the operations are related to synchronization operations, although some do not imply actual synchronization, such as ‘failed’ or initialization events.

A fork event has a single corresponding start event. The fork event always appears before the corresponding start. There is no fork event without a matching start event or a start event without a fork event. The main thread has no start or end event. A join event has a single corresponding end event. The end event always appears before the corresponding join. But not all end events have a matching join. A detached or non-joinable thread produces an end event, which does not match any join in the trace.

We assume that lock-unlock, or rdlock-unlock, pairs are balanced. A lock is always followed by an unlock by the same thread.

The events signal, broadcast, and wait do not use the condition variable on which they operate as an argument. Instead they have a condition instance c_i. A condition instance c_i is a specific use of a condition variable c. A condition variable c can be used in a program multiple times and can build different inter-thread edges. In this trace model each of these edges is considered an independent instance of the condition variable. Each condition instance has exactly one signal or broadcast event associated. If a condition instance is associated with a signal, there are zero or one wait events associated, because a

main	worker
fork(worker)	
	lock(m)
	lock(p)
	unlock(p)
	unlock(m)
lock(p)	
lock(m)	
unlock(m)	
unlock(p)	
join(worker)	

(a)

main	worker
fork(worker)	
lock(p)	
lock(m)	
unlock(m)	
unlock(p)	
	lock(m)
	lock(p)
	unlock(p)
	unlock(m)
join(worker)	

(b)

main	worker
fork(worker)	
	lock(m)
lock(p)	

DEADLOCK

(c)

Figure 4.3.: Different interleavings for program in Figure 4.2

signal can only wake up one thread. If a condition instance is associated with a broadcast, there are any number of wait events associated, because a broadcast can wake up multiple threads. Signal or broadcast events without associated waits are cases of *lost_signal*. They do not impose any synchronization and are kept for replay purposes. How the condition variables instances are computed is described in Section 4.3.2 along with the trace capture.

Two events, barrier_enter and barrier_exit, represent waiting at a barrier. The barrier_exit(t, b) event always happens after the matching barrier_enter(t, b). There is no other event performed by thread t between these two events.

Initialization events, barrier_init and sem_init, do not carry thread identifiers and are mere annotations of the initial values of the corresponding synchronization constructs. For each barrier b in the trace there is a single barrier_init for b. For each semaphore s in the trace there is a single sem_init for s. Initialization events are not used later during reproduction.

Failing events: flock, frdlock, fwait, and sem_fwait represent unsuccessful versions of some synchronization primitive. These events only appear in calls that can 'fail', such as conditional locking. These events do not produce any kind of inter-thread synchronization. Their position in the trace is needed for a more precise replay of the original program. Similarly, the atom event is only needed to know the position of atomic memory operations and order non-deterministic atomic operations during the replay step.

4.3.1. Example of a Non-Deadlocked Trace

We assume that a single execution of the program in Figure 4.2 has not produced a deadlock and finished successfully, as in the interleaving shown in Figure 4.3a. This interleaving is represented in the trace model as the trace in Figure 4.4.

fork(t_1, t_2)
start(t_2)
lock(t_2, m)
lock(t_2, p)
unlock(t_2, p)
unlock(t_2, m)
lock(t_1, p)
lock(t_1, m)
unlock(t_1, m)
unlock(t_1, p)
end(t_2)
join(t_1, t_2)

Figure 4.4.: Trace with no deadlock

The thread `main` is identified as t_1, the thread `worker` as t_2. At the beginning, the fork of t_2 by t_1 appears before the start of t_2. In this trace t_2 is the first to acquire and release both mutexes m and p. Afterwards, t_1 acquires and releases these mutexes. The lock-unlocks events on m and p are balanced. The thread t_1 also performs a join on t_2, which appears after the corresponding end event of t_2. For deadlocks only the synchronization events are needed, no other events are recorded in the trace.

4.3.2. Capturing the Trace

The description of the trace events in Table 4.1 associate each event with some function or point in the program, a tracing point. To capture the event at each tracing point we use two techniques: function wrapping and program instrumentation. The input program is first compiled in LLVM IR. The LLVM

IR is modified to generate an instrumented version which will be compiled to machine code. Auxiliary functions needed by the instrumentation are part of an external library that is linked to the instrumented program. The execution of the final binary of the instrumented version will output a trace.

The start, end, and atom events do not directly correspond to any function call. Auxiliary functions are inserted at the beginning and end of each thread function to emit start and end events. Before each LLVM IR atomic instruction we insert an auxiliary function, which will emit the atom event.

Calls to the functions of the synchronization library, the POSIX threads library, are outlined and wrapped in the LLVM IR with auxiliary functions. These auxiliary functions emit the event for the corresponding function, with a unique thread identifier and the needed arguments. Each auxiliary wrapping function also calls its original synchronization function. The trace model events are mapped one-to-one to POSIX threads calls, with some exceptions. Barrier_enter and barrier_exit are generated by the function wrapper during a single call to a barrier_wait, i.e. after and before the call to the original function. 'Failed' events depend on the return value of the original function. If a try_lock fails because the lock is held, then flock is emitted. If successful, then a normal lock event is emitted.

Capturing signal-wait events

Capturing signal-wait or broadcast-wait events is not as straightforward as capturing the other events. As explained in Section 2.1.2, the wait call is surrounded by a loop that checks the precondition. If the precondition is met before reaching the loop, the wait call is not seen at all. It is also possible that we see the wait call multiple times. The waiting thread can sporadically wake up, not exit the loop and call the wait function again.

This issue is solved using the exit of the waiting loop as the point where the wait event is emitted instead of the call itself. For each waiting loop an additional function call -`wait_post`- is introduced after the loop, as in the example in Figure 4.5.

The algorithm in Figure 4.6 checks if a wait call has the expected structure and instruments it. First it checks if the wait call is in a loop. If not a warning is issued. The loop is canonicalized; it contains a single entry edge from outside (the header), a single back edge, and all exit blocks are dominated by the header. The algorithm checks whether the loop header is dominated by a lock operation on the mutex used by the wait, and whether all exits of the loop are post-dominated by the corresponding unlock. As explained in

```
int flag = 0;
mutex m;
cond_var c;
void main() {
    fork(worker);
    lock(m);
    flag = 1;
    signal(c);
    unlock(m);
    join(worker);
}

void worker() {
    lock(m);
    while (flag == 0) {
        wait(c, m);
    }
    wait_post(c, m);
    unlock(m);
}
```

Figure 4.5.: Program with instrumented wait loop

Section 2.1.2, a wait loop must be surrounded by a lock and an unlock. If any of these conditions is not met, a warning is issued and the wait is ignored. Then each exit of the loop is instrumented with a wait_post function call. The instrumenting function takes the condition variable and mutex of the wait call as arguments. Non-instrumented waits lead to missing wait events in the trace. The instrumenting function is implemented as a LLVM Pass that uses LLVM to find loops and compute dominance.

During runtime the instrumented wait call will not emit any event. Instead, wait_post produces the corresponding wait event. For each wait-loop a single wait event is emitted, regardless of whether the wait call is executed zero, once, or multiple times. As a wait implicitly unlocks and locks the mutex, the wait event is not emitted alone. The wait event is always preceded by an unlock event on the mutex and followed by a lock event on the mutex, as show in the trace in Figure 4.7.

We also need to match the signal or broadcast to the corresponding waits. The trace model defines that signal and wait events operate on condition variable instances, i.e. uses of a condition variable. Initially these events are emitted

```
Let $wait\_set$ be the set of all wait, try_wait, and timed_wait calls present
in a module
for each w(c, m) call in set $wait\_set$, where $c$ is a condition variable and $m$
a mutex do
    if $w$ not in a loop then
        Issue a warning and continue
    end if
    Let $l$ be the loop surrounding $w$ in canonical form
    if $\nexists$ lock on $m$ dominating the header of $l$ then
        Issue a warning and continue
    end if
    for each unique exit block $e$ of the loop $l$ do
        if $\nexists$ unlock on $m$ postdominating $e$ then
            Issue a warning and continue
        end if
    end for
    for each unique exit block $e$ of the loop $l$ do
        Insert wait_post($c$,$m$) in $e$
    end for
end for
```

Figure 4.6.: Wait loop instrumentation algorithm

with the identifier of the condition variable. The trace is preprocessed to compute the condition variable instances. The preprocessing matches each wait to the nearest and previous signal or broadcast available and assigns to each match a unique identifier for the instance. During runtime, it is ensured that the signal event appears in the trace before the wait.

fork(t_1, t_2)
start(t_2)
lock(t_1, m)
signal(t_1, c)
unlock(t_1, m)
lock(t_2, m)
unlock(t_2, m)
wait(t_2, c)
lock(t_2, m)
unlock(t_2, m)
end(t_2)
join(t_1, t_2)

Figure 4.7.: Trace of a signal-wait program

4.4. CSP Model

The CSP model is the core idea of RaceQuest. A trace represents a single interleaving. From a single trace we extrapolate alternative interleavings that occur when the synchronization events have a different timing. The events in the trace are mapped to CSP events. The behavior of the individual threads and the synchronization constructs are modeled as CSP processes. These processes are combined with different standard operators to obtain a final process called $PROGRAM$. The idea is that the set of traces $traces(PROGRAM)$ contains the ordering of the input trace as well as the alternative ones.

Events

Each event in the trace is mapped one-to-one to CSP events. The mapping uses the CSP dot notation. The event type is the channel. The event arguments are fields in the CSP event, in the same order. For example, a trace event lock(t_1, m) is translated directly to CSP as $lock.t_1.m$.

The initialization events, barrier_init and sem_init, are ignored and not translated into CSP events. They are only used to provide starting values to the CSP processes representing barriers and semaphores.

Threads

Each thread in the trace is represented in two ways in the CSP model. Each CSP event carries an identifier of the performing thread in the first field, like the events in the trace carry the thread identifier.

The CSP model also defines a process $THREAD_i$ for thread i. Each of these processes represents the total ordering of the events in the trace for that thread. Each $THREAD_i$ process is a chain of CSP prefixed expressions terminated in $SKIP$. Each prefix corresponds to one event in trace for thread i, and the prefix expressions follow the same order as the events in the trace. For example, if the events in the trace for thread t are: $e_1, e_2, e_3, ..., e_n$, then the following CSP process is built:

$$THREAD_t = e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow ... \rightarrow e_n \rightarrow SKIP$$

An unrestricted interleaving of all thread events is achieved using the interleaving operator on all $THREAD$ processes. This new interleaved process freely mixes the events of the different threads, but maintains the internal order of each one. To determine that all threads have ended we concatenate this interleaving composition with the emission of an auxiliary event *endthreads*. The resulting process is called $INTER$:

$$INTER = (|||_{i \in Threads}\, THREAD_i))\ ;\ endthreads \rightarrow SKIP$$

Main process

The CSP process $PROGRAM$ represents the input trace but also the alternative reorderings, which could include interleavings that reveal a deadlock.

$$PROGRAM = INTER \underset{sync\_events}{\parallel} SYNC$$

$$sync\_events = \{|fork, start, join, end, lock, rdlock, unlock, signal, broadcast, \\ wait, barrier\_enter, barrier\_exit, sem\_post, sem\_wait|\} - independent$$

$PROGRAM$ is a parallel composition of two processes $INTER$ and $SYNC$. The combination of all the possible orderings of the threads is represented by the process $INTER$. The event names do not have any meaning for the process algebra, e.g. lock or signal are arbitrary identifiers. Their semantics are subject to our use case. The process $INTER$ can exhibit traces that violate the semantics of the synchronization events. For example, it is possible that $\langle lock.t_1.m, lock.t_2.m\rangle \in traces(INTER)$, because if both events appear in different $THREAD$ processes, the interleaving operator can shuffle them. But a mutex locked by t_1 could not be immediately locked by t_2. To provide meaning to the synchronization events, we define a process that describes the valid orders of the synchronization events. These orders are represented by the process $SYNC$.

The parallel composition of $INTER$ and $SYNC$ removes the invalid orders because of the synchronization events, for example $\langle lock.t_1.m, lock.t_2.m\rangle \notin traces(PROGRAM)$. All synchronization events are contained in the set *sync_events*. To perform any event in *sync_events*, both processes need to agree to perform the event due to the synchronous rules of CSP and the interfaced parallel operator. The set *independent* is excluded. This set contains: end events without matching join, signal, and broadcast events without matching wait.

Synchronization

The order between threads imposed by the synchronization constructs is represented in the CSP model with the process $SYNC$. $SYNC$ is a combination, interleaving, of sets of independent processes. Each of these sets represents a type of synchronization present in the trace: fork-join edges, mutexes, signal-wait, barriers, and semaphores.

$$\begin{gathered} SYNC = FORKJOINS \mathbin{|||} MUTEXES \mathbin{|||} WAITS \\ \mathbin{|||} BARRIERS \mathbin{|||} SEMAPHORES \end{gathered}$$

Fork-join

The creation of thread t' through the fork-start pair is represented by a $FORK_{t'}$ process:

$$FORK_{t'} = fork?t.t' \rightarrow start.t' \rightarrow SKIP$$

The start event of the children thread t' cannot be performed, because t' is blocked at the beginning, until any other thread t executes a fork operation creating t'. The use of ? instead of a dot means that t could be any thread, while t' is defined by the parameter of the process $FORK_{t'}$. Only one fork event that creates t' exists in the trace.

The joining on a thread t' through the end-join pair is represented by a $JOIN_{t'}$ process:

$$JOIN_{t'} = end.t' \rightarrow join?t.t' \rightarrow SKIP$$

In the trace, there is only one join event per thread created. Again, any thread t that can perform the join event is specified with the use of the ? separator.

The process $FORKJOINS$ combines all the $FORK$ and $JOIN$ processes. The processes are simply interleaved as they do not interact. All thread identifiers, with the exception of the main thread, the only one not created with a fork, compose the set $CreatedThreads$. The set $JoinedThreads$ contains all thread identifiers that are joined at some point in the trace. The main thread, non-joinable, and detached threads are excluded from $JoinedThreads$.

$$\begin{aligned} FORKJOINS = {} & (|||_{t \in CreatedThreads}\ FORK_t) \\ & |||\ (|||_{t \in JoinedThreads}\ JOIN_t) \end{aligned}$$

Mutexes

A mutex m that can be used by any thread is represented by the process $MUTEX_m$:

$$MUTEX_m = lock?t.m \to unlock.t.m \to MUTEX_m$$

The process accepts a lock on the mutex m by any thread, which represents the transition to the acquired state of a mutex. Afterwards the process can only return to the original state, thanks to the recursive call to $MUTEX_m$, with an unlock event. The unlock event must be performed by the same thread that has executed the previous lock event, as it is bound to t. Figure 4.8 represents the equivalent labeled transition system.

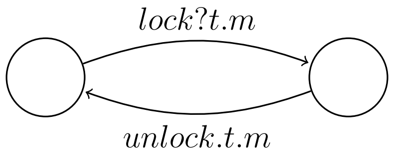


Figure 4.8.: Labeled transition system for process $MUTEX_m$

If the trace contains a rdlock event associated with a particular mutex m, then m is considered a read-write mutex. A read-write mutex m is modeled by a $RWMUTEX_{m,max}$ process, where max defines the maximum number of threads that can share the mutex in read mode:

$$\begin{aligned} RWMUTEX_{m,max} &= RWMUTEX_{m,max,0} \\ RWMUTEX_{m,max,0} &= lock?t.m \to unlock.t.m \to RWMUTEX_{m,max} \\ &\quad \Box\ rwlock?t.m \to RWMUTEX_{m,max,1} \\ RWMUTEX_{m,max,i} &= rdlock?t.m \to RWMUTEX_{m,max,i+1} \\ &\quad \Box\ unlock?t.m \to RWMUTEX_{m,max,i-1} \\ RWMUTEX_{m,max} &= unlock?t.m \to RWMUTEX_{m,max,i-1} \end{aligned}$$

At the beginning a $RWMUTEX$ process accepts any lock or rwlock event. A lock event puts the read-write mutex in write mode. Then it works similar to a normal mutex. No other locking event is possible until the corresponding thread frees it with an unlock event. Alternatively, the mutex can go into read mode, which happens when it is acquired with a rdlock event from the initial state. From this point on, only new rdlock events or unlock events are allowed. The parameter i is a counter of the needed unlocks to bring back the process to the initial state after a series of rdlock events. A read-write mutex has no limit on the number of threads that can acquire it in read mode. To limit the number of possible states in the model due to the different values of i, we limit i with the maximum value max. The value max is equivalent to the number of different threads in the trace that have a rdlock event on the mutex m. The maximum value of max is the number of different threads in the trace. If max is zero, there is no rdlock event on the trace, and the behavior is equivalent to a simple $MUTEX$ process, so $MUTEX_m \equiv RWMUTEX_{m,0}$. Figure 4.9 represents the equivalent labeled transition system.

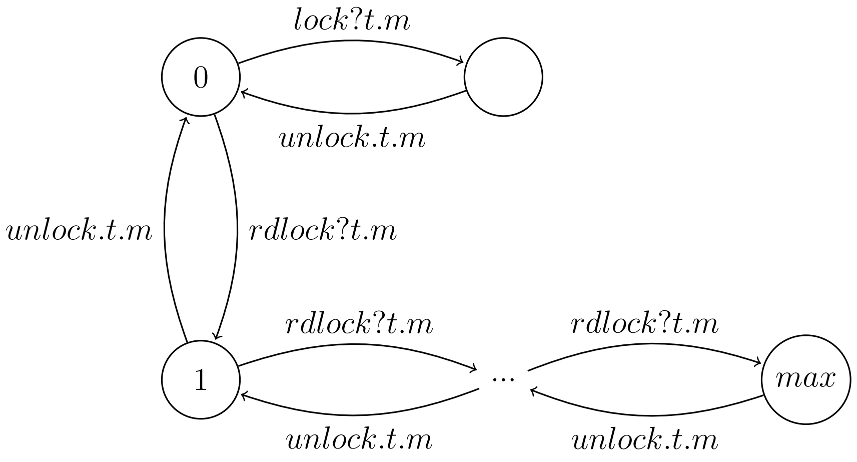


Figure 4.9.: Labeled transition system for process $RWMUTEX_{m,max}$

All the mutexes in a trace are represented with the process $MUTEXES$. The set $Mutexes$ is the set of all normal mutexes defined on the trace. The set $RWMutexes$ is a set of pairs with the id of a mutex, as well as the number of threads with rdlock events on the mutex, if this number is greater than zero.

$$\begin{aligned} MUTEXES = {} & (|||_{m \in Mutexes}\ MUTEX_m) \\ & ||| \ (|||_{(m,max) \in RWMutexes}\ RWMUTEX_{m,max}) \end{aligned}$$

We can simplify it further and define the $MUTEXES$ combination only using $RWMUTEX$. In this case the set $AllMutexes$ is a set of pairs with the id of a mutex and the number of threads with rdlock events on it.

$$MUTEXES = |||_{(m,max) \in AllMutexes} RWMUTEX_{m,max}$$

Signal-wait

Each synchronization due to a conditional variable instance c_i is represented by a $SIGNAL_{c_i}$ process.

$$\begin{aligned} SIGNAL_{c_i} &= signal?t.c_i \rightarrow wait?t.c_i \rightarrow SKIP \\ &\quad \Box\, broadcast?t.c_i \rightarrow NWAIT_{c_i} \\ NWAIT_{c_i} &= wait?t.c_i \rightarrow NWAIT_{c_i} \end{aligned}$$

If the condition variable instance is associated with a signal, then the signal is followed by a single wait. If associated with a broadcast, then the recursive process $NWAIT_{c_i}$ allows any number of waits on c_i. A thread can only perform a wait on c_i after another thread has performed the corresponding signal/broadcast event on c_i. Figure 4.10 represents the equivalent labeled transition system.

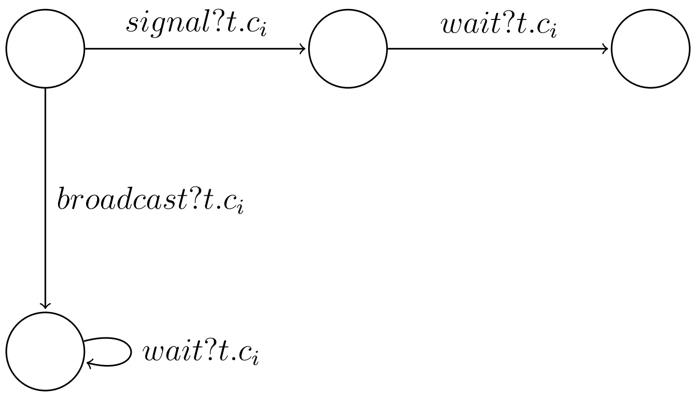


Figure 4.10.: Labeled transition system for process $SIGNAL_{c_i}$

The process $SIGNALS$ represents all the signal-wait event pairs in a trace, where the set $CondVarInstances$ is the set of all condition variable instances with at least one wait in the trace.

$$SIGNALS = |||_{c_i \in CondVarInstances} SIGNAL_{c_i}$$

Barriers

A barrier b is modeled with the $BARRIER_{b,max}$ process, where max is the number of threads that must reach the barrier to open it, a value provided by the only related barrier_init event on b in the trace.

$$\begin{aligned}
BARRIER_{b,max} &= barrier\_enter?t.b \rightarrow BARRIER\_U_{b,max,1} \\
BARRIER\_U_{b,max,i} &= barrier\_enter?t.b \rightarrow BARRIER\_U_{b,max,i+1} \\
BARRIER\_U_{b,max,max} &= barrier\_exit?t.b \rightarrow BARRIER\_D_{b,max-1} \\
BARRIER\_D_{b,max,i} &= barrier\_exit?t.b \rightarrow BARRIER\_D_{b,max,i-1} \\
BARRIER\_D_{b,max,0} &= BARRIER_{b,max}
\end{aligned}$$

Initially the process $BARRIER_{b,max}$ only accepts barrier_enter events. When the counter i reaches the max value, the process only accepts barrier_exit events. When a thread in the trace performs a barrier_enter, the next event is always a barrier_exit on the same barrier. This process accumulates the barrier_enter events of multiple threads, until there are max threads, then all these threads can execute their respective barrier_exit events. Once all the threads have left the barrier, the process is restarted and can be reused. Figure 4.11 represents the equivalent labeled transition system.

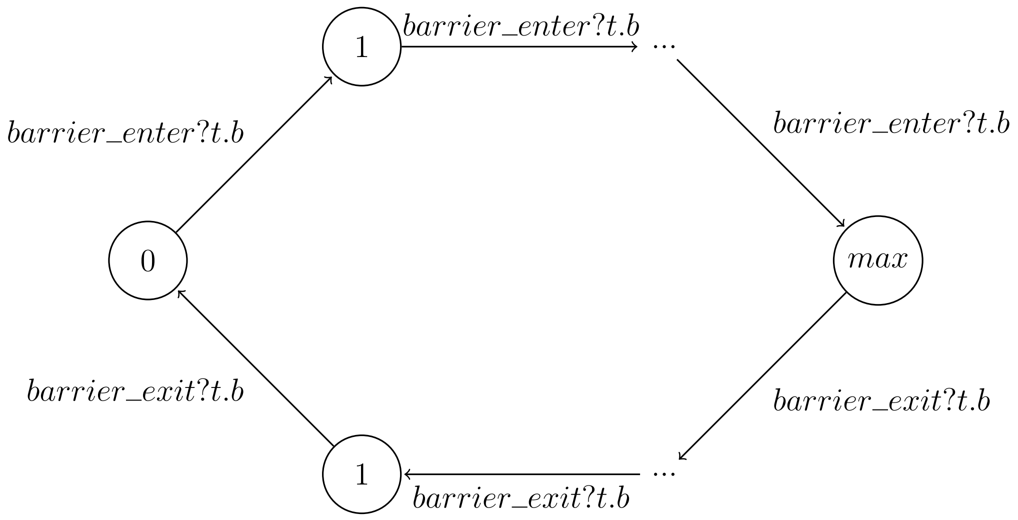


Figure 4.11.: Labeled transition system for process $BARRIER_{b,max}$

All the barriers in a trace are represented by the $BARRIERS$ process. The set $Barriers$ is the set of pairs with the id of a barrier and the number of threads needed to unblock the barrier given by the corresponding barrier_init event in the trace.

$$BARRIERS = |||_{(b,max)\in Barriers}\ BARRIER_{b,max}$$

Semaphore

A semaphore s is modeled as a counter with the $SEMAPHORE_{s,init,max}$ process, where $init$ is the initial value of the semaphore provided by the corresponding sem_init event. Max defines the maximum number that the semaphore can reach and is equal to the total number of sem_post events on s in the trace.

$$\begin{aligned}
SEMAPHORE_{s,0,max} &= sem\_post?t.s \rightarrow SEMAPHORE_{s,1,max}\\
SEMAPHORE_{s,i,max} &= sem\_post?t.s \rightarrow SEMAPHORE_{s,i+1,max}\\
&\quad \square\ sem\_wait?t.s \rightarrow SEMAPHORE_{s,i-1,max}\\
SEMAPHORE_{s,max,max} &= sem\_wait?t.s \rightarrow SEMAPHORE_{s,max-1,max}
\end{aligned}$$

An event sem_post by any thread increases the counter by one, while an event sem_wait decreases the counter by one. If the counter is zero, threads executing a sem_wait event on the semaphore are blocked until another executes a sem_post. A semaphore starts with an initial value $init$, so the whole process can start in any of the states. Figure 4.12 represents the equivalent labeled transition system.

