FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Concolic Testing of Concurrent Software in the Context of Weak Memory Models

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Martin Dobiasch
Matrikelnummer 0828302

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Helmut Veith
Mitwirkung: Dipl.-Inf.(FH) Dr.techn. Andreas Holzer, M.Sc.

Wien, 30.09.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Concolic Testing of Concurrent Software in the Context of Weak Memory Models

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Martin Dobiasch
Registration Number 0828302

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Helmut Veith
Assistance: Dipl.-Inf.(FH) Dr.techn. Andreas Holzer, M.Sc.

Vienna, 30.09.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Martin Dobiasch
Ramperstorffergasse 46/6, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I would like to thank Andreas Holzer for his continued support and valuable suggestions. Thank you also for teaching and explaining me a lot about scientific writing and research. Thank you for the countless hours you spent in meetings with me. They helped tremendously with the progress of this thesis.

Of course, I would like to thank my wonderful girlfriend Marion for all her support. Thank you for all the countless hours of listening to my gibberish about executions and uncommited values. Moreover, thanks for your humorous comments on my thesis and drawings in early drafts of it. They made writing this thesis a lot easier.

Last but not least, I would like to thank Professor Helmut Veith for giving me the opportunity to write this thesis in his FORSYTE group.

Abstract

With software becoming ubiquitous and increasingly complex, software reliability also becomes more important. Research on software testing and verification tries to create automated solutions which exhaustively search for software bugs. One promising example for these research efforts is concolic testing, a testing technique that combines concrete with symbolic execution and which is implemented in various recent testing tools.

However, testing of concurrent software still remains a challenge. One of the main difficulties for testing concurrent software is determining the order in which the statements from different threads have to be executed. The reason for this being a challenge is that the number of possible executions can be intractably big. Furthermore, weak memory models, which describe the memory architectures of modern processors, can further increase the number of possible executions. Weak memory models allow certain deviations from the expected thread-local program order. For example, the effects of a write from one thread to a memory location might not become immediately visible in another thread which reads from this particular memory address. This results from modern memory hierarchies when computations are distributed over several computing cores. The consequence of such behaviours can be software bugs that are hard or even impossible to detect with conventional testing techniques. Most testing tools are based on the assumption of the Sequential Consistency model, i.e., that these effects cannot occur.

This thesis presents CONCRESTWMM, a concolic testing tool for concurrent software that is able to simulate the effects of weak memory models during an execution of a program under test. CONCRESTWMM is implemented as an extension of the tool CONCREST, a testing tool for concurrent software. As a central component, CONCRESTWMM adds a WMM-Scheduler. By using WMM-schedules it is possible to select a WMM and to trigger effects of this model at specific points during the execution of a test case. Thus, the tool CONCRESTWMM offers the possibility to discover bugs that can occur on real-world processor architectures that do not adhere to Sequential Consistency but to another WMM.

Two WMMs are implemented as part of this thesis. First, the WMM Sequential Consistency which can be used to simulate the behaviour of CONCREST and second, the Partial Store Order (PSO) WMM, a model which allows to delay the effect of a write event. Memory barriers are a common means to limit the deviation of executions from Sequential Consistency. By using memory barriers it is possible to force the effect of writes to become visible. Support for memory barriers is also implemented in CONCRESTWMM. Since many concurrent applications and data structures make use of a Compare and Swap (CAS) operation, a CAS operation is also implemented as part of CONCRESTWMM. The CAS operation updates a memory cell with a new value if the cell contains an expected old value. The built-in operation offers the

same semantics as the Boolean CAS operation offered by the compiler GCC and is atomic. Moreover, the built-in CAS operation is equipped with symbolic information. This enables CONCRESTWMM to search for test cases covering both branches of the CAS-operation, i.e., finding test cases where the value at the memory location is changed and test cases where the value is not changed.

To test the implementation of CONCRESTWMM and to show its capabilities, several experiments were performed. These examples are taken from the literature. The experiments show that on average CONCRESTWMM is only a constant factor of ~ 1.15 slower than CONCREST when comparing both tools.

Kurzfassung

Mit zunehmend komplexerer und gleichzeitig allgegenwärtig werdender Software wird die Verlässlichkeit von Software immer wichtiger. Forschung zu Testen und Verifikation von Software versucht automatisierte Lösungen zu finden, welche Software gründlich nach Fehlern durchsuchen. Ein Beispiel für Forschungserfolge ist Concolic Testing, eine Kombination aus konkretem und symbolischem Testen, welches in vielen aktuellen Test-Werkzeugen verwendet wird.

Allerdings ist das Testen von nebenläufiger Software nach wie vor ein offenes Problem. Eine der Hauptschwierigkeiten beim Testen solcher Software ist das Bestimmen der Reihenfolge in welcher die Befehle der verschiedenen Programmstränge ausgeführt werden sollen. Der Grund für die Schwierigkeit dahinter liegt in der Zahl der Reihenfolgen der verschiedenen Reihenfolgen welche unlösbar groß sein kann. Hinzukommen sogenannte Weak Memory Models (WMMs), welche Speicherarchitekturen beschreiben und die Zahl der Reihenfolgen weiter erhöhen können. Weak Memory Models erlauben Abweichungen von dem erwarteten Prozesslokalem Programmablauf. Zum Beispiel kann es passieren, dass der Effekt eines Schreibzugriffs von einem Programmstrang auf eine Speicheradresse in einem anderen Programmstrang, welcher von dieser Adresse liest, nicht sofort für alle andere Threads sichtbar wird. Der Grund für diese Abweichungen liegt in modernen Speicherhierarchien welche es erlauben, dass Berechnungen über mehrere Prozessor-Kerne aufgeteilt werden. Aus diesem abweichenden Verhalten können Fehler resultieren, welche mit konventionellen Testtechniken nur schwer wenn nicht sogar unmöglich zu finden sind. Die meisten Test-Werkzeuge basieren auf der Annahme, dass diese Effekte nicht auftreten können. Dieses Annahme wird auch als sequentielle Konsistenz bezeichnet

Diese Arbeit präsentiert CONCRESTWMM, ein Programm welches Concolic Testen verwendet um nebenläufige Software zu testen und welches in der Lage ist die Effekte von Weak Memory Models während der Ausführung von Tests zu simulieren. CONCRESTWMM wurde als eine Erweiterung von CONCREST, ein Test-Werkzeug für nebenläufige Software, implementiert. Der WMM-Scheduler bildet eine zentrale Komponente von CONCRESTWMM. Durch die Verwendung von WMM-Schedules ist es möglich die Effekte eines WMMs auszuwählen und an einem bestimmten Punkt während der Ausführung eines Testfalls auszuführen. Als Konsequenz daraus ist es möglich mit CONCRESTWMM Fehler, welche auf modernen sequentiell inkonsistenten Prozessoren auftreten können, zu entdecken.

Zwei Weak Memory Models werden als Teil dieser Arbeit präsentiert. Als erstes das WMM Sequential Consistency, welches verwendet werden kann um das Verhalten von CONCREST zu simulieren und als zweites das Partial Store Order (PSO) WMM, ein Model, welches es ermöglicht die Effekte eines Schreib-Events zu verzögern. Speicher-Barrieren sind ein gebräuchliches

Mittel um Abweichungen von sequentieller Konsistenz zu vermeiden. Durch die Verwendung von Speicher-Barrieren ist es möglich die Sichtbarkeit der Effekte von Schreib-Events sichtbar zu machen. Deshalb sind Speicher-Barrieren in CONCRESTWMM unterstützt. Da viele nebenläufige Programme und Datenstrukturen Compare and Swap (CAS) Operationen verwenden, ist sie auch in CONCRESTWMM realisiert. Die CAS Operation aktualisiert eine Speicheradresse mit einem neuen Wert sollte der Wert an der Adresse mit einem erwarteten Wert übereinstimmen. Die eingebaute Operation bietet die gleiche Semantik wie die vom GCC Compiler angebotene Boolsche CAS Operation und ist atomar. Zusätzlich ist die eingebaute CAS Operation mit symbolischer Information ausgestattet. Das ermöglicht es CONCRESTWMM nach Testfällen zu suchen welche beide Stränge der CAS Operation verwenden, d.h., dass Testfälle gesucht werden, welche den Wert an der Speicheradresse verändern und Testfälle welche dies nicht tun.

Um die Implementierung von CONCRESTWMM zu Testen und ihre Fähigkeiten zu demonstrieren werden mehrere Beispiele durchgeführt. Diese Beispiele stammen aus der Literatur. Die Experimente zeigen, dass im Durchschnitt CONCRESTWMM nur einen konstanten Faktor von ~ 1.15 langsamer ist als CONCREST wenn die Werkzeuge verglichen werden.

Contents

CHAPTER 1

Introduction

1.1 Motivation & Problem Statement

Software becomes more and more ubiquitous in our daily and, at the same time, the complexity of software increases. This poses a major challenge for the reliability of software. To mitigate this problem, research on software testing and verification tries to create automated solutions which exhaustively search for software bugs. Although there has been progress in developing automated testing tools for sequential testing [9], testing concurrent software still remains as a challenge. Concurrency increases the amount of possibilities for tests exponentially since the scheduling of statements from different threads adds an additional factor, i.e., it is also necessary to investigate in which order program statements are executed across different threads.

Weak Memory Models. Most of the test and verification tools for concurrent software rely on the assumption of *sequential consistency*. This means that the execution of a concurrent program is the same as if the statements from different threads had been brought to a sequential order and executed sequentially while the thread-local order remains unchanged for all threads. However, unlike sequential consistency, so-called *Weak Memory Models* (WMMs) allow certain deviations from a thread-local program order. For example on modern processors a write to a shared variable might be stored in a local buffer before it is stored in global memory. Thus, the executing thread can see the new value of this variable while other threads might still see the old value. We say that a write is committed when all threads see the new value. Another interesting aspect of modern CPUs is that they have multiple pipelines for commands to be executed. This stems from the fact that they have different processing units (e.g. for integer operations, for floating point operations, ...). Whenever an instruction is being processed but has not been finished yet, it is referred to as being in an *in-flight* state. As a result of WMMs, an instruction can remain in an in-flight state while other subsequent instructions have been committed already. By doing so the processor can save time since it does not have to wait for an instruction - like a write - to finish until the next instruction can be executed. This can result in a behaviour that for thread A it seems that another thread B does not respect the thread-local order. One aspect that

Thread A	Thread B
$x = 1$	$r1 = y$
$y = 1$	$r2 = x$
Initial state: $x = 0 \wedge y = 0$	
Forbidden: $r1 = 1 \wedge r2 = 0$	

Figure 1.1: Message Passing Example from [12]

increases the difficulty of understanding these models of computation is that they are usually not published in a comprehensible manner but instead hidden in long technical documentations of a specific processor [12]. One major problem during testing is that testing a software on a machine with a WMM can lead to unexpected program results which are difficult to explain and are hard to reproduce since WMMs allow WMM-specific effects, but do not require the computation to perform the same effects again when running the program anew. In summary, WMMs can make identifying faults via manual code inspection extremely hard if not impossible.

Figure 1.1 shows an example with two threads. The first thread performs two writes on two shared variables x and y while the second thread reads the values from these two shared variables and writes them into two local variables $r1$ and $r2$. Moreover, the example defines a state which is infeasible when considering sequential consistency (denoted as 'forbidden'). In the initial state both x and y are initialised with the value 0. Under sequential consistency the forbidden state cannot be reached. However, this state is feasible for ARM and POWER architectures [12]. One possibility to reach this state is as follows: First thread A starts executing the write $x = 1$ but does not commit it. Then the second write $y = 1$ is executed but this time the write is committed immediately. Next, thread B takes over reading the value of y (1) and writing it to $r1$. After this, it executes the second read, thus storing the initial value of x (0) to $r2$. The forbidden state is thereby reached. From a technical point of view there are two possibilities for this scenario. The first possibility is that the compiler or processor performed an instruction reordering on thread A. As a consequence of this, the write $y = 1$ is executed as first instruction from thread A. Next, thread B read the written value and stores it to $r1$ ($r1 = y$) and subsequently reads $x = 0$ and writes the value to $r2$ ($r2 = x$). Thus, a state $r1 = 1 \wedge r2 = 0$ is reached. The second possibility is that the executing processor uses a store buffer. While x is buffered for thread A, y is written directly to the memory. As a consequence of this an execution will write 1 to the cache for x and 1 to the memory address of y. The cache for x is not cleared immediately, i.e., the memory still contains 0 as value for x. Thread B will thus read the values 1 for y and 0 for x and reach the forbidden state.

Concolic Testing. In past years various testing tools have been developed featuring concolic testing [9], a testing approach that combines the concrete execution of a program with a simultaneous symbolic execution. However, only a few of these tools [17] can test concurrent software. A testing approach derived from concolic testing which can test concurrent software is $(con)^2colic$ testing (*con*crete and symb*olic* execution of a *con*current program) [7]. Figure 1.2 illustrates the high-level architecture of $(\text{con})^2$colic testing which can mainly be divided into two parts: an execution engine and a reasoning engine. The execution engine gathers symbolic

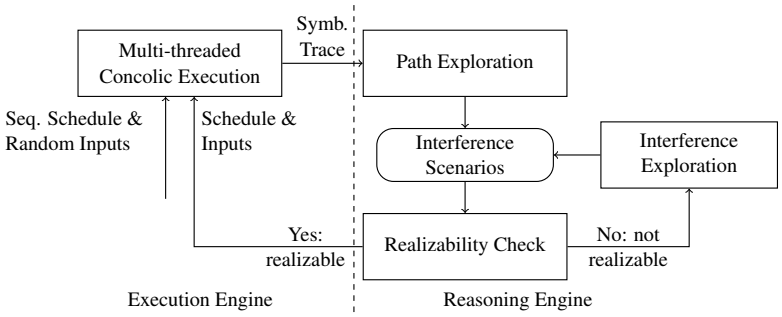


Figure 1.2: **Overview of $(\mathbf{Con})^2$colic Testing [7].**

constraints which the reasoning engine then modifies to generate new execution paths. The reasoning engine explores interference scenarios by combining and modifying gathered constraint systems. The $(\text{con})^2$colic algorithm works in the following way: First, the program under test is executed and (symbolic) information is collected. Based on gathered information alternate program runs are then generated and tested for their realisability. However, there could be an intractable amount of such alternate program runs. The tool CONCREST [7] implements $(\text{con})^2$colic testing as an extension of CREST[1]. However, CONCREST has no means to test programs with respect to the effects of weak memory models and instead relies on sequential consistency. As a consequence of this, certain aspects of concurrent behaviour remain untested. *The problem to be solved in this thesis was to add support for weak memory models in $(con)^2$colic testing.*

1.2 Results

This thesis implemented the tool CONCRESTWMM which extends the execution platform of CONCREST with the capability to simulate effects of weak memory models. CONCRESTWMM provides an interface to enable the precise control of the effects of a WMM. To support exist different WMMs, CONCRESTWMM is configurable with regard to which WMM is used while performing program executions. Two WMMs have been implemented: *Sequential Consistency* as a default WMM-model which simulates the behaviour of CONCREST and the WMM *Partial Store Order* (PSO). CONCRESTWMM was designed to preserve CONCREST's feature to execute dynamic analysis tools, like Valgrind, in parallel to a test execution.

1.3 Organisation

In Chapter 2 $(\text{con})^2$colic testing and CONCREST an implementation of it are discussed. For this purpose first concolic testing is introduced in Section 2.2. Next, $(\text{Con})^2$colic is introduced in Section 2.3. CONCREST as an implementation of $(\text{con})^2$colic testing is then explained in detail in Section 2.4. Chapter 3 contains implementation details for the implemented simula-

[1] `http://code.google.com/p/crest/`

tor. First the architecture of CONCRESTWMM is described in Section 3.1. After a description of event scheduling in Section 3.2, the implemented memory models are discussed in Sections 3.3. Section 3.4 introduces memory barriers and discusses how they can be used with CONCRESTWMM. Next, Section 3.6 describes Compare and Swap and its implementation in CONCRESTWMM. Subsequently, the limitations and possible extensions of CONCRESTWMM are outlined in Section 3.7, followed by a discussion of the capabilities of CONCRESTWMM in Sections 3.8 and 3.9. The conducted experiments are outlined in Chapter 4. Additionally, benchmarks are presented in the last section of this chapter. In Chapter 5 an overview over related work is given. It contains a description of partial orders in Section 5.1 and furthermore how these orders are used within CBMC in order to perform bounded model checking for WMMs. Next, Section 5.2 presents RELAXER a testing tool which is able to simulate WMM-effects during the execution of a program under test. Finally, a conclusion and future work is given in Chapter 6.

CHAPTER 2

CONCREST and (Con)^{2}colic Testing

This chapter describes the testing approach (con)^{2}colic testing and the tool CONCREST which implements it. In Section 2.1 some basic definitions are introduced which are used throughout this chapter. In Section 2.3 the testing approach (con)^{2}colic testing is presented. Section 2.4 then presents CONCREST as an implementation of (con)^{2}colic testing.

2.1 Glossary and Definitions

Program under test is computer program which is subject to examined for included bugs. It can vary between a simple program consisting only of an entry point with some additional functions contained in a single source code file and a large scale application consisting of several source files.

Propositional logical formulae are formulae comprised of variables and constants over the symbols **T** (*true*) and **F** (*false*) while using (at least) the connectives $\wedge$ and $\neg$. The evaluation of any syntactically correct formula results in either true or false. A formula is satisfiable iff there exists a variable assignment so that the evaluation of the formula under that assignment evaluates to true.

First-Order Theory First order logical formulas may be quantified by universal and existential quantifiers and contain functions and predicates. In addition to quantification a formula may make use of first-order theories extending the expressiveness to, for example, uninterpreted functions. A first-order theory provides a set of predicates and function symbols and additionally, axioms which hold on these introduced predicates and functions.

Static Single Assignment is used when data-flow in a program has to be modelled. For each write access to a variable a new variable-symbol is introduced. For example, the initialisation $x = 0$ gets transformed into $x_0 = 0$.

```
1 int abs(int x) {
2   if( x < 0 )
3     return -x;
4   if( x == 1 )
5     return -x;
6   return x;
7 }
```

Figure 2.1: Faulty implementation of a function returning the absolute value of its parameters. Adopted from [5].

SAT solver is a program which is able to decide whether a given formula is satisfiable or unsatisfiable. For satisfiable formulas it is furthermore able to provide evidence via a variable assignment.

SMT solver (**S**atisfiable **M**odulo **T**heory) is a program which is able to decide whether a given formula using a set of first-order theories is satisfiable or unsatisfiable modulo the used theories.

2.2 Concolic Testing

Concolic testing is the combination of *conc*rete and symb*olic* testing. Concolic testing was independently developed in several papers [5, 9, 20] which propose similar techniques. The technique has several other names such as *directed systematic test generation* [9] or *execution generated testing* [5].

One of the roots of concolic testing is another automatised approach called random testing where inputs for a program are guessed in order to generate test cases for the program under test. However, the approach of random testing has several problems for testing software in a comprehensive manner. For example for a program fragment like `if(x==1) abort();` it is hard to reach the error state using random testing since there are 2^{32} possible inputs, assuming x is an 32-bit integer and there are no further constraints on it. As a result of this it is quite impracticable to generate a test case which will follow a path through a program to this exact error/problem state. The problem is better illustrated in Figure 2.1 where an example of a faulty implementation of a function returning the absolute value of the function parameter is shown. The implementation returns an incorrect value only if the input is 1; for all other possible inputs the function is correct. While it is hard for random testing to generate a test case exploiting the bug, concolic testing, however, is able to detect a bug like this easily.

Figure 2.2 shows the concolic execution of the `abs` function defined in Figure 2.1. Prior to the first step no constraints are gathered and thus the random value 15 is chosen for x. The program is then executed visiting the statements in lines $1, 2, 4$ and 6 and returning 15 as output value. During this execution the two path-constraints $x < 0$ and $x = 1$ are gathered. In the next step $x = -1$ is generated as a solution for the constraint $x < 0$. Thus, the program execution visits the lines $1, 2$ and 3 returning -1. For the remaining constraint $x = 1$ is generated as

Constraints	Input	Execution
$\{\}$	$x = 15$	$1, 2, 4, 6$
$\{\{\mathbf{x<0}\}, \{x = 1\}\}$	$x = -1$	$1, 2, 3$
$\{\{\mathbf{x = 1}\}\}$	$x = 1$	$1, 2, 4, 5$

Figure 2.2: Concolic execution of abs-function in 2.1.

input value. This causes the program execution to traverse lines $1, 2, 4$ and 5. As a result of the outlined concolic testing of the `abs` function the three test cases $x = 15$, $x = -1$ and $x = 1$ are generated. When used by a testing tool which verifies the output of the `abs` function these test cases will exploit the bug of the function.

However, symbolic testing without using concrete values has limitations for certain classes of programs as can be observed when testing a simple example as shown in Figure 2.3. The branch created by the `if` statement in line 2 has a non-linear constraint, thus causing most symbolic testing tools to stop the test execution since most solvers are not able to reason about non-linear arithmetic [9]. Concolic testing tries to overcome the problems of random testing and symbolic testing by combining concrete and symbolic testing. As a result of this it is possible to use the concrete values as a fall-back and continue the (symbolic) execution when the symbolic execution fails to provide the values due to circumstances like limitations of the used solver or theory. Thus, it is possible for concolic testing to correctly examine the program in Figure 2.3 with the result that the `abort` statement in line 7 is not reachable but is reachable in line 4. The statement is not reachable since the $x^3 <= 0$ branch can only be reached if x is negative. Thus it is not possible to find input values so that $x > 0 \wedge y = 20$ is satisfied. However, as mentioned before, concolic testing will correctly report that the error state in line 4 is reachable.

```
1 void nonlinear(int x, int y){
2   if (x*x*x > 0){
3     if (x>0 && y==10)
4       abort();
5   } else {
6     if (x>0 && y==20)
7       abort();
8   }
9 }
```

Figure 2.3: Non linear path constraints. Taken from [9].

Another example where pure symbolic testing fails to discover the bug can be seen in Figure 2.4. When assuming that no symbolic information about the $hash$ function is at hand a symbolic execution has to stop. Concolic testing, however, can just use the concrete value provided during the execution. Thus, the testing of the function could proceed as follows. The initial inputs of $x = 0$ and $y = 0$ are guessed and used for the next test case of the program.

```
1  int obscure(int x, int y) {
2    if( x == hash(y) )
3      abort();
4    return 0;
5  }
```

Figure 2.4: Obscure-Function. Taken from [8].

```
1  int obscure(int x, int y) {
2    LOAD(y);
3    tmp = hash(y);
4    STORE(tmp);
5    LOAD(x);
6    LOAD(tmp);
7    BRANCH(EQ);
8    if( x == tmp )
9      abort();
10   return 0;
11 }
```

(a) Instrumented Code

Input	Symbolic Information
$x = 0, y = 0$	$x = 0 \wedge y = 0$ $\wedge(y = 0 \rightarrow tmp = 42)$ $\wedge\neg(x = tmp)$
$x = 42, y = 0$	$x = 42 \wedge y = 0$ $\wedge(y = 0 \rightarrow tmp = 42)$ $\wedge(x = tmp)$

(b) Symbolic Informatione

Figure 2.5: Concolic execution of `obscure`.

During this execution the evaluation of $hash(0)$ delivers the concrete value 42. Thus, the *else*-branch is taken. In order to cover the *then*-branch the algorithm can now select 42 as input value for x. When executing the function again with the input values $x = 42$ and $y = 0$ the execution will take the *then*-branch[1] and discover the `abort` statement.

Working Principle

Figure 2.5 illustrates a concolic execution of the `obscure` function. Most concolic testing tools (like [5, 20]) work by first instrumenting a program in order to be able to gather symbolic information during the concrete execution of the program. This means that additional statements are added to the original source, thus generating a new source file. The instrumented code of `obscure` is presented in subfigure 2.5a. Some tools, like CREST, for example, also simplify or change the program statements (without changing the semantics of the program under test). For the instrumentation automatic tools like CIL [15] are used [5, 17, 20]. The instrumented program under test rather than the original program is then compiled and executed during the performed tests. During each test run the instrumented code will then gather the symbolic information while being executed and without changing the behaviour of the program under test. This symbolic information contains amongst other data constraints as well as path constraints. The gathered

[1] Assuming a deterministic hash function.

constraint systems can then be used to generate new test cases. In subfigure 2.5b the gathered symbolic information for two test cases is illustrated for the `obscure` function. For each test case the symbolic information in the second column of the table shows in the first row input data constraints, in the second row data constraints and in the third path constraints.

Path constraints contain the information about the evaluation of branching conditions within the program. A path constraint thus can also be seen as an equivalence class for inputs since it describes exactly which elements will result in the same control flow, i.e., lie in the same class of inputs. In general, the solution to these constraints is not unique. The path constraints are usually constructed from branching conditions within the program under test. In order to cover all branches of the program under test a path constraint can be flipped and then be used for obtaining new input values. These input values should then guide the execution to the previously unvisited branch. However, the program run might fail to visit the predicted branch. This can happen, for example, if the program makes use of non-deterministic functions.

Limitations of concolic testing

One advantage from the developer's point of view of concolic testing over some other automated testing approaches, like random testing, is that once a bug is found using concolic testing the developer can be presented with a test case causing the bug and additionally also with a trace for this bug. Having a trace and a test case can often reduce the effort needed for analysing the problem.

A problem concolic testing suffers from is the dependence on deterministic programs. When, for example, the `hash` function from the example in Figure 2.4 is non-deterministic, concolic testing will probably fail to discover the bug. Moreover, an incomplete concolic testing approach could for other more complex programs claim that no bugs exists.

Another problem of concolic testing are irreversible functions like, for example, the hashing function MD5. Concolic testing makes always use to overcome this issue. However, the execution will not always discover new branches. For a simple program like the program in Figure 2.4 the fall-back concrete values can lead to new input values thus guiding the search to successfully testing the program. However, for other programs the concrete values might not be successful in triggering a new branch.

Due to the fact that most concolic testing tools rely on SMT solvers for solving path constraints and the like, concolic testing is incomplete. Currently there are several undecidable theories like non-linear arithmetic, integer division and modulo. The fact that they are undecidable affects symbolic testing and thus also concolic testing. However, as mentioned before concolic testing can try to overcome this problem by falling back on the concrete values recorded during the execution.

Finally, the design of most concolic testing tools is not yet suited for testing concurrent software.

2.3 $(\text{Con})^2$colic Testing

A testing approach that is able to test concurrent software is $(con)^2colic$ testing. The term $(\text{con})^2$colic testing assembles from the terms *con*crete, symb*olic* and *con*current. $(\text{Con})^2$colic testing is based on concolic executions of concurrent software. Based on the concolic executions of a program under test it derives inputs and schedules for the program so that its execution space is explored systematically [7].

Preliminaries

In order to be able to describe $(\text{con})^2$colic testing in more depth some more definitions have to be explained upfront.

Coverage. One of the main challenges for testing concurrent software is defining coverage of a concurrent program. While for sequential programs it is straight-forward to define useful criteria like condition, branch and statement coverage, for concurrent software it remains a hard problem with no commonly agreed standard solution. A reason for this is that in addition to the usual criteria mentioned above, interferences between processes have to be taken into account. This means that for shared variables it has to be taken into account in which order the various processes access (i.e., read or write) variables within the program under test. A brute force search to solve this problem would be intractable since the number of all possible program orders is exponential. *$(Con)^2$colic testing provides a coverage guarantee over the space of program inputs and interleavings* [7].

Concurrent Programs. A program usually consists of a finite sequence of program statements which are executed in some order. This order is of course not always from top to bottom since programs usually consist of conditional (e.g. branches and loops) and unconditional (e.g. function calls) jumps. A concurrent program consists of a countable amount of threads $T = \{T_1, T_2, ...T_n\}$, where each of these threads consists of a finite amount of program statements.

Schedules. As mentioned before the main challenge of testing concurrent software is to find the order of statements throughout the different threads causing a bug. On modern systems several statements can be executed at the same time, i.e., the execution of a program can be concurrent. However, there exists a total order over the statements. Using this total order the concurrent execution can be simulated by bringing the statements from the threads of the program under test into a sequential order. The way the statements from the different threads are aligned, i.e., the number of instructions a thread can perform before it has to wait for its next turn, is called a *schedule* σ. Hence, a schedule is a sequence of tupels (T_{i_j}, n_j) where T_{i_j} is a thread and n_j is the number of statements from this particular thread to be executed.

Events. In order to be able to track the relevant information of the execution the events *tf*(T_i), *ac*(l), *rel*(l), *br*(ψ), *rd*(x, r) and *wt*(x, val) are defined. The first event represents the forking of

```
1  int x, y;
2  int main() {
3    ...
4    pthread_create(&thread1, NULL, Thread1, NULL);
5    ...
6  }
7
8  int Thread1(void* param) {
9    LOAD(x);
10   LOAD(y);
11   BRANCH(EQ);
12   if( x == y ) {
13     LOAD(1);
14     STORE(X);
15     x = 1;
16   } else
17     abort();
18
19   return 0;
20 }
```

(a) Instrumented Code

Line	Event
4	*tf*(*Thread1*)
10	*rd*(x, x_0)
11	*rd*(y, y_0)
12	*br*$(x_0 = y_0)$
14	*wt*$(x, 1)$

(b) Events

Figure 2.6: A small program which creates a second thread. In this thread the two shared variables x and y are compared. Depending on the outcome of the compare the program is aborted or the value of x is set to 1.

a thread. For example, line 4 in Figure 2.6a creates a new thread which is recorded as the event *tf*(*Thread1*). The next two events (*ac* and *rel*) are required to be able to deal with locks, i.e., acquiring and releasing a lock l. Whenever a program branches using a predicate ψ this information is recorded using the *br* event. The *rd* event represents a read from a shared variable x where the result/value of this read is represented by the symbolic value r. In Figure 2.6a the value of x has to be loaded before the comparison to y this is recorded as the event *rd*(x, x_0). When a write to x using the symbolic value val is observed during the program execution it is represented by the write event *wt*. For example, setting x to 1 in line 15 is recorded as the event *wt*$(x, 1)$. Thus, a concurrent program consists of the set of threads T, the set of input variables *IN*, the set of shared variables *SV*, the set of local variables *LV*, and the set of locks L. To be semantically correct the information of all events are recorded using a *Static Single Assignment* (SSA). Using these events it is possible to track the flow of information between the various threads T_i of a program.