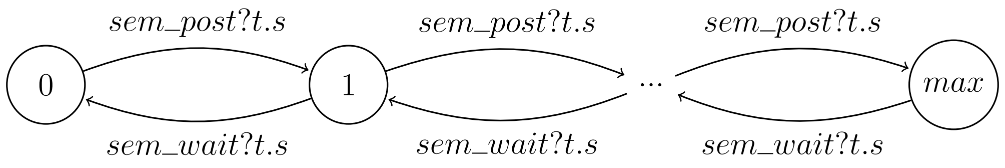

$$\begin{tikzcd}
0 \arrow[r, bend left, "sem\_post?t.s"] & 1 \arrow[l, bend left, "sem\_wait?t.s"] \arrow[r, bend left, "sem\_post?t.s"] & \cdots \arrow[l, bend left, "sem\_wait?t.s"] \arrow[r, bend left, "sem\_post?t.s"] & max \arrow[l, bend left, "sem\_wait?t.s"]
\end{tikzcd}$$

Figure 4.12.: Labeled transition system for process $SEMAPHORE_{s,init,max}$

All the semaphores in a trace are represented by the $SEMAPHORES$ process. $Semaphores$ is the set of 3-tuples with the id of a semaphore, the initial value, and the maximum value of the counter, which are given by the corresponding sem_init event in the trace.

$$SEMAPHORES = |||_{(s,init,max)\in Barriers}\ SEMAPHORE_{s,init,max}$$

4.4.1. CSP Model Example

Figure 4.13 contains the corresponding CSP model for the trace example in Figure 4.4.

$$
\begin{aligned}
THREAD_{t_1} &= fork.t_1.t_2 \to lock.t_1.p \to lock.t_1.m \to unlock.t_1.m \to \\
&\quad unlock.t_1.p \to join.t_1.t_2 \to SKIP \\
THREAD_{t_2} &= start.t_2 \to lock.t_2.m \to lock.t_2.p \to unlock.t_2.p \to \\
&\quad unlock.t_2.m \to end.t_2 \to SKIP \\
INTER &= (THREAD_{t_1} \mid\mid\mid THREAD_{t_2}) \mathbin{;} endthreads \to SKIP \\
FORK_{t'} &= fork?t.t' \to start.t' \to SKIP \\
JOIN_{t'} &= end.t' \to join?t.t' \to SKIP \\
MUTEX_i &= lock?t.i \to unlock.t.i \to MUTEX_i \\
SYNC &= FORK_{t_2} \mid\mid\mid JOIN_{t_2} \mid\mid\mid MUTEX_m \mid\mid\mid MUTEX_p \\
PROGRAM &= INTER \underset{sync\_events}{\parallel} SYNC \\
sync\_events &= \{\mid fork, start, join, end, lock, rdlock, unlock, signal, \\
&\quad broadcast, wait, barrier\_enter, barrier\_exit, sem\_post, \\
&\quad sem\_wait \mid\} - independent \\
independent &= \emptyset
\end{aligned}
$$

Figure 4.13.: CSP model of the trace in Figure 4.4

The trace contains only two threads, so the model only uses the identifiers t_1 and t_2 in the events and there are only two $THREAD$ processes. Each $THREAD_t$ process only consists of the events in the trace performed by thread t, and the events follow the same order as in the trace. The $SYNC$ process is composed of three other processes: $FORK_{t_2}$, $JOIN_{t_2}$ and a $MUTEX_p$. As described previously, $FORK_{t'}$, $JOIN_{t'}$ and $MUTEX_i$ are the templates for any fork, join, or mutex synchronization. They are instantiated in the composition of $SYNC$. The $FORK_{t_2}$ represents the creation of the thread t_2. The $JOIN_{t_2}$ represents the joining of t_2. The joining thread cannot execute the join event until t_2 has executed its end event. The $MUTEX_p$ model the accesses by any thread to the mutex p, after one lock has been performed; another lock is not possible until the corresponding unlock. The set *independent* is empty, because there are no independent events in this model. We could simplify the set *sync_events* to $sync\_events = \{\mid fork, start, join, end, lock, unlock \mid\}$.

The set *traces*($PROGRAM$) contains the trace of the interleaving in Figure 4.4. The set also contains other traces, i.e. other reorderings of the input events. The order of the events of the same thread is maintained in all reorderings. For example, $lock.t_1.p$ always appears after $fork.t_1.t_2$, because it was so defined in the process $THREAD_{t_1}$. The processes representing the synchronization constructs, such as $MUTEX_i$ and $FORK_{t'}$, ensure that the orders of their events follow the semantics of the synchronization. For example, there is no trace in the set where $start.t_2$ appears before $fork.t_1.t_2$, because $FORK_{t_2}$ forces the fork to always appear first. Similarly, there is no trace with two lock events on the same mutex without an unlock between them; the process $MUTEX_i$ guarantees this.

$$\begin{gathered}
\mathit{traces}(PROGRAM) = \{\langle fork.t_1.t_2\rangle, \langle fork.t_1.t_2, start.t_2\rangle, ..., \\
\langle fork.t_1.t_2, start.t_2, lock.t_2.m, lock.t_2.p, unlock.t_2.p, unlock.t_2.m, \\
lock.t_1.p, lock.t_1.m, unlock.t_1.m, unlock.t_1.p, end.t_2, join.t_1.t_2, endthreads\rangle, \\
\langle fork.t_1.t_2, start.t_2, lock.t_1.p, lock.t_1.m, unlock.t_1.m, unlock.t_1.p, \\
lock.t_2.m, lock.t_2.p, unlock.t_2.p, unlock.t_2.m, end.t_2, join.t_1.t_2, endthreads\rangle, \\
\langle fork.t_1.t_2, start.t_2, lock.t_2.m, lock.t_1.p\rangle, ...\}
\end{gathered}$$

We see that the goal of the CSP modeling is achieved, i.e. to obtain a set of possible interleavings from variations of the input interleaving, such as all the interleavings depicted in Figure 4.3. The trace corresponding to the deadlocked interleaving in Figure 4.3c also appears in the set:

$$\langle fork.t_1.t_2, start.t_2, lock.t_2.m, lock.t_1.p\rangle$$

In the next section we extract this trace from *traces*($PROGRAM$), using a CSP refinement check and the event *endthreads*. Some traces in the set exhibit the auxiliary event *endthreads* as the last event. This event is present in the model, but not in the input trace. The event *endthreads* is one of the keys to differentiate between a trace where the program halts and a trace where the program deadlocks.

4.5. Error Checking

Once we have a model of the input and the alternative traces in the CSP process $PROGRAM$, we have to find and extract a possible trace in $PROGRAM$ leading to the target error.

The error or correct behavior check must be defined as a refinement relationship between the two processes $SPEC$ and $IMPL$:

$$SPEC \sqsubseteq IMPL(PROGRAM)$$

The $IMPL$ process must be a composition where one of the components is the $PROGRAM$ process. A CSP model checker has the task to explore the composing processes and check if the refinement holds or not. The model checker can find more than one state where the refinement does not hold; these states belong to different schedulings and/or different errors. If the refinement does not hold, the $PROGRAM$ process has executed a sequence of steps that it should not. This sequence of steps can be used later to reproduce the erroneous behavior.

This refinement relationship and auxiliary processes are the property specification. We must describe the processes $SPEC$ and $IMPL$ for the type of error that we want to detect. The following section describes the general property specification for deadlocks.

4.5.1. Deadlock Checking

In CSP a process is deadlocked after a trace if it cannot perform any event at all, i.e. its refusal set is composed of all the events in the system. The default CSP process $STOP$ is always deadlocked. After the only possible trace, $\langle\rangle$ (the empty trace), $STOP$ cannot perform anything.

Using the *failures* semantic model and a refinement relationship, it is possible to check if a process deadlocks or not in CSP terms. The *traces* semantic model is not enough here. We need to compare which sequences the processes are able to perform and if they are willing to perform more events; in a deadlock they cannot engage in any event. To find deadlocks in the original program, we search for deadlocks in the constructed $PROGRAM$ process.

But when the process $PROGRAM$ performs the *endthreads* event, it will be CSP-deadlocked, i.e. after executing all the events in both thread processes there are no more events to execute in $PROGRAM$. We have to distinguish between the situation in $PROGRAM$ reaching a CSP-deadlock because *endthreads* has been reached, and any other CSP-deadlock. These other CSP-deadlocks are the ones that we want to find and apply to the original program as deadlocks. The following refinement relationship for deadlock freedom is as follows:

$$\begin{aligned}&LIVE = live \rightarrow LIVE\\&LIVE \sqsubseteq_{\mathrm{F}} (PROGRAM \Theta_{endthreads} SKIP) \setminus \Sigma \,;\, LIVE\end{aligned}$$

We use the exception operator Θ on $PROGRAM$ to transit to a $SKIP$ process if the event *endthreads* is reached. Reaching $SKIP$ means reaching the process $LIVE$ thanks to the sequential composition. The process $LIVE$ is a recursive process that always emits and accepts the *live* event. It never deadlocks. We also hide all events, Σ, except the *live* event.

The idea is the following: if $PROGRAM$ cannot deadlock under any sequence of events, then it will always reach the *endthreads* event, which leads to $SKIP$ and finally to $LIVE$. Both sides of the refinement will behave like the $LIVE$ process and the refinement holds, always emitting and accepting the *live* event. If $PROGRAM$ deadlocks at any other point without reaching the *endthreads* event, then, for some trace, it will not behave like $LIVE$ and will refuse the event *live*. But the $LIVE$ process on the left side of the refinement cannot refuse the event *live*, it always accepts *live*. As the left side of the refinement always accepts *live*, but the right side can refuse *live* for some traces, then the refinement does not hold.

When the refinement holds there is no deadlock in any reordering of the input trace, i.e. $PROGRAM$ is deadlock-free. When the refinement does not hold, the process $PROGRAM$ contains a reordering of the input trace leading to the detected deadlock. This reordering is a counterexample of the refinement, a witness of the falsification of the relationship.

Note that the refinement check has always the same definition, independently of the construction of the $PROGRAM$ process. Neither the number of events, composition of $THREAD$ processes, or synchronization processes modify how the deadlock check is built.

4.5.2. Deadlock Check Example

Figure 4.14 contains the previous example model in Figure 4.13 along with the deadlock refinement check. The final CSP model with the check is the union of the processes defining the possible reorderings, such as $PROGRAM$ and $THREAD_t$, the definition of the auxiliary process $LIVE$, and the refinement check.

In this example the refinement does not hold and a counterexample is:

$$\langle fork.t_1.t_2, start.t_2, lock.t_2.m, lock.t_1.p\rangle$$

The counterexample is equivalent to the deadlocked interleaving in Figure 4.3c. The thread t_2 first acquires m, and before acquiring p the thread t_1 locks on p. After this trace the process $PROGRAM$ cannot engage in any new event, it is

$$\begin{aligned}
THREAD_{t_1} &= fork.t_1.t_2 \rightarrow lock.t_1.p \rightarrow lock.t_1.m \rightarrow unlock.t_1.m \rightarrow \\
&\quad unlock.t_1.p \rightarrow join.t_1.t_2 \rightarrow SKIP \\
THREAD_{t_2} &= start.t_2 \rightarrow lock.t_2.m \rightarrow lock.t_2.p \rightarrow unlock.t_2.p \rightarrow \\
&\quad unlock.t_2.m \rightarrow end.t_2 \rightarrow SKIP \\
INTER &= (THREAD_{t_1} \mid\mid\mid THREAD_{t_2}) \,;\, endthreads \rightarrow SKIP \\
FORK_{t'} &= fork?t.t' \rightarrow start.t' \rightarrow SKIP \\
JOIN_{t'} &= end.t' \rightarrow join?t.t' \rightarrow SKIP \\
MUTEX_i &= lock?t.i \rightarrow unlock.t.i \rightarrow MUTEX_i \\
SYNC &= FORK_{t_2} \mid\mid\mid JOIN_{t_2} \mid\mid\mid MUTEX_m \mid\mid\mid MUTEX_p \\
PROGRAM &= INTER \underset{sync\_events}{\parallel} SYNC \\
sync\_events &= \{| fork, start, join, end, lock, rdlock, unlock, signal, \\
&\quad broadcast, wait, barrier\_enter, barrier\_exit, sem\_post, \\
&\quad sem\_wait |\} - independent \\
independent &= \emptyset
\end{aligned}$$

$$\begin{aligned}
LIVE &= live \rightarrow LIVE \\
LIVE &\sqsubseteq_{\mathrm{F}} (PROGRAM \Theta_{endthreads} SKIP) \setminus \Sigma \,;\, LIVE
\end{aligned}$$

Figure 4.14.: CSP model of the trace in Figure 4.4 with deadlock refinement

deadlocked. The $INTER$ process (due to the individual $THREAD$ processes) is only willing to execute the following events: $lock.t_1.m$ and $lock.t_2.p$. But the $SYNC$ (due to the $MUTEX$ processes) will only engage in: $unlock.t_1.p$, $unlock.t_2.m$ or $end.t_2$. As these two sets are disjointed and all the events are part of $sync\_events$, $INTER$ and $SYNC$ cannot agree on the parallel operator to execute a common event. The combination ($PROGRAM$) refuses to engage in any event at all and is effectively deadlocked after that trace. The machine-readable version of this example in CSP$_M$ is in Appendix A.2.1.

4.6. Counterexample Reproduction

The final step in RaceQuest consists of reproducing the failure. The reproduction has two goals: to discard infeasible counterexamples, see Section 4.7.2, and to allow the developer to reproduce the error multiple times, always observing the same interleaving.

The counterexample is an ordered sequence of events that lead to the erroneous state. We use the counterexample directly as a schedule to replay the instrumented program. The input trace was obtained from a set of tracing points, such as start and end of a thread, thread creation, mutex locking and unlocking, etc., as described previously in Table 4.1. These points are now used as scheduling points.

Each time a thread arrives at one of these scheduling points, the algorithm in Figure 4.15 is evaluated. The thread id of the current thread is compared against the thread id of the first event in the counterexample. If the ids are different, the current thread is blocked and must wait to be woken up. If the current thread matches the first event in the counterexample, then the type of the operations is compared. For example, if the first event in the counterexample is a lock, then the current scheduling point of the current thread should be a *lock*. When the types do not match, the counterexample is infeasible and the reproduction is aborted. If the types match, the real operation is executed, i.e. the lock. If the expected operation is a 'failed' event, such as flock or fwait, the failed result is mimicked instead of executing the real operation. For example, returning a 'lock busy' result if we are in a try_lock. Then the first event in the counterexample is discarded and the next event is enforced. If the thread for the next event is already blocked, it is woken up explicitly.

The reproduction is partially deterministic. After enforcing the last event in the counterexample, the program reaches a state where the failure manifests itself. The program continues its execution freely, with the consequences of the failure, and the enforcing algorithm no longer applies.

During the enforcement the algorithm prevents the execution of any scheduling point that does not match the first event in the counterexample. Instructions not covered by scheduling points are executed freely. If a program does not exhibit the failure, the counterexample can be discarded.

4.6.1. Deadlock Reproduction Example

The counterexample obtained from the previous model in Figure 4.14 is:

$$\langle fork.t_1.t_2, start.t_2, lock.t_2.m, lock.t_1.p \rangle$$

Following this counterexample we re-execute the original program from Figure 4.2. At the beginning only the `fork` by the thread `main` is possible. After that, the thread `worker` (t_2) can start or `main` can acquire the mutex `p`. If

```
Let $sched$ be the sequence of events to be enforced
Let $t$ be the current thread
Let $op$ be the type of operation in the current scheduling point
Let $t_s$ be the thread id in $head(sched)$
if $t \neq t_s$ then
    Block thread $t$
end if
Let $op_s$ be the operation in $head(sched)$
if $op \neq op_s$ then
    infeasible enforcement, abort
end if
if $op_s$ is not a fail operation then
    $result \leftarrow$ execute original operation
else
    $result \leftarrow$ a failed result
end if
Remove $head(sched)$
Let $t_w$ be the thread id in $head(sched)$
if $t_w$ is blocked then
    Wake up $t_w$
end if
return $result$
```

Figure 4.15.: Scheduling point enforcing algorithm

`main` reaches first `lock(p)`, then its thread id will not match the thread id of the next event, $start.t_2$, and `main` will be blocked by the algorithm. The thread `worker` is able to begin, i.e. to complete the *start* event. The thread `worker` also successfully executes `lock(m)`. The next thread is `main`, so it is unblocked. The thread `main` executes `lock(p)`. The counterexample is now totally consumed and the program will run freely. But both mutexes are acquired by different threads and both threads will try to acquire the other mutex, thus the program deadlocks as expected.

4.7. Limitations

The presented approach has several limitations in the model depending on its size, which events are represented, and the possible reorderings. In this section we discuss all these limitations, as well as their consequences.

4.7.1. Unsupported Concurrency Mechanisms

The CSP model only creates inter-thread orderings based on the semantics of the synchronization operations. There are two mechanisms that are not supported by RaceQuest:

- Ad-hoc synchronization: any non-standard synchronization construct created by the application developer, as described in Section 2.1.3. Ad-hoc synchronization does not use any standard function call or fixed pattern. Its purpose is dependent on the specific implementation.
- Lock-free programs: in general, these are parallel programs that cannot be blocked by other threads. They do not use mutexes or other common synchronization constructs. Instead, through a clever use of atomic variables and data separation, they enable concurrent and safe access without introducing concurrency errors.

These two mechanisms are completely transparent to RaceQuest. RaceQuest can neither identify the points in the source code where the synchronization happens, nor its expected behavior. The CSP model will not reflect specific behavior of any of either of these two mechanisms – no additional orders between events. This results in the appearance of false positives in the model. Also, as the schedule re-execution is based on synchronization operations, no order is enforced in code with any of these mechanisms which results in non-deterministic execution.

A solution could be to add annotations to the code to capture the order observed in the input manually or automatically. The annotations would emit auxiliary events and the CSP model should define the order of these events. This order would simply be like a signal-wait, an order that would be maintained in all interleavings. As the semantics of the mechanism are unknown it is not possible to implement more complex reorderings. But the false positives due to these mechanisms would be reduced.

4.7.2. Infeasible Reordering

The CSP model is optimistic and assumes that all orders of events in the set $traces(PROGRAM)$ are possible in the original program, i.e. that they are feasible. A reordering of synchronization events in the original program could lead to a different program path, where the remaining events of the new order cannot be enforced. These other program paths are unknown by the model as they do not exist in the input trace.

The program in Figure 4.16 can mainly exhibit two interleavings depending on which thread executes the access to the atomic `flag` first, as shown in Figure 4.17. If the thread `main` executes the store on `flag` first, as in Figure 4.17a, then the thread `worker` will enter in the if-body and execute the calls `lock` and `unlock` on `m` and `p`. But if `worker` reads the `flag` value first, then it will not execute the mutex operations and will end, as in Figure 4.17b.

```
atomic flag = false;
mutex m, p;
void main() {
    fork(worker);
    lock(p);
    lock(m);
    flag = true;
    unlock(m);
    unlock(p);
    join(worker);
}

void worker() {
    if (flag == true) {
      lock(m);
      lock(p);
      unlock(p);
      unlock(m);
    }
}
```

Figure 4.16.: Program with multiple control flow

The behavior of RaceQuest depends on the input trace. If the input trace is the one in Figure 4.17b, no deadlock is detected and no counterexample is generated. If the input trace is the one in Figure 4.17a, then a deadlock is detected and the counterexample could be:

$$\langle fork.t_1.t_2, start.t_2, atomic.t_2.flag, lock.t_2.m, lock.t_1.p\rangle$$

But this counterexample is infeasible. After the events $fork.t_1.t_2$, $start.t_2$, the event $atomic.t_2.flag$ is performed in the if-clause. As the predicate is false, the worker thread will not enter the if-body and the thread can only execute its corresponding end event, $end.t_2$. The counterexample demands that the next event is a lock. As this demand cannot be fulfilled, the counterexample cannot

main	worker
fork(worker)	
lock(p)	
lock(m)	
flag = true	
unlock(m)	
unlock(p)	
	if (flag)
	lock(m)
	lock(p)
	unlock(p)
	unlock(m)
join(worker)	

(a)

main	worker
fork(worker)	
	if (flag)
lock(p)	
lock(m)	
flag = true	
unlock(m)	
unlock(p)	
join(worker)	

(b)

Figure 4.17.: Different interleavings for program in Figure 4.16

be completed; it is infeasible. We consider this counterexample a false positive of the modeling step. As we automatically reproduce all counterexamples provided by the model checker, we can detect which can be completed and which cannot. The infeasible schedulings are explicitly removed after enforcing them, and not provided to the user.

4.7.3. Scalability & Trace Windowing

As the number of events in the trace increases, the size of the CSP model grows quickly. A bigger model requires more resources for its exploration, and may eventually become intractable. The size of the trace imposes a limit on the scalability of the approach.

To limit the size of the model without limiting the size of the trace, we split the trace into a series of fixed size windows, e.g. 1,000 events. Each window is treated as an individual trace with its own CSP model and individual checks. Multiple checks of smaller models take fewer resources than a single check on a larger model.

Previous windows are still needed to setup the initial conditions. The CSP model for the window i has to take into consideration the state of the system at the end of the window $i-1$, e.g. if mutexes are acquired or the counters in semaphores. The structure of the *PROGRAM* process is replaced by the following:

$$PROGRAM = (PREFIX\,;INTER) \underset{sync\_events}{\parallel} SYNC$$

The process $INTER$ is the interleaving of $THREAD$ processes with only the events in window i. The process $SYNC$ is the interleaving of the synchronization processes for any synchronization construct that appears in the windows 0 to i. The $PREFIX$ process is a $SKIP$ ending process where all the events in the windows 0 to $i-1$ appear as a sequence of prefix operators. In $PREFIX$ there is no concurrency, the events from different threads are not split into subprocesses and the events follow the order in the input trace. A $PREFIX$ process looks like this:

$$PREFIX = fork.t_1.t_2 \rightarrow start.t_2 \rightarrow lock.t_1.m \rightarrow lock.t_2.p \rightarrow SKIP$$

$SYNC$ is also in parallel with $PREFIX$. As $PREFIX$ is always a valid order, the order executed by the program, then $SYNC$ will synchronize in all its events. After the end of $PREFIX$, all synchronization processes in $SYNC$ can be in a non-initial state, e.g. mutexes can already be acquired. When $INTER$ starts, after the $SKIP$ in $PREFIX$, it will run in parallel with the last state of $SYNC$.

A disadvantage of trace windowing is the possible introduction of false negatives. To be detected, an error would need a reordering between elements on two different windows. But as only the elements on one window are reordered, the error would be missed, a false negative.

A positive side-effect of trace windowing is that the probability of obtaining an infeasible schedule is reduced. In Section 4.7.2 we have shown that it is possible to obtain a counterexample that cannot be applied to the original program. For a long trace the probability of becoming infeasible is higher, due to a large divergence from the original trace. The windowing only allows new reorderings from the start of the window, not from the start of the trace. The events prior to the examined window cannot become infeasible.

4.8. Deadlock Detection Evaluation

In this section we evaluate the RaceQuest capability for deadlock detection. We apply RaceQuest to a set of scenarios that could deadlock under specific timings. We observe whether we could deadlock the programs using the counterexample provided by RaceQuest. We compare RaceQuest against another predictive approach: a lock-order detector. A lock-order detector issues a warning if a set of two or more locks have more than one acquisition order in a single execution of a program. Different order-acquisitions are an indication of a possible deadlock under a different scheduling. We also present two more scenarios in detail, where the lock-order detector fails but RaceQuest does not.

4.8.1. Experimental Setup

This evaluation has been carried out in an 8x Intel Xeon CPU E5-1620 v2 at 3.70GHz, 64 GB RAM, running CentOS 6.7 x64. The programs are written on C or C++ and use POSIX threads for parallelization. As RaceQuest instrumentation and replay are based on LLVM, the programs were compiled with clang 3.7 for x64 architecture. FDR3.3 was used as CSP model checker. FDR3 was configured to search for a single counterexample. After it has been found the search is aborted as the refinement does not hold. The lock-order detector tested is included in ThreadSanitizer V2 [SI09]. The trace window size is 10,000 events. The traces of all the benchmarks fit into a single window.