Symbolic Traces. When a concurrent program is executed concolically it will yield a finite string π containing the events of the program in their order of occurrence. Furthermore, $\pi|_{T_i}$

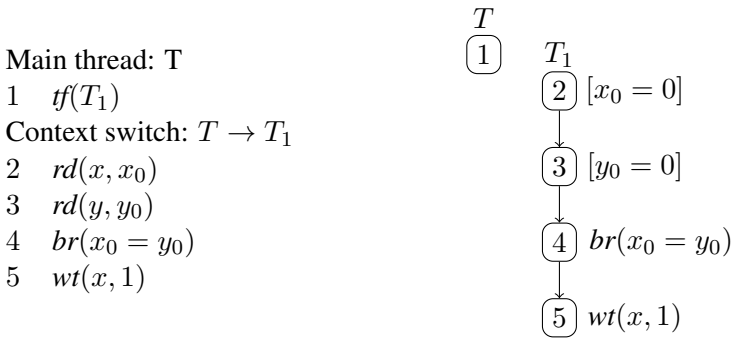


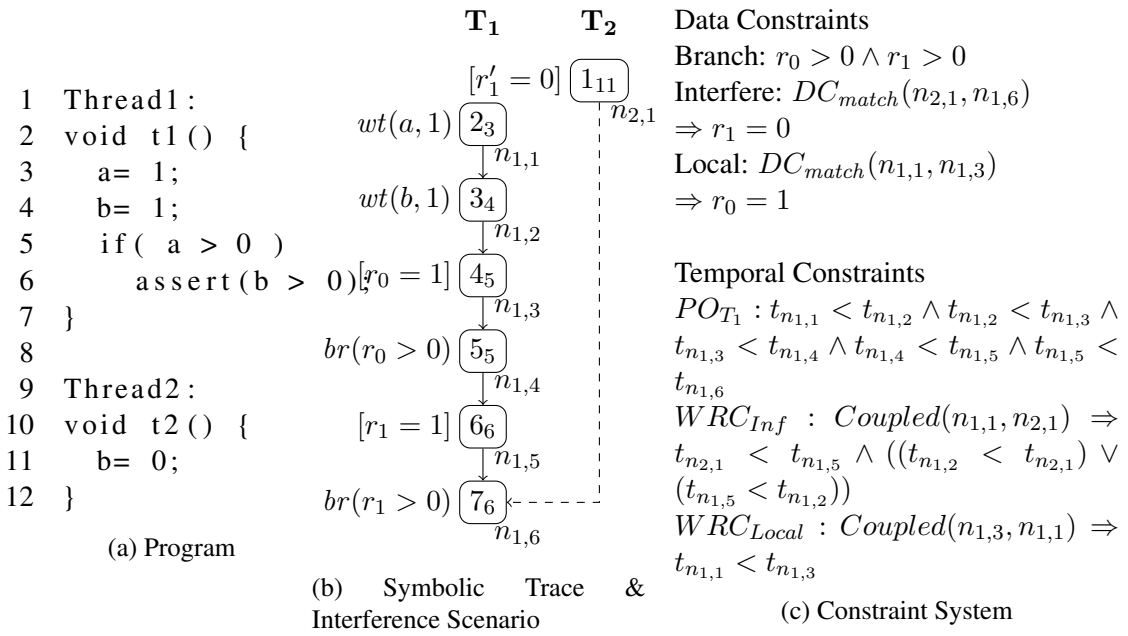
Figure 2.7: Symbolic trace

denotes the events involving thread T_i. A symbolic trace for the program in Figure 2.6 is shown in Figure 2.7. The trace is computed for a program execution where $x = 0$ and $y = 0$.

Interferences. The interesting program runs are those where a thread T_j reads a value written by another thread T_i, i.e., an *interference* occurs. To systematically record and examine all interferences of interest $(\text{con})^2$colic testing defines so-called *interference scenarios* IS. An interference scenario describes a class of program executions where the same interferences will happen during the program execution. An IS is a set of thread-local program executions (traces) extended with the information of interferences between them. The nodes in the IS represent the events from the program executions. The edges in the IS consist of two disjoint sets E_L and E_I where E_L is the set of thread local edges and E_I is the set of interference edges connecting two threads. E_L can further be subdivided into the disjoint sets E_{T_i} of the threads of the program. Each of these E_{T_i} induces a subforest G_{T_i} consisting only of nodes of the thread T_i. An interference scenario I is called a *realizable interference scenario* iff there exists a feasible partial program run where the extracted interference scenario from its symbolic trace coincides with I [7]. If the sink n of an interference scenario C is a branching event, i.e., $Ac(n) = br(\psi)$, then C is called an interference scenario candidate (ISC) for n.

Constraint Systems

In order to check whether an interference scenario is realisable two constraint systems are used: Data Constraints DC and Temporal-Consistency Constraints TC. The data constraints DC for an interference scenario consist of branch, interference and local constraints. The branch constraints ensure that the program execution will follow the desired path in the program, i.e., so that the evaluation of control statements, like *if*-statements within the program will result in the desired evaluation (*true/false*). An Interference constraint relates the read from a shared variable to the symbolic value of a write event. Any solution to the interference constraints will include the read-write interferences. In addition the local constraints are used to block out interferences from other threads for a read in a thread which should receive the value from a thread-local write. Thus, any solution to the data constraints of an interference scenario defines an input vector to the concurrent program.



(a) Program

(b) Symbolic Trace & Interference Scenario

(c) Constraint System

Figure 2.8: A small program with a concolic execution and a constraint system

Temporal-consistency constraints TC for an interference scenario, on the other hand, can be used to find schedules for a concurrent program. In order to relate events in a time-concerning manner a set of integer variables t_n is introduced encoding for every event n its index within the symbolic trace π. The TC constraints can further more be subdivided into four categories of constraints. The first category defines thread-local program-order consistency. Constraints from this category ensure that any potential schedule for the program needs to execute the statements from the various threads according to their order within the threads. The second category, the thread-fork consistency constraints, ensure that no thread can be scheduled before it has been created, i.e., forked by another thread. Lock consistency constraints define the third category of constraints ensuring that for every lock no pair of threads can acquire the lock at the same time. In order to do so for every lock acquisition event the corresponding lock release event is identified thus defining a lock block. For every lock it can then be guaranteed that in a potential schedule no overlapping blocks exist. Furthermore, it is ensured that for a lock which is never released in a schedule it is only acquired once all other threads have released this particular lock. The last category of constraints define write-read consistency within a potential schedule. For a pair of a read and write event a coupled block is defined in the constraints so that no other write or read event can occur between the events in a potential schedule. The coupled read-write events are formed from interferences and local read-write events.

Figure 2.8 shows an example for a constraint system. In Subfigure 2.8a a small program is outlined. It consists of two threads and two shared variables a and b. The first thread sets both a and b to 1 and then tests whether $a > 0$. If this is the case then $b > 0$ is asserted. The second thread contains a data race since it sets b to 0. Subfigure 2.8b shows a symbolic

trace of an execution of the program. This execution first executed thread 2 and subsequently thread 1. Thus, the assertion was not violated. Moreover, an interference scenario containing an interference from the write to b in thread 2 to the read of b required for verifying the assertion is shown. Subfigure 2.8c shows the constraint system constructed for the interference scenario. Note, that since the program does not contain any locks also the constraint system does not contain constraints for the lock. Moreover, since only a program stub is depicted thread-fork consistency constraints are also omitted.

Algorithm

In this section a high level description of the $(\text{con})^2$colic testing algorithm is outlined. The exact algorithm and more information can be found in [7]. The algorithm tries to iteratively increase branch coverage. During its iteration the number of interferences k is increased up to k_{max}.

The algorithm uses two list of sets: a list of worklists W^i for ISCs having degree i with $0 \leq i \leq k_{max}$ and a list of ISCs UN^i for ISCs which are unrealisable having degree i with $0 \leq i \leq k_{max}$. Furthermore, an interference forest *forest* is used as a central data structure.

The algorithm starts off with an initial concolic execution of each thread of the program under test thus gathering a symbolic trace for every execution. These initial executions are performed using the same random values as inputs for all executions. Every execution uses a schedule where only one of the created threads will perform its operations; once this thread has finished its execution the program under test is terminated. From each of the gathered traces the set of ISCs is then extracted. All extracted ISCs form the initial set of W^0, the set of ISCs with no interferences. As a result of this the exploration can then start off with a set of ISCs for in depth exploration.

In its main loop the algorithm takes a candidate ISC C from the current worklist W^k and a realisability check is performed. If C is not realisable it is added to UN^i for later exploration since it might become realisable when more interferences have been introduced. However, C can still be used in this iteration to be extended to a set of ISCs which target the sink of C. In order to obtain a set of ISCs from C the write nodes of *forest* are used to introduce new interferences to read nodes in C. The obtained ISCs might have a degree d different from k and are thus added to their corresponding worklists W^d if d does not exceed k_{max}. If, on the other hand, C is realisable, inputs and a schedule are obtained by the realisability-check. The program under test is then executed concolically using the generated inputs and schedule. The resulting trace π is once more examined for further ISCs which are then added to the current worklist W^i. Newly discovered write nodes in π are used to analyse previously unrealisable ISCs. For this matter all ISCs from UN^i for $0 \leq i < k$ are examined for possible new ISCs. This is done by probing all read nodes from the ISC C to a newly discovered write node. If a read-write pair can be used to extend C the a new ISC C' is created by extending C with the interference (i.e., the read-write pair) and C' is added to the list of newly discovered ISCs. A newly discovered ISC isc is then, like for an unrealisable C, analysed for its degree d and added to W^d accordingly.

The presented algorithm is capable of achieving full branch coverage for a program P under the following conditions. First, standard assumptions for concolic testing have to be satisfied, i.e., P has to be deterministic, does not use non-linear arithmetic, and does not contain calls to external library functions. In practise concolic testing can, however, fall back to concrete values

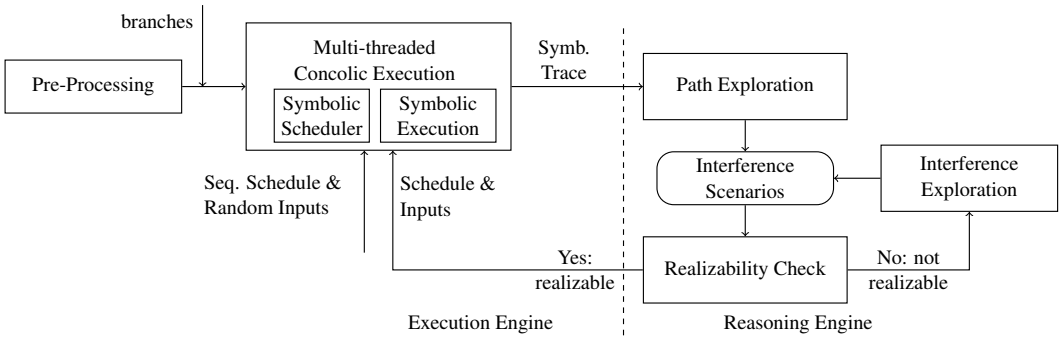


Figure 2.9: Architecture of CONCREST

and thus might successfully test P. Secondly, the number of interferences required to achieve full branch coverage does not exceed k_{max}.

2.4 CONCREST

The tool CONCREST [7] implements the $(\text{con})^2$colic testing approach as an extension of CREST. Figure 2.9 illustrates the high-level architecture of CONCREST. CONCREST can be divided into two parts: an execution engine and a reasoning engine. The execution engine is responsible for executing a program with given inputs and schedule. Additionally, it gathers information like symbolic constraint systems during the execution of the program. This information is then used to generate alternate program runs. Since there could be an intractable amount of them the previously mentioned interference scenarios are used to limit the number of possible alternate runs. Using the realisability checker it is checked whether a schedule for the threads in the program and inputs to the program exist so that the scenario can be realised.

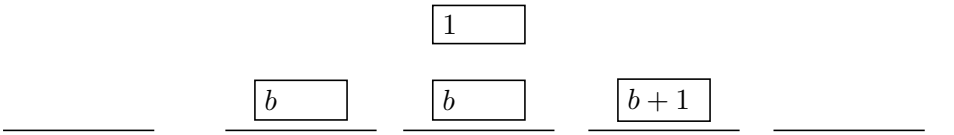
Assertions are modelled as two branches. This means that in the instrumented program an *if*-statement is introduced where one branch models the violation of the assertion and the other one the affirmation. CONCREST will then try to traverse both branches introduced by the assertion. In case the assertion is violated this fact is also recorded in the information about the program execution before it is terminated. In addition to assertions CONCREST introduces *assumptions*. Like assertions, assumptions are modelled using two branches and CONCREST will try to cover both of them. If the assumption does not hold the program is terminated otherwise the program execution continues. Using assumptions it is possible to guarantee that certain statements are true when a program execution reaches a specific point. As a result of this the assume can be used, for example, to model a join of two threads, i.e., a thread waits until another thread terminates.

Pre-Processing

CONCREST relies mostly on an instrumentation process for the pre-processing. In the first step the program under test is processed by the CIL tool suite [15]. The C Intermediate Language (CIL) is used in order to generate the instrumented program. Non-atomic statements

```
1    __CrestLoad(3, (unsigned long )(& b), (long long )b);
2    __cil_tmp6 = b;
3    __CrestSchedulerAfter(3, (unsigned long )(& b));
4    __CrestLoad(4, (unsigned long )0, (long long )1);
5    __CrestApply2(5, 0, (long long )(__cil_tmp6 + 1));
6    __CrestSchedulerBeforeWrite(6, (unsigned long )(& a));
7    __CrestStore(6, (unsigned long )(& a));
8  #line 1
9    a = __cil_tmp6 + 1;
10   __CrestSchedulerAfter(6, (unsigned long )(& a));
```

(a) Instrumented operation



(b) Empty stack (c) Load b (d) Load 1 (e) Apply $+$ (f) Store to a

Figure 2.10: Stack operations

within the program are split up into atomic operations thus modelling an assembler like execution. This means that, for example, a statement like $a = b + 1$ is split up into the operations of loading the value of b and putting the value on the stack, putting 1 on the stack, adding the two values on the stack and writing the result of the addition to a. Figure 2.10 illustrates the pre-processing. In Subfigure 2.10a the instrumented version of $a = b + 1$ is shown. The execution starts with an empty stack (Subfigure 2.10b). In line 1 b is loaded, i.e., put on the stack (Subfigure 2.10c) subsequently the current value of b is stored in a temporary variable (Line 2). The next load is performed in line 4 and puts 1 on the stack (Subfigure 2.10d. After that the operator is applied in line 5, thus removing two elements and putting the result on the stack (Subfigure 2.10e). In line 7 the result of the operation is stored to the symbolic variable a thus removing the result from the stack (Subfigure 2.10f). Finally, in line 9 the concrete result is computed and written to the concrete variable a. Additionally, every load and write operation within the translated program is guarded with the scheduling instructions `__CrestSchedulerBeforeRead`/`__CrestSchedulerBeforeWrite` and `__CrestSchedulerAfter`. As mentioned earlier, true concurrency can be simulated by bringing the statements into a sequential order and executing the statements in this order. Technically, CONCREST does this by introducing a global lock gl, the token of execution. The instructions above will enable the execution engine to pass the token of execution between the threads of the program while keeping track of how many steps a thread has been executing. Moreover, information about the structure of the program, i.e. how the program can be traversed, is gathered. This means that the control flow graph CFG (written to the file `cfg`) of the program is generated using the tool suite. The CFG models precisely where in the program

branching positions and calls to functions are positioned. During the initial code analysis the process distinguishes between functions contained within the program under test (i.e. functions which are also subjected to tests) and built-in functions or external functions which are not tested explicitly. For built-in and external function no symbolic information is computed. Hence, there is no information about their branches. The information about the functions within the program under test is written to `cfg_func_map`. All computed information about the control flow is written to `cfg`.