4.8.2. Benchmark & Results

RaceQuest deadlock detection is evaluated on a set of 13 scenarios and applications from the deadlock detection literature. Hawknl is a test for a custom mutex implementation in the HawkNL game-oriented network API. Sqlite-1672 is a test harness for a mutex implementation in the SQLite database engine, related to the bug 1672. Hg02, tc13, and tc14 are tests from the Helgrind [Val07] suite, tc14 is an implementation of the classical dining philosophers problem. Cp-9 is the scenario in Figure 9 in the *causally proceeds* paper [SES$^+$12]. Tsandl1 and tsandl2 are two cases of a data race benchmark [Goo09] which also includes some deadlocks. Guarded-lock, guarded-lock2, mul-lockorders, sem-deadlock, and barrier-deadlock have been explicitly written for this benchmark. Scenarios with a deadlock are timed to show a non-deadlocked interleaving, but in another interleaving the deadlock is still possible. The scenarios and applications have 37 lines of code and 13 synchronization operations in average.

The results appear in Table 4.2. *Expected* indicates if the scenario contains a deadlock for any interleaving. *RaceQuest* indicates if a deadlock is found and reproduced. *ThreadSanitizer* shows if a lock order violation is identified, which means a possible deadlock.

Hawnknl, hg02, tc14, cp-9, tsandl1, tsandl2, and guarded-lock are different scenarios where there is a violation of the lock order. RaceQuest and ThreadSanitizer correctly detected the deadlock. RaceQuest not only detected the

Table 4.2.: Deadlock detection benchmark results

Scenario	**Expected**	**RaceQuest**	**ThreadSanitizer**
hawknl	TRUE	TRUE	TRUE
sqlite-1672	TRUE	**FALSE**	TRUE
hg02	TRUE	TRUE	TRUE
tc13	FALSE	FALSE	**TRUE**
tc14	TRUE	TRUE	TRUE
cp-9	TRUE	TRUE	TRUE
tsandl1	TRUE	TRUE	TRUE
tsandl2	TRUE	TRUE	TRUE
guarded-lock	TRUE	TRUE	TRUE
guarded-lock2	FALSE	FALSE	**TRUE**
mul-lockorders	FALSE	FALSE	**TRUE**
sem-deadlock	TRUE	TRUE	**FALSE**
barrier-deadlock	TRUE	TRUE	**FALSE**
Σ True Positives	10	9	8
Σ True Negatives	3	3	0
Σ False Positives	-	0	3
Σ False Negatives	-	1	2
Precision	-	1	0.7273
Recall	-	0.9	0.8

deadlock, it also generated a counterexample that forces the program into a deadlock.

In sqlite-1672 there is also a lock-order violation. RaceQuest detected the deadlock and generated a feasible and enforceable counterexample, but the deadlock situation was not achieved. In this case, after completing the steps defined in the counterexample, the deadlock still depends on a non-deterministic access to a shared variable that is not considered in the model or in the counterexample. ThreadSanitizer correctly emitted a warning about the lock-order violation.

The tc13 and mul-lockorders scenarios contain multiple orders to acquire the same mutexes. These lock orders never happen in parallel, due to other synchronization operations. RaceQuest takes all the synchronization operations into account and does not report any deadlock. ThreadSanitizer emitted incorrect warnings in both cases. A lock-order detector is only aware of mutex operations and produces false warnings in these two cases.

Similarly, sem-deadlock, and barrier-deadlock are deadlocks that happen when semaphore and barrier primitives are reordered. RaceQuest was able to find and reproduce both deadlocks, but ThreadSanitizer missed them completely. Two of these examples are examined more deeply in the following subsection.

In summary, the lock-order detector of ThreadSanitizer missed two deadlocks and issued three false warnings. In the cases where a lock-order detector issued a warning, it could not be used directly to reproduce the suspected deadlock, as it did not contain a scheduling sequence. RaceQuest produced no false positives, and a single false negative: sqlite-1672. For all reported positives, RaceQuest was also able to induce the deadlock in the original program using the counterexample obtained from the CSP model.

The average time used by RaceQuest for all scenarios, except tc14, is 554ms with an average trace size of 22.25 events, while for ThreadSanitizer it was 57ms. For tc14 the time is 1,921ms for RaceQuest, with 2,020 events, and 1,015ms for ThreadSanitizer. RaceQuest needs more resources, but a lock-order detector with more resources would detect the same errors. The limitations on the lock-order detector lie in the algorithm itself, as it cannot take non-mutex synchronization into account. Eighty-five percent of the time spent by RaceQuest in these scenarios is in the off-line step, which is dominated by the refinement checker.

4.8.3. Additional Detailed Examples

Here we describe two additional examples in detail, to show their CSP models and cases where a lock-order detector fails.

Multiple lock orders

This example corresponds to the scenario mul-lockorders. Figure 4.18 shows the equivalent source code. Two threads run in parallel and both acquire the mutexes `m` and `p` two times at two different moments. The threads clearly have two phases divided by the synchronization on barrier `b`.

A trace of the program appears in Figure 4.19. The calls to `barrier_wait` in the program have produced two pairs of consecutive events: barrier_enter and barrier_exit. There is no other event from the same thread between enter and exit.

RaceQuest would generate the CSP model that appears in Figure 4.20. The CSP model contains two $THREAD$ processes, the corresponding $FORK$ and

$JOIN$ processes, two $MUTEX$ processes, and a single $BARRIER$ process. Note that the $BARRIER$ process is instantiated with a maximum value of two, as indicated by the single barrier_init event in the trace.

In this case the refinement relationship holds. There is no alternative reordering where $PROGRAM$ does not reach the *endthreads* event. As there is no deadlock, no counterexample is generated.

A lock-order detector would issue a warning. It would observe the following acquisition orders in the trace: $p \to m$ and $m \to p$. But the different acquisition orders happen in non-concurrent parts of the trace: one is before the barrier and the other is after it. There is no possibility of deadlock in any alternative interleaving. A lock-order detector does not consider the barrier calls: it simply ignores them and the inter-thread ordering that they define. The machine-readable version of this example in CSP_M is in Appendix A.2.2.

```
barrier b;
mutex m, p;
void main() {
    b = barrier_init(2);
    fork(worker);
    lock(p);
    lock(m);
    unlock(m);
    unlock(p);

    barrier_wait(b);

    lock(m);
    lock(p);
    unlock(p);
    unlock(m);
    join(worker);
}

void worker() {
    lock(p);
    lock(m);
    unlock(m);
    unlock(p);

    barrier_wait(b);

    lock(m);
    lock(p);
    unlock(p);
    unlock(m);
}
```

Figure 4.18.: Program with different lock orders and a barrier

barrier_init(b, 2)
fork(t_1, t_2)
start(t_2)
lock(t_1, p)
lock(t_1, m)
unlock(t_1, m)
unlock(t_1, p)
lock(t_2, p)
lock(t_2, m)
unlock(t_2, m)
unlock(t_2, p)
barrier_enter(t_2, b)
barrier_enter(t_1, b)
barrier_exit(t_1, b)
barrier_exit(t_2, b)
lock(t_2, m)
lock(t_2, p)
unlock(t_2, p)
unlock(t_2, m)
lock(t_1, m)
lock(t_1, p)
unlock(t_1, p)
unlock(t_1, m)
end(t_2)
join(t_1, t_2)

Figure 4.19.: Trace of program in Figure 4.18

$$
\begin{aligned}
THREAD_{t_1} &= fork.t_1.t_2 \rightarrow lock.t_1.p \rightarrow lock.t_1.m \\
&\rightarrow unlock.t_1.m \rightarrow unlock.t_1.p \rightarrow barrier\_enter.t_1.b \\
&\rightarrow barrier\_exit.t_1.b \rightarrow lock.t_1.m \rightarrow lock.t_1.p \\
&\rightarrow unlock.t_1.p \rightarrow unlock.t_1.m \rightarrow join.t_1.t_2 \rightarrow SKIP \\
THREAD_{t_2} &= start.t_2 \rightarrow lock.t_2.p \rightarrow lock.t_2.m \\
&\rightarrow unlock.t_2.m \rightarrow unlock.t_2.p \rightarrow barrier\_enter.t_2.b \\
&\rightarrow barrier\_exit.t_2.b \rightarrow lock.t_2.m \rightarrow lock.t_2.p \\
&\rightarrow unlock.t_2.p \rightarrow unlock.t_2.m \rightarrow end.t_2 \rightarrow SKIP \\
INTER &= (THREAD_{t_1} \mathbin{|||} THREAD_{t_2}); \\
&endthreads \rightarrow SKIP \\
FORK_{t'} &= fork?t.t' \rightarrow start.t' \rightarrow SKIP \\
JOIN_{t'} &= end.t' \rightarrow join?t.t' \rightarrow SKIP \\
MUTEX_i &= lock?t.i \rightarrow unlock.t.i \rightarrow MUTEX_i \\
BARRIER_{b,max} &= barrier\_enter?t.b \rightarrow BARRIER\_U_{b,max,1} \\
BARRIER\_U_{b,max,i} &= barrier\_enter?t.b \rightarrow BARRIER\_U_{b,max,i+1} \\
BARRIER\_U_{b,max,max} &= barrier\_exit?t.b \rightarrow BARRIER\_D_{b,max,max-1} \\
BARRIER\_D_{b,max,i} &= barrier\_exit?t.b \rightarrow BARRIER\_D_{b,max,i-1} \\
BARRIER\_D_{b,max,0} &= BARRIER_{b,max} \\
SYNC &= FORK_{t_2} \mathbin{|||} JOIN_{t_2} \mathbin{|||} MUTEX_m \mathbin{|||} MUTEX_p \\
&\mathbin{|||} BARRIER_{b,2} \\
PROGRAM &= INTER \underset{sync\_events}{\|} SYNC \\
sync\_events &= \{|fork, start, join, end, lock, rdlock, unlock, signal, \\
&broadcast, wait, barrier\_enter, barrier\_exit, sem\_post, \\
&sem\_wait|\} - independent \\
independent &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
LIVE &= live \rightarrow LIVE \\
LIVE &\sqsubseteq_{\mathrm{F}} (PROGRAM \Theta_{endthread} SKIP) \setminus \Sigma \,;\, LIVE
\end{aligned}
$$

Figure 4.20.: CSP model of the trace in Figure 4.19 with multiple lock orders and a barrier

Semaphore deadlock

This example corresponds to the scenario sem-deadlock. Figure 4.21 shows the equivalent source code. Two threads run in parallel and they synchronize using a semaphore. The developer's intention here is that `first_function` is called before `second_function`, and that `third_function` is called after `second_function`. To generate this order, the thread `worker` should wait until the thread `main` allows it to continue. Afterwards it is the main thread that waits until the worker allows it to continue.

A trace of the program appears in Figure 4.22. This trace shows the intended order of the synchronization events. The thread t_1 calls sem_post, allowing t_2 to complete sem_wait, which issues another sem_post and allows t_1 to complete its own sem_wait.

RaceQuest would generate the CSP model that appears in Figure 4.23. The CSP model contains two $THREAD$ processes, the corresponding $FORK$ and $JOIN$ processes, and a single $SEMPAHORE$ process. Note that the process $SEMAPHORE$ is instantiated with a maximum value of ten and an initial value of zero, as indicated by the single sem_init event in the trace.

In this case the refinement relationship does not hold, because there is an alternative reordering where $PROGRAM$ does not reach the *endthreads* event. A counterexample that reveals a deadlock is the following:

$$\langle fork.t_1.t_2, start.t_2, sem\_post.t_1.s, sem\_wait.t_1.s\rangle$$

In this case, after `main` has increased the semaphore, it decreases it again with its own `sem_wait` call. When `worker` calls `sem_wait`, the semaphore counter is zero again, and the `worker` thread will be blocked forever. Then `main` will be blocked at the `join` call. With both threads blocked the program is in a deadlock.

A lock-order detector would generate no warnings in this case, i.e. false negatives. It would neither take the wait at the semaphore nor the end-join order into consideration. The machine-readable version of this example in CSP$_M$ is in Appendix A.2.3.

```
semaphore s;
void main() {
    s = sem_init(0, 10);
    fork(worker);
    first_function();
    sem_post(s);

    long_computation();

    sem_wait(s);
    third_task();
    join(worker);
}

void worker() {
    do_work();
    sem_wait(s);

    second_task();

    sem_post(s);
    do_work();
}
```

Figure 4.21.: Program with different lock orders and a barrier

sem_init(b, 0, 10)
fork(t_1, t_2)
start(t_2)
sem_post(t_1, s)
sem_wait(t_2, s)
sem_post(t_2, s)
sem_wait(t_1, s)
end(t_2)
join(t_1, t_2)

Figure 4.22.: Trace of program in Figure 4.21

$$
\begin{aligned}
THREAD_{t_1} &= fork.t_1.t_2 \rightarrow sem\_post.t_1.s \rightarrow sem\_wait.t_1.s \\
&\rightarrow join.t_1.t_2 \rightarrow SKIP \\
THREAD_{t_2} &= start.t_2 \rightarrow sem\_wait.t_2.s \rightarrow sem\_post.t_2.s \\
&\rightarrow end.t_2 \rightarrow SKIP \\
INTER &= (THREAD_{t_1} \mid\mid\mid THREAD_{t_2}); \\
&endthreads \rightarrow SKIP \\
FORK_{t'} &= fork?t.t' \rightarrow start.t' \rightarrow SKIP \\
JOIN_{t'} &= end.t' \rightarrow join?t.t' \rightarrow SKIP \\
SEMAPHORE_{s,0,max} &= sem\_post?t.s \rightarrow SEMAPHORE_{s,1,max} \\
SEMAPHORE_{s,i,max} &= sem\_post?t.s \rightarrow SEMAPHORE_{s,i+1,max} \\
&\Box\ sem\_wait?t.s \rightarrow SEMAPHORE_{s,i-1,max} \\
SEMAPHORE_{s,max,max} &= sem\_wait?t.s \rightarrow SEMAPHORE_{s,max-1,max} \\
SYNC &= FORK_{t_2} \mid\mid\mid JOIN_{t_2} \mid\mid\mid SEMAPHORE_{s,0,10} \\
PROGRAM &= INTER \underset{sync\_events}{\mid\mid} SYNC \\
sync\_events &= \{\mid fork, start, join, end, lock, rdlock, unlock, \\
&signal, broadcast, wait, barrier\_enter, \\
&barrier\_exit, sem\_post, sem\_wait \mid\} - independent \\
independent &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
LIVE &= live \rightarrow LIVE \\
LIVE &\sqsubseteq_{\mathrm{F}} (PROGRAM \Theta_{endthread} SKIP) \setminus \Sigma\ ; LIVE
\end{aligned}
$$

Figure 4.23.: CSP model of the trace in Figure 4.22 with a semaphore

4.8.4. Summary

In this chapter we presented RaceQuest and used it to detect deadlocks. The main idea of RaceQuest is to use a trace of a successful single execution to generate a model in the CSP process algebra. The model represents each thread and the synchronization constructs as individual processes. The semantics of each synchronization construct are enforced by a concrete process, which defines the valid orders of events between threads. The model not only contains the observed trace but also plausible reorderings, i.e. alternative interleavings that the program could take.

We explore the model using a refinement check to find deadlocks in alternative interleavings. The key to detect the deadlock is to map a program deadlock to a CSP-deadlock. The detected deadlock is obtained with an interleaving that can be used as schedule to replay the deadlock.

We evaluated the approach in 13 scenarios and compared it with a lock-order detector. RaceQuest obtained a higher precision and recall than the other detector.

We saw the limitations of RaceQuest and the steps taken to mitigate them. Some concurrency mechanisms are unsupported and would require additional annotations. The model also contains infeasible reorderings, which are discarded by the replay step. Longer traces imply bigger models; to handle the state explosion we split the trace to control the size at the cost of more false negatives.

The following chapter expands RaceQuest to enable race detection. The core of RaceQuest continues to be a CSP model that generalizes a trace. New events to represent memory accesses are added, as well as extra steps in the workflow to manage the huge number of race candidates.

5. Data Race Detection

This chapter describes the use of RaceQuest for data race detection and the extensions needed. RaceQuest performs the same steps: tracing, CSP model generation, checking, and replay, but with some modifications. Checking for data races requires monitoring and testing all memory accesses performed by the program. The number of memory accesses in a program is huge; to manage this, a few extensions in RaceQuest reduce the size of the trace. The error check must look for data races: concurrent read-write or write-write accesses. These patterns are defined as a new property specification, a CSP refinement check. As in deadlocks, RaceQuest will infer new interleavings and possible races in the CSP model.

Figure 5.1 displays the additional steps in the RaceQuest workflow, introduced before and after the trace step. Memory grouping computes the memory overlap between memory accesses off-line, relieving the model checker of this task, which is described in Section 5.2.2. Reduction refers to the race candidate elimination algorithms presented in Section 5.5.

5.1. Motivational Data Race Example

A data race is a situation when two threads access the same memory address simultaneously, and at least one access is a write. A program with a possible data race appears in Figure 5.2. In the program two threads increment two shared variables `x` and `y`. While variable `y` is correctly protected, variable `x` is not. Three possible interleavings appear in Figure 5.3. A happens-before detector output would depend on the observed interleaving. The detector

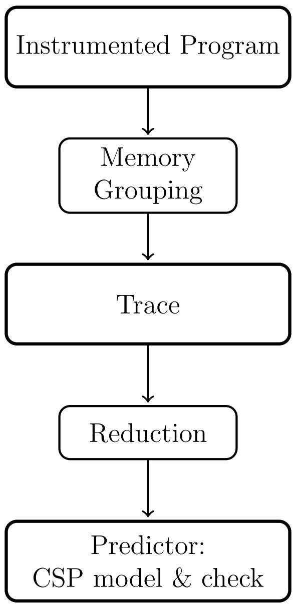


Figure 5.1.: Auxiliary steps in RaceQuest workflow

would not find the race in the interleaving in Figure 5.3a, because a happens-before relation between the `unlock` and `lock` calls hides the race on `x`. But the race would be detected in the interleavings in Figures 5.3b and 5.3c, because the accesses to `x` are not happens-before ordered. With RaceQuest the goal is to generate a racy interleaving from any of the three interleavings as input, where the conflicting accesses are not happens-before ordered. In other words, and from the scenario in Figure 5.3a, to obtain any of the schedules of Figures 5.3b or 5.3c. Any one of these two schedules is enough to reveal the failure.

5.2. Incorporating Memory Accesses

In the previous chapter RaceQuest captured synchronization events to be aware of possible inter-thread orderings. Race detection must track which memory operations take place in the program, so we need to incorporate them into the trace and CSP models. This section describes how they are extracted and defined in the trace; the following section introduces them in the CSP model.

First, for each memory operation we need to know the type (read or write), the starting address, and the size, because a single instruction can affect multiple bytes. A pair of memory accesses will race if their memory blocks overlap. We capture a trace including the executed read and writes, along with the performing thread, address, and size, as explained in Section 5.2.1.

```
int x = 0, y = 0;
mutex m;
void main() {
    fork(worker);
    x++;
    lock(m);
    y++;
    unlock(m);
    join(worker);
}

void worker() {
    lock(m);
    y++;
    unlock(m);
    x++;
}
```

Figure 5.2.: Program with a data race

If we use the address and size in the CSP model directly to check for races, the model size would increase greatly, because the required if-conditions are very expensive. Instead, we pre-compute the overlaps off-line in an auxiliary step before generating the final trace. With the pre-computed overlaps, the CSP model does not have to perform this comparison. This step is described in Section 5.2.2.

The final trace will not have any reference to address or size of the accesses. Instead the memory events refer to unique identifiers for overlapping blocks of memory, which we call memory intervals. Section 5.2.3 explains the extended trace model.

5.2.1. Capturing Memory Events

The capturing of events is an extension of the trace capture explained in Section 4.3.2. Memory accesses are also captured using instrumentation in LLVM IR. Each load and store instruction is a tracing point, so before each instruction an instrumenting function call is added. The instrumentation function will emit if it is a read (load) or a write (store), as well as the performing thread, the memory address, and the size of the memory access. The instrumentation excludes the following cases:

main	worker
fork(worker)	
x++	
lock(m)	
y++	
unlock(m)	
	lock(m)
	y++
	unlock(m)
	x++
join(worker)	

(a)

main	worker
fork(worker)	
	lock(m)
	y++
	unlock(m)
	x++
x++	
lock(m)	
y++	
unlock(m)	
join(worker)	

(b)

main	worker
fork(worker)	
x++	
	lock(m)
	y++
	unlock(m)
	x++
lock(m)	
y++	
unlock(m)	
join(worker)	

(c)

Figure 5.3.: Different interleavings for program in Figure 5.2

- inside a single LLVM basic block a load before a store on the same target without any function call in-between. Without any function call between both instructions, there is no synchronization. When the load races with any other instruction, the store will also race. Both instructions will always have the same interleaving. It is enough to detect one race to have an interleaving for both races.
- operations where the target does not escape, i.e. if the target is not accessed outside the current scope. The capture tracking algorithm [LLV] integrated in LLVM determines this.

Also, during runtime all memory accesses before the first fork event in the program are ignored. At the beginning of the execution there is always a single thread, and its accesses cannot race with any other future access.

5.2.2. Memory Grouping in Intervals

After capturing the trace, we translate the addresses and sizes of memory accesses to the more convenient memory intervals. The captured trace differs from the trace model in the parameters of the memory events.

We define a memory interval as the longest sequence of consecutive memory positions that are always accessed by the same threads. A memory interval could represent a single byte, if any thread addresses it individually at any point, or multiple bytes, e.g. the bytes of an integer variable that is always accessed as a whole. No two memory intervals overlap, and each byte addressed by the program is contained in a single memory interval.

The algorithm in Figure 5.4 describes how we transition from memory events with address and size to memory events with interval ids. First, for each memory access a tuple is defined with the starting address, the final address (sum of starting and size minus one), operation (read or write), and the set of threads accessing it (initially each tuple only has the performing thread). These intervals are stored in the set *intervals*, sorted by their starting address. Then iteratively the following conditional rules are applied to each pair of consecutive intervals:

- if the two intervals have the same operation and exactly the same range (start and end addresses), they are combined into a single one. Note that the set of accessing threads can be different, but in the combined interval the sets of threads are joined.
- if they are strictly consecutive, i.e. one starts where the other ends, and the operation and threads involved are the same, they are fused into a larger interval.
- if they overlap, they are split into smaller non-overlapping intervals. These intervals can subsequently be merged with other intervals.

This process generates a list of non-overlapping intervals of memory addresses, which are accessed by the threads with read or write operations. A unique id is assigned to each element in the set *intervals*. Then each memory access in the trace is replaced by a new set of memory accesses targeting the corresponding memory intervals.

Subsequently we can remove some memory intervals and related memory accesses from the trace if any of these conditions apply:

- a memory interval is only referenced by read events. A memory interval must be written at some point to have a race.

```
Let $intervals := \emptyset$
for each Memory access with address $a$, size $s$, operation $o$ done by thread $t$ in the trace do
    Add $(a, e, o, tids)$ to $intervals$ where $e = a + s - 1, tids = \{t\}$
end for
Sort $intervals$ by the address $a$ of the tuples
for each Two consecutive intervals $m$, $n$ in $intervals$ do
    if $m_a = n_a \wedge m_e = n_e \wedge m_o = n_o$ then
        Add $(m_a, m_e, m_o, m_{tids} \cup n_{tids})$ to $intervals$ at $m$ position
        Remove $m$ and $n$ from $intervals$
    else if $m_e = n_a \wedge m_o = n_o \wedge m_{tids} = n_{tids}$ then
        Remove $n$ from $intervals$
        $m_e := n_e$
        $m_{tids} := m_{tids} \cup n_{tids}$
    else if $m_e > n_a$ then
        Add $(m_a, n_a, m_o, m_{tids})$ to $intervals$ at $m$ position
        Add $(n_e, m_e, m_o, m_{tids})$ to $intervals$ at $m$ position
        Add $(n_a, m_e, n_o, n_{tids})$ to $intervals$ at $m$ position
        Add $(m_e, n_e, n_o, n_{tids})$ to $intervals$ at $m$ position
        Remove $m$ and $n$ from $intervals$
    end if
end for
```

Figure 5.4.: Memory interval algorithm

- a memory interval is only referenced by a single thread. A memory interval accessed by a single thread cannot have a race.

Grouping Example

Figure 5.5 shows an example of memory interval grouping. Thread t_1 writes at address 0, size 1, address 1, size 3, and address 4, size 3. Thread t_2 writes at address 2, size 2, and reads at address 4, size 1. The corresponding memory interval ids are in the right column. The rows reflect which memory addresses are included in the interval. We see that address 0 and 1 have been merged in interval 1, because they are only accessed by t_1. Addresses 2 and 3 conform interval 2, as they are accessed by t_1 and t_2

After computing the memory intervals, we replace the address and size of each memory operation with the corresponding memory interval id. If an original memory operation affects more than one memory interval, a new memory event per affected memory interval is added just after the original. For example, the write by t_1 at address 4, size 3, is replaced by two operations, a write by t_1 on interval 3 and a write by t_1 on interval 4. The read by t_2 at address 0, size1, is replaced by a read by t_2 on interval 3.

To compare if two memory operations affect the same memory, instead of checking for overlapping, we only need to compare the memory interval ids. The write by t_1 on interval 3 and the read by t_2 on interval 3 affect the same memory interval 3 and they could conflict.

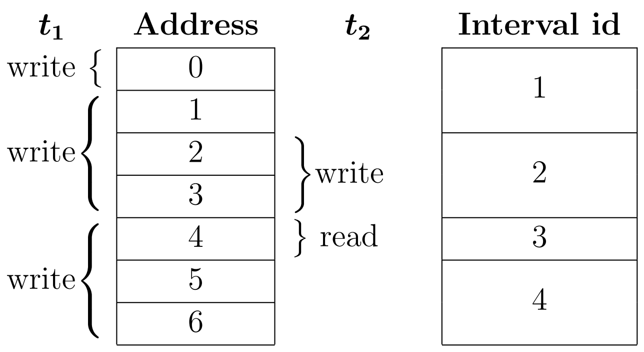


Figure 5.5.: Memory interval example

5.2.3. Extending the Trace Model with Memory Events

The final trace model is an extension of the model presented in Section 4.3. Table 5.1 lists the additional events included in the model.

Table 5.1.: Memory access events

Event	Description
read(t, r)	thread t reads memory interval r
write(t, r)	thread t writes memory interval r

Read and write represent types of access to a memory interval by a thread. Atomic accesses will always generate a read event in addition to the atomic event, and do it independently if they are atomic reads or writes. Atomic accesses can only race with non-atomic writes, so considering them as reads simplifies the classification of events. None of the other rules and events presented about the trace model in Section 4.3 are modified.

5.2.4. Example of a Trace

Figure 5.6 shows the final trace of an execution of the racy program in Figure 5.2, concretely the corresponding to the interleaving in Figure 5.3a.

fork(t_1, t_2)
start(t_2)
write(t_1, r_x)
lock(t_1, m)
write(t_1, r_y)
unlock(t_1, m)
lock(t_2, m)
write(t_2, r_y)
unlock(t_2, m)
write(t_2, r_x)
end(t_2)
join(t_1, t_2)

Figure 5.6.: Trace of program in Figure 5.2

The thread `main` is t_1 and the thread `worker` is t_2. In the program the variables `x` and `y` are in disjoint memory regions, so their corresponding intervals will not overlap. In this case the resulting memory intervals are mapped one-to-one to the variables in the program, the memory interval r_x contains the addresses of `x` and r_y the addresses of `y`. Each auto-increment in the original program implies a read and a write on the same memory interval. The reads are removed from the trace as the writes are sufficient to find the race.

5.3. CSP Model & Data Race Checking

As explained previously in Section 4.4, the CSP model is derived from the trace. The only addition is the new read and write events. These events are also mapped one-to-one to CSP events. Two new channels are defined, read and write, where the first field is the thread id, and the second field the id of the memory interval. Read and write events are not synchronization events, so they only appear in the $THREAD_i$ processes.

5.3.1. Data Race Checking

To find the data races in the CSP model, we need a definition of a data race in terms that we can translate to the model. Two events $a = op_a(t_a, k_a)$ and $b = op_b(t_b, k_b)$ in a trace could conflict if:

- both are memory accesses, $op_a, op_b \in \{read, write\}$.
- one is a write, $op_a = write \vee op_b = write$.
- the thread id on both events is different, $t_a \neq t_b$.
- both affect the same memory interval, $k_a = k_b$.
- a and b are concurrent, $a||b$. If there is no other event between a and b in the trace, they are concurrent. If we consider the trace a total order of elements, then $\nexists x | a < x < b \Rightarrow a||b$. This requirement for concurrency between two memory accesses is strict but sufficient, as the process $PROGRAM$ contains all possible combinations of the events. For example, this requirement is fulfilled in the interleaving in Figure 5.3b, where the auto-increments on x are seen one after the other.

If we want to find all the data races we should check the above properties for each pair of events in the trace, i.e. a search in the model for each pair. Instead we approximate, and assume that it is enough to find one race per memory interval. In this case we build a single check per memory interval, instead of per pair of memory events.

To translate the previous conditions to CSP, we need to find if, in the set $traces(PROGRAM)$, there is any sequence that contains one of the following subsequences for the memory interval k:

$$\langle write.t.k, write.t'.k\rangle \vee \langle read.t.k, write.t'.k\rangle \quad | \quad t \neq t'$$

We must also describe this property as a refinement expression. The following is a refinement relationship that holds if the memory interval k is data race free:

$$\begin{aligned}
race\_events_k &= \{read.t.k | t \in threads\} \cup \\
&\qquad \{write.t.k | t \in threads\} \cup sync\_events \\
RACE_k &= RACE\_ERR_k \triangle (\Box_{z:race\_events_k}\, z \rightarrow RACE_k) \\
RACE\_ERR_k &= read?t.k \rightarrow write?t' : threads - \{t\}.k \rightarrow race \rightarrow SKIP \\
&\quad \Box\; write?t.k \rightarrow write?t' : threads - \{t\}.k \rightarrow race \rightarrow SKIP \\
STOP &\sqsubseteq_T (PROGRAM \underset{race\_events_k}{||} RACE_k) \upharpoonright \{race\}
\end{aligned}$$

We build the implementation process as a combination of $PROGRAM$ and a process $RACE_k$ that will monitor the data race subsequences on the memory interval k. The process $RACE$ will emit a single $race$ event in case the patterns is matched. $RACE_k$ can monitor the relevant events of $PROGRAM$, thanks to the parallel composition. The relevant events in the monitoring are the synchronization events and memory accesses related to the memory interval k. We ignore all the events performed by the composition except the warning event *race*, using the project operator $\upharpoonright$. The specification process is only composed of empty traces, $traces(STOP) = \emptyset$. So, if $RACE_k$ reaches the *race* event, the refinement does not hold and a counterexample exists. Note that for data races the trace semantic model of CSP is sufficient. In Chapter 4 we used the failures model for deadlock detection.

The key elements in $RACE_k$ are the racy sequences, read-write and write-write, described as alternative paths in $RACE\_ERR_k$. When any of these paths are completed, the *race* event is emitted. The process $RACE_k$ returns to process $RACE_k$ through the interrupt operator $\triangle$. The interrupt operator is used to reset the pattern and avoid the process blocking itself when confronted with another event. Any combination of events can be the prefix of the racy pair. The equivalent labeled transition system for $RACE_k$ is shown in Figure 5.7.

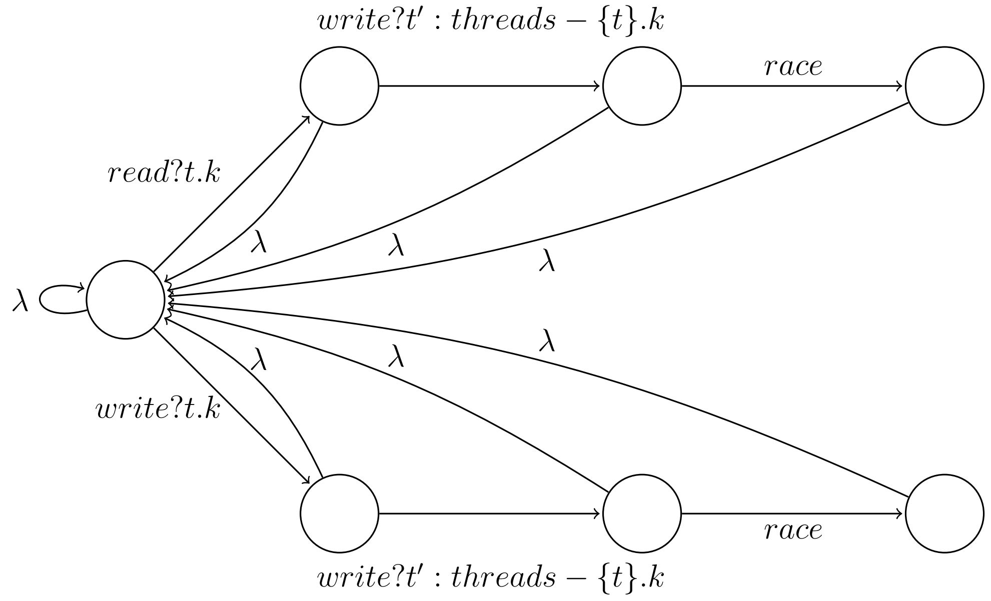


Figure 5.7.: Labeled transition system for process $RACE_k$. Where $\lambda \in race\_events_k$

Each memory interval k must be checked by its own refinement relationship. But the definitions of $RACE_k$ and $RACE\_ERR_k$ are the same; only the refinement itself must be instantiated for each memory interval. For each refinement that does not hold we obtain a counterexample, i.e. a counterexample per memory interval.

5.3.2. Model Example

Figure 5.8 contains the CSP model along with the data race refinement check for the trace in Figure 5.6. We observe that the structure of the processes that define $PROGRAM$ are the same as in the deadlock case. In the $THREAD_i$ processes the new memory events appear, e.g. $write.t_1.r_y$ is the write by t_1 on the memory interval r_y.

As there are two memory intervals present in the model, r_x and r_y, we have two refinement relationships too. For r_y the refinement holds, as the memory accesses to r_y are correctly protected and the lock m avoids the two writes appearing contiguously. For r_x the refinement does not hold, because there is a sequence of events that triggers the event *race*:

$$\langle fork.t_1.t_2, start.t_2, lock.t_2.m, write.t_2.r_y, unlock.t_2.m, write.t_2.r_x, write.t_1.r_x\rangle$$

The counterexample is equivalent to the interleaving shown in Figure 5.3b, where the worker thread acquires the mutex first. In this case the two auto-increments on variable x appear one after the other. The memory interval r_x corresponds to the variable x in the example. As the two memory accesses to r_x appear together, they are not *happens-before* ordered, because any of them could be the first. The machine-readable version of this example in CSP$_M$ to be executed with the FDR3 refinement checker is in Appendix A.2.4.

5.4. Counterexample and Reproduction

The counterexample obtained from $PROGRAM$ is composed of synchronization and memory events. Because of the reductions in Sections 5.2 and 5.5, neither the trace nor the CSP model contain a memory event per memory instruction executed. During reproduction it is not possible to match and enforce all performed memory operations with events in the counterexample. Instead, the memory events on the counterexample are completely ignored, i.e. they do not work as enforcing points. Only the synchronization events are enforced in the reproduction. The reproduction operates exactly as described in Section 4.6.

For deadlocks a successful reproduction leads to the program being deadlocked. For data races a successful reproduction leads to a state where the memory

$$\begin{aligned}
THREAD_{t_1} &= fork.t_1.t_2 \rightarrow write.t_1.r_x \rightarrow lock.t_1.m \rightarrow write.t_1.r_y \rightarrow \\
&\quad unlock.t_1.m \rightarrow join.t_1.t_2 \rightarrow SKIP \\
THREAD_{t_2} &= start.t_2 \rightarrow lock.t_2.m \rightarrow write.t_1.r_y \rightarrow unlock.t_2.m \rightarrow \\
&\quad write.t_2.r_x \rightarrow end.t_2 \rightarrow SKIP \\
INTER &= (THREAD_{t_1} \mid\mid\mid THREAD_{t_2}) \,;\, endthreads \rightarrow SKIP \\
FORK_{t'} &= fork?t.t' \rightarrow start.t' \rightarrow SKIP \\
JOIN_{t'} &= end.t' \rightarrow join?t.t' \rightarrow SKIP \\
MUTEX_i &= lock?t.i \rightarrow unlock.t.i \rightarrow MUTEX_i \\
SYNC &= FORK_{t_2} \mid\mid\mid JOIN_{t_2} \mid\mid\mid MUTEX_m \\
PROGRAM &= INTER \underset{sync\_events}{\parallel} SYNC \\
sync\_events &= \{| fork, start, join, end, lock, rdlock, unlock, signal, \\
&\quad broadcast, wait, barrier\_enter, barrier\_exit, sem\_post, \\
&\quad sem\_wait |\} - independent \\
independent &= \emptyset
\end{aligned}$$

$$\begin{aligned}
race\_events_k &= \{read.t.k | t \in threads\} \cup \{write.t.k | t \in threads\} \\
&\quad \cup sync\_events \\
RACE_k &= RACE\_ERR_k \triangle (\Box_{z:race\_events_k}\, z \rightarrow RACE_k) \\
RACE\_ERR_k &= read?t.k \rightarrow write?t' : threads - \{t\}.k \rightarrow race \rightarrow SKIP \\
&\quad \Box\; write?t.k \rightarrow write?t' : threads - \{t\}.k \rightarrow race \rightarrow SKIP \\
STOP &\sqsubseteq_{\mathrm{T}} (PROGRAM \underset{race\_events_{r_x}}{\parallel} RACE_{r_x}) \upharpoonright \{race\} \\
STOP &\sqsubseteq_{\mathrm{T}} (PROGRAM \underset{race\_events_{r_y}}{\parallel} RACE_{r_y}) \upharpoonright \{race\}
\end{aligned}$$

Figure 5.8.: CSP model of the trace in Figure 5.6

interval is accessed by two threads in a non-happens-before order. To confirm that the accesses are not happens-before ordered, we execute a pure happens-before detector in parallel during the reproduction. The detector will compute the vector clock of the accesses and confirm if the race exists or not, as described in Section 2.3.1. If the detector emits no warning, the counterexample is discarded. A case where the race does not exist under a feasible counterexample is show in Section 5.4.2.

RaceQuest does not store any information that links a memory event with the original source code line. The reductions on the trace would increase the imprecision of this information. Instead we rely on the reproduction step to obtain more information about the race: stack trace, code lines, memory addresses, etc. The race detector executed in parallel will report all this information in a known format.

5.4.1. Reproduction Example

If we take the counterexample from the model in Figure 5.8 for memory interval r_x and we ignore the memory accesses events, we have the following sequence:

$$\langle fork.t_1.t_2, start.t_2, lock.t_2.m, unlock.t_2.m\rangle$$

During reproduction the program is going to behave like the interleaving shown in Figures 5.3b or 5.3c. In Figure 5.3b the accesses to `x` are seen together, as in the assumption for the data race pattern. But in Figure 5.3c they are not together, as the critical section of the worker thread is seen in between. It does not matter which of these two interleavings are observed in the reproduction. In both cases the accesses to `x`, memory interval r_x, are not happens-before ordered.

5.4.2. No Race Under Feasible Reordering

The use of a happens-before detector helps to distinguish the situation where the counterexample is feasible but a suspicious data race does not exist. An example is the program in Figure 5.9.

This program can mainly exhibit two interleavings, as in Figure 5.10, depending on which thread first executes the access to the critical section. If the thread `main` executes the critical section first, Figure 5.10a, then the thread `worker` updates `y` to two and executes the access to `x`. In this case the accesses to `x` are happens-before ordered. If `worker` executes the critical section first, as in

```
int x = 0, y = 0;
mutex m, p;
void main() {
    fork(worker);
    x++;
    lock(m);
    y++;
    unlock(m);
    join(worker);
}

void worker() {
      int flag = 0;
      lock(m);
      y++;
      flag = y;
      unlock(m);
      if (flag > 1) {
          x++;
      }
}
```

Figure 5.9.: Feasible program with multiple control flow

Figure 5.10b, it will not execute the if-body as the condition is false, because the value of `flag` is one.

The result of RaceQuest depends on the input trace and the program paths covered by the trace. If the input trace is the one in Figure 5.10b, there is single thread accessing `x` and no data race is reported. If the input trace is the one in Figure 5.10a, then a race on `x` is reported with a simplified counterexample:

$$\langle fork.t_1.t_2, start.t_2, lock.t_2.m, unlock.t_2.m\rangle$$

This counterexample is feasible. The counterexample forces the program to behave like the interleaving in Figure 5.10a during reproduction. Without an additional check, we would report that there is a data race on `x`. But the parallel happens-before detector will report no race at all, so we can also discard this counterexample.

main	worker
fork(worker)	
x++	
lock(m)	
y++	
unlock(m)	
	lock(m)
	y++
	flag = y
	unlock(m)
	if (flag > 1)
	x++
join(worker)	

(a)

main	worker
fork(worker)	
	lock(m)
	y++
	flag = y
	unlock(m)
	if (flag > 1)
x++	
lock(m)	
y++	
unlock(m)	
join(worker)	

(b)

Figure 5.10.: Different interleavings for program in Figure 5.9

5.5. Reduction Techniques

The memory accesses executed by a program outnumber the synchronization operations. The complexity of the model increases with the number of events, which requires more resources and eventually makes the model intractable. Capturing, storing, and processing a trace with a high number of events is also costly. It is imperative to minimize the number of events to reduce the consumed resources and to support larger programs and traces. To achieve this goal we have applied several additional strategies along the RaceQuest workflow. Trace windowing is a general technique already shown in the previous chapter. As we know the characteristics of data races we can apply specific reduction techniques.

5.5.1. On-Line Redundant Accesses Removal

Memory accesses from different instructions to the same memory address in the same thread segment generate redundant memory events. For example, in Figure 5.11 there are two auto-increments. The statement `x++` accesses the four bytes of `x`. But the statement `*ptr++` modifies a single byte at an unknown address. If the pointer `ptr` points to variable `x`, e.g. `ptr = &x`, and `x` races with another part of the program, then the access through the pointer will also race. The scheduling for the two races would be the same, as they are in the same thread segment. A single counterexample would reveal both races. In this case the memory event generated by the pointer access is redundant for race detection.

```
int x = 0;
char *ptr;
void increment() {
    x++;
    *ptr++;
}
```

Figure 5.11.: Example with redundant accesses

We remove redundant memory accesses on-line using a cache. During trace capture the memory events are not emitted directly, instead they are stored in a small thread-local cache. Each time a memory access is going to be added to the cache, we check if it overlaps any access already stored in the cache. If the whole memory block accessed by the new access is contained in a previous memory access, the new one is ignored and not added to the cache. When the cache is full or the thread segment ends, because a synchronization operation is reached, then the cache is immediately flushed. Afterwards, the corresponding memory access event is generated for each entry in the cache.

In the example the access to `*ptr++` happens after `x++`, and the memory addresses that it uses are a subset of the already contained accesses of `x++`. The memory events of `*ptr++` can be ignored. When the value of `ptr` is not the address of `x` or one of its bytes, as this reduction works at runtime, the event generated by the memory accesses of `*ptr++` would not overlap the memory events generated by `x++`. In the reverse case, if `x++` happens after `*ptr++`, then all the memory accesses of both statements would be in the cache. The cache would contain redundant events and they would be emitted. This redundancy can still be removed by the next techniques.

5.5.2. Hybrid Algorithm Data Race Filtering

Exploring the CSP model for a race is more expensive than comparing two vector clocks for a happens-before relationship or comparing two locksets. If all accesses to a memory interval are protected by a common lock in the input trace, the CSP check is not going to report any race. Similarly, accesses ordered by a fork-start are going to be ordered in all interleavings in the CSP model.

We use a weakened version of the happens-before algorithm[1] combined with

[1]A version that ignores the happens-before edges of the mutex operations.

the lockset algorithm to remove memory accesses that cannot race in any interleaving in the CSP model. Alone and in combination, these algorithms produce false positives. But false positives are not a problem, as they will be checked later by the CSP model. False negatives are not introduced, as the *weak* happens-before and lockset algorithms are conservative. This step is applied after obtaining the trace model and computing the memory intervals.

The combined algorithm is shown in Figure 5.12. Each memory event m is compared against any other memory event n. First, we check if they are from different threads, target the same memory interval and at least one is a write. Then we apply both algorithms: the weak happens-before and the lockset algorithms. If the events are parallel due the weak happens-before relationship or do not share any lock, then they are added to the set *candidate*. All events that are not present in the set *candidate* are removed from the trace and will not be present in the CSP model. Only the events in the set *candidate* are considered candidates for a possible data race.

```
Let $candidates = \emptyset$ be the set of possible racy memory accesses
for each Memory access $m$ in the trace do
    for each Memory access $n$ in the trace where $m \neq n$ do
        if $\begin{pmatrix} m_{tid} \neq n_{tid} \land m_{interval} = n_{interval} \land \\ (m_{op} = write \lor n_{op} = write) \end{pmatrix}$ then
            if $VC(m) \not< VC(n) \land VC(n) \not< VC(m)$ then
                if $lockset(m) \cap lockset(n) = \emptyset$ then
                    Add $m$ to candidates
                    Add $n$ to candidates
                end if
            end if
        end if
    end for
end for
Remove from the trace each memory access $m$, such $m \notin candidates$
```

Figure 5.12.: Hybrid filtering algorithm

Firstly, we check if both accesses could be parallel using vector clocks for a weak happens-before relationship. The properties of the vector clocks are explained in Section 2.3.1. In the standard algorithm the happens-before edges are constructed for all synchronization constructs. In this weak version we only construct the edges for the following constructs: fork-start, end-join, signal-wait and broadcast-wait. Memory accesses ordered by only these constructs in the trace will be always ordered by them in the CSP model. It is less expensive to prune these memory accesses with vector clocks. This modification removes false negatives from the original happens-before algorithm, at the cost of producing false positives.

Secondly, we run a standard lockset analysis, as described in Section 2.3.2. The differences here only are in the implementation, where we have two locksets per memory accesses, one for read locks and one for write locks. The locksets of memory accesses are compared against each other. We do not use an accumulative $C(x)$ lockset per memory interval. The lockset analysis will remove the memory accesses from the trace to an interval that is always correctly protected. The remaining accesses will be checked in the CSP model.

Figure 5.14 illustrates this hybrid algorithm with an example. Two threads concurrently access the intervals a, b, and c. The original trace is on the left, Figure 5.13a. The filtered trace is on the right, Figure 5.13b. The two accesses to c are *weak* happens-before ordered by the end-join of t_2, so they are removed. All the accesses to b are protected by the mutex m, so they cannot be in a race on b and are removed. The accesses to a are neither weak happens-before ordered, because the mutexes does not impose any weak happens-before edge, nor protected by a common mutex. The accesses to a are not filtered, and they will be checked in the CSP model.

fork(t_1, t_2)	fork(t_1, t_2)
start(t_2)	start(t_2)
write(t_1, a)	write(t_1, a)
lock(t_1, m)	lock(t_1, m)
write(t_1, b)	
unlock(t_1, m)	unlock(t_1, m)
lock(t_2, m)	lock(t_2, m)
write(t_2, b)	
unlock(t_2, m)	unlock(t_2, m)
write(t_2, c)	
write(t_2, a)	write(t_2, a
end(t_2)	end(t_2)
join(t_1, t_2)	join(t_1, t_2)
write(t_1, c)	
(a) Original	**(b)** Filtered

Figure 5.14.: Hybrid filtering algorithm example

This filtering reduces the size of the CSP model and the number of checks.

5.5.3. Same Thread Segment Reduction

After the previous filtering with the hybrid algorithm, it is still possible to have multiple racy memory intervals that will generate the same counterexample. It is a similar case to the motivation for redundant access removal in Section 5.5.1. When two memory intervals have exactly the same memory accesses in the same set of thread segments in the trace, and there is a race in one of the intervals, then there will be a race in the other interval. The counterexamples will be equivalent, containing the same order of synchronization operations.