During the start-up of CONCREST, i.e., before the exploration of the program under test starts, the CFG is parsed and analysed further. For every branch within the program all branches directly reachable in one step are computed using a bounded Dijkstra algorithm. The weights on the edges in the CFG are set to 0 if they do not end in a branching node or to 1 otherwise. As a result of this reachability computation the information about which branches can be visited next from a given branch is known. This information is then used during the exploration. The result of the computation is stored in the binary file `cfg_branches` containing for every branch the list of branches reachable with cost 1.

Execution

As mentioned previously, the $(\text{con})^2$colic testing algorithm can be divided into two units, the execution engine and the exploration/reasoning engine. While the latter is responsible for the main part of the algorithm described in the previous section, the execution engine is responsible for executing the program under test with the desired input and schedule while gathering all relevant symbolic information.

Program Initialisation

In the first step the schedule and inputs generated by the reasoning engine are written to the files `schedule` and `input`. Subsequently, the instrumented program is launched and the exploration engine waits for the program under test to terminate. Due to its instrumentation the program will, as a first step, read the configuration (i.e., schedule and inputs) from the files. In order to achieve this, the instrumenter added a function `__globinit_<name>` to the code and a call to this function as the first operation in the `main` function of the program under test where this initialisation function itself performs a call to `__CrestInit`, the initialisation of CONCREST. This function is responsible for enabling the concolic execution. In order to do so it first reads the inputs for the current test and passes them to the symbolic interpreter. Subsequently, a wrapper for the pthread library is enabled. The wrapper enables the execution engine to keep track of all thread-related events, like the creation of a thread. Furthermore, a handler for segmentation faults is created. By doing so any problems during the execution can be captured and no information should be lost. In a last step `__CrestAtExit` is registered as an exit-callback.

Scheduling

One of the main parts of the execution engine is the execution of the program under test according to the provided schedule. This is done by the symbolic scheduler which is a central part of the execution engine of CONCREST. The symbolic scheduler has control over the token of execution gl. As mentioned previously, the instrumenter adds calls to the commands `__CrestSchedulerBeforeRead`, `__CrestSchedulerBeforeWrite`, and `__CrestSchedulerAfter` to the translated program under test surrounding read and write operations. These commands contain calls to the symbolic scheduler of CONCREST which decides whether the token of execution has to be passed on to the next thread. Thus, the symbolic scheduler keeps track of the steps each thread has taken. In order to do so the symbolic scheduler keeps track of the threads the program has created and stores information about the environment they currently run in.

Gathering Symbolic Information

Another feature of the execution engine is its ability to gather the symbolic information during the execution. For this sake the following commands are instrumented to the program under test.

`__CrestLoad` This command pushes a value to the symbolic stack of the current thread (Subfigures 2.10c and 2.10d). It takes always two parameters: a concrete value and an address. Doing so the symbolic interpreter is able to gather the information of both the load from a symbolic variable as well as the load of a concrete value. The latter case includes loading values from variables which are not examined by CONCREST as well as loading concrete values for operations like an increment. A `__CrestLoad` command records a read event in the symbolic trace.

`__CrestStore` Is used to record the symbolic information of a write event. The command takes only one parameter namely the address of the shared variable. In order to record the information it pops a value from the symbolic stack of the current thread and uses it as value for the write to the symbolic variable representing the shared variable within the program under test (Subfigure 2.10f). A `__CrestStore` command records a write event in the symbolic trace.

`__CrestApply2` Whenever a condition needs to be checked this command is added by the instrumenter. It takes as parameters an operator which has to be applied and the concrete value of the condition. The operation pops two values from the symbolic stack and applies the desired operator on them. Subsequently the result is pushed back to the symbolic stack (Subfigure 2.10e). The implemented operations include arithmetic operations like addition, bitvector operations like left shift and compare operations like lower than. When a compare operation has been performed and both operands have had symbolic information a symbolic predicate is stored for later use by the symbolic interpreter. Otherwise only an empty predicate is stored.

`__CrestBranch` This command is used to gather symbolic information about the branching behaviour of the program under test. It pops a value from the symbolic stack, thus emp-

tying the stack. If a symbolic predicate has been recorded this predicate is logged as a path constraint for the current branch. Otherwise, only the information about entering a new branch is recorded. A `__CrestBranch` command records a branching event in the symbolic trace.

`__CrestAssertionFailed` As mentioned before CONCREST models assertions using two branches. On one branch the assertion is violated which is indicated using this command. The other branch is left empty besides a call to CONCREST's branching function. The assertion-failed command indicates the failed assertion by creating a file. Additionally, a debug message stating in which file and line the error has occurred is printed but no additional information is stored since CONCREST has all required information due to previous commands such as the branch command. After indicating the violated assertion by creating the file `crest_assertion_violation_found` the program under test is terminated immediately.

Before the program terminates `__CrestAtExit` is invoked. This function is used to safely terminate the execution of a test. Thus, it writes the recorded symbolic information about the execution to the file `szd_execution`. Furthermore, the log-file for the current execution is written to `concrest_log`.

Once the execution of the program under test has terminated the time required for the test is recorded. Subsequently, the information stored during the termination of the test is read back from the files and used to update the forest data structure of the algorithm. Moreover, it is checked whether the execution of the program under test reached the predicted branch. If the branch was not reached this misbehaviour is recorded.

Exploration

The exploration engine uses the gathered information about previous program executions and decides about the next step to be performed. It uses a predefined set of strategies in order to make its decisions.

The algorithm terminates whenever one of the following conditions is met. The timeout feature was enabled and a timeout has occurred. It is possible to test a program using CONCREST and limit the total execution time of CONCREST to a fixed amount of time. Whenever a program execution has finished CONCREST will check whether the total amount of time spent has exceeded the maximal value. In case the timeout was reached the exploration will stop. The second condition to be checked every iteration is the total number of iterations CONCREST has performed. Like the timeout, the maximum number of iterations can be specified as an option. The third condition checks whether the exploration strategy has decided to stop the testing. The exploration strategy will do this for example when all branches are covered and thus no further tests are necessary.

After checking the conditions for termination the exploration engine checks which operation has to be performed next. There are three operations the exploration can decide to perform during one iteration of the algorithm. The basic operation is to test an execution. Once enough information is gathered the exploration engine will decide to perform another test and hand over

control to the execution engine. The execution engine will be given exact instructions about which schedule and inputs have to be used for the execution. The second operation which can be chosen is to perform a realisability check for a given scenario. To do so the next ISC in the queue is selected. For this scenario the constraint systems are then generated and checked for their solutions. If a solution exists the scenario is realisable and a schedule and inputs to the program exists. Thus, an execution job using the schedule and inputs is created. Otherwise, the unrealisability is reported. Furthermore, the unrealisable ISC is examined for possible additional interferences which could make it realisable in a future iteration of the algorithm. In the affirmative case an interference exploration job is added. Otherwise, the failure to generate a schedule is reported and a counter for unrealisable ISCs is increased. The third operation which can be performed by the exploration engine is to do an inference exploration. During this interference exploration every thread in the execution forest is analysed for its possible interferences with other threads.

In order to check for completion CONCREST also tracks the coverage of the program under test. This is done using a predefined coverage measurement strategy. As mentioned previously, the statistics are updated after each (successful) execution of the program under test. For each branch in the program under test a flag is stored marking whether it has been visited or is still undiscovered by the test cases. The same is done for all functions within the program under test. Additionally, the timestamp of the first visit to the branch is recorded for every branch within the program under test.

CHAPTER 3

ConCrestWMM

This Chapter presents CONCRESTWMM, the tool implemented for this thesis. The following sections describe the different parts of the implemented tool and how they interact with CONCREST and also with each other. Section 3.1 describes the architecture of CONCRESTWMM. Next, in Section 3.2 the scheduling of WMM-events is discussed. The implemented weak memory models are shown in Section 3.3. Special aspects of these models, i.e., memory barriers and Compare and Swap are discussed in Sections 3.4 and 3.6. Limitations and possible extensions of the implementation of CONCRESTWMM are presented and discussed in Section 3.7. An overview of the capabilities of CONCRESTWMM with respect to its completeness for WMM-effects is outlined in Section 3.8. Finally, the ability to limit the execution to a desired behaviour is described Section 3.9.

3.1 Architecture of ConCrestWMM

The architecture of the WMM-Support is kept simple with only a few ties to CONCREST. Thus, the WMM-Support can be turned off without changes to the behaviour of CONCREST. The WMM-Support was added to CONCREST by adding callbacks to certain functions of CONCREST. These functions are all functions which are instrumented to the program under test during the instrumentation process of CONCREST. Hence, they are no integral part of the execution or reasoning engine but rather its tools. Thus, CONCREST's working principle and its algorithms remain unchanged. Figure 3.1 illustrates the architecture of CONCRESTWMM.

WMMScheduler

The symbolic scheduler of CONCREST offers, as will be described in Section 2.4, the possibility to control the execution of the program under test. For this purpose the symbolic scheduler of CONCREST uses the token of execution gl which is passed to a thread of the program under test for the amount of steps the thread has to perform. In order to schedule the effects of a weak memory model a new scheduler, the WMM-Scheduler, was implemented. The WMM-Scheduler

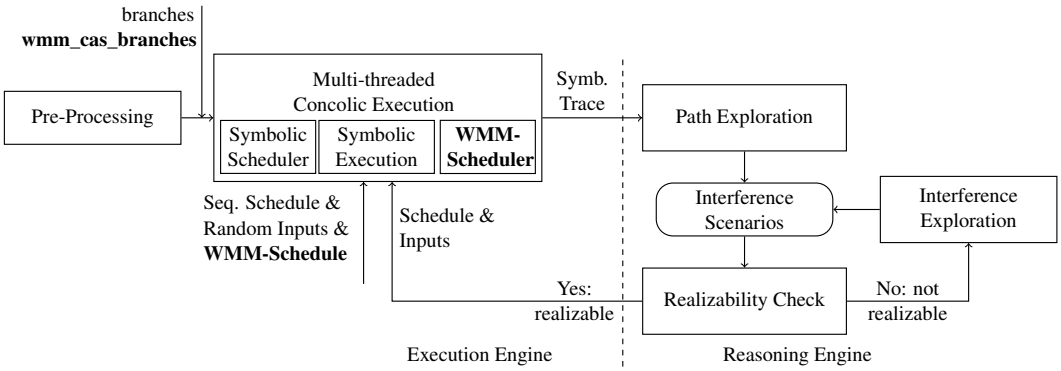


Figure 3.1: Architecture of CONCRESTWMM

Action	Description
Store	Record a write event
Commit	Make a write effect visible to all threads
Load	Read a value
Compare & Swap	Perform a compare and swap operation
Memory Barrier Store-Store	Perform a Store-Store barrier
Memory Barrier Store-Load	Perform a Store-Store barrier
Memory Barrier Load-Load	Perform a Store-Store barrier

Table 3.1: Interface of a Weak Memory Model

offers the possibility to schedule an event at a given point of time, i.e., after a given amount of steps of the symbolic scheduler. This timing relation is realised through callbacks to the WMM-Scheduler which are embedded into the scheduling functions of CONCREST. Additionally these callbacks intercept all actions from the program under test. The WMM-Scheduler will then pass the actions to the desired WMM.

Additionally, the WMM-Scheduler offers the possibility to directly invoke events like compare and swap and memory barriers, as will be explained later. As a result of this, it will be possible to examine the behaviour of the program under test under different environments by simulating different systems and compilers through schedules and instrumentation. The instrumentation processes of CONCREST can also be seen as a pre-compilation/translation step. For example, a compiler might add a memory fence like a `LWSYNC` directive to the compiled program due to added constraints and directives in the program under test. Using schedules it is possible to imitate behaviour like this using CONCRESTWMM.

WMM

In order to handle the effects of a specific WMM a new class is introduced. Using object-oriented techniques various different models can be implemented. Like CONCREST the WMM keeps track of all changes to the declared shared variables. A copy of both the concrete and

symbolic memory is kept within the WMM-class. It is thus possible to analyse which value is written to which variable at which point of time during the execution of the program.

Table 3.1 shows the interface of a WMM. The three basic operations of a WMM are Store, Commit and Load. A store operation is performed for every write access to an address which was declared as shared variable. This operation executed whenever a `__CrestStore` is performed in the instrumented program under test. Load operations are performed every time a read access from a shared variable occurs, i.e., a `__CrestLoad` operation is performed in the instrumented program under test. The load operation has to return the value for the executing thread will observe for the address with respect to the modelled WMM. A commits are used to make the effect of a write visible to all threads. Hence, there is no corresponding operation in CONCREST. The Compare and Swap operation is command introduced by CONCRESTWMM which can be added to the program under test. A detailed description of the operation and its semantics are in Section 3.6. Memory barriers are inserted in programs in order to guarantee a certain state of the memory model with respect to uncommitted writes. Memory barriers and their effects are discussed in more detail in Section 3.4.

WMM-Wrapper

In order to comply with the standards of CONCREST the possibility to invoke functionality of the WMM-Scheduler directly from non-object oriented C-code had to be added. This was done in a similar fashion as it is done in CONCREST by creating a wrapper around the WMM-Scheduler. To do so a single reference to the WMM-Scheduler was introduced which is used over the whole runtime of CONCREST. However, since CONCREST currently does not use concurrency in its algorithms no data-races occur. Additionally, the program under test is only executed in a single threaded manner and all operations of the WMM-Scheduler can be seen as atomic. The reference is initialised using a created "Init"-function to which a callback was added to `__CrestInit`.

Building CONCRESTWMM

The build script of new features for CONCRESTWMM was added to the existing build infrastructure of CONCREST. Thus, there are no high level changes in the build process.

Since all WMM related code within CONCRESTWMM is guarded using preprocessor directives the WMM-Support can be ruled out by removing the define of the `USE_WMM` constant from the build process. Thus, a version of CONCREST will be built.

3.2 Scheduling of Events for Weak Memory Models

As described previously the WMM-Scheduler allows to schedule events during the execution of the program under test.

Challenges for WMM-Schedules

An event of the WMM-Scheduler, like a commit, needs in addition to its time-code the variable/address for which it has to be applied. However, one of the main challenges for developing a

scheduler was creating a way of declaring variables for which the scheduler should be able to examine WMM-effects and thus, using them in its schedules. The difficulty mainly arises from the fact that at the moment the generation of schedules for the WMM-Scheduler is not automatic. As a result of this schedules are written by hand to perform experiments in addition to the schedules generated by CONCREST. Thus, neither CONCREST nor the WMM-scheduler has information about the shared variables for which the simulation of effects is of interest. CONCREST already has a facility to declare shared variables. However, this facility is not enough at the moment. Shared variables are declared to CONCREST by explicitly stating this fact in the code by adding a statement like CREST_shared_int(x); for a variable x at the beginning of the program under test. The problem with this approach is that there is still no information about shared variables outside the code, i.e., the information is only available during runtime. Thus, at runtime only the addresses of shared variables were known to CONCREST, while the WMM-schedules needed some way of stating which variable has to be affected by a particular event in the schedule. The problem with addresses is of course that they are changing with each program run. Thus, if a schedule would be created using addresses it would not be applicable for a future re-run of the test case.