To avoid redundant refinement checks, we check the thread segment equivalence of each pair of memory intervals before the CSP model generation. The algorithm appears in Figure 5.15. Each thread segment in the trace is numbered. For each memory interval we define two sets, one for thread segments with only a read access and other with the thread segments with at least one write access to this memory interval. Then we check in each thread segment which memory intervals are written and which are only read. The thread segment is added to the appropriate set of each memory interval. Then all memory intervals are compared in pairs. If two memory intervals have exactly the same thread segment sets, the memory accesses to one of them are removed from the whole trace. One memory interval effectively disappears from the trace.

```
Let $seg\_written_k = \emptyset$ and $seg\_read_k = \emptyset$ sets of thread segments for memory
interval $k$
for each Thread segment $ts$ in the trace do
    if Memory interval $r$ is written in $ts$ then
        Add $ts$ to $seg\_written_r$
    else if Memory interval $r$ is read in $ts$ then
        Add $ts$ to $seg\_read_r$
    end if
end for
for each Memory intervals $r$ and $s$ in the trace do
    if $seg\_written_r = seg\_written_s \wedge seg\_read_r = seg\_read_s$ then
        Remove all memory accesses to $s$ from the trace
    end if
end for
```

Figure 5.15.: Thread segment reduction algorithm

One example is shown in Figure 5.16. In Figure 5.16a there is a trace, where two threads access the intervals a, b, and c (the events pertinent to each thread have been justified to each side). The thread t_1 has three relevant thread segments:

ts_1, ts_2, and ts_4. Before the fork and after the join there are also thread segments, but without any memory accesses. The thread t_2 has one relevant thread segment, ts_3; again, before the lock and after the unlock there are also thread segments but they are empty. In Figure 5.16b we see which memory intervals are accessed in each thread segment. Intervals a and b have exactly the same thread segment sets. The counterexample for a will also trigger the race in b, so we can safely remove b from the trace and not check it. However, interval c has different sets and will have to be checked. The number of race checks required is reduced from three to two.

$$
\begin{array}{rlrl}
 & \text{fork}(t_1,\, t_2) & & \\
 & & \text{start}(t_2) & \\
ts_1\left\{\begin{array}{l}\text{write}(t_1,\, a)\\ \text{write}(t_1,\, b)\end{array}\right. & & & \\
 & \text{lock}(t_1,\, m) & & \\
ts_2\left\{\begin{array}{l}\text{read}(t_1,\, a)\\ \text{read}(t_1,\, b)\end{array}\right. & & & \\
 & \text{unlock}(t_1,\, m) & & \\
 & & \text{lock}(t_2,\, m) & \\
 & & \left.\begin{array}{r}\text{write}(t_2,\, a)\\ \text{write}(t_2,\, b)\\ \text{write}(t_2,\, c)\end{array}\right\} ts_3 & \\
 & & \text{unlock}(t_2,\, m) & \\
ts_4\{\ \text{read}(t_1,\, c) & & & \\
 & & \text{end}(t_2) & \\
 & \text{join}(t_1,\, t_2) & &
\end{array}
$$

(a) Trace

Interval	**seg_written**	**seg_read**
a	$\{ts_1, ts_3\}$	$\{ts_2\}$
b	$\{ts_1, ts_3\}$	$\{ts_2\}$
c	$\{ts_3\}$	$\{ts_4\}$

(b) Segment sets

Figure 5.16.: Thread segment reduction example

5.6. Evaluation

RaceQuest data race detection was evaluated in two parts. RaceQuest is first compared against two happens-before detectors that, like RaceQuest, work on C/C++ programs: the state of the art ThreadSanitizer V2 [SI09] and Helgrind [Val07]. This comparison uses a set of unit tests with known data races and a set of applications commonly used in data race benchmarking. RaceQuest is also compared with other high-coverage race detectors, and we observe how they perform with an increasing number of possible interleavings.

5.6.1. Experimental Setup

The first half of the evaluation was executed in an 8x Intel Xeon CPU E5-1620 v2 at 3.70GHz, 64 GB RAM, running CentOS 6.7 x64. The programs are written in C or C++ and use POSIX threads for parallelization. As RaceQuest instrumentation and replay are based on LLVM, the programs were compiled with clang 3.7 for x64 architecture. FDR3.3 was used as CSP model checker. We requested FDR to provide a single counterexample, after it is found the search is aborted. This makes no difference if the refinement holds. The on-line event cache has a size of 100 events, and the trace window size is 10,000 events.

5.6.2. Unit Test Benchmark

The unit test benchmark is a set of 113 scenarios. Thirteen scenarios are from the Helgrind test suite [Val07]. Six scenarios appear in the *causally proceeds* paper [SES+12]. The other 94 are part of the unit test suite [Goo09] for the first version of ThreadSanitizer, which is based on Valgrind. This suite contains specific tests for the tool functionalities and multiple data race scenarios. We have only used the data race scenarios that do not require special support, such as custom annotations. The scenarios use different synchronization primitives: locks, signals, barriers, semaphores and multiple threads. The scenarios use sleeps to always show the same interleaving to the detectors.

The results appear in Table 5.2. *True positives* are scenarios with a real race detected. *True negatives* are race-free scenarios where no race is detected. *False positives* are race-free scenarios where the tool reports a race that in fact does not exist. *False negatives* are scenarios with races where the tool does not report any warning. *Precision* is the ratio between *True positives* and the sum of *True positives* and *False positives*. *Recall* is the ratio between *True positives* and the sum of *True positives* and *False negatives*. The detailed results are in Appendix B.

The results show that RaceQuest detected fewer incorrect scenarios than Helgrind with 18 and ThreadSanitizer with 17. Whereas RaceQuest only shows one false negative, ThreadSanitizer shows 15 and Helgrind 13. RaceQuest explores more interleavings than the other tools. RaceQuest generated a counterexample that reproduces the warning in all cases of true positives. Helgrind failed, deadlocked, in one scenario.

Both RaceQuest and ThreadSanitizer produced two false positives, while Helgrind produced six. The two cases uses ad-hoc synchronization. None of the tools can detect non-standard synchronization. Neither ThreadSanitizer

Table 5.2.: Data race detection unit test benchmark results

	RaceQuest	**ThreadSanitizer**	**Helgrind**
True positives	57	43	45
True negatives	53	53	48
False positives	2	2	6
False negatives	1	15	13
Precision	0.9661	0.9556	0.8824
Recall	0.9828	0.7414	0.7759

nor Helgrind can build the corresponding happens-before edge between both threads, and RaceQuest does not build any kind of synchronization CSP processes to describe the ad-hoc synchronization, as stated in Section 4.7.1. Nevertheless, RaceQuest generated a counterexample in both cases. But during the reproduction the races were erroneously detected, as the race verification depends on a happens-before detector which is also insensitive to the ad-hoc synchronization. Ad-hoc synchronization detection is an orthogonal problem to race detection, which can be solved with specific algorithms or annotations.

In 12 of the cases marked as true negatives, RaceQuest initially generated a counterexample. During reproduction these cases showed no warning and the counterexamples were discarded. These cases show that the happens-before detector that runs during reproduction is needed to discard possible false positives.

5.6.3. Application Benchmark

The application benchmark is composed of open source projects used in the data race detection literature. Aget is a parallel application to download files through http. BoundedBuffer and prodcons are producer-consumer implementations. Bzip2smp and pbzip2 are parallel implementations of the BZIP compression algorithm. Ctrace is a library for debugging and tracing multithreaded programs. Hawknl is a test for a custom mutex implementation in the HawkNL game-oriented network API. Qsort_mt is a parallel implementation of the quicksort algorithm. Pfscan is a parallel file scanner. Sqlite-1672 is a test harness for a mutex implementation in the sqlite database engine, related to bug number 1672. Blackscholes, fmm, fft, lu, lu-non, streamcluster, and water-nsquared are applications and kernels from the SPLASH-2 multithread benchmark [WOT+95]. The applications contain an unknown number of data races. Table 5.3 lists which synchronization operations are used in each application (mutexes, signal-wait, barrier, or semaphores), the number of threads, and the size in lines of code. Some applications are CPU intensive despite their size and generate long traces, as shown in Table 5.7. A single execution of ThreadSanitizer or Helgrind is much faster than a single execution of RaceQuest. The user could repeatedly execute any of these happens-before detectors in the same amount of time, and hope to observe more interleavings. To be fair, the evaluation compares a single execution of RaceQuest against multiple executions of ThreadSanitizer and Helgrind, so all tools have the same amount of resources.

Table 5.3.: Application benchmark characteristics

Application	LOC	Threads	Mutex	Signal	Barrier	Sema
aget	848	3	✓			
blackscholes	327	5			✓	
boundedBuffer	252	5	✓	✓		
bzip2smp	4,210	4	✓	✓		
ctrace	772	2	✓	✓		✓
fft	886	2	✓		✓	
fmm	3,385	2	✓		✓	
hawknl	6,992	3	✓			
lu	917	2	✓		✓	
lu-non	718	2	✓		✓	
pbzip2	1,491	4	✓	✓		
pfscan	632	3	✓	✓		
prodcons	67	5	✓	✓		
qsort_mt	511	3	✓	✓		
sqlite-1672	48,253	3	✓			
streamcluster	1,255	4			✓	
water-nsquared	1,373	2	✓		✓	

Race detection results

Table 5.4 contains the number of unique locations involved in a race for each tool. The first column is the application name, while the second, third, and fourth columns correspond to the results of RaceQuest, ThreadSanitizer, and Helgrind. All tools emit a set of warnings, where each warning corresponds to a data race. A data race involves one or two locations, i.e. one or two different instructions. To compare the tools, all warnings for an application have been collected, all locations extracted, and all differences counted. For RaceQuest there is a single execution per application. For ThreadSanitizer and Helgrind there are multiple executions, therefore multiple sets of warnings per application. For these two tools all sets of warnings for each application are combined, so the results are the aggregation of multiple executions.

Table 5.4.: Number of racy locations

Application	RaceQuest	ThreadSanitizer	Helgrind
aget	4	5	2
blackscholes	0	0	0
boundedBuffer	0	0	0
bzip2smp	0	0	0
ctrace	0	2	2
fft	0	0	0
fmm	75	64	-
hawknl	0	0	0
lu	0	0	0
lu-non	0	0	0
pbzip2	7	6	6
pfscan	7	0	0
prodcons	5	3	3
qsort_mt	8	8	2
sqlite-1672	0	0	0
streamcluster	2	2	0
water-nsquared	0	0	0
Total	108	90	15

Table 5.5 shows the number of times that ThreadSanitizer and Helgrind were executed in the time needed for a single RaceQuest execution, hence each tool ran during the same amount of time. Application is the application name. Time is the total time in milliseconds required by RaceQuest in a single execution. The sum of tracing time, off-line predictive time, and reproduction time is included. The third and fourth columns are the total number of executions of ThreadSanitizer and Helgrind during the same amount of time.

Table 5.5.: Time and number of executions equivalent to a single RaceQuest execution

Application	Time(s)	ThreadSanitizer	Helgrind
aget	1.89	18	3
blackscholes	0.46	30	2
boundedBuffer	0.54	40	2
bzip2smp	0.43	6	1
ctrace	2.78	2	2
fft	5.05	390	18
fmm	480.36	1,373	-
hawknl	1.03	14	2
lu	2,488.21	7,171	461
lu-non	67.03	3,529	190
pbzip2	1.49	14	2
pfscan	4.49	5	12
prodcons	0.73	68	3
qsort_mt	1.21	70	5
sqlite-1672	0.62	30	2
streamcluster	1.11	72	4
water-nsquared	1,956.19	653	332

The applications blackscholes, boundedBuffer, bzip2smp, fft, hawknl, lu, lu-non, and water-nsquared seem to be race-free. None of the three tools in the evaluation emitted any warning about them.

In qsort_mt and streamcluster, RaceQuest matched the results of ThreadSanitizer, with eight and two erroneous locations. Helgrind falls behind with two and zero locations.

In aget, ThreadSanitizer found one more data race location than RaceQuest. The additional data race location is mutually exclusive to another racy location. Both locations are in different program paths and cannot be found in a single execution. RaceQuest examines a single trace, so like any other dynamic approach it is highly dependent on the path explored by that trace. ThreadSanitizer suffers this dependency too, but in this case it has been executed 18 times. Not all of the executions of ThreadSanitizer took the same path, allowing it to find an additional data race location. RaceQuest and ThreadSanitizer performed better than Helgrind, which only found two lines.

In ctrace, ThreadSanitizer and Helgrind found two locations, but RaceQuest none. RaceQuest correctly suspected the memory address where the data race took place, and generated a counterexample. The counterexample given by the model checker was not enforceable in the original program, i.e. it was infeasible, so RaceQuest missed the data race. The given counterexample diverges very early from the program path, and its schedule cannot be completed. In this case, we requested the model checker to provide additional counterexamples, but none were generated.

Sqlite-1672 is a similar case to ctrace. None of the tools emitted any kind of warning, but in this case we know, through manual inspection, that there is one possible data race. RaceQuest suspected a race, but the counterexample obtained was not feasible and was discarded.

In the fmm, pbzip2, pfscan, and prodcons applications, RaceQuest found more locations than ThreadSanitizer and Helgrind. In fmm RaceQuest found 75, while ThreadSanitizer only found 64. Helgrind crashed in fmm, which makes the most difference in the total number of warnings between the three tools. It is relevant here that ThreadSanitizer was executed 1,373 times, and still could not match the number of locations found by RaceQuest. Note that a single execution of ThreadSanitizer found 52 locations. Most of the remaining 1,372 executions only revealed 12 more. This high number of executions shows that ThreadSanitizer wastes a lot of time exploring the same interleaving, but the off-line exploration of interleavings by RaceQuest discovers more failures.

In pbzip2 RaceQuest found seven locations, while the other tools only found six. In prodcons RaceQuest found two more than ThreadSanitizer and Helgrind. In pfscan neither ThreadSanitizer nor Helgrind found any location at all, while RaceQuest found seven.

The results show that a single execution of RaceQuest found more races than a dynamic happens-before detector in multiple executions. A non-predictive race detector relies on reaching a specific timing to be able to see some races. RaceQuest explores many more off-line.

RaceQuest and dynamic on-line race detectors are not necessarily mutually exclusive. A user would start by using a dynamic on-line detector, as a considerable portion of the data races can be found in a single run. RaceQuest would only be applied if more races are suspected, or the user requires more confidence.

Counterexample feasibility

Table 5.6 describes how many counterexamples RaceQuest generated. *Application* is the application name. *Candidate* is the number of counterexamples obtained in the predictive step. *Feasible and erroneous* is the number of counterexamples that can be successfully reproduced and where the parallel race detector emits at least one warning.

Table 5.6.: Counterexample feasibility

Application	Candidate	Feasible & Erroneous
aget	2	2
blackscholes	0	0
boundedBuffer	0	0
bzip2smp	0	0
ctrace	1	0
fft	0	0
fmm	121	12
hawknl	2	0
lu	0	0
lu-non	0	0
pbzip2	2	1
pfscan	2	2
prodcons	1	1
qsort_mt	1	1
sqlite-1672	1	0
streamcluster	1	1
water-nsquared	0	0

Fmm produced many race candidates, but only 10% are feasible. We have seen that fmm contains many racy locations. Most of the discarded counterexample candidates are infeasible interleavings; reproduction is aborted as soon as infeasibility is detected. For the other applications 14 counterexample candidates were produced, out of which eight were feasible and contained at least a data race warning.

As previously observed there is a data race in sqlite-1672 and ctrace, and RaceQuest generated a counterexample. Unfortunately, the reordering provided by the model checker is not feasible.

In all applications, the number of final counterexamples is smaller than the number of data races. A single counterexample can reveal multiple data races, thanks to the trace reduction strategies explained in Section 5.5. For example: when accessing different fields in the same structure, the whole structure is a single interval in the trace. In this case a single counterexample will reveal a race per field.

Race detectors usually only provide the location of the race, but no information on when and how the program has reached that state. Mixing the race detector with interactive debugging makes the erroneous state difficult to reach. RaceQuest provides a step by step counterexample of the synchronization events that is immune to the interference of interactive debugging.

Trace reductions

Tables 5.7 and 5.8 show the trace reduction effects of the techniques presented in Section 5.2 and 5.5.1.

Table 5.7 shows the number of events at different stages in the RaceQuest workflow until the generation of the trace in the trace model. *No cache* and *with cache* show how many events are emitted if the on-line cache is disabled or enabled. The size of the on-line cache is 100. Other runtime techniques are always in effect, such as the elimination of read in read-write pairs, and no tracing before the first fork. *Total* shows the final size of the trace after the memory intervals have been computed, from the *with cache* emitted events. The number of synchronization events emitted and presented in the trace model is shown in *synch.*

The on-line cache drastically reduces the number of events emitted by the tracing mechanism. The average reduction for the entire application set is 46.2%, with a maximum of 93.64% for blackscholes and 87.66% for water-nsquared. For the biggest applications – fmm, lu and water-nsquared – the reduction is particularly relevant as the storage requirement and IO load is significantly smaller.

Grouping the memory accesses into memory intervals and removing read-only intervals reduces the trace further to the value in the *total* column. The memory intervals reduce the trace size by an average of 72.29%, with peak values of 99.98% for lu and 99.95% for water-nsquared. The combined reduction due to the on-line cache and the grouping of memory accesses into intervals leads to an average trace size reduction of 82.41%. For blackscholes and bzip2smp, these reductions remove all possible memory access events, leaving no race candidates.

Table 5.7.: Trace reduction: cache and memory intervals

	Events emitted		**Trace size**	
Application	**No cache**	**With cache**	**Total**	**# Synch**
aget	1,609	1,142	213	89
blackscholes	4,044	257	20	20
boundedBuffer	879	521	209	129
bzip2smp	35	26	16	16
ctrace	886	762	235	88
fft	84,005	36,166	1,028	22
fmm	4,963,725	1,064,670	12,308	2,460
hawnknl	439	266	81	47
lu	93,484,264	58,993,887	7,239	142
lu-non	1,514,864	921,251	3,048	46
pbzip2	346	229	91	63
pfscan	1,243	1,018	446	273
prodcons	242	209	114	89
qsort_mt	7,289	1,741	795	46
sqlite-dl	78	59	31	22
streamcluster	666	149	61	18
water-nsquared	222,853,344	27,511,033	12,391	8,304

Table 5.8 shows the number of events removed by the hybrid filter and the same segment reduction, as well as the number of redundant CSP refinement checks. *Hybrid* indicates the number of events from the trace eliminated by the hybrid race filter algorithm. *Same segment* shows how many events are eliminated due to the same thread segment reduction. *Redundant* indicates the number of refinement checks that are not performed because of the events eliminated in the *same segment* column. The number of remaining refinement checks to be performed appears in the *final* column.

The hybrid algorithm removed all memory access events in several applications: boundedBuf, fft, lu, lu-non, and water-nsquared. These cases have no remaining race candidates that should be checked with the CSP model. The average reduction in the trace size by the hybrid algorithm is 47.73%. Blackscholes and bzip2smp are excluded as they have no remaining memory events.

The same segment reduction is shown by the number of removed events and

Table 5.8.: Trace reduction: hybrid filter & segment merge

	Events removed		**Refinement checks**	
Application	**Hybrid**	**Same segment**	**Redundant**	**Final**
aget	53	0	0	2
blackscholes	0	0	0	0
boundedBuffer	80	0	0	0
bzip2smp	0	0	0	0
ctrace	124	15	2	1
fft	1,006	0	0	0
fmm	4,081	5,396	2,091	143
hawnknl	15	4	1	4
lu	7,097	0	0	0
lu-non	3,002	0	0	0
pbzip2	19	2	1	2
pfscan	160	0	0	2
prodcons	18	0	0	1
qsort_mt	737	2	1	3
sqlite-dl	0	0	0	1
streamcluster	34	6	2	1
water-nsquared	4,087	0	0	0

the number of saved refinement checks. For blackscholes, boundedBuffer, bzip2smp, fft, lu, lu-non, and water-nsquared, the other reductions have eliminated all possible race candidates, so the thread segment reduction is not needed. This technique has no effect on aget, pfscan, prodcons, and sqlite-dl, which all require one or two checks to be performed. In ctrace, fmm, pbzip2, qsort_mt, and streamcluster, at least 20% of the initial checks are eliminated. Fmm is an extreme case where 93.6% of the checks are removed as they are equivalent to the remaining 6.4%. The number of times that the CSP model must be explored is greatly reduced from 2234 to 143.

We can cross-compare the number of remaining checks with the number of counterexamples obtained, which are already shown in the *candidate* column of Table 5.6. As we obtain a maximum of one counterexample per check, we see that in the 86.7% of the final checks RaceQuest found a racy interleaving.

5.6.4. Scalability Comparison

In this section we compare the scalability of RaceQuest, in number of interleavings, with other dynamic high-coverage tools, such as CHESS and RVPredict. CHESS [MQB07], which is described in Section 3.1.3, is an influence scheduler that exhaustively explores the interleavings of a program. RVPredict [HMR14] is a trace-based predictive tool, described in Section 3.1.2. These tools work for different platforms to RaceQuest; CHESS is used for C# programs and RVPredict for Java programs, so we cannot use the previous benchmarks. Therefore, we implemented a program with variable number of interleavings in the three languages: C, C#, and Java.

The implemented program is depicted in Figure 5.17, and it is similar to the motivational example of this chapter in Figure 5.2. The program contains a configurable number of interleavings. It contains a race that only occurs under a specific schedule, when the thread `worker` finalizes before the thread `main` performs the increment on `x`. In all other interleavings, the race on `x` is hidden because of the happens-before relation between the `unlock` and `lock` calls. In this program each thread executes the correctly protected critical section `N` times, so the number of possible interleavings depends on `N`; but there is still only one interleaving where the race takes place.

This evaluation is performed on a 2x Intel Core i5 4300 at 1.9GHz with 8GB RAM, running a Debian Jessie for RaceQuest and RVPredict, and Windows 10 for CHESS. The configuration of RaceQuest is the same as in previous experiments. CHESS uses a default configuration with race detection enabled. For RVPredict we extended the trace window size to 10,000 events, matching the trace window size of RaceQuest.

Table 5.9 contains the time, in seconds, needed by each tool to find the race. Each column corresponds to an instance of the previous program with `N` loop iterations in each thread.

CHESS starts with 3.83 seconds for a program with 10 iterations but increases exponentially to more than half an hour when reaching the 200 iterations in each thread. CHESS explores all possible interleavings, with a bounded number of preemptions, until it finds the race. It executes the whole program each time, which results in hundreds of executions. This program does not perform any significant work outside locking/unlocking. If there was any relevant work in any thread, CHESS would pay the cost of executing this work in each explored interleaving, something that RaceQuest and RVPredict do not as they only explore an abstract model.

```
int x = 0, y = 0;
mutex m;
void main() {
    fork(worker);
    x++;
    for(int i = 0; i< N; i++) {
      lock(m);
      y++;
      unlock(m);
    }
    join(worker);
}

void worker() {
    for(int i = 0; i< N; i++) {
      lock(m);
      y++;
      unlock(m);
    }
    x++;
}
```

Figure 5.17.: Program with a race and N loop iterations

RVPredict detection time quickly increases with the number of iterations, but not as fast as CHESS. When the number of iterations is 200 RVPredict exhaust the available memory of the machine and crashes. If we use the standard trace window size of RVPredict, which is 1,000 events, it does not crash for 200 iterations, because the internal models are smaller. But it would not find the race in the cases with 150 and 200 iterations.

RaceQuest can handle all cases without a huge increment in execution time. For RaceQuest the time includes the cost of replaying the counterexample schedule. RaceQuest scales better than the other two approaches, as it minimizes the number of events in the trace and creates a smaller abstract model.

5.7. Summary

In this chapter we have shown that RaceQuest, with some extensions, can detect data races. The CSP model only requires two additional event types to represent the memory operations. The race detection is performed with

Table 5.9.: Detection time(s) with different number of iterations

Number of iterations	**10**	**50**	**100**	**150**	**200**
RaceQuest	1.30	1.60	2.04	2.86	3.79
RVPredict	1.69	4.04	50.78	85.59	Fail
CHESS	3.83	70.57	385.50	1,013.46	2,235.16

a specific refinement check. This check searches for read-write or write-write sequences of events in the model and reports if a race is found.

The number of memory operations executed by a program is vast and all of them can be race candidates. To reduce the size of the trace and the number of candidates and checks in the model, we presented several techniques and reductions: selective instrumentation, on-line caching, interval computation, hybrid race filtering, and same segment elimination. In our evaluation we see that the combination of all these techniques removes on average 87% of the events executed by the program.

RaceQuest race detection was compared against ThreadSanitizer and Helgrind happens-before detectors in a benchmark of 113 test scenarios and 17 real applications. In both cases RaceQuest outperformed the other race detectors and found more races. A single execution of RaceQuest detects more races than multiple, even thousands, of executions of the other race detectors. This shows that the other detectors explore redundant interleavings, but RaceQuest achieves greater coverage. We also compared the scalability of RaceQuest against the high-coverage race detectors CHESS and RVPredict. RaceQuest scales much better than the other tools as the number of interleavings increases.

The next chapter enables developers to customize RaceQuest similar to how it is done for race detection. They can generate additional instrumentation to the program to generate new events, which are mapped to CSP, like read and write events. The users can specify their own refinement checks in CSP to detect other ordering errors.