In order to solve this problem the WMM-Scheduler is equipped with the function `NewSharedVariable` mapping addresses to an identifier and a type. The identifiers are then substituted for concrete addresses when a call to the function `ReadSchedule` is performed. The function `NewSharedVariable` takes the parameters address, type, value and name. The name is a text representation of the variable name which can then be used to address this variable in a schedule for the WMM-Scheduler. However, it is not required that the name of the variable in the source code coincides with the name handed to the WMM-Scheduler via the declaration function. It has to be mentioned that in the current implementation the type parameter might seem to be unnecessary as it imitates the behaviour of CONCREST where the declaration function has to be chosen according to the data type of the variable. Nevertheless, the type parameter is necessary since the WMM-Support also deals with concrete values and thus needs to copy them, for which the type of the variable is needed. The value parameter of the function `NewSharedVariable` was introduced in order to pass the current value of the variable to the internal memory of the WMM-Scheduler.

The function `ReadSchedule` is intended to be invoked once all shared variables are declared. First it will check if there exists a schedule for the WMM-Scheduler. This check was introduced in order to enable the execution engine to execute tests without WMM-effects. In case no schedule file exists the WMM-Scheduler will load Sequential Consistency as default model and turn on the auto-commit functionality. This auto-commit functionality will invoke the commit operation of the WMM-Scheduler after each store operation. If there exists a schedule then the WMM-Scheduler will create a copy of this schedule and substitute all declared shared-variables for their addresses using the previously mentioned variable names. After this substitution the substituted schedule file is read line by line by the scheduler. While the first line specifies which memory model has to be used the remaining lines specify which effect of the selected WMM will happen at which point of time. Schedules entries are related to points of times. As a consequence of this WMM-schedules have to be created with respect to schedule for CONCREST. The schedule for CONCREST fixes when store and load operations will occur

Event ID	Name	Description
1	Commit	Commit the effect of a previous write
2	Memory Barrier Store-Store	Perform a Store-Store barrier on a given thread
3	Memory Barrier Store-Load	Perform a Store-Load barrier on a given thread
4	Memory Barrier Load-Load	Perform a Load-Load barrier on a given thread

Table 3.2: WMM Event Types

on the threads of the program. Based on the timings of this schedule the WMM-schedule can then be fixed.

WMM-Schedules

The scheduling of WMM-effects follows a similar pattern as the scheduling of events in several threads in CONCREST does. Instead of creating additional functions for the WMM-Scheduler which then have to be instrumented to the program under test, callbacks to the WMM-Scheduler were added at the end of the CONCREST functions `__CrestSchedulerBeforeRead`, `__CrestSchedulerBeforeWrite` and `__CrestSchedulerAfter`. The first two functions are instrumented before every read or write operation within the program under test irrespective of whether the read/write concerns a shared variable or not. The after-function is instrumented at the end of each of such a store/load operation. This means that CONCREST can decide whether before or after a store/load a context switch from one process to another can happen. As a result of this, the schedules for CONCREST are stating which thread can perform how many operations. This means that each line of the schedule states a thread ID and the number of steps it has to perform. In contrast to this behaviour, the WMM-Scheduler expects a schedule stating after how many steps which effect has to occur.

Format

The schedule for the WMM-Scheduler is a plain text file. The first line of the schedule file contains the memory model which has to be loaded for performing the tests. Every following line will describe an event marked with a time code, i.e.

`<timecode> <event type> <event data>.`

The `timecode` refers to the number of steps the scheduler has performed. While the time cod, is a single unconstrained number, the event type is limited to be exactly one of the numbers 1 to 4 since each of these numbers represents one of the implemented WMM-effects. The meaning of the numbers is as follows: 1 denotes a commit, 2 a store-store memory barrier, 3 a store-load barrier and 4 a load-load barrier. The structure of the event data depends on the type of the event, as will be described later. It is necessary that the events are ordered by time since the parser will not sort them. Figure 3.2 shows an example WMM-schedule. Line 1 states that the WMM-schedule uses the PSO model. Line 2 states that variable y is committed on thread 1

```
1 | PSO
2 | 5 1 1 y
3 | 9 3 2
```

Figure 3.2: Example Schedule

after 5 steps of the WMM-Scheduler (Timecode: 5, event type: 1/commit, event data: 1 y, i.e., commit y on thread 1). Line 3 states that after 9 steps of the WMM-Scheduler a Store-Load memory barrier is issued on thread 2 (Timecode: 9, event type: 3/Store-Load memory barrier, event data: 2).

Commit Events

A commit event consists always of a memory address and the thread on which it should occur. However, not all data is necessary in all cases. For example, for a potential model of the Total Store Order WMM an address parameter will have no effect since commits affect all addresses used by a processor. However, it still has to be specified and not left blank. On the other hand, if SC is loaded, a commit will have no effect since in SC the semantic of a store operation is defined so that it does not need a commit.

Other than memory barriers, the commit operation is currently not subjected to be directly invokable. This means that it cannot be instrumented or written manually to the program under test. The reason for this is that it would not make sense to allow developers to commit writes manually, since ensuring the global visibility of a write can be done using memory barriers as will be explained later.

Memory Barriers Events

Memory barrier events (event type 2,3 and 4) require as event data only the ID of the thread on which it should be applied. The reason that the thread has to be specified is that otherwise the requirement for synchronisation mechanisms between schedules for the WMM-Scheduler and CONCREST would arise. The type of the barrier (Store-Store, Load-Load or Store-Load) is specified by the event type. Memory barriers can also be instrumented to the program under test or used manually in a program. More details about memory barriers can be found in section 3.4

Introducing New Event Types

The design of the WMM-Scheduler is done in a way so that it is straight-forward to add new event types. To add a new event type two functions of the WMM-Scheduler have to be changed. First, the parser in the function `ReadSchedule` has to be adopted in order to be able parse events of the new type. Secondly, a callback for the function has to be added to the function `PerformOperations` of the WMM-Scheduler. In addition to changing the functions of the WMM-Scheduler also the models have to be adopted for the new requirements. Every functionality should be modelled as a member function of the `WMM`-class. Implementations of a specific WMM should then overwrite these member functions.

Action	Effect
Store	Perform write to main memory
Commit	No effect
Load	Return value from main memory
Compare & Swap	Compare and Swap operation
Memory Barrier Store-Store	Not supported
Memory Barrier Store-Load	Not supported
Memory Barrier Load-Load	Not supported

Table 3.3: Sequential Consistency (SC) model

Action	Effect
Store	Record write event for requesting thread
Commit	Commit write for requested thread-address combination
Load	Return value from main memory or thread-local buffer
Compare & Swap	Commit all pending writes for address. Then perform CAS
Memory Barrier Store-Store	Commit all writes on requesting thread
Memory Barrier Store-Load	Commit all writes on requesting thread
Memory Barrier Load-Load	Not supported

Table 3.4: Partial Store Order (PSO) model

3.3 Implemented Weak Memory Models

Sequential Consistency

The default behaviour of CONCREST is Sequential Consistency. In order to simulate the same behaviour in CONCRESTWMM SC can be loaded as memory model. A store operation in SC will store the value immediately in the main memory. Thus, a commit will have no effect since its effect is incorporated in the store operation. A load operation will return the value from the requested memory address irrespective of the thread from which the load was requested. The Compare and Swap operation will behave exactly like the semantics described in Section 3.6. Since all writes in SC will become visible immediately no memory barriers are needed and thus, no memory barriers are supported for SC. A summary of the SC model is shown in Table 3.3.

Partial Store Order

The Partial Store Order (PSO) model, as summarised in Table 3.4 is implemented similarly to the PSO-model described in [4]. It allows writes on a processor to be reordered past different writes on the same processor. Thus, for every processor and every memory address PSO uses a FIFO-Queue to store writes to memory addresses. A store will append the value to the queue for the address on the requesting thread. Whenever the queue for a processor-address combination is non-empty the last value of this queue is used by the load operation. Otherwise, the value

from the global memory is returned. Thus, every process can see its own writes while other process cannot see them until they are committed. The commit operation removes the first write of the corresponding processor-address queue and writes the value to the global memory. A CAS operation with parameters address, expected value and new value will first commit the pending writes for all thread for the address before determining the result of the actual CAS operation. Furthermore, the PSO model offers Store-Store and Store-Load memory barriers. Both barriers commit all pending writes of the thread on which they are performed.

Total Store Order (TSO) is a stricter model as PSO since it allows writes to be reordered passed a different load but not a later store to the same address [4]. Thus, PSO can be seen as a generalisation of TSO and PSO can imitate the behaviour of TSO.

3.4 Memory Barriers

Some algorithms require specific guarantees for their writes. Thus, weak memory models need a mechanism to force effects of writes to become visible. This mechanism is added by so-called memory barriers or memory fences. One difficulty for providing support for memory barriers comes from the fact that they vary between the different processor architectures. Thus, the implementation for the WMM-Support does not use specific names for the implemented memory barriers which can be found as instruction names in programming manuals provided by processor manufacturers. However, the naming of the barriers is done in the same way as proposed by [4].

The current implementation offers three different types of memory barriers:

- Store-Store
- Store-Load
- Load-Load

For the implemented memory model PSO the Store-Store and Store-Load barriers are available. Both barriers ensure that all writes for a thread are committed. This is done by looping over all memory addresses for the requesting thread. In this loop all pending writes for an address are committed. Sequential Consistency, on the other hand, does not include memory barrier in its model, since the effect of a write becomes globally visible immediately under the assumptions of this model. Thus, the implementation of SC does not offer any memory barriers.

These memory barriers can be requested to be performed by the WMM-Scheduler in two ways. The first way is to add a corresponding entry to the wmm-schedule. As mentioned previously the only parameter for the instruction in the schedule is the thread on which the memory barrier has to be applied. None of the barriers can be applied only to a specific memory address which also reflects the behaviour of instructions like ARM's `DMB` or POWER's `sync` [12]. The second possibility is to add the corresponding instruction to the (instrumented) program under test. For each of the barriers a function was added to the WMM-Wrapper. Thus a call to a memory barrier can be added to the program without any additional constraints using one of the functions from the wrapper. The commands are named `__WMM_membar_xx` where `xx` can

```
10 int main(int argc, char* argv[])
11 {
12   ...
13   CREST_shared_int(x);
14
15   __WMM_new_shared_int(&x, "x");
16   __WMM_READ_SCHEDULE();
17
18   ...
19 }
```

(a) Test program

```
35 int main(int argc , char **argv )
36 {
37   ...
38   {
39   __globinit_test();
40   ...
41   CrestSharedInt(& x);
42   ...
43   __WMM_new_shared_int(& x, "x");
44   ...
45   __WMM_READ_SCHEDULE();
46   ...
47   }
48   ...
49 }
50
51 void __globinit_test(void)
52 {
53
54   {
55   __CrestInit();
56 }
57 }
```

(b) Instrumented test program

Figure 3.3: Test program

either be `ss`, `sl` or `ll`. These commands can be placed at any position in the program under test. For neither of the ways the memory model has to be known/specified upfront. However, not every WMM supports all three memory barriers.

The fact mentioned above, namely that not every WMM offers all three of the above memory barriers imposes another difficulty for the implementation. This means that the implementation has to be done in way which prevents (semantic) errors when non-existing memory barriers are requested. Still, the interface for WMMs defines all three memory barriers. If a specific implementation of a WMM does not support a specific memory barrier this is not identified by the WMM-Scheduler. However, if a call to a not supported barrier is requested by the WMM-Scheduler, then an error message will be printed indicating the wrong request and the execution of the program under test will be terminated.

3.5 Structure of a Program Run

In the following section the testing of a program run is illustrated. The skeleton of a test program and its instrumented version are shown in Figure 3.3.

1. The instrumentation process of CONCRESTWMM adds a call to the function `glob_init` as the first instruction in the main function to the program under test (Line 39 in Subfig-

ure 3.3b). In this `globinit`-function the `__CrestInit` function is called (Line 55 in Subfigure 3.3b) which initialises CONCRESTWMM and thus also the WMM-Scheduler. Additionally, SC is loaded as the default WMM. This pre-loading of SC is done in order to avoid problems when the execution engine of CONCRESTWMM performs calls to, for example, store-operations which require a WMM to be loaded before the actual WMM-schedule is read.

2. After this initialisation the shared variables can be declared. The declaration can be seen in line 15 in Subfigure 3.3a and line 43 in Subfigure 3.3b. The declaration of shared variables has to be done by the programmer in the source code of the program under test. Since this declaration is done by calling a function the declaration is also part of the instrumented program. During the execution of the instrumented program the WMM-Scheduler stores information like address and type of each of the declared variables.

3. Once all variables are declared the WMM-Scheduler can start scheduling events. This process also needs to be invoked by the programmer in the original source code by a call to the function `ReadSchedule` of the WMM-Scheduler (Line 16 in Subfigure 3.3a and line 45 in Subfigure 3.3b). The WMM-Scheduler will then substitute the variable-names in the `wmm-schedule` file by their current addresses and create a file named `wmm-schedule-sub`. After that it will load the events from the substituted schedule.

4. Throughout the program-run the WMM-Scheduler works in the background using the earlier mentioned callbacks added to scheduling events of CONCRESTWMM in order to react to certain events:

   - Each time a load operation is performed in the instrumented program the WMM-Scheduler will, like CONCREST, put the value stored currently at the address on an internal stack. If the address for which the load is performed is a shared variable the WMM-Scheduler will load the current values of both concrete and symbolic value for this address on the current thread by using its internal load functions. Thus, the current values according to the WMM are stored. Otherwise it will create a symbolic expression using the value handed to the store operation and put it on the stack together with this value.
   - The store operation is performed in a similar way. If the address, for which the store operation is performed, is a shared variable the WMM-Scheduler takes both a symbolic and concrete value from the stack. These values are then stored in the loaded memory model using the store operation of the WMM. Otherwise it will just remove the values from the stack.
   - Every time a call to the CONCREST-Scheduler is performed the call is intercepted beforehand as the WMM-Scheduler checks if there was a context switch, i.e., if the current process is different from the process observed during the last check. If the processes are different the WMM-Scheduler will poll all the current values for the shared variables in the loaded WMM and write them into the internal memory of CONCRESTWMM. This is of course done for both concrete and symbolic memory.

```
atomic CAS(memory, old, value) {
  if( *memory == old ) {
    *memory= value;
    return 1;
  }
  return 0;
}
```

Figure 3.4: Semantics of the implemented Compare and Swap operation

- At the end of each call to the CONCREST-Scheduler the WMM-Scheduler will take over. It will check for due WMM-events like commits and memory barriers for the current point of time and issue them.

3.6 Compare and Swap

Compare and Swap (CAS) (also known as *Compare and Exchange*) is an atomic processor instruction which checks whether the value of a certain register is equal to another fixed value and in the affirmative case swaps the value of the register with a new value. In addition to the possible changing of the register value, the result of the compare operation is stored in a special purpose register. Other than the name might suggest, the register containing the new value remains unchanged [11]. The intended semantics of the implemented function can be found in Figure 3.4. There are various implementations of CAS having different semantics. For example, GCC uses two built-in atomic operations: a Boolean operation returning true in case the register was updated and a "value" version where the previous value of the memory cell is returned [10]. Intel's CMPXCHG instruction, on the other hand, sets the ZF flag if the value was not changed and clears it otherwise [11].

CAS is widely used in multi-threaded applications where it is used to synchronise different processes or to implement concurrent data structures where the access to specific elements needs to be synchronised [13]. Moreover, it can be used in order to avoid spinlocks within applications. A spinlock is a lock where a process trying to acquire the lock will loop until the process is able to acquire it. During this loop the process will repeatedly check whether it is possible to acquire the lock. Spinlocks can decrease the overall performance of a system while increasing the energy consumption when processes tend to hold the lock for a long time while other processes are waiting/spinning for this lock. Thus, most modern operating systems make heavy use of the CAS operation in order to achieve a good performance.