6. Detection of Custom Ordering Errors

Previous chapters have shown the approach of RaceQuest for general concurrency failures: deadlocks and data races. In this chapter we extend RaceQuest further to detect program specific errors that involve ordering issues, such as atomicity and order violations. The idea is similar to the approach for data races: capture additional relevant events for the error, represent them in the CSP model, and use a custom refinement check to detect the error.

Only the user can provide the necessary knowledge about the domain, the failure, and the correct behavior. The user has to provide a definition of the additional events and the refinement check. To specify the additional events, we use Aspect-Oriented Programming. The user describes an instrumentation specification, which includes the new events for the trace model and how these events are generated. The new events in the trace model are automatically translated into the CSP model by RaceQuest. The user also provides a description of the property as a CSP refinement check. The check is combined with the CSP model and fed to the model checker. When the check does not hold, a counterexample is obtained. The counterexample is used to reproduce the interleaving to reach the erroneous state.

6.1. Motivational Example

An example of an order violation is the file access example in Figure 6.1.[1] Two threads use a file API. The thread `main` opens and closes the file handle. The

[1]As already described in Chapter 1

spawned thread `worker` writes the output of the function `compute` to the file.

```
file *f;
void main() {
    fork(worker);
    f = open_file("foo");

    some_work();

    join(worker);
    close_file(f);
}

void worker() {
    data = compute();
    write_file(f, data);
    clear(data);
}
```

Figure 6.1.: Concurrent use of file API (again)

The user expects that the program performs the interleaving in Figure 6.2a. After the file is opened and `worker` is created, the function `some_work` runs in parallel with `compute` and the file writing. But the interleaving in Figure 6.2b is also possible. Here, `write_file` happens before the file is open. In this case `write_file` could simply fail without any warning, and the data written to the file would be lost.

The idea with RaceQuest is that if finds the erroneous interleaving in Figure 6.2b when given any interleaving of the program, such as the one in Figure 6.2a. We need to know the valid orders of API calls to find an interleaving that would produce a failure, so RaceQuest can check if any interleaving in the CSP model does not follow the valid orders. The valid set of API orders is only known by the developer, who has to provide these orders to RaceQuest. In the case of the file API, the valid orders are: at the beginning the file can only be opened, afterwards the file can be read or written any number of times. An opened file can be closed at any time. Then the file returns to the original state and can only be accessed after being opened again.

main	worker
`fork(worker)`	
`f = open_file("foo")`	
`some_work()`	
	`data = compute()`
	`write_file(f, data)`
	`clear(data)`
`join(worker)`	
`close_file(f)`	

(a)

main	worker
`fork(worker)`	
	`data = compute()`
	`write_file(f, data)`
`f = open_file("foo")`	
`some_work()`	
	`clear(data)`
`join(worker)`	
`close_file(f)`	

(b)

Figure 6.2.: Different interleavings for program in Figure 6.1

6.2. Customizing the Trace Model

The basic trace model presented in Chapter 4 only contains synchronization related events. To check properties not directly related to synchronization events, we need to extract the additional information from the program execution. In Chapter 5 the events read and write were added, with the sole purpose of detecting data races. These memory events were obtained by instrumenting all memory operations and generating events at runtime.

Through instrumentation specification the user is able to define new events and the corresponding tracing points to generate them. The instrumentation specification is based on Aspect-Oriented Programming, which is introduced in Section 2.5. The grammar for the instrumentation specification is shown in Figure 6.3.

The whole specification is defined as the set *Instrumentation*, a set of several instrumentation orders. Each instrumentation order contains the following: *event*, *where*, *what* and *target*. These describe the join-points, pointcuts, and

```
⟨Instrumentation⟩ ::= (⟨event⟩ ⟨where⟩ ⟨what⟩ ⟨target⟩ '\n')*

⟨where⟩  ::= 'before'
           | 'after'

⟨what⟩   ::= 'call'
           | 'body'
           | 'read-global'
           | 'write-global'
           | 'line'

⟨event⟩  ::= ⟨identifier⟩ ( '(' ⟨field⟩ (',' ⟨field⟩)* ')' ) ?

⟨field⟩  ::= 'string'
           | '*'* '#index'
           | '*'* '#return'
           | '#size'
           | '#address'
           | '#value'
```

Figure 6.3.: Summary of instrumentation specification grammar

advice in AOP terminology. The join-points are described by the combination of *where* and *what*. These define the type of element in source code that will be instrumented. The pointcuts (join-point instances) are denoted with the identifier *target*. *Where* describes if the instrumentation takes place before or after *what* in the source code. The following list contains the possible values of *what* and the meaning of the associated *target*:

- `call` - a call to a function; *target* is the name of the function.
- `body` - the body of a function (if available); *target* is the name of the function.
- `read-global` - a load instruction for a global variable; *target* is the name of the global variable.
- `write-global` - a store instruction for a global variable; *target* is the name of the global variable.
- `line` - a source code line; *target* is the source code file and line.

An advice is in all cases the emission of an event, described by *event*. *Event* is composed of an identifier and a sequence of fields. The identifier is the name of the new event in the trace, such as the already existing lock, unlock, or fork events. The fields parameterize the different events. They carry additional information in the trace model, e.g. a lock event carries the identifier of a mutex. Most of the fields will take different values at runtime, based on the real values in the program during the execution.

Different values of *what* allow different fields; Table 6.1 summarizes which *fields* are available for each join-point.

Table 6.1.: Valid *field* elements in each join-point

	`call`	`body`	`read-global`	`write-global`	`line`
`string`	✓	✓	✓	✓	✓
`#`*`index`*	✓	✓			
`#return`	✓	✓			
`#size`			✓	✓	
`#address`			✓	✓	
`#value`			✓	✓	

The following list describes the behavior of each type of *field*:

- *`string`* - an alphanumeric string, which is used directly as an identifier in the trace model. This value is constant across all the instances of the same event.
- `#`*`index`* - captures the argument in the *`index`* position as an integer, targeted for capturing pointer addresses or integers.
- `#return` - captures the returning value of the function as an integer, targeted for capturing pointer addresses or integers. Only valid if the value of *where* is `after`.
- `#size` - captures the width of a memory access as an integer.
- `#address` - captures the address of a memory access as an integer.
- `#value` - captures the value of a memory access as an integer.

Any `#`*`index`* or `#return` field can be prefixed by the `*` symbol. For each instance of the symbol, the argument or the return value will be dereferenced before emitting the contained value. The dereferencing is used to capture the content of pointers.

Additionally, all events have automatically associated the thread identifier of their performing thread as first field. The user does not specify that in the instrumentation specification, but it appears in the trace model. For example, an *event* element defined as **access()**, would emit the event access(t_x) at runtime.

6.2.1. Example of a Custom Instrumentation

Figure 6.4 shows the relevant events for the API used by the file access program in Figure 6.1.

Each line defines an additional event in the trace, starting with the identifier that will appear in the trace, such as **open**, **close** and **access**, and followed by when they should be logged. In this case the three events are logged before a function call. The synchronization events are always automatically instrumented, as they are needed to generate the inter-thread orderings. The functions in the source code are `open_file`, `close_file`, `write_file`, and `read_file`. The instrumentation of the call to `open_file` is placed after the call itself, because we need to track the output of the call. The output of the call is the file handle pointer that is used later with the other functions; we capture the pointer with the `#return` field. The program is automatically instrumented before the `close_file`, `write_file`, and `read_file` calls. For these three calls the events also carry the pointer of the file handle, which is always the first argument and is captured with the `#0` field. The instrumentation logs the previous identifiers (**open**, **close** and **access**) in the trace. The **access** event represents the calls to `read_file` and `write_file`.

```
open(#return) after call open_file
close(#0) before call close_file
access(#0) before call write_file
access(#0) before call read_file
```

Figure 6.4.: Instrumentation specification for file API

Figure 6.5 displays a trace of the program in Figure 6.1, obtained with the previous instrumentation in Figure 6.4. It is obtained from the expected interleaving in Figure 6.2a. The thread `main` is t_1, the thread `worker` is t_2. The trace contains the open, access, and close events as expected. The first field for each one is the thread identifier. Note that the user did not specify the thread identifiers, it is done automatically. The second field is the address of the target file captured as an integer value. Here we simply represent the value as f. There is no trace of the functions that are not specified in the instrumentation: `some_work`, `compute`, and `clear`.

fork(t_1, t_2)
open(t_1, f)
start(t_2)
access(t_2, f)
end(t_2)
join(t_1, t_2)
close(t_1, f)

Figure 6.5.: Simplified trace in tuple format

6.3. CSP Model & Custom Properties

The transformation of a trace with custom events to the CSP model corresponds to the process outlined in Section 4.4. The additions are the custom events, which are translated one-to-one to CSP events. For each type of event a channel is defined, and the fields of the event in the trace are fields of the CSP events. The custom events are not synchronization events, so they only appear in the processes $THREAD_i$.

6.3.1. Custom Properties

The second input that the user must provide is the property specification. The property specification is exactly a refinement check in CSP and the auxiliary processes and events, as explained in Section 4.5. The processes $SPEC$ and $IMPL$ can take any form that the user wishes. As in the previous refinement checks, the only requirement is that the user references the process $PROGRAM$ in the implementation process, because the counterexample is extracted automatically from $PROGRAM$. The refinement check can use any semantic model. $SPEC$ and $IMPL$ may reference both synchronization and custom events in the model. The generated CSP model is combined with the property specification and fed to the model checker.

Property check patterns

The refinement check can take multiple forms and have different goals. We identified two main patterns: correctness check and sequence search.

In correctness checking, the user defines in CSP the correct behavior for a set of events, e.g. a process $VALID$. Then the CSP model of the trace is checked for violations of that correct behavior. This case is a direct implementation of the refinement checking definition:

$$\begin{aligned}
&SPEC = VALID\\
&IMPL = PROGRAM \upharpoonright valid\_set\\
&SPEC \sqsubseteq_{\mathrm{T}} IMPL
\end{aligned}$$

The correct behavior is the specification process. The implementation process is the process *PROGRAM*, but the events that are never performed by the correct behavior are hidden. The set of possible events performed by *VALID* is defined in the set *valid_set*. If at any point the process *IMPL* causes an ordering not defined in *VALID*, the refinement does not hold and the counterexample is retrieved.

In sequence searching the user provides a description of an erroneous interleaving and uses the refinement check to find it. An example is race detection, where the read-write and write-write subsequences are searched, independently of the previous and the subsequent events. In CSP there is no direct approach to search for a subsequence in the *traces* of a process, as each trace of a process begins from the first event. Instead we have to build a watchdog process that will follow all the relevant events of the process *PROGRAM* and emit a special event to indicate the error. We offer a pre-defined watchdog process called *WD*. The structure of *WD* and the associated refinement check is as follows:

$$\begin{gathered}
WD_{E,events} = (E\,;error \rightarrow SKIP) \triangle (\Box_{z:events}\; z \rightarrow WD_{E,events})\\
STOP \sqsubseteq_{\mathrm{T}} (PROGRAM \underset{err\_set}{\parallel} WD_{ERR,err\_set}) \upharpoonright \{error\}
\end{gathered}$$

The process *WD* will follow all the events performed by *PROGRAM* in *err_set*. The refinement does not hold if *WD* emits the special event *error*; in this case the counterexample is extracted from *PROGRAM*. *WD* is configurable by the user, it requires a process *ERR* defining the relevant subsequence. The subsequence must terminate with *SKIP*, in order for the event *error* to be emitted. A set with the relevant events for the subsequence ordering is also needed, and is defined in *err_set*.

6.3.2. Counterexample Reproduction

A counterexample will be a combination of synchronization and custom events. As the custom events also have tracing points, they will be used as enforcing points.

6.3.3. Example of a Custom Property

Figure 6.6 shows the corresponding CSP model for the trace in Figure 6.5. The model contains the two corresponding thread processes $THREAD_i$ and the required synchronization processes $FORK$ and $JOIN$. The model only contains the standard synchronization events and the events specified in Figure 6.4 that occur at runtime.

$$\begin{aligned}
THREAD_{t_1} &= fork.t_1.t_2 \to \mathit{open}.t_1.f \to \mathit{join}.t_1.t_2 \to \mathit{close}.t_1.f \to SKIP \\
THREAD_{t_2} &= \mathit{start}.t_2 \to \mathit{access}.t_2.f \to \mathit{end}.t_2 \to SKIP \\
FORK_{t'} &= fork?t.t' \to \mathit{start}.t' \to SKIP \\
JOIN_{t'} &= \mathit{end}.t' \to \mathit{join}?t.t' \to SKIP \\
INTER &= (THREAD_{t_1} \mathbin{|||} THREAD_{t_2}) \mathbin{;} \mathit{endthreads} \to SKIP \\
SYNC &= FORK_{t_2} \mathbin{|||} JOIN_{t_2} \\
PROGRAM &= INTER \underset{\{|fork, join|\}}{\|} SYNC
\end{aligned}$$

Figure 6.6.: CSP model of the trace in Figure 6.5

The property, in the form of a CSP refinement check, is shown in Figure 6.7. The process $FILE_p$ represents the valid orders in CSP for the file p, but uses the events defined in the instrumentation specification. The events include the thread identifier and the file identifier as fields. The process allows any event to be performed by any thread, because in each event the thread field accepts any value.

The refinement check is an application of the correctness pattern, where $FILE_p$ describes the allowed order of file-related events. We can have multiple file identifiers in the program and the trace; therefore, for each file a process $FILE_p$ is instantiated. These processes are combined with an interleaving operator in the process $FILES$ that represents the combined orderings of file operations in all files in the trace. The set Files contains all file identifiers in the model. In the implementation side of the refinement check we only show the file-related events of $PROGRAM$. All the synchronization events are hidden, as none appear in $FILE_p$.

The refinement check with the process $PROGRAM$ from the model in Figure 6.6 does not hold. In $PROGRAM$ the $\mathit{access}.t_2.f$ event can occur before the $\mathit{open}.t_1.f$ event, an order not allowed by $FILE_p$. The counterexample here

$$
\begin{aligned}
FILE_p &= open?t.p \rightarrow OPEN_p \\
OPEN_p &= access?t.p \rightarrow OPEN_p \\
&\quad \Box\ close?t.p \rightarrow STOP \\
FILES &= |||_{x \in Files}\ FILE_x \\
FILES &\sqsubseteq_{\mathrm{T}} PROGRAM \restriction \{|open, access, close|\}
\end{aligned}
$$

Figure 6.7.: File access property

is:

$$fork.t_1.t_2, start.t_2, access.t_2.f$$

The CSP$_M$ version of the property model and checks can be found in Appendix A.2.5.

6.4. Use Cases

In this section we present additional use cases of RaceQuest with custom properties.

For each case, including the file access example, two small programs have been coded and tested to evaluate the validity of the defined properties. One is a scenario that violates the relevant property, e.g. the file opening does not ever happen before the first access. The other is the corrected version of that scenario. RaceQuest detected 100% of the property violations and generated the corresponding counterexample. No warning was reported in any of the correct test cases. The average trace size in all the 14 tests is 15 events. Due to their small size, as in the deadlock evaluation in Section 4.8, the average time used by RaceQuest is 554ms, out of which 85% is the off-line prediction step, dominated by the call to the refinement checker.

6.4.1. Deadlock

In Chapter 4 we used deadlock detection as a motivational example for the basic modeling in RaceQuest. Although deadlock detection is built into RaceQuest, we can easily replicate the property it as a custom property. The instrumentation specification is empty, as no additional events are needed, the synchronization events are enough. The property specification is exactly the one defined in Section 4.5.1. It is a special instance of correctness checking, as the specification process is a non-deadlocking one. The user would only need to provide the description in Figure 6.8 as property specification.

$$LIVE = live \rightarrow LIVE$$
$$LIVE \sqsubseteq_{\mathrm{F}} (PROGRAM \Theta_{endthreads} SKIP) \setminus \Sigma \, ; LIVE$$

Figure 6.8.: Deadlock property

6.4.2. Custom Race Detector

Race detection was covered in Chapter 5, which presented additional algorithms to reduce the huge number of initial candidates. Users can also create their own race detectors if they want to check a specific global variable. For example, a user wants to check if the accesses to the shared variables x and y of a program suffer from a race. The instrumentation specification in Figure 6.9 instruments all accesses to the shared variables x and y of a program, but only the direct accesses to the global variables, not through alias. During runtime these memory access events will populate the trace, such as $x.t_1.store$ for a write by the thread t_1 to variable x.

```
x(load) before read-global x
x(store) before write-global x
y(load) before read-global y
y(store) before write-global y
```

Figure 6.9.: Instrumentation specification for custom race detector

The data race property for checking races on both variables is shown in Figure 6.10. A data race is an instance of the sequence search pattern. The set err_set_k for each variable are the events related to the variable and the synchronization events. The same parameterized process ERR_k can be used for both variables. The process WD is the same as presented in the pattern description. For each variable there is a specific refinement check. Both refinement checks have the same structure and only the process ERR and the set err_set change. The whole structure of the race check is the same as the one used in Section 5.3.1, distinguished only by the structure of the events and the process we use, WD.

The CSP$_M$ version of the property can be found in Appendix A.2.6.

$$\begin{aligned}
err\_set_k &= \{|k|\} \cup sync\_events \\
ERR_k &= k?t.load \rightarrow k?t' : threads - \{t\}.store \rightarrow SKIP \\
&\quad \Box\ k?t.store \rightarrow k?t' : threads - \{t\}.store \rightarrow SKIP \\
WD_{E,events} &= (E\,;error \rightarrow SKIP) \bigtriangleup (\Box_{z:events}\, z \rightarrow WD_{E,events}) \\
STOP &\sqsubseteq_{\mathrm{T}} (PROGRAM \underset{err\_set_x}{\parallel} WD_{ERR_x,err\_set_x}) \upharpoonright \{error\} \\
STOP &\sqsubseteq_{\mathrm{T}} (PROGRAM \underset{err\_set_y}{\parallel} WD_{ERR_y,err\_set_y}) \upharpoonright \{error\}
\end{aligned}$$

Figure 6.10.: Data race property

6.4.3. Rover Uplink

The following example describes the ordering in API calls of a data uplink from a planetary rover to a space craft, a fictional example presented by Havelund [Hav08]. The functions in the API are listed in Figure 6.11. The API has the following four requirements:

R1: A connection is opened, accessed zero or more times, and subsequently either closed or canceled. An access is either a write operation or a commit operation.

R2: The commit operation must be followed by an acknowledgment before any other operation can be performed, except a cancellation.

R3: It is prohibited to have more than one connection opened at any time.

R4: A write operation or an assignment to the variable `header`, collectively referred to as an update, should be followed by a commit operation before the connection is closed, unless the transmission is canceled.

```
char* header;
Connection open_connection(char* name) {...}
bool close_connection(Connection connection) {...}
void cancel_transmission(Connection connection) {...}
void write_buffer(Connection connection, int data) {...}
void commit_buffer(Connection connection) {...}
void acknowledge() {...}
```

Figure 6.11.: Rover uplink API

We want to check if any program using this API violates any of the requirements. We need to define the instrumentation and property specifications for RaceQuest. The instrumentation specification is shown in Figure 6.12. An event is created for each function and for the accesses to the global variable `header`. The assignment to `header` and the write operation on the buffer produce the common event *update* (R4). As it is only permissible to open one connection (R3), we do not need to differentiate the function calls by their arguments, i.e. the connections.

```
open before call open_connection
close before call close_connection
cancel before call cancel_transmission
update before call write_buffer
commit before call commit_buffer
update before write-global header
ack before call acknowledge
```

Figure 6.12.: Instrumentation specification for rover example

The property check is displayed in Figure 6.13. The property follows the correctness check pattern, where the process $UPLINK$ represents the specification to be fulfilled. The four requirements are defined in the processes $UPLINK$, $OPEN$, and $COMMIT$. $UPLINK$ ensures that the first action must be always to open the connection (R1) and that only one connection is open at any time (R3). $UPLINK$ may be canceled at any state (derived from R1, R2, and R4). $OPEN$ describes the operations on an open connection: closing (R1), committing (R1), or updating (R4). $COMMIT$ ensures that a commit operation is always followed by an acknowledgment operation (R2). Finally, all synchronization events and read operations on the header variable are hidden from $PROGRAM$, as they are not relevant for $UPLINK$.

Any program that makes use of this API can be checked with RaceQuest to detect violations of any of the four requirements. For example, a violation is a write or commit after a cancel by another thread. Only the instrumentation and property specifications are needed. The CSP$_M$ version of the property can be found in Appendix A.2.7.

6.4.4. SQLite Core API

The SQLite API for C/C++ contains a core set of functions that are sufficient to access the database and make requests. The API manual [SQL] describes

$$
\begin{aligned}
UPLINK &= open?t \rightarrow ((OPEN \bigtriangleup cancel?t \rightarrow SKIP)\,;UPLINK)\\
OPEN &= closed?t \rightarrow SKIP\\
&\quad \Box\ COMMIT\\
&\quad \Box\ update?t \rightarrow COMMIT\\
COMMIT &= commit?t \rightarrow ack?t \rightarrow OPEN\\
UPLINK &\sqsubseteq_{\mathrm{T}} PROGRAM \restriction \{|open, close, cancel, update, commit, ack|\}
\end{aligned}
$$

Figure 6.13.: Rover example property

the relevant set of functions and the order in which they are used. The API contains two sets of functions, one for managing a connection to a database and another for managing SQL statements. A connection to a database can be created and closed with the functions `sqlite3_open` and `sqlit3_close`. SQL statements can be compiled and prepared on an open connection with the function `sqlite3_prepare`. This preparation can only be done once per statement. The function `sqlite3_step` evaluates a statement and retrieves the first row of results. Subsequent calls to `sqlite3_step` on the same statement deliver the next rows. The columns of the current row are explored with the function `sqlite3_column`. Starting with the first column, following `sqlite3_column` calls show the next column. A statement is destroyed with the function `sqlite3_finalize`. Prepared statements can be reused using the function `sqlite3_reset`. A prepared statement can be parameterized with the function `sqlite3_bind`. This must happen before evaluating the statement, and before calling `sqlite3_step`.

The instrumentation specification in Figure 6.14 defines the events for the different functions. The events carry at least one parameter to identify the connection or statement to which they refer. The function `sqlite3_prepare` carries both arguments, as it depends on both. In all cases we capture the pointer address of the structure so they can be uniquely identified. The functions `sqlite3_open` and `sqlite3_prepare` use one of their arguments to return a pointer to the created connection or statement. In these two cases we need to dereference these arguments so they match the captured arguments of the other functions. For example, the `*#3` for `sqlite3_prepare` dereferences the third argument of the function call and then emits its content as an integer in the event.

The custom property is defined in Figure 6.15. The processes DB_x and $OPEN_x$

```
open(*#1) after call sqlite3_open
close(#1) before call sqlite3_close
prepare(#0, *#3) after call sqlite3_prepare
finalize(#0) before call sqlite3_finalize
bind(#0) before call sqlite3_bind
reset(#0) before call sqlite3_reset
step(#0) before call sqlite3_step
column(#0) before call sqlite3_column
```

Figure 6.14.: Instrumentation specification for the SQLite core API.

define the behavior of a single database connection. The connection can be opened. Once open, multiple prepare events can be performed until the connection is finally closed. The process $STMT_x$ initiates a database statement, which can be initially bound or not, and transits into the process $READY_x$. $STMT_x$ allows multiple step events, being reset (with an optional bind), multiple column events, and finalized. The process SQL combines all instances of the processes DB_x and $STMT_x$. All DB_x processes are interleaved, as are all $STMT_x$ processes. The interleaved DB_x processes are composed in parallel with the $STMT_x$ processes communicating over the prepare events; this composition is the process SQL. No $STMT_x$ begins until the correspondent prepare event has been performed by the corresponding DB_x. The refinement check follows the correctness pattern, where SQL describes the correct behaviors. The CSP$_M$ version of the property can be found in Appendix A.2.8.

6.4.5. Atomicity Violation

Figure 6.16 shows a program with an atomicity violation. In the program the threads `main` and `worker` execute the function `increment`, which increases the global variable `x` by one. Note that the variable `x` is atomic, so there are no data races during the access to `x`. While `main` is performing the local instruction `temp++`, `worker` can concurrently execute its call to `increment`. Then the value of `x` that `main` read is no longer valid and `main` will overwrite the work done by `worker`. The function `increment` function should be executed atomically, i.e. its calls should be serialized. But nothing is enforcing the atomicity.