A naïve implementation could just add a pre-written CAS-function to the program under test in a pre-processing step. This function would then also be instrumented by CONCRESTWMM. However, this would cause two problems. First of all, the operation would not be atomic. Since when using shared variables CONCRESTWMM will instrument the function in a way such that the function will be interrupted and thus cause an unexpected behaviour. To prevent this one could add a global lock. By using such a lock it can be assured that only one process at a time

is allowed to change the variable. However, the second problem makes it necessary to add a built-in support for CAS to CONCRESTWMM: since CAS is an instruction at processor level its behaviour depends highly on the processor and thus of course on the implemented WMM. As a consequence of this, CAS is a part of the definition of a WMM-model.

In a first step to add CAS-support to CONCREST a simple test program was written where a simple CAS-function was used. This function used no lock or instruction for the compiler in order to be atomic. Therefore, CONCREST immediately found a "bug" within the code since a thread executing the CAS-function could just be interrupted and control be given to a different thread, thus causing a faulty behaviour. However, for a correct implementation of CAS such a behaviour is not possible. Thus, this simple program can also serve as a test case testing whether CAS is atomic. The instrumented version of the CAS operation was taken as a starting point for a built-in operation. Therefore, all scheduling functions allowing CONCREST to perform a context switch were removed. In the next step the instrumented code of the CAS-function was analysed with respect to created branches and other dependencies created by CONCREST. After this step, the code was analysed further in order to remove dependencies between the CONCREST-library and WMM related code. The behaviour of the necessary functions of CONCREST were implemented directly within the new CAS operation, i.e., calls to the symbolic interpreter were invoked directly from the WMM-related code. As a result of this the only direct connection between CONCREST and the WMM-Support was created by a reference to the symbolic interpreter which is necessary to issue calls to the symbolic interpreter. However, the code is written in a way such that when the WMM-Support is turned off during the build process of CONCREST the code is still compiling and CONCREST runs without errors for programs which do not rely on the WMM-CAS-operation.

In order to implement the CAS-function correctly without the dependence on a handwritten function in the program under test the corresponding information about the built-in CAS-function was added to CONCRESTWMM. As described in Section 2.4, the information about the program under test is created in two phases. First CIL is used to instrument the program and write information about the control flow graph of the program into several files. In the second phase these files are read by the `process_cfg` program and a binary file containing the adjacency list of the reduced control flow graph is written. The reduced CFG is then computed by computing for every branch all branches reachable in one step. The information generated in these two phases is then used during the $(\text{con})^2$colic execution in order to gather information about coverage and additionally to decide about the execution strategy, i.e., which scenario should be explored next. Thus, in order to fully embed CAS into CONCREST matching information has to be added to the control flow graph after its generation. This is done by adding a mock function with two branches to the graph. To guarantee minimal interference with existing functions while not harming the performance of CONCREST too much, the IDs used for creating the function in the control flow graph are dynamic. This means that the maximal branch ID is computed using the information written by the instrumenter and a configurable offset is added to this maximal number. The IDs of the *then* and *else* branch of the CAS-function are then written to the file `wmm_cas_branches`. The information from the `wmm_cas_branches` file is then used by the WMM-Scheduler when a CAS operation is requested by the program under test. After the branches are read by the exploration engine the control flow graph CFG is

constructed. Subsequently, the new function is added to the CFG using the previously computed numbers. As a consequence of this CONCREST will not see a difference between the built-in CAS function and a handwritten function from the program under test. Thus, CONCREST will try to cover both branches, i.e., once changing the value and once not changing the value. When the `ReadSchedule` method of the WMM-Scheduler is invoked it will read the previously written branch IDs and will use them every time the CAS function is invoked.

3.7 Limitations and possible extensions

There are several possible solutions for extending CONCRESTWMM so that is able to examine concurrent programs while identifying WMM related bugs systematically. The first simple possibility is to let the user decide which schedules for the program under test have to be tested using WMM-semantics. This means that CONCRESTWMM could be equipped with a search facility aiming for identifying WMM related bugs for a given schedule. The search for WMM related bugs could then be done using heuristics similarly to the operational semantics used by RELAXER [4][1]. A possible heuristic could first identify all reads to a written value and try to delay the commit of the write until all identified reads have been scheduled. Another, more sophisticated and thus computationally more expensive heuristic could build on the similar pattern of searching for WMM-related bugs for a given program run. First it would also have to identify the read-write relations of the given program run. These relations could be extracted from the constraint systems generated by CONCRESTWMM. In the next step the heuristic could then one by one increase the number of reads from a certain address scheduled before the commit event. For each iteration a new WMM-schedule has to be generated and the program under test executed again using the obtained WMM-schedule in combination with the fixed schedule.

Another, more complicated, possibility for integrating the full support of WMM-effects into CONCRESTWMM is to encode semantics of WMM-effects into the constraint systems of CONCRESTWMM. The semantics could be encoded in a similar fashion as proposed by [2][2]. The modified constraint systems need to be constructed in a way so that a solution to them can be transformed into schedules for both symbolic scheduler and the WMM-Scheduler of CONCRESTWMM.

Yet another possibility similarly to the previously described solution is the reformulation of the constraint systems in a way so that a solution to the constraint systems yields interferences where a relaxation of the write-consistency can be used in order to violate an assertion in the program under test. The identified interferences can then be further analysed in order to obtain the offset from the write-event in the schedule to its corresponding commit-event in the WMM-schedule.

[1] A description of RELAXER can be found in section 5.2.

[2] These encodings are also described in Section 5.1.

3.8 Simulating Possible Effects

CONCRESTWMM is able to simulate all currently known effects of weak memory models. For the simulation it relies on WMM-schedules as described in Section 3.2. In these schedules a write is not committed until stated explicitly in the schedule or forced implicitly by a memory barrier. As a consequence of this, it is necessary that schedules for the WMM-Scheduler reflect the actual semantics of the desired WMM.

Currently, the implementation supports the simulation of effects from the PSO model. However, it is straight-forward to implement other weak memory models such as TSO or PSLO. The main work for adding support for a different WMM to CONCRESTWMM is to add a new implementation of the `WMM`-class which models the effects of the desired WMM. More details for the purpose of adding additional WMMs can be found in Chapter 3.

Moreover, the tool also supports the possibility of simulating multiple processes running one processor. The interface of the WMM-Scheduler provides load and store methods which are used for simulating the memory access during the execution of the program under test. Each of these methods takes a thread-number as argument. In the current implementation the logical-thread-ID assigned by CONCRESTWMM is passed along whenever a store or load operation occurs during the execution. However, this behaviour can be changed in order to add support for architectures where processes can share caches and thus have access to the same uncommitted values. In case this option has to be implemented CONCRESTWMM would need to store for each combination of thread and point of time a "physical" processor ID. This ID then has to be passed along to the WMM-Scheduler for load and store operation. In addition to these changes to CONCRESTWMM also the WMM-Support would require the addition of an event capturing the possibility of reallocating a process from one processor to another.

Another aspect of program executions which are highly dependent on the semantics of the used WMM are memory barriers. These memory barriers are invoked using manufacturer-dependent commands such as `sync`, `lwsync` or `dmb`. However, most processor manufactures implement a single WMM for a series of processors, thus making it easier to examine the effects of memory barriers. The possibility to examine the effects of memory barriers explicitly was added through the support of memory barrier instructions by the WMM-Scheduler. Effects of a memory barrier can be triggered via instrumentations to the program under test or entries in the WMM-schedule and there is no difference between the effects of a scheduled or instrumented memory barrier. A correct implementation of a compiler inserts certain memory barriers into the program under test automatically. Since CONCRESTWMM works on a source- code level it has no sense of memory barriers inserted by the compiler and treats the program under test as if there are no memory barriers inserted by the compiler. CONCRESTWMM is able to simulate the effect of a compiler using two variants: memory barriers which are inserted into the source code (automatically or manual) or by extending the generator of WMM-schedules so that it adds memory barriers in the same way as a specific compiler does.

3.9 Limiting Impossible Effects

One issue when simulating the effects of weak memory models using CONCRESTWMM is to distinguish between effects possible on real-world processors and effects possible using the implemented model. A WMM-model consists of the definition of the semantics of the WMM-operations, i.e., the implementation of the WMM-interface (see Section 3.1) and additionally, the valid combinations of these operations. However, the WMM-Scheduler performs the enforced events from the WMM-schedule without checking the validity of the operations with respect to the defined model. As a consequence of this not every WMM-schedule is valid with respect to the WMM-model. Therefore, it is the responsibility of the user to create WMM-schedules reflecting the behaviour of the WMM-model

The first possibility for ensuring a valid (with respect to the desired WMM-model) WMM-schedule for the program under test is committing the written values. As shown in Section 3.5 variables can be declared as shared variables with WMM-effects explicitly during the start-up of the program under test. When a memory model other than Sequential Consistency is selected the effects of writes to these shared variables are delayed until a commit event happens for the specified memory address. In contrast to memory barriers, commit events can only happen by enforcing them in the WMM-schedule for the current test case.

Another possibility of ensuring correct behaviour of the program is the use of memory barriers. As mentioned earlier a barrier can be enforced either by adding it explicitly to the program under test or by enforcing it using the WMM-schedule. A memory barrier event is comprised of the type of the barrier and the thread on which it has to be applied. For the implemented PSO-model a store-store and a store-load barrier exist. However, their semantics do not differ. Both barriers commit all pending writes on the thread it was requested on.

Another step towards achieving semantically correct executions is to include semantic checks of the WMM-operations into the execution model of CONCRESTWMM. These semantic checks could be performed in a similar fashion to the presented ideas in the first section of this chapter. The WMM-Scheduler records all store and load events during the execution. Additionally, it is equipped with the `PerformOperations`-function which is invoked after every step of the symbolic scheduler of CONCREST. This function can be extended to include a semantic check of the current status of the memory with respect to the selected WMM. An approach like this can be used to commit writes at the latest possible point of time whereas the WMM-schedule can be used to commit writes earlier than this latest point of time.

The major challenge is to distinguish between allowed and disallowed executions which remains open since manufacturers tend not to provide precise information about their memory models. However, there are research results providing useful information about the behaviour of modern processors for sets of test cases. In [12], for example, a variety of examples featuring allowed program executions is outlined. Additionally, [3] describes an approach for constructing an axiomatic framework modelling the behaviour of certain processor architectures. These tools can be used to obtain information about the correctness of developed models.

CHAPTER 4

Experiments

This Chapter presents a series of conducted experiments. In Section 4.1 general remarks on the performed experiments are given. In particular, the section will cover remarks about schedules and naming schemes for the test cases. Next, Sections 4.2 - 4.7 will each present a simple program. Most of these programs are so-called litmus tests. Furthermore, each of these Section will also present several test cases for the program it presents. Section 4.8 concludes the Chapter by presenting benchmark results.

4.1 General Remarks

Schedules

As mentioned earlier the additional WMM-Support does not change the default behaviour of CONCRESTWMM when WMM-effects are not turned on explicitly. The reason for this is that if no WMM-schedule is present CONCRESTWMM uses SC as the default WMM-model and thus simulates the behaviour of CONCREST. However, if a `wmm-schedule` file is present the effects of the enforced WMM-model are simulated. In order to test and simulate different WMM-effects several programs have been implemented exploiting different problems for testing when considering WMM and additionally, demonstrating different features of the WMM-Support in CONCRESTWMM.

Since the automatic generation of WMM-schedules is not part of this thesis all schedules for the WMM-Scheduler were created manually. Most of the presented test cases will use a schedule for the symbolic scheduler which CONCRESTWMM using SC generated during an examination of the program under test. However, for some test cases CONCRESTWMM using SC cannot reach the desired state. Therefore, for these test cases a schedule for the symbolic scheduler was created manually (This fact will be mentioned explicitly).

Test cases

All experiments in this chapter provide a makefile which is able to run all test cases. These makefiles rely on the two constants `CONCREST_HOME` and `CONCREST` which are required to run CONCREST and build the test case. As a rule of thumb the test case will try to exploit the violation of an assertion which could not be found using CONCRESTWMM with SC. The targets of the makefile will follow the following naming scheme:

- A simple `make` without any additional parameters will run the test program using CONCREST without any WMM-effects, i.e., using SC as WMM-model.

- A target `<testname>-pso[<number>]` will execute a test using the PSO-model by executing the instrumented version of the test program with supplied predefined schedules for CONCREST and the WMM-Scheduler. These test cases run a hard copy of the instrumented program. This means that changes to the original version of the program under test will not affect the PSO test cases. In order to update the changes for these test cases the `<testname>-pso.c` file has to be replaced by a copy of the `<testname>.cil.c` file placed in the `temp-concrest` directory.

- A target `<testname>-membar` will run a copy of the instrumented program where a memory barrier instruction is enforced by either the WMM-schedule or an instrumented memory barrier in the program under test. Similarly to the PSO test cases also for the memory barrier test cases there exists a hard copy of the instrumented program named `<testname>-membar.c`. Additionally, an instruction invoking the memory barrier is added to this copy. This has to be kept in mind when replacing the file with an updated version of the instrumented program.

Thread finishes

The descriptions of program executions and schedules in this chapter will make frequent use of the phrase "*thread finishes*". When testing a concurrent program using CONCREST in some cases a post-condition has to be checked. For this purpose it has to be ensured that each thread terminates properly before the post condition is evaluated. Otherwise, the final state cannot be checked in a semantically correct way. However, usual `pthread_join(thread)` commands are not supported when testing a program using CONCREST. In order to overcome this issue each thread is equipped with a flag-variable `<tid>_done` which is globally initialised to 0. The flag for the thread is then set to 1 at the end of the corresponding thread. After declaring the flags as shared variables to CONCRESTWMM the termination of a thread can be ensured by using `concrest_assume(tid_done)`. As mentioned earlier an assume works similar like an assert but the violation of an assumption is not considered as an error like an assertion-violation. In contrast to an assertion, CONCREST tries to find a schedule such that the assumption is not violated. These flags are declared as shared variables to CONCRESTWMM, however, they are not declared as shared variables to the WMM-Scheduler. This is done since the flags only serve to mark a thread to be completed and thus, delaying a write to these flags

Name	Description
`mp-concrest`	Test the program with CONCRESTWMM and SC
`mp-pso1`	Reach forbidden state using PSO and handwritten schedule
`mp-pso2`	Reach forbidden state using PSO and generated schedule
`mp-membar`	Forbid state using an instrumented memory barrier
`mp-membar-schedule`	Forbid state with a memory barrier in `wmm-schedule`

Table 4.1: Overview of test cases for MP

would only result in complications and undesired behaviour of the program under test. Nevertheless, for correctly computing the time codes in the WMM-schedule the operations on these flags have to be considered, i.e., each thread needs two additional steps in a schedule in order to terminate which corresponds to setting the value of the flag to 1.

4.2 Message Passing

Message passing (MP) is a simple example already capable of demonstrating some effects of WMM. Thus, it is used frequently in literature on the effects of WMM.

Thread 1	Thread 2
$x = 1$	`while`$(y == 0)\{\}$
$y = 1$	$r2 = x$
Initial state: $x = 0 \wedge y = 0$	
Forbidden: $r2 = 0$ on Thread 2	

Figure 4.1: Base form of Message Passing Example from [12]

Figure 4.1 shows a simple program using two shared variables x and y. While y serves as a guard or flag for a data variable, x represents exactly this data written by the program. This means that once y is set to 1 other processes should be able to see a (semantically) "correct" value for x. Thus, other processes will have to loop/wait until y is set to 1 as can be seen in Thread 2. However, it is not necessary to always execute the program as a complete program where Thread 2 loops until Thread 1 has written its data. Instead it is possible to just look at a single execution of the program by transforming it into a simplified loop-free program and modifying the post-condition accordingly. Instead of checking that after executing the loop in Thread 2 x has the correct value, the condition can be reformulated to a check ensuring that when the write flag has been set then the data in x has to be correct, i.e., non zero. Figure 4.2 shows the loop-free message passing code in a similar fashion as it is also shown in [12]. Table 4.1 lists the test cases for MP.

Tests like MP are commonly referred to as *Litmus Tests*. Litmus tests, like the test presented in Figure 4.2, are small concurrent programs which have a defined initial state and some constraints on their final state. This means that a test case can be executed on a particular pro-

cessor of a certain architecture and the executions of the test case can then be examined whether forbidden states have been observed [12].

Thread 1	Thread 2
$x = 1$	$r1 = y$
$y = 1$	$r2 = x$
Initial state: $x = 0 \wedge y = 0$	
Forbidden: $r1 = 1 \wedge r2 = 0$ at any time	

Figure 4.2: Message Passing Example from [12]

For Sequential Consistency the final state $r1 = 1 \wedge r2 = 0$ not reachable. In order to violate the assertion an execution would be required to execute the write to y in Thread 1 before the write to x, the read of y in Thread 2, the read of x in Thread 2 and finally the write to $r2$ in Thread 2. A program order like this, however, would violate the thread-local program order and is thus not possible in the SC WMM. Thus, CONCRESTWMM using SC cannot to generate inputs such that the assertion is violated. However, when testing using the PSO-model it is not guaranteed that the assertion is never violated. While for x86-TSO and SPARC TSO the mentioned state is forbidden too, architectures like ARM and POWER allow this particular state in their execution models. However, it has to be pointed out that the likelihood of observing an execution where the state is observed is relatively low. For example, during 4.9×10^9 tests the state was observed 10^7 times on a PowerG5 CPU and during 3.8×10^9 tests it was observed 4×10^7 times on a Tegra2 CPU in the experiments presented in [12].

Base Case

As standard schedule (test case `mp-pso1`) the schedules shown in Figure 4.3 are used. Schedules for the symbolic scheduler have the following syntax: on each line the first number denotes the thread-ID and the second entry is either the number of steps or the sign $-$, for a unlimited amount of steps. The schedule for the symbolic scheduler was created manually since this schedule is more intuitive than any schedule generated by CONCRESTWMM for this program. It first executes Thread 1 and subsequently Thread 2 is executed. Doing so, as will be explained later, it can be shown that the assertion can be violated when using PSO. Figure 4.5 illustrates the execution of the test case.

schedule

```
0 1
1 6
0 1
2 10
0 -
```

wmm-schedule

```
PSO
5 1 1 y
14 1 2 r1
14 1 2 r2
```

Figure 4.3: Schedules for `mp-pso1`

schedule

```
0 1
1 2
0 1
1 2
2 8
1 2
0 2
2 2
0 -
```

wmm-schedule

```
PSO
4 1 1 y
14 1 2 r1
14 1 2 r2
```

Figure 4.4: Schedules for `mp-pso2`

schedule	Thread	Execute	wmm-schedule	WMM-Operation
`0 1`	main	start Thread 1		
`1 6`	Thread 1	$x = 1$		
		$y = 1$	`5 1 1 y`	Commit write to y
		set done flag for Thread1		
`0 1`	main	start Thread 2		
`2 10`	Thread 2	$r1 = x$		
		$r2 = y$	`14 1 2 r1`	Commit write to $r1$
			`14 1 2 r2`	Commit write to $r2$
		set done flag for Thread2		
`0 -`	main	Test assertion $\neg(r1 = 1 \wedge r2 = 0)$		

Figure 4.5: Scheduled execution of `mp-pso1`

The second test case (test case `mp-pso2`) for this example program uses a schedule generated by CONCRESTWMM during the execution of the generic test case. The schedule is shown in Figure 4.4 and the contrast to the handwritten schedule can be observed by comparing it to the schedule in Figure 4.4. The generated schedule contains more context switches and is thus harder to comprehend. Semantically there is not much difference between the two schedules since the second schedule (`mp-pso2`) first executes the first write in Thread 1 to x, then Thread 2 is created (but none of its statements executed), after this the second write of Thread 1 to y is executed, now Thread 2 takes over and runs all its statements, after this Thread 1 finishes, then Thread 2 finishes and control is handed back to the main thread.

When using Sequential Consistency the schedules will result in an execution where Thread 2 will see the value 1 for y and 1 for x. However, under PSO, as shown in Figure 4.4, the two writes can be delayed causing Thread 2 to see, for example, the value 0 for y and 1 for x, which would cause the program to terminate in a forbidden state. The test case demonstrates this by using a `wmm-schedule` which delays the writes long enough. Thus, the assertion is violated.

Name	Description
`concrest`	Test the program with CONCRESTWMM and SC
`relaxer1-pso1`	Reach forbidden state using PSO and handwritten schedule
`relaxer1-pso2`	Reach forbidden state using PSO and generated schedule

Table 4.2: Overview of test cases for Relaxer1

Memory Barriers

The forbidden state can be ruled out by the programmer or compiler by inserting memory barriers in the program. Two variants have been implemented for the current example: one test case where a store-load barrier has been added to the source code (target `mp-membar.c`) and one test case where the corresponding event was added as an event to the wmm-schedule (target `mp-membar-schedule`). In both cases the assertion will not be violated.

4.3 Relaxer1

The next example (adopted from [4]), as outlined in 4.6 presents a simple program where two flags are set using data written by a different process. The program contains two data races. The first one is the race of the write of x in Thread 1 to the read of x in Thread 2. The second race concerns the variable y in a similar fashion. Either the write to $t1$ or to $t2$ has to be the last statement to be executed. Thus the statement is supposed to see an earlier write by the respective other thread. As a result of this the forbidden state can never be reached under the Sequential Consistency model. However, certain WMM allow the writes to be delayed in a way such that their effect occurs after the corresponding read was performed, i.e., the writes to $t1$ and $t2$ are performed before the writes to x and y. The consequence of this is that the forbidden state can be reached [4]. Table 4.2 lists the test cases for this example.

Thread 1	Thread 2
$x = 1$ $t1 = y$	$y = 1$ $t2 = x$
Initial state: $x = 0 \wedge y = 0 \wedge t1 = 0 \wedge t2 = 0$	
Forbidden: $t1 \neq 1 \wedge t2 \neq 1$	

Figure 4.6: Example 1 from [4]

Similarly to the MP example the first test case (`relaxer1-pso1`) uses a naïve handwritten schedule for the symbolic scheduler where first Thread 1 is executed until it finishes and, then, Thread 2 is executed subsequently. The second test case (`relaxer1-pso2`) is performed using the schedule generated by CONCRESTWMM shown in Figure 4.7. This schedule will first perform the store of 1 to y in Thread 2 and subsequently load the value of x observing 0. Next, Thread 1 will perform the store of 1 to x. Since x has been read before, the natural behaviour is that when next Thread 2 performs the store to $t2$ it will write the value 0. Now Thread 1

schedule

```
0 2
2 4
1 2
2 2
1 2
2 2
1 4
0 -
```

wmm-schedule

```
PSO
12 1 1 t1
12 1 1 x
14 1 2 t2
14 1 2 y
```

Figure 4.7: Schedules for `relaxer1-pso2`

reads the value of y and under the assumption of Sequential Consistency it will observe 1. After Thread 2 had finished, Thread 1 stores the value in $t1$ and thus the program will not reach the forbidden state under the assumption of SC. CONCRESTWMM will not find a schedule where the assertion can be violated. However, as mentioned before the assertion can be violated when the write to x and y are delayed to the end of the respective threads. The according schedule for the WMM-Scheduler shows that the assertion can be violated.

4.4 Relaxer2

Like the previous examples also the next example as outlined in Figure 4.8 contains no error when examined under Sequential Consistency. Its structure is similar to MP since it also contains several writes on one thread and reads these values from a different thread. The interesting cases are when Thread 1 finished as one would expect that $done$, y and x are indeed 1 which is true under Sequential Consistency. However, under PSO the effect of the write to x in Thread 1 can be delayed such that Thread 2 observes $done = 1$ but $x = 0$ [4]. Note again that this is not possible under SC and thus most test and verification tools will fail to correctly output that the ERROR state is reachable. The test cases for this example are shown in Table 4.3

Thread 1	Thread 2
$x = 1$ $y = 1$ $done = 1$	$if(done)\{$ $\quad if(x == 0)$ $\quad\quad$ ERROR $\quad local = y$ $\}$
Initial state: $x = 0 \wedge y = 0 \wedge t1 = 0 \wedge t2 = 0$	
Forbidden: ERROR	

Figure 4.8: Example 2 from [4]

Name	Description
`concrest`	Test the program with CONCRESTWMM and SC
`relaxer2-pso1`	Reach forbidden state using PSO and handwritten schedule
`relaxer2-pso2`	Reach forbidden state using PSO and generated schedule
`relaxer2-membar`	Forbid state using an instrumented memory barrier

Table 4.3: Overview of test cases for Relaxer2

schedule

```
0 1
1 4
0 1
1 2
2 -
```

wmm-schedule

```
PSO
4 1 1 y
6 1 1 done
10 1 1 x
```

Figure 4.9: Schedules for relaxer2-pso2

Using the schedules shown in Figure 4.9 it can be shown that when using the PSO-model the error state can be reached. This execution will proceed in the following way. First Thread 1 is created. Next, the write to x is performed, but the effect does not become visible to other threads immediately. Subsequently the writes to y and $done$ are performed in Thread 1 and immediately committed. Thus, Thread 1 sees $x = 1, y = 1, done = 1$ but Thread 2 sees $x = 0, y = 1, done = 1$. Now, Thread 2 is created and loads $done$ observing 1. It will go to the `then`-branch of the `if` statement. There, however, it will load the value 0 for x. Thus, Thread 2 will follow the `then`-branch of the second `if`. Afterwards, Thread 1 can commit the write-effect for x but the program will still terminate in the ERROR-state.

This example demonstrates once more that WMM-effects can lead to results which are hard to explain. As outlined above the program can terminate in an error state. When examining the program and considering Sequential Consistency, however, one will be puzzled how the ERROR-state could have been reached. Moreover, since the effect of the write to x can become visible just before the program terminates in the ERROR-state the value of x will be 1 as expected by SC, rather than 0 which caused the program to reach the error state. A result like this is hard to explain since the values observed are as expected for SC but the program went to a branch which differs from the observed values since the write to x became visible only after the value has been read.

Reaching the ERROR-state in this example can be prevented, even when the PSO-model is used, by adding a memory-barrier or some other synchronisation mechanism to the program under test [4]. The test case `relaxer2-membar` demonstrates this behaviour. It uses the same schedule for the symbolic scheduler and WMM-schedule as the test case `relaxer2-pso` but a store-store memory barrier is added to the block where the write to $done$ happens. This means that the barrier has to be executed before CONCRESTWMM can perform a context switch to a

Name	Description
non-blocking-counter-nowmm	Test non-atomic, lock-free program with CONCRESTWMM and SC
non-blocking-counter-lock	Test program using locks, same settings as above
non-blocking-counter-wmm	Test atomic version, same settings as above

Table 4.4: Overview of test cases for non-blocking counter

different thread. Note that "instrumented" memory barriers do not consume additional steps for the schedulers thus the schedules can remain unchanged.

4.5 Non-Blocking Counter

The next example demonstrates the usage of the newly introduced CAS-operation (see Section 3.6 for more details). It demonstrates the implementation of a concurrent counter in a simple program. The counter only features an increment function but can easily be extended to feature a decrement function. Additionally, the program can also be extended to feature multiple counters. The pseudo code of the increment function is outlined in Figure 4.10. Unlike a normal counter the variable is not incremented directly. Instead, the current value is fetched and written to old_count. The incremented value is then written to new_count. In the next step the CAS-operation is used in attempt to update the counter. The operation will succeed, i.e., return 0 as result, if the value of $count$ has not been changed since the current thread fetched the value. Otherwise the operation will return 1 and thus the loop will run again and fetch the new value of $count$. Using a CAS-operation will prevent data races and gets rid of the necessity of locks or other synchronisation mechanisms. The test cases for this example are shown in Table 4.4

```
1  void increment() {
2    int old_count;
3    int success;
4    do {
5      old_count = count;
6      int new_count = old_count+1;
7      // Atomically increment the counter.
8      success = CAS(&count, &old_count, &new_count);
9      // If the counter was changed by someone else --> restart
10   } while (!success);
11 }
```

Figure 4.10: Safe incrementing of a variable using CAS

The test program uses two threads each incrementing the value of the counter. Thus, at the end of the program the value of the counter is supposed to be 2 (modelled as an assertion).

Since the test program uses assumptions, CONCREST is required to find a schedule such that one CAS-operation will fail to update the value. This means that a schedule has to be found where a context switch is performed after line 5, i.e., after the value of $count$ was fetched. For testing purposes, each thread counts the loop iterations performed for incrementing the counter. A first assumption states that both threads have to finish. The next states that either Thread 1 or Thread 2 is required to have looped twice while the respective other thread is required to have looped once. Thus, it is clear that the value $count$ has to be 2.

Three test programs have been implemented. The programs only differ in the implementation of the CAS-operation they use. Each test program is examined using CONCRESTWMM. The first, test program, which can be tested by building the `non-blocking-counter-wmm` target, uses the newly implemented CAS-operation. CONCRESTWMM can find a schedule where a thread has to perform its loop twice. Moreover, at the end of the program the value of the counter is correct, i.e., 2.

Test program `non-blocking-counter-lock` uses a semantically correct implementation of the CAS-operation using a mutex from the pthread library in order to be atomic. This version also delivers correct results. However, it turns out that the test cases using the lock runs roughly a factor 1.3 slower than the test cases using the built-in operation.

The third test program (`non-blocking-counter-nowmm`) uses a non-atomic CAS-operation. Running CONCRESTWMM on this test program will result in the finding of a schedule where the assertion of $counter = 2$ is violated. The violation shows that the implementation of the non-atomic CAS-operation is not semantically correct.

4.6 Store-Buffering

The next example presents a pattern regularly appearing in mutual exclusion algorithms such as Dekker's algorithm which will be presented in the next example. Store-buffering consists like most of the previous examples of two threads and is similar in its structure to the simplified versions of MP. Both threads write to a shared variable and write the value written by the other thread to a register. The example is outlined in Figure 4.11. When examining store-buffering under the assumption of Sequential Consistency the final state of $r1 = 0 \wedge r2 = 0$ will be forbidden since there is no program under which both reads occur before both writes. However, the final state is allowed for relaxed architectures such as x86 or Sparc [12]. Table 4.5 shows the test cases for this example.

Thread 1	Thread 2
$x = 1$ $r1 = y$	$y = 1$ $r2 = x$
Initial state: $x = 0 \wedge y = 0$	
Allowed: $r1 = 0 \wedge r2 = 0$	

Figure 4.11: Store-buffering. Adopted from [12]

In order to be able to observe whether the state is reachable its negation is added to the program under test using an assertion, thus stating that it cannot happen. The test program can be tested

Name	Description
`sb-concrest`	Test the program with CONCRESTWMM and SC
`sb-pso1`	Show reachability of allowed state using CONCRESTWMM and PSO
`sb-membar`	Forbid state using an instrumented memory barrier

Table 4.5: Overview of test cases for SB

using CONCREST by building the default target or by building the `sb-concrest` target. As expected CONCREST will not be able to violate the assertion in the program under test.

As mentioned before, the state is reachable, for example, under PSO but not under SC. The schedules in Figure 4.12 is used for the test case `sb-pso1`. The schedule creates both threads and subsequently executes the write to x in Thread 1 but does not commit it. In the next step it will similarly execute the write to y in Thread 2, again not committing it. Thus, when next the value of y is read in Thread 1 and stored to $r1$ it will still receive 0 as value. Subsequently, the value of x is read in Thread 2 putting 0 to the stack. After the immediate context-switch used to mark Thread 1 as finished the loaded value is stored to $r2$. In the main program the $done$-flag for Thread 1 is read before the done-flag is set on Thread 2. After that, the main program continues until its termination. Due to the observed values it will terminate in an error state, since the assertion will be violated.

schedule

```
0 2
1 2
2 2
1 4
2 2
1 2
2 2
0 2
2 2
0 -
```

wmm-schedule

```
PSO
5 1 2 y
11 1 1 x
20 1 1 r1
20 1 2 r2
```

Figure 4.12: Schedules for `sb-pso1`

However, the state can easily be disallowed by adding memory barriers between the first and the second statement in each thread. For example, a `DMB` barrier can be added for an ARM processor or `sync` barrier for a Power processor. It has to be pointed out that a Load-Load barrier for the PSLO model would not forbid the state [4, 12]. In the sub-test case `sb-membar` a Store-Load barrier is added between the store and load commands on each thread. Thus, while using the same schedules[1] the assertion is not violated.

[1] Memory barriers do not consume scheduling steps

Name	Description
`dekker-concrest`	Test the program with CONCRESTWMM and SC
`dekker-pso1`	Visit both critical sections with CONCRESTWMM and PSO
`dekker-membar`	Forbid visiting both critical sections using a memory barrier

Table 4.6: Overview of test cases for Dekker

4.7 Dekker's Algorithm

Dekker's algorithm is an algorithm used for guaranteeing mutual exclusion. The algorithm is considered to be the first solution to the mutual exclusion problem for two threads. It guarantees that only one process can enter a critical section while also preventing deadlocks. In this critical section a shared resource can be modified, for example.

While the algorithm is error-free for sequentially consistent architectures it is unsafe for architectures with weak memory semantics. A reason for this lies of course in the fact that it was invented at a time when only sequentially consistent architectures where used. Figure 4.13 shows a simplified version of the algorithm. The simplification was done in a similar fashion as it was done for the MP-example. The test case consists of two threads each equipped with a flag indicating its intention to enter its critical section. Each critical section consists of a flag marking if it was visited during the execution. Table 4.6 shows the test cases for this example.

Thread 1	Thread 2
$flag1 = 1$	$flag2 = 1$
$if(flag2 == 0)$	$if(flag1 == 0)$
$c1 = 1$	$c2 = 1$
Initial state: $flag1 = 0 \wedge flag2 = 0 \wedge c1 = 0 \wedge c2 = 0$	
Forbidden: $c1 = 1 \wedge c2 = 1$	

Figure 4.13: Example for Dekker's algorithm, adopted from [1].

The following condition was added as an assertion to the program in order to be able to observe whether the forbidden-state is reachable.

$$(c1 = 1 \wedge c2 = 0) \vee (c1 = 0 \wedge c2 = 1)$$

The assertion will be checked once both threads have terminated. All variables are declared as shared variables to CONCRESTWMM but only the flag variables are declared as shared variables to the WMM-Scheduler. This is done since the flags $c1$ and $c2$ only serve to show whether a critical section has been visited. Thus, delaying a write to these variables would only result in complications and undesired behaviour of the program under test.

The test program can be examined using CONCRESTWMM by building the default target or by building the `dekker-concrest` target. As expected CONCRESTWMM using SC will

not be able to violate the assertion in the program under test. Still it will be able to cover all branches.

Using the target `dekker-pso1` it can be seen that the assertion can be violated using the PSO-model. The reason that the assertion can be violated follows immediately from the Store-buffering example described previously. However, unlike the previous example no generated schedule can be used since CONCRESTWMM is not able to generate a schedule where on both threads the `then`-branch is visited. The schedule for the test case is obtained by increasing the steps of the thread which did not execute the `then`-branch in the CONCRESTWMM generated schedule. Figure 4.14 shows the schedules violating the assertion. After the creation of two threads in the main program Thread 2 is started and writes 1 to $flag2$ without committing the write immediately. Subsequently, the value of $flag1$ is read which will observe 0 leading Thread 2 into the `then`-branch. Next, Thread 1 will write 1 to $flag1$ also not committing its write immediately. After that, Thread 2 will enter the critical section setting $c2$ to 1. The next scheduling entry hands control to Thread 1 which will read $flag2$ observing 0 since the write has not yet been committed. Thus, also Thread 1 enters the critical section where it will set $c1$ to 1 and will finish afterwards. Subsequently, the main program will observe that Thread 1 has finished before handing control to Thread 2 which will then finish. Now, the main program will observe that both threads entered the critical section and thus terminate in an error state.

schedule

```
0 2
2 4
1 2
2 2
1 6
0 2
2 2
0 -
```

wmm-schedule

```
PSO
10 1 2 flag2
11 1 1 flag1
```

Figure 4.14: Schedules for `dekker-pso1`

Similarly to previous examples also Dekker's algorithm can be made safe when using memory barriers. Target `dekker-membar` executes the program under test with an added Store-Load barrier. However, for this test case a schedule different to the schedule of `dekker-pso1` has to be used since in this test case only one critical section will be visited.

4.8 Benchmarks

For all examples presented in this chapter the program was tested 100 times and the runtime was captured. In order to capture the runtime the standard UNIX utility `time` was used and user-time was recorded. The examples were tested using CONCRESTWMM and additionally with a version of CONCREST which was built by disabling the WMM-Support during the build-processes.

Benmark	**CONCREST**	**CONCRESTWMM**	δ
MP	0,153	0,176	0,022
SB	0,149	0,171	0,022
Relaxer-Fig1	0,156	0,179	0,023
Relaxer-Fig2	0,131	0,155	0,024
Dekker	0,162	0,186	0,024
aget	0,288	0,309	0,021
apache1	3,563	3,588	0,024
apache2	0,427	0,449	0,022

Table 4.7: Benchmakrs

In order to be able to test the program under test without WMM-Support any dependencies to the WMM-Support were removed prior to testing for these benchmarks. In addition a similar benchmark was also performed for `aget`, `apache1` and `apache2` from the benchmarks of CONCREST. For this set no modifications to the program under test were made since they do not include dependencies to the WMM-Support. The experiments were performed on a quad-core 64-bit Linux machine with 2.7GHz and 4GB RAM. A summary of the captured data can be found in Table 4.7. The experiments showed that on average CONCRESTWMM is only a constant factor of ~ 1.15 slower than CONCREST when comparing both tools. This small factor gives rise to the assumption that the additional time is spent during the additional parsing of the control flow graph in CONCRESTWMM.

CHAPTER 5

Related Work

There are various other tools than CONCREST for testing and verifying software. Table 5.1 summarises the tools mentioned in this chapter and gives information about their availability. Like CONCREST, SAGE [9] and CUTE [17] implement the DART (Directed Automated Random Testing) approach but have no support for multi-threading. While the test tool CUTE (Concolic Unit Testing and Explicit Path Model-Checking) can only test sequential C programs, jCUTE can test concurrent Java programs for data races, deadlocks, infinite loops and uncaught exception errors [17]. ConTest [6] takes a different approach: It takes a Java program and a test for the program and tries to increase the coverage of two concurrency coverage criteria. The first coverage criteria requires a context switch for each method in the Java program, while the second coverage criteria requires a context switch for every method to all other methods in the Java program. ConTest tests a program only on a single core CPU. The tool instruments the program using sleep, yield and priority to force a context switch from one thread to a different thread during execution [6]. Similar to ConTest, Poirot [16] also focuses on systematically investigating context switches. Poirot translates a concurrent program into a sequential program. This sequential program is forced to contain context switches which are considered to be crucial for possible bugs. The sequential program is then analysed using static analysis techniques. Chess [14] uses model checking for testing concurrent programs. It tries to enumerate possible thread schedules giving priority to schedules with fewer preemptions. To apply Chess to a program under test one has to provide a test harness. The framework provides wrapper functions for common concurrency API-functions such as the Win32 API such that the execution of the test can be instrumented and recorded. Fusion [18, 19] also records a concurrent trace of a program execution but then transforms this trace into a logical formula that also encodes certain properties to be checked. If the formula has a solution then a property violation has been determined or otherwise the trace does not violate the given properties. As an extension, the concept of interference abstractions, a concept similar to interference scenarios, is introduced [18].

None of the previously mentioned tools is able to test a program under test with respect to effects of WMM-models. However, there are two tools that are able to test concurrent programs using WMMs which are discussed in more depth in the following sections. The first tool, CBMC,

Tool	Basic principle	WMM support	Source code available
CBMC [2]	bounded model checking	yes	yes
CREST [1]	concolic testing	no	no
CONCREST [7]	$(\text{con})^2$colic testing	no	yes
SAGE [9]	concolic testing	no	no
Cute [17]	concolic testing	no	yes
jCute [17]	concolic testing	no	yes
CHESS [14]	model checking	no	yes
Poirot [16]	static analysis	no	no
Fusion [18, 19]	model checking	no	no
ConTest [6]	dynamic analysis	no	no
RELAXER [4]	dynamic analysis	yes	no

Table 5.1: Overview of concurrency testing tools

has an extension which makes bounded model checking with respect to a given WMM possible. One problem with this approach is that bounded model checking does not scale well for larger concurrent software [2]. The second tool is RELAXER [4], which facilitates an active random testing technique and combines it with the search for data races using different WMMs. This means that the program uses an analysis to predict potential bugs. After this initial analysis the program tries to trigger these predicted bugs by actively controlling the scheduler.

5.1 CBMC

Alglave et al. [2] propose a method for using bounded model checking to verify concurrent software under weak memory semantics. The underlying idea is that instead of modelling the events from the program under test as a total order, thus assuming Sequential Consistency, the events are ordered using partial orders. This relaxation makes it possible to model architectures weaker than SC. Their method proposes a symbolic decision procedure for partial orders. In particular, they implemented their encoding as an extension of the model checker CBMC.

Partial Orders

An event is a tuple consisting of a unique identifier for the event, a direction (*r*ead or *w*rite), an identifier for the memory address and a value written to or read from the memory address. For example the event $(e)\ Wx1$ denotes the write event e where 1 is written to x. The event structure E consists of a set of events $\mathbb{E}$ and a program order po which is a total order over the events for one processor. By dp a subset of po ($\mathsf{dp} \subseteq \mathsf{po}$) is denoted which includes all

[1] `http://code.google.com/p/crest/`

dependencies between events in the program under test, for example, whenever the address for an access is computed using a previous load. This can be seen, for example, when pointer arithmetic is used. The write serialisation ws is a total order for every address and the writes to it, i.e., the writes to one address are ordered and only one write to a specific address can occur at a time. rf is a relation which links write events to read events so that a read r will read precisely the value written by write w. Then, an execution of a program can be modelled as an execution witness X which consists of the two relations ws and rf. The relation rf can further be separated into internal read-from rfi and external read-from rfe events. rfi events are events where both a write w and a read r happen on the same processor whereas w and r happen on distinct processors for rfe events. By using rf and ws it can be ensured that a write w_0 ordered before the write w_1 when $(w_0, w_1) \in$ ws and for some write r $(w_0, r) \in$ rf. Furthermore, the relation from-read fr can be used to express that r happens before w_1. A pair $(r, w_1) \in$ fr is the equivalent of stating that there exists a w_0 such that $(w_0, r) \in$ rf and $(w_0, w_1) \in$ ws.

The orders described above model Sequential Consistency in case they are total. Weak Memory Models can be modelled by relaxing sub-relations of po and rf. This means that, for example, reads from the rf relation are not included in the global happens before relation. The internal reads rfi can also be relaxed in order to model architectures which feature store buffers. Moreover, also the fact that some architectures allow processors to communicate privately via a cache can be modelled by relaxing the external reads rfe. This relaxation of rfe also means that writes atomicity is relaxed. If a write event e_1 is non-atomic then another write event e_2 to the same memory location can happen between the write event e_1 and the corresponding read event from the rf relation. Subsets of the relations which are guaranteed to be in order for an architecture are called the preserved program order ppo. Another subset of the program order are fences. A fence is called cumulative if it makes writes atomic, i.e., sequences of rfe cannot be relaxed. Non-cumulative fences guarantee that certain events pairs surrounding the fence, i.e., one event before and one event after the fence, are in order. The union of fences and cumulativity is denoted as ab.

An architecture is a concrete realisation of the order-relations described above and thus determines which of the relations are safe, i.e., which relations cannot be relaxed. ws and fr are considered to be always safe whereas read-from is in most cases relaxed. Thus, the subset grf is introduced which denotes the subset of the read-from relation which is safe on all processors for this particular architecture.

A candidate for an execution of the modelled program is valid for a specific architecture when the following three conditions hold. The first condition is that accesses to the same address are sequentially consistent. This means that the program order for same locations (po-loc) is compatible with the communication between processors, i.e., ws $\cup$ rf $\cup$ fr $\cup$ po-loc does not contain a cycle. No causal loop exists for the values, i.e., rf $\cup$ dp does not contain a cycle. The third condition is that the events can be linearised in a global happens before structure, i.e., ws $\cup$ grf $\cup$ fr $\cup$ ppo $\cup$ ab does not contain a cycle.

Translation to SAT

The decision procedure as described in the previous section gives the possibility to determine whether a particular execution witness is valid for a specific architecture. However, the aim of

the presented work is to reason about properties and possible executions of a program. Thus, a formula is constructed which is satisfiable iff there exists an execution for the encoded architecture which violates the reachability property also encoded in the formula. The formula consists of two conjuncts. The first conjunct of the formula, ssa, models the data and control flow for every thread of the program using static single assignment for capturing writes. For every write on every thread a fresh new index is introduced.

The second conjunct, pord (partial order constraints), models the communication between the threads of the program under test. It thus highly depends on the assumptions for the architecture it is targeted to represent. In order to be able to define the order of events, clock variables $clock_x$ are introduced for all events within the program. Additionally, clock constraints are defined which can then be used to reason about the time-relation between events. A clock constraint c_{xy} states that for the events x,y the relation $clock_x < clock_y$ holds. Furthermore, variables s_{wr} are introduced to represent a read-from relation between a write w and a read r. Thus, the two conjuncts ssa and pord are modelled as representations for the presented orders.

The constraints for the preserved program order ppo contain a clock constraint c_{e_1,e_2} for every pair of events e_1, e_2 from the same thread if the following two conditions hold. First, the pair is safe on the selected architecture A. Secondly, there is a control flow path between e_1 and e_2 and it is not in the transitive closure of C_{ppo}[2]. The read-from relation is encoded as the sets C_{rf} and C_{grf}. For every (w, r), which is safe for A, a constraint $s_{wr} \rightarrow c_{wr}$ is added to C_{grf}. Additionally, for each such pair in rf