This atomicity violation is a semantic error, so we need to know that the function `increment` should be serialized and check whether the serializability is violated. It is not in the scope of RaceQuest hypothesize which parts of a

$$
\begin{aligned}
DB_x &= open?t.x \rightarrow OPEN_x \\
OPEN_x &= prepare?t.x?s \rightarrow (OPEN_x \;\square\; close?t.x \rightarrow SKIP) \\
STMT_x &= prepare?t?d.x \rightarrow (bind?t.x \rightarrow READY_x \;\square\; READY_x) \\
READY_x &= step?t.x \rightarrow READY_x \\
&\quad \square\; reset?t.x \rightarrow (READY_x \;\square\; bind?t.x \rightarrow READY_x) \\
&\quad \square\; column?t.x \rightarrow READY_x \\
&\quad \square\; finalize?t.x \rightarrow SKIP \\
SQL &= (|||_{x \in Connections}\, DB_x) \underset{\{|prepare|\}}{\|} (|||_{x \in Statements}\, STMT_x) \\
SQL &\sqsubseteq_{\mathrm{T}} PROGRAM \restriction \\
&\quad \{|open, close, prepare, finalize, bind, reset, step, column|\}
\end{aligned}
$$

Figure 6.15.: SQLite example property

program are supposed to be atomic. This information should come from program annotations or other tools, e.g. from Vaziri [VTD06] or Atomizer [FF08]. However, with RaceQuest we can check whether the atomicity of a supposed atomic region has been violated.

We describe the instrumentation specification in Figure 6.17, where the enter and exit events correspond to the beginning and end of the body of the `increment` function, and the access event correspond to the accesses to the variable `x`.

The work of Vaziri [VTD06] defines multiple atomicity violation patterns, such as *read-write-write* or *write-read-write* by different threads. The five single variable atomicity patterns are condensed here in the process AV_t. The process $PREAV_t$ allows any memory access as prefix of AV_t. The process ERR ensures that the previous patterns are only active inside the defined serializable region, delimited by the enter-exit pair of events. The refinement check follows the sequence search pattern.

The atomicity violation property in CSP can be reused for any single variable atomicity violation check. But the instrumentation must be adapted to the specific serializable block and variable to be checked. The CSP$_M$ version of the property can be found in Appendix A.2.9.

```
atomic x = 0;
void main() {
    fork(worker);

    increment();

    join(worker);
}

void worker() {
    increment();
}

void increment() {
    int temp = x;

    temp++;

    x = temp;
}
```

Figure 6.16.: Atomicity violation example

6.4.6. Resource Management System

The following example describes a resource management system presented by Havelund [HR15]. The system contains only the four functions listed in Figure 6.19.

The system has the following requirements:

R1: A resource may be requested by any task with the request function.

R2: A requested resource may be denied or granted, i.e. the request function returns a zero or a non-zero.

R3: A granted resource may be rescinded or canceled, a rescinded resource is still granted.

R4: A resource may only be requested by a task if that task does not currently hold the resource.

R5: A granted resource must eventually be canceled.

```
enter before body increment
exit after body increment
x(load) before read-global x
x(store) before write-global x
```

Figure 6.17.: Instrumentation specification for an atomicity violation check

$$
\begin{aligned}
err\_set_k &= \{|enter, exit, k|\} \\
AV_{t,k} &= k.t.load \rightarrow k?t' : threads - \{t\}.store \rightarrow k.t?any \rightarrow SKIP \\
&\quad \Box\ k.t.store \rightarrow k?t' : threads - \{t\}.load \rightarrow k.t.store \rightarrow SKIP \\
&\quad \Box\ k.t.store \rightarrow k?t' : threads - \{t\}.store \rightarrow k.t?any \rightarrow SKIP \\
PREAV_{t,k} &= AV_{t,k} \bigtriangleup (\Box_{z:\{|k|\}}\ z \rightarrow AV_{t,k}) \\
ERR_k &= enter?t \rightarrow (PREAV_{t,k} \bigtriangleup exit.t \rightarrow ERR_k) \\
WD_{E,events} &= (E\,;error \rightarrow SKIP) \bigtriangleup (\Box_{z:events}\ z \rightarrow WD_{E,events}) \\
STOP &\sqsubseteq_{\mathrm{T}} (PROGRAM \underset{err\_set_k}{\|} WD_{ERR_k,err\_set_k}) \upharpoonright \{error\}
\end{aligned}
$$

Figure 6.18.: Atomicity property

R6: A resource should only be held by at most one task at any one time. A resource should be canceled before being granted to another task, i.e. mutual exclusion.

R7: Two resources can be declared to be mutually exclusive (they conflict), i.e. they cannot be granted to any task at the same time.

We will consider that each task is performed by a single thread.

The instrumentation specification is shown in Figure 6.20. We capture one event per function call and the first argument, which refers to the resource. For the function `request` we also capture the return value, as it is needed to know if the resource has been granted or not. The function `conflict` declares that the two given resources are mutually exclusive.

Figure 6.21 displays the property. The processes RES_k and $GRANTED_{t,k}$ describe the basic behavior of the resource management. In RES_k, after a request event for a resource, the return value, field s, is used to decide if the resource has been granted or not. A resource granted to thread t transits the process to the process $GRANTED_{t,k}$. A denied resource returns to RES_k, so

```
int request(resource *r) {...}
void rescind(resource *r) {...}
void cancel(resource *r) {...}
void conflict(resource *r1, resource* r2) {...}
```

Figure 6.19.: Resource management API

```
request(#0, #return) after call request
rescind(#0) after call rescind
cancel(#0) after call cancel
conflict(#0,#1) after call conflict
```

Figure 6.20.: Resource management

it can be requested again. All the RES_k processes are interleaved. The process $CONFI_{m,n}$ represents the entry point for the conflict management, i.e. the mutual exclusion between resources m and n. $CONFI_{m,n}$ must first observe the corresponding conflict event, afterwards the mutual exclusion becomes active with the process $CONF_{m,n}$. $CONF_{m,n}$ observes the request events on m or n. After a denial of the request, the process returns to $CONF_{m,n}$ to observe the next request. After a granted request, non-zero s, the only valid event is a cancel on the same resource. The paired resource cannot be requested until the first one is released. For each pair of resources, m and n, an independent $CONFI_{m,n}$ is instantiated in the process $CONFS$. All the instances of $CONFI_{m,n}$ run in parallel and synchronize on the events request and cancel. Different conflicting pairs will have to agree on these events. Similarly, all these conflict processes also run in parallel with the interleaved RES_k processes, again synchronizing on request and cancel events. With all the defined processes synchronizing simultaneously on the request and cancel events, we ensure that all conditions, e.g. resources not acquired and not conflicting with any other, for each event must be fulfilled at the same time. The refinement check is an instance of the correctness pattern, where the process $RESS$ describes the correct behaviors. The CSP$_M$ version of the property can be found in Appendix A.2.10.

6.5. Summary

In this chapter we have extended RaceQuest to support custom checks. Users can define their own properties to validate that the behavior of a program

$$
\begin{aligned}
RES_k &= request?t.k?s \rightarrow \\
&\quad (RES_k \lessdot s == 0 \gtrdot GRANTED_{t,k}) \\
GRANTED_{t,k} &= rescind.t.k \rightarrow GRANTED_{t,k} \\
&\quad \Box\ cancel.t.k \rightarrow RES_k \\
CONFI_{m,n} &= conflict?.m.n \rightarrow CONF_{m,n} \\
&\quad \Box\ request?t.m \rightarrow CONF_{m,n} \\
&\quad \Box\ request?t.n \rightarrow CONF_{m,n} \\
&\quad \Box\ cancel?t.m \rightarrow CONF_{m,n} \\
&\quad \Box\ cancel?t.n \rightarrow CONF_{m,n} \\
CONF_{m,n} &= request?t.m?s \rightarrow \\
&\quad (CONF_{m,n} \lessdot s == 0 \gtrdot cancel?t.m \rightarrow CONF_{m,n} \\
&\quad \Box\ request?t.n?s \rightarrow \\
&\quad (CONF_{m,n} \lessdot s == 0 \gtrdot cancel?t.n \rightarrow CONF_{m,n} \\
CONFS &= \underset{\{|request,cancel|\}}{\|}\ {}_{x,y \in Resources}\ CONFI_{x,y} \\
RESS &= (|||_{x \in Resources}\ RES_x) \underset{\{|request,cancel|\}}{\|} CONFS \\
RESS &\sqsubseteq_{\mathrm{T}} PROGRAM \upharpoonright \{|request, rescind, cancel, conflict|\}
\end{aligned}
$$

Figure 6.21.: Resource management property

follows the correct order. Users must specify which events are relevant in a program and extract them with the help of Aspect-Oriented Programming. The orders to be checked, i.e. the properties, are described in CSP. The violation of the properties provides a counterexample, such as in deadlocks and data races, which users can use to analyze the problem. We provide two main patterns to facilitate the writing of new properties.

We illustrated the use of RaceQuest for custom properties with seven different examples. The examples include cases with descriptions of only the accepted behaviors, such as API calling orders, and cases with descriptions of erroneous behaviors, such as atomicity violations. All examples have been tested with correct and faulty versions of the same program.

7. Conclusion

This final chapter presents a summary of the RaceQuest predictive approach, as well as several ideas and directions for research and future work.

7.1. Summary

In this work we developed RaceQuest, an innovative automatic dynamic approach to find concurrency failures. We stated in Section 1.3 that a model-based analysis of a trace should fulfill the following requirements: high coverage of interleavings, no false positives, reproducibility of the detected failures, extensibility, and minimal size of the trace. Throughout this work we have shown how RaceQuest achieves these requirements.

Our model takes a trace of an execution, and models the observed interleaving and alternative interleavings using the CSP process algebra. These alternative interleavings are based on the reorderings of the observed instructions, always following the semantics of the present synchronization operations. The exploration of the model that represents this set of interleavings enables the detection of more failures than other dynamic tools, which are sensitive to the timing and observe a narrow set of interleavings. The off-line exploration of interleavings, and its smaller abstract model, allow RaceQuest to cover more interleavings with a better scalability without re-executing the program for each interleaving.

RaceQuest only observes real addresses and values, and thus eliminates false positives due to aliasing. RaceQuest has some limitations that can introduce false positives, such as ad-hoc synchronization or lock-free algorithms, but these could be improved with additional annotations. The interleaving inference by the model could generate infeasible interleavings, which could lead to false positives. These infeasible interleavings are explicitly detected and eliminated replaying the obtained counterexamples. The outputs of RaceQuest exclusively consist of failure-triggering counterexamples.

The counterexamples are schedulings provided by RaceQuest when a failure is found in any interleaving of the model. Replaying can also be used by developers to investigate the failure. The locations in the source code are not always enough to understand the cause of the problem and identify the defect. Thanks to the schedule provided, the developer can re-run the program multiple times and repeatedly observe the same behavior.

Our model can be used for multiple purposes. We have covered deadlocks, data races, and ordering errors defined by the user. All the errors are defined directly in the process algebra, without any kind of extension to the algebra.

In each case the captured trace and the model are composed of the least number of events relevant to the failure. This implies less overhead during the capture of the trace and the reproduction of the program, as well as faster exploration of the CSP model due to its smaller size. Race detection is a taxing case, as the initial number of operations is huge. RaceQuest contains additional steps that reduce the size of the trace as well as the number of explorations of the model.

RaceQuest is a viable and effective approach to help the developers of multi-threaded shared memory systems to find and debug concurrency failures. As the tool does not emit false positives, no time is wasted tracking non-existent failures. The developers can focus on replaying the given counterexamples to understand the causes of the failure without the timing influence of a debugger. RaceQuest simplifies the life of developers and saves time when facing concurrency failures, which is one of the main disadvantages of the now ubiquitous parallel programming.

7.2. Future Work

An improvement would be the reduction of infeasible counterexamples, to save resources by avoiding the testing of invalid schedules. To achieve this without incorporating sequential consistency in all memory operations is difficult. Tracking sequential consistency in all memory operations requires capturing all accesses and their values. That is, the amount of resources needed to reduce infeasibility in the model should be smaller than what is needed to perform the active testing. One possibility is to only enable memory consistency for atomic variables. This approach should reduce the number of interleavings without excessively increasing the size of the trace and the model.

In cases where no error is detected, the model checker performs an exhaustive search. We have not limited the number of resources in each search. But in case of more limited resources we would want to prioritize some reorderings above others. Alternatively, we would want to prioritize interleavings that are similar to the original trace to avoid path divergence. One direction of work would be to incorporate this prioritization using the process algebra or the model checker algorithms.

The Aspect-Oriented grammar can be significantly expanded with ideas from more complex languages such as AspectJ. We could add advice customization, more join-points, or conditional behavior to the tracing. The current implementation of the Aspect-Oriented instrumentation works at the level of LLVM. An alternative could be to change it to operate directly on C or C++ code, which would offer more possibilities, such as the use of type information.

Applying RaceQuest to other languages or libraries, even if they use LLVM as backend, implies technical changes, such as identifying the functions to synchronization calls. More importantly, we need to support the semantics of its synchronization constructs, i.e. add or modify the CSP processes that represent these constructs. For example, in Java we find a recursive mutex as built-in mutex and a latch is a non-reusable barrier.

Although this work covered shared memory systems, the main idea could be translated to distributed memory. Each process of the distributed program would take the role of one thread in the model. The synchronization mechanisms would need to be redefined.

Appendices

A. Example Models in CSP$_M$

This appendix contains a brief introduction to the machine-readable format CSP$_M$ and the CSP$_M$ versions of the examples seen in previous chapters. The examples can be applied as they are to the FDR3 model checker.

A.1. CSP$_M$ Language

The CSP$_M$ is a lazy functional language to describe CSP models and checks. CSP$_M$ representation differs slightly from the *blackboard* CSP description used in the previous chapters. In a CSP$_M$ description we identify three main elements: type definitions, CSP processes, and checks. Table A.1 describes the CSP$_M$ equivalent, in plain ASCII, for *blackboard* CSP symbols.

In CSP$_M$ the events are typed. Type is assigned with the keyword `channel`. For example, `channel coin: Integer` defines a channel coin that always has exactly a single field and is an integer; `coin.100` or `coin.-50` are valid instances of the event, but `coin.gold` is not as gold is not an integer. New types are defined with the keyword `datatype`. For example, `datatype metal = gold | silver` creates the type metal that has two values: gold or silver. A set of types can be bonded together to a name using the keyword `nametype`. For example, `nametype validcoins = {10,50}` binds the set composed of the integers 10 and 50 to the name `validcoins`; this set can also be used as another type. Datatypes and nametypes can be used in channel definitions, e.g. `channel coin: metal` or `channel coin: values`.

An assertion asks the model checker to verify if an expression is true. An assertion uses the keyword `assert`, as in `assert P [T= Q` where it verifies if Q

Table A.1.: CSP$_M$ symbol equivalence

CSP	**CSP$_M$**
$a \rightarrow P$	a -> P
$P \mathbin{\Box} Q$	P [ ] Q
$P \setminus \mathit{event\text{-}set}$	P \event-set
$P \upharpoonright \mathit{event\text{-}set}$	P |\event-set
$P \mathbin{;} Q$	P ; Q
$P \Theta_{\mathit{event\text{-}set}} Q$	P [| event-set |> Q
$P \mathbin{\|\|\|} Q$	P ||| Q
$P \underset{\mathit{event\text{-}set}}{\|\|} Q$	P [| event-set |] Q
$P \bigtriangleup Q$	P /\Q
$P \lhd b \rhd Q$	if b then P else Q
$\{\|a, b\|\}$	{| a, b |}
Σ	Events
$S \sqsubseteq_{\mathrm{T}} I$	S [T= I
$S \sqsubseteq_{\mathrm{F}} I$	S [F= I
$\mathit{m\text{-}set} \cup \mathit{n\text{-}set}$	union(m-set, n-set)
$\bigcup_{x \in \mathit{sets}} x$	Union(sets)
$\mathit{m\text{-}set} \setminus \mathit{n\text{-}set}$	diff(m-set, n-set)

refines P using the trace model. Assertions also accept additional options, such as `:[partial order reduce]`, which applies a safe state space reduction technique.

CSP$_M$ has more capabilities not described here: functions, error handling, modules, subtypes, etc. A more complete description of CSP$_M$ is available in the FDR3 manual [GRABR13].

A.2. Example Versions in CSP_M

A.2.1. Model with Deadlock in Figure 4.14

```
nametype thread = {0,1}
nametype mutex = {0,1}
channel lock : thread.mutex
channel unlock : thread.mutex
channel fork : thread.thread
channel join : thread.thread
channel start : thread
channel end : thread
channel endthreads

THREAD_0 = fork.0.1 -> lock.0.0 -> lock.0.1 -> unlock.0.1 ->
         unlock.0.0 -> join.0.1 -> SKIP
THREAD_1 = start.1 -> lock.1.1 -> lock.1.0 -> unlock.1.0 ->
         unlock.1.1 -> end.1 -> SKIP
INTER = (THREAD_0 ||| THREAD_1) ; endthreads -> SKIP
MUTEX_0 = lock.0.0 -> unlock.0.0 -> MUTEX_0
        [] lock.1.0 -> unlock.1.0 -> MUTEX_0
MUTEX_1 = lock.0.1 -> unlock.0.1 -> MUTEX_1
        [] lock.1.1 -> unlock.1.1 -> MUTEX_1
FORK_1 = fork.0.1 -> start.1 -> STOP
JOIN_1 = end.1 -> join.0.1 -> STOP
SYNC = MUTEX_0 ||| MUTEX_1 ||| FORK_1 |||  JOIN_1
sync_set = {|fork,join,start,end,lock,unlock|}
PROGRAM = INTER [| sync_set |] SYNC

channel aux
LIVE = aux -> LIVE
assert LIVE [F= (((PROGRAM [| {endthreads} |> SKIP)\Events);
            LIVE) :[partial order reduce [precise]]
```

A.2.2. Model with Multiple Lock-Orders in Figure 4.20

```
nametype thread = {0,1}
nametype mutex = {0,1}
nametype barrier = {0}
channel lock : thread.mutex
channel unlock : thread.mutex
channel fork : thread.thread
channel join : thread.thread
channel start : thread
channel end : thread
channel barrier_enter : thread.barrier
channel barrier_exit : thread.barrier
channel endthreads

THREAD_0 = fork.0.1 -> lock.0.0 -> lock.0.1 -> unlock.0.1 ->
         unlock.0.0 -> barrier_enter.0.0 -> barrier_exit.0.0
         -> lock.0.1 -> lock.0.0 -> unlock.0.0 -> unlock.0.1
         -> join.0.1 -> SKIP
THREAD_1 = start.1 -> lock.1.0 -> lock.1.1 -> unlock.1.1 ->
         unlock.1.0 -> barrier_enter.1.0 -> barrier_exit.1.0
         -> lock.1.1 -> lock.1.0 -> unlock.1.0 -> unlock.1.1
         -> end.1 -> SKIP
INTER = (THREAD_0 ||| THREAD_1) ; endthreads -> SKIP
MUTEX_0 = lock.0.0 -> unlock.0.0 -> MUTEX_0
        [] lock.1.0 -> unlock.1.0 -> MUTEX_0
MUTEX_1 = lock.0.1 -> unlock.0.1 -> MUTEX_1
        [] lock.1.1 ->  unlock.1.1 -> MUTEX_1
FORK_1 = fork.0.1 -> start.1 -> STOP
JOIN_1 = end.1 -> join.0.1 -> STOP
BARRIER_U_0(2) = barrier_exit.0.b -> BARRIER_D_B(1)
               [] barrier_exit.1.b -> BARRIER_D_B(1)
BARRIER_U_0(i) = barrier_enter.0.b -> BARRIER_U_B(i+1)
               [] barrier_enter.1.b -> BARRIER_U_B(i+1)
BARRIER_D_0(0) =barrier_enter.0.b -> BARRIER_U_B(1)
               [] barrier_enter.1.b -> BARRIER_U_B(1)
BARRIER_D_0(i) = barrier_exit.0.b -> BARRIER_D_0(i-1)
               [] barrier_exit.1.b -> BARRIER_D_0(i-1)
SYNC = MUTEX_0 ||| MUTEX_1 ||| FORK_1 |||  JOIN_1
     ||| BARRIER_D_0
sync_set = {|fork,join,start,end,lock,unlock,barrier_enter,
         barrier_exit|}
PROGRAM = INTER [| sync_set |] SYNC
```

```
channel aux
LIVE = aux -> LIVE
assert LIVE [F= (((PROGRAM [| {endthreads} |> SKIP)\Events);
             LIVE) :[partial order reduce [precise]]
```

A.2.3. Model with Deadlock & Semaphore in Figure 4.23

```
nametype thread = {0,1}
nametype semaphore = {0}
channel fork : thread.thread
channel join : thread.thread
channel start : thread
channel end : thread
channel sem_post : thread.semaphore
channel sem_wait : thread.semaphore
channel endthreads

THREAD_0 = fork.0.1 -> sem_post.0.0 -> sem_wait.0.0 ->
         join.0.1 -> SKIP
THREAD_1 = start.1 -> sem_wait.1.0 -> sem_wait.1.0 ->
         end.1 -> SKIP
INTER = (THREAD_0 ||| THREAD_1) ; endthreads -> SKIP
FORK_1 = fork.0.1 -> start.1 -> STOP
JOIN_1 = end.1 -> join.0.1 -> STOP
SEMAPHORE_0(0) = sem_post.0.0 -> SEMAPHORE_0(1)
               [] sempost.1.0 -> SEMAPHORE_0(1)
SEMAPHORE_0(10) = sem_wait.0.0 -> SEMAPHORE_0(1)
                [] semwait.1.0 -> SEMAPHORE_0(1)
SEMAPHORE_0(i) = sem_post.0.0 -> SEMAPHORE_0(i+1)
               [] sem_post.1.0 -> SEMAPHORE_0(i+1)
               [] sem_wait.0.0 -> SEMAPHORE_0(i-1)
               [] sem_wait.1.0 -> SEMAPHORE_0(i-1)
SYNC = FORK_1 |||  JOIN_1 ||| SEMAPHORE_0
sync_set = {|fork,join,start,end,lock,unlock,sem_post,
         sem_wait|}
PROGRAM = INTER [| sync_set |] SYNC

channel aux
LIVE = aux -> LIVE
assert LIVE [F= (((PROGRAM [| {endthreads} |> SKIP)\Events);
            LIVE) :[partial order reduce [precise]]
```

A.2.4. Model with a Data Race in Figure 5.8

```
nametype thread = {0,1}
nametype mutex = {0}
nametype memory = {0,1}
channel lock : thread.mutex
channel unlock : thread.mutex
channel fork : thread.thread
channel join : thread.thread
channel start : thread
channel end : thread
channel read: thread.memory
channel write: thread.memory
channel endthreads
THREAD_0 = fork.0.1 -> write.0.0 -> lock.0.0 -> write.0.1 ->
         unlock.0.0 -> join.0.1 -> SKIP
THREAD_1 = start.1 -> lock.1.0 -> write.1.1 -> unlock.1.0 ->
         write.1.0 -> end.1 -> SKIP
INTER = (THREAD_0 ||| THREAD_1) ; endthreads -> SKIP
MUTEX_0 = lock.0.0 -> unlock.0.0 -> MUTEX_0
        [] lock.1.0 -> unlock.1.0 -> MUTEX_0
FORK_1 = fork.0.1 -> start.1 -> STOP
JOIN_1 = end.1 -> join.0.1 -> STOP
SYNC = MUTEX_0 ||| FORK_1 |||  JOIN_1
sync_set = {|fork,join,start,end,lock,unlock|}
PROGRAM = INTER [| sync_set |] SYNC

channel race
RACE(k,event_set) = RACE_ERR(k) /\
                   ([] z:event_set @ z -> RACE(k,event_set))
RACE_ERR(k) = (read?t1:thread!k ->
               write?t2:diff(thread,{t1})!k -> race -> SKIP)
            [] (write?t1:thread!k ->
               write?t2:diff(thread,{t1})!k -> race -> SKIP)
events_0 = union({read.0.0,read.1.0,write.0.0,write.1.0},
         sync_set)
events_1 = union({read.0.1,read.1.1,write.0.1,write.1.1},
         sync_set)
assert STOP [T= (PROGRAM [| events_0 |] RACE(0,events_0)) |\
            {race} :[partial order reduce [precise]]
assert STOP [T= (PROGRAM [| events_1 |] RACE(1,events_1)) |\
            {race} :[partial order reduce [precise]]
```

A.2.5. File Example in Figures 6.6 and 6.7

```
nametype thread = {0,1}
nametype mutex = {0}
channel lock : thread.mutex
channel unlock : thread.mutex
channel fork : thread.thread
channel join : thread.thread
channel start : thread
channel end : thread
channel endthreads
THREAD_0 = fork.0.1 -> open.0.0 ->  join.0.1 -> close.0.0 ->
         SKIP
THREAD_1 = start.1 -> access.1.0 -> end.1 -> SKIP
INTER = (THREAD_0 ||| THREAD_1) ; endthreads -> SKIP
FORK_1 = fork.0.1 -> start.1 -> STOP
JOIN_1 = end.1 -> join.0.1 -> STOP
SYNC = FORK_1 |||  JOIN_1
sync_set = {|fork,join,start,end,lock,unlock|}
PROGRAM = INTER [| sync_set |] SYNC

files = Union({open_arg1,access_arg1,close_arg1})
channel open: thread.files
channel access: thread.files
channel close: thread.files
FILE(p) = open?t.p -> OPENED(p)
OPENED(p) = access?t.p -> OPENED(p)
          [] close?t.p -> FILE(p)
FILES = ||| x : files @ FILE(x)
assert FILES [T= PROGRAM |\ {|open,access,close|}
             :[partial order reduce [precise]]
```

A.2.6. Custom Race Detector Example in Figure 6.10

```
channel x : thread.{load,store}
channel y : thread.{load,store}

err_set(k) = union({|k|},sync_set)

ERR(k) = k?t1!load -> k?t2:diff(thread,{t1})!store -> SKIP
       [] k?t1!store -> k?t2:diff(thread,{t1})!store -> SKIP

channel error
WD(E,events) = (E; error -> SKIP) /\
             ([] z : events @ z -> WD(E,events))

assert STOP [T= ( PROGRAM [| err_set(x) |] WD(ERR(x),
   err_set(x))) |\ {error} :[partial order reduce [precise]]
assert STOP [T= ( PROGRAM [| err_set(y) |] WD(ERR(y),
   err_set(y))) |\ {error} :[partial order reduce [precise]]
```

A.2.7. Rover Example in Figure 6.13

```
channel open: thread
channel close: thread
channel update: thread
channel cancel: thread
channel ack: thread
channel commit: thread

UPLINK = open?t -> ((OPENED /\ cancel?t -> SKIP); UPLINK)
OPENED = close?t -> SKIP
       [] COMMIT
       [] update?t -> COMMIT
COMMIT = commit?t -> ack?t -> OPENED
assert UPLINK [T= PROGRAM |\{|open,close,cancel,update,
              commit,ack|} :[partial order reduce [precise]]
```

A.2.8. SQL Example in Figure 6.15

```
db = Union({open_arg1,close_arg1,prepare_arg1})
st = Union({prepare_arg2,bind_arg1,step_arg1,finalize_arg1,
   reset_arg1,column_arg1})

channel open: thread.db
channel close: thread.db
channel prepare: thread.db.st
channel bind: thread.st
channel step: thread.st
channel finalize: thread.st
channel reset: thread.st
channel column: thread.st

DB(x) = open?t!x -> OPEN(x)
OPEN(x) = prepare?t!x?s -> (OPEN(x) [] close?t!x -> SKIP)
STMT(x) = prepare?t?d!x ->
        (bind?t!x -> READY(x) [] READY(x))
READY(x) = step?t!x -> READY(x)
         [] reset?t!x -> (READY(x) [] bind?t!x -> READY(x))
         [] COLUMN(x)
         [] finalize?t!x -> SKIP
COLUMN(x) = column?t!x -> (COLUMN(x) [] READY(x))
SQL = ||| x : db @ DB(x) [| {|prepare|} |]
      ||| x : st @ STMT(x)
assert SQL [T= PROGRAM |\ {|open,prepare,close,bind,step,
   reset,column,finalize|} :[partial order reduce [precise]]
```

A.2.9. Atomicity Violation Example in Figure 6.18

```
channel x : thread.{store,load}
channel y : thread.{store,load}
channel enter: thread
channel exit: thread

err_set(k) = {|enter,exit,k|}

AV(t1,k) = k.t1.load -> k?t2:diff(thread,{t1})!store ->
                                          k.t1?any -> SKIP
         [] k.t1.store -> k?t2:diff(thread,{t1})!load ->
                                          k.t1.store -> SKIP
         [] k.t1.store -> k?t2:diff(thread,{t1})!store ->
                                          k.t1?any -> SKIP
PREAV(t,k) = AV(t,k) /\ ([] e:{|x|} @ e -> AV(t,k))
ERR(k) = enter?t -> (PREAV(t,k) /\ exit.t -> ERR(k))

channel error
WD(E,events) = (E;error -> SKIP)
             /\ ( [] z : events @ z -> WD(E,events)))
assert STOP [T= ( PROGRAM [| err_set(x) |] WD(ERR(x),
   err_set(x))) |\ {error} :[partial order reduce [precise]]
```

A.2.10. Resource Management Example in Figure 6.21

```
resources = Union({conflict_arg1,conflict_arg2,request_arg1,
   rescind_arg1,cancel_arg1})
size = request_arg2
channel conflict : thread.resources.resources
channel request : thread.resources.size
channel rescind : thread.resources
channel cancel : thread.resources

RES(k) = request?t!k?s ->
       (if s==0 then RES(k) else GRANTED(t,k))
GRANTED(t,k) = rescind.t.k -> GRANTED(t,k)
             [] cancel.t.k -> RES(k)

CONFI(m,n) = conflict?t!m!n -> CONF(m,n)
           [] request?t!m?s -> CONFI(m,n)
           [] request?t!n?s -> CONFI(m,n)
           [] cancel?t!m -> CONFI(m,n)
           [] cancel?t!n -> CONFI(m,n)
CONF(m,n) = request?t!m?s ->
       (if s==0 then CONF(m,n) else cancel?t!m -> CONF(m,n))
          [] request?t!n?s ->
       (if s==0 then CONF(m,n) else cancel?t!n -> CONF(m,n))
CONFB = [|{|request,cancel|}|] x:conflict_arg1,
     y:conflict_arg2 @ CONFI(x,y)
RESS = ( ||| x:resources@ RES(x))
     [| {|request,cancel|} |] CONFB

assert RESS [T= PROGRAM |\ {|request,rescind,cancel,
            conflict|} :[partial order reduce [precise]]
```

B. Data Race Unit Test Results

Table B.1 contains the detailed results for the data race unit test evaluation presented in Section 5.6.2. *Test name* identifies the test case and the corresponding test suite. *Expected* shows the expected result of the test, TRUE if it contains a data race, FALSE otherwise. The third, fourth, and fifth columns show the results for each tool: RaceQuest, ThreadSanitizer, and Helgrind. The values reflect whether a warning was emitted or not. An emitted warning in RaceQuest also implies a feasible counterexample. If the result of a tool does not match the expected value, the result is boldfaced. A summary at the end shows the number of true and false positives and negatives, along with the precision and recall.

Table B.1.: Data race detection unit test benchmark results - detailed

Test name	Expected	RaceQuest	TSan	Helgrind
Helgrind suite				
hg01_all_ok	FALSE	FALSE	FALSE	FALSE
hg03_inherit	TRUE	TRUE	TRUE	TRUE
hg04_race	TRUE	TRUE	TRUE	TRUE
hg05_race2	TRUE	TRUE	TRUE	TRUE
hg06_readshared	FALSE	FALSE	FALSE	FALSE
lock_vs_unlocked1	TRUE	TRUE	TRUE	TRUE
tc01_simple_race	TRUE	TRUE	TRUE	TRUE
tc02_simple_tls	FALSE	FALSE	FALSE	FALSE
tc03_re_excl	FALSE	FALSE	FALSE	FALSE
tc05_simple_race	TRUE	TRUE	TRUE	TRUE
tc06_two_races	TRUE	TRUE	TRUE	TRUE
tc16_byterace	FALSE	FALSE	FALSE	**TRUE**
tc21_pthonce	TRUE	TRUE	TRUE	TRUE
Causally-preceedes				
Fig 3	TRUE	TRUE	**FALSE**	**FALSE**
Fig 5	TRUE	TRUE	**FALSE**	**FALSE**
Fig 6	TRUE	TRUE	**FALSE**	**FALSE**
Fig 7	TRUE	TRUE	**FALSE**	**FALSE**
Fig 8	TRUE	TRUE	**FALSE**	**FALSE**
Fig 9	FALSE	FALSE	FALSE	FALSE
Data-race-test				
1	TRUE	TRUE	TRUE	TRUE
2	FALSE	FALSE	FALSE	FALSE
4	FALSE	FALSE	FALSE	FALSE
5	FALSE	FALSE	FALSE	FALSE
8	FALSE	FALSE	FALSE	FALSE
9	TRUE	TRUE	TRUE	TRUE
10	TRUE	TRUE	TRUE	TRUE
11	FALSE	FALSE	FALSE	FALSE
12	FALSE	FALSE	FALSE	FALSE

Continued on next page

Table B.1 – continued from previous page

Test name	Expected	RaceQuest	TSan	Helgrind
14	FALSE	FALSE	FALSE	FALSE
22	TRUE	TRUE	TRUE	TRUE
23	FALSE	FALSE	FALSE	FALSE
28	FALSE	FALSE	FALSE	FALSE
29	FALSE	FALSE	FALSE	FALSE
32	FALSE	FALSE	FALSE	FALSE
33	FALSE	FALSE	FALSE	FALSE
36	FALSE	FALSE	FALSE	FALSE
37	FALSE	FALSE	FALSE	FALSE
38	FALSE	FALSE	FALSE	FALSE
40	FALSE	FALSE	FALSE	FALSE
43	FALSE	FALSE	FALSE	FALSE
44	FALSE	FALSE	FALSE	FALSE
45	FALSE	FALSE	FALSE	FALSE
46	TRUE	TRUE	**FALSE**	**FALSE**
47	TRUE	TRUE	**FALSE**	**FALSE**
48	TRUE	TRUE	TRUE	TRUE
49	TRUE	TRUE	TRUE	TRUE
50	TRUE	TRUE	TRUE	TRUE
51	TRUE	TRUE	TRUE	TRUE
52	TRUE	TRUE	TRUE	TRUE
53	FALSE	FALSE	FALSE	FALSE
57	FALSE	FALSE	FALSE	FALSE
58	FALSE	**TRUE**	**TRUE**	**TRUE**
64	TRUE	TRUE	TRUE	TRUE
65	TRUE	TRUE	TRUE	TRUE
66	FALSE	**TRUE**	FALSE	FALSE
68	TRUE	TRUE	TRUE	TRUE
69	FALSE	FALSE	FALSE	FALSE
75	TRUE	TRUE	**FALSE**	TRUE
76	FALSE	FALSE	FALSE	FALSE
77	FALSE	FALSE	FALSE	FALSE
78	FALSE	FALSE	FALSE	FALSE

Continued on next page

Table B.1 – continued from previous page

Test name	Expected	RaceQuest	TSan	Helgrind
79	FALSE	FALSE	FALSE	FALSE
80	FALSE	FALSE	FALSE	FALSE
81	FALSE	FALSE	FALSE	FALSE
82	TRUE	TRUE	TRUE	TRUE
83	TRUE	TRUE	TRUE	TRUE
84	FALSE	FALSE	FALSE	FALSE
89	TRUE	TRUE	TRUE	TRUE
90	FALSE	FALSE	FALSE	FALSE
91	FALSE	FALSE	FALSE	FALSE
92	FALSE	FALSE	FALSE	**FAILED**
94	TRUE	TRUE	TRUE	TRUE
95	TRUE	TRUE	TRUE	TRUE
96	FALSE	FALSE	FALSE	FALSE
101	FALSE	FALSE	FALSE	FALSE
102	TRUE	TRUE	TRUE	TRUE
103	FALSE	FALSE	FALSE	FALSE
104	TRUE	TRUE	TRUE	TRUE
105	FALSE	FALSE	FALSE	FALSE
108	FALSE	FALSE	**TRUE**	**TRUE**
109	FALSE	FALSE	FALSE	FALSE
110	TRUE	TRUE	TRUE	TRUE
114	FALSE	FALSE	FALSE	**TRUE**
116	FALSE	FALSE	FALSE	FALSE
117	FALSE	FALSE	FALSE	**TRUE**
119	TRUE	TRUE	TRUE	TRUE
120	TRUE	TRUE	TRUE	TRUE
121	TRUE	**FALSE**	TRUE	TRUE
122	TRUE	TRUE	TRUE	TRUE
125	FALSE	FALSE	FALSE	**TRUE**
131	FALSE	FALSE	FALSE	FALSE
132	TRUE	TRUE	TRUE	TRUE
133	TRUE	TRUE	TRUE	TRUE
134	FALSE	FALSE	FALSE	FALSE

Continued on next page

Table B.1 – continued from previous page

Test name	**Expected**	**RaceQuest**	**TSan**	**Helgrind**
139	TRUE	TRUE	**FALSE**	**FALSE**
146	TRUE	TRUE	TRUE	TRUE
148	TRUE	TRUE	**FALSE**	**FALSE**
150	TRUE	TRUE	TRUE	TRUE
152	FALSE	FALSE	FALSE	FALSE
153	FALSE	FALSE	FALSE	FALSE
154	TRUE	TRUE	TRUE	TRUE
302	TRUE	TRUE	**FALSE**	**FALSE**
305	TRUE	TRUE	**FALSE**	**FALSE**
306	TRUE	TRUE	TRUE	TRUE
307	TRUE	TRUE	TRUE	TRUE
308	TRUE	TRUE	TRUE	TRUE
310	TRUE	TRUE	**FALSE**	**FALSE**
311	TRUE	TRUE	**FALSE**	**FALSE**
312	TRUE	TRUE	TRUE	TRUE
313	TRUE	TRUE	TRUE	TRUE
315	FALSE	FALSE	FALSE	FALSE
CyclicBarrier	TRUE	TRUE	**FALSE**	TRUE
RWLockVsRWLock	TRUE	TRUE	TRUE	TRUE
Summary				
Σ True Positives	58	57	43	45
Σ True Negatives	55	53	53	48
Σ False Positives	-	2	2	6
Σ False Negatives	-	1	15	13
Precision	-	0.9661	0.9556	0.8824
Recall	-	0.9828	0.7414	0.7759

Bibliography

[Bar06] R. Barik, “Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs,” in *18th International Workshop on Languages and Compilers for Parallel Computing*, 2006, pp. 152–169. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69330-7_11

[BH02] S. Bensalem and K. Havelund, “Reducing False Positives in Runtime Analysis of Deadlocks,” 2002. [Online]. Available: http://ntrs.nasa.gov/search.jsp?R=20030002784

[BLR02] C. Boyapati, R. Lee, and M. Rinard, “A Type System for Preventing Data Races and Deadlocks in Java Programs,” in *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2002, pp. 211–230. [Online]. Available: https://www.cse.umich.edu/techreports/cse/2006/CSE-TR-525-06.pdf

[BMT$^+$05] A. Bouajjani, M. Markus, T. Touili, M. Müller-Olm, and T. Touili, “Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems,” in *CONCUR 2005 16th International Conference on Concurrency Theory*, 2005, pp. 473–487. [Online]. Available: http://dx.doi.org/10.1007/11539452_36

[Boe11] H. Boehm, “How to miscompile programs with “benign” data races,” in *HotPar'11 Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2001255

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani, “System Deadlocks,” *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, 1971. [Online]. Available: http://doi.acm.org/10.1145/356586.356588

[CT15a] L. M. Carril and W. F. Tichy, “Interleaving Generation for Data Race and Deadlock Reproduction,” in *Proceedings of the 2Nd*

International Workshop on Software Engineering for Parallel Systems. Pittsburgh, PA, USA: ACM, 2015, pp. 26–34. [Online]. Available: http://doi.acm.org/10.1145/2837476.2837480

[CT15b] ——, "Predicting and Witnessing Data Races Using CSP," in *NASA Formal Methods: 7th International Symposium, NFM 2015*, K. Havelund, G. Holzmann, and R. Joshi, Eds. Pasadena, CA, USA: Springer International Publishing, 2015, pp. 400–407. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-17524-9_28

[CWC14] Y. Cai, S. Wu, and W. K. Chan, "ConLock: a constraint-based approach to dynamic checking on deadlocks in multithreaded programs," *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pp. 491–502, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2568225.2568312

[EA03] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 237–252. [Online]. Available: http://dl.acm.org/citation.cfm?id=945468

[FF08] C. Flanagan and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," *Science of Computer Programming*, vol. 71, no. 2, pp. 89–109, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=964023

[FF09] C. Flanagan and S. N. S. Freund, "FastTrack: efficient and precise dynamic race detection," in *PLDI '09 Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 2009, pp. 121–133. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542490

[Gai86] J. Gait, "A probe effect in concurrent programs," *Software: Practice and Experience*, vol. 16, no. 3, pp. 225–233, 1986.

[Goo09] Google, "Data-race-test Benchmark for ThreadSanitizer," 2009. [Online]. Available: https://github.com/lmcarril/data-race-test-

[GRABR13] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe, "Failures Divergences Refinement (FDR) Version 3," 2013. [Online]. Available: https://www.cs.ox.ac.uk/projects/fdr/

[GRABR14] ——, "FDR3 - A Modern Refinement Checker for CSP," *Tools and Algorithms for the Construction and Analysis of*

Systems, vol. 8413, pp. 187–201, 2014. [Online]. Available: http://www.cs.ox.ac.uk/files/6001/Document.pdf

[Hav00] K. Havelund, “Using runtime analysis to guide model checking of Java programs,” in *7th International SPIN Workshop, Stanford, CA, USA*, no. 7th International SPIN Workshop, 2000, pp. 245–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=672085

[Hav08] ——, “Runtime Verification of C Programs,” in *Testing of Software and Communicating Systems*, 2008, pp. 7 – 22. [Online]. Available: http://www.springerlink.com/content/31526744648107p1

[HMR14] J. Huang, P. Meredith, and G. Rosu, “Maximal sound predictive race detection with control flow abstraction,” in *PLDI '14 Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 337–348. [Online]. Available: http://dl.acm.org/citation.cfm?id=2594315

[Hoa78] C. Hoare, “Communicating Sequential Processes,” *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978. [Online]. Available: http://www.cs.ucf.edu/courses/cop4020/sum2009/CSP-hoare.pdf

[HQR15] J. Huang, L. Qingzhou, and G. Rosu, “GPredict: generic predictive concurrency analysis,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, 2015, pp. 847–857. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818856

[HR15] K. Havelund and G. Reger, *Specification of Parametric Monitors*, 2015. [Online]. Available: http://link.springer.com/10.1007/978-3-658-09994-7

[ISZBM99] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai, “Towards Integration of Data Race Detection in DSM Systems,” *Journal of Parallel and Distributed Computing - Special issue on software support for distributed computing*, vol. 59, no. 2, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=339736

[JNPS09] P. Joshi, M. Naik, C.-S. Park, and K. Sen, “CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs,” in *Proceedings of the 21th International Conference on Computer Aided Verification (CAV '09)*, 2009, pp. 675–681. [Online]. Available: http://dl.acm.org/citation.cfm?id=1575118

[JT08] A. Jannesari and W. F. Tichy, "On-the-fly race detection in multi-threaded programs," in *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD '08)*, 2008, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1390841.1390847

[JT10] ——, "Identifying ad-hoc synchronization for enhanced race detection," in *Proceedings of the International Symposium on Parallel & Distributed Processing (IPDPS '10)*, 2010, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5470343

[KH08] E. Koskinen and M. Herlihy, "Dreadlocks: Efficient Deadlock Detection," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures - SPAA '08*, 2008, p. 297. [Online]. Available: http://dl.acm.org/citation.cfm?id=1378585

[KVBA+99] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," *11th Euromicro Conference on Real-Time Systems*, no. June, pp. 114–122, 1999. [Online]. Available: http://dx.doi.org/10.1109/EMRTS.1999.777457

[LA04] C. Lattner and V. S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." in *International Symposium on Code Generation and Optimization*, 2004, pp. 75–88. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/CGO.2004.1281665

[Lam78] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978. [Online]. Available: http://dl.acm.org/citation.cfm?id=359563

[Lee06] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1137289

[LKK+99] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "Runtime assurance based on formal specifications," in *Proceed-*

ings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 1999, pp. 279–287. [Online]. Available: http://repository.upenn.edu/cgi/viewcontent.cgi?article=1311&context=cis_papers

[LLV] LLVM, "LLVM Capture Tracking." [Online]. Available: http://llvm.org/docs/doxygen/html/CaptureTracking_8h_source.html

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, vol. 36, no. 4, pp. 471–482, 1987. [Online]. Available: http://dx.doi.org/10.1109/TC.1987.1676929

[MQB07] M. Musuvathi, S. Qadeer, and T. Ball, "CHESS: A systematic testing tool for concurrent software," Tech. Rep. MSR-TR-2007-149, 2007. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=70509

[NM92] R. H. B. Netzer and B. P. Miller, "What are race conditions? Some issues and formalizations," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 1, pp. 74–88, 1992. [Online]. Available: http://dl.acm.org/citation.cfm?id=130623

[NPSG09] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 386–396. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5070538

[NS07] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1250746

[NWT$^+$07] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 22–31, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1250738

[PFH11] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Practical static race detection for C," *ACM Transactions on Programming Languages and Systems*, vol. 33, no. 1, pp. 3:1—-3:55, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1890000

[PS07] E. Pozniansky and A. Schuster, “MultiRace: efficient on the fly data race detection in multithreaded C++ programs,” *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1228969

[Ram94] G. Ramalingam, “The Undecidability of Aliasing,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, sep 1994. [Online]. Available: http://doi.acm.org/10.1145/186025.186041

[RB99] M. Ronsse and K. D. Bosschere, “RecPlay: a fully integrated practical record/replay system,” *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 133–152, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=312203.312214

[Ros10] A. Roscoe, *Understanding Concurrent Systems*, 1st ed. Springer-Verlag New York, Inc., oct 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1941861

[SBN$^+$97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: a dynamic data race detector for multithreaded programs,” *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391–411, nov 1997. [Online]. Available: http://doi.acm.org/10.1145/265924.265927

[Sch99] S. Schneider, *Concurrent and Real-Time Systems: The CSP approach*, 1st ed. John Wiley & Sons, Inc., 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=555233

[SES$^+$12] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, “Sound predictive race detection in polynomial time,” *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’12*, p. 387, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2103656.2103702

[SI09] K. Serebryany and T. Iskhodzhanov, “ThreadSanitizer: data race detection in practice,” in *WBIA ’09 Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009, pp. 62–71. [Online]. Available: http://dl.acm.org/citation.cfm?id=1791203

[SN16] A. Sankar and V. K. Nandivada, “Improved MHP Analysis,” in *CC 2016 Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 207–217. [Online]. Available: http://dl.acm.org/citation.cfm?id=2897144

[SOA08] A. Sen, V. Ogale, and M. S. Abadir, "Predictive runtime verification of multi-processor SoCs in SystemC," in *Proceedings - Design Automation Conference*, 2008, pp. 948–953. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1391469.1391708

[SQL] SQLite, "An Introduction To The SQLite C/C++ Interface." [Online]. Available: https://www.sqlite.org/cintro.html

[Ste93] N. Sterling, "WARLOCK - A Static Data Race Analysis Tool," in *USENIX Technical Conference*, 1993, pp. 97–106.

[Sut05] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, pp. 1–9, 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[SWYS11] M. Said, C. Wang, Z. Yang, and K. Sakallah, "Generating data race witnesses by an SMT-based analysis," in *NFM'11 Proceedings of the Third international conference on NASA Formal methods*, 2011, pp. 313–327. [Online]. Available: http://dl.acm.org/citation.cfm?id=1986334

[Val07] Valgrind, "Helgrind: a data-race detector," 2007. [Online]. Available: http://valgrind.org/docs/manual/manual.html

[VJL07] J. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code," in *ESEC-FSE '07 Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 205–214. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287654

[VTD06] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," *ACM SIGPLAN Notices*, vol. 41, no. 1, pp. 334–345, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1111320.1111067

[WOT$^+$95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs," in *Proceedings of the 22nd annual international symposium on Computer architecture - ISCA '95*, no. June, 1995, pp. 24–36. [Online]. Available: http://dl.acm.org/citation.cfm?doid=223982.223990

[XPZ$^+$10] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, "Ad Hoc Synchronization Considered Harmful," in *Proceedings of* *the 9th USENIX conference on Operating systems design and implementation.* USENIX Association, 2010, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924955

[YCG08] Y. Yang, X. Chen, and G. Gopalakrishnan, "Inspect: A Runtime Model Checker for Multithreaded C Programs," Tech. Rep. i, 2008. [Online]. Available: http://formalverification.cs.utah.edu/publications/conferences/pdf/spin07.pdf

[YRC05] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack : Efficient Detection of Data Race Conditions via Adaptative Tracking," *Computing*, vol. 39, pp. 221–234, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1095832

[ZT07] P. Zhou and R. . Y. Z. Teodorescu, "HARD : Hardware-Assisted Lockset-based Race Detection," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, 2007, pp. 121–132. [Online]. Available: http://dl.acm.org/citation.cfm?id=1318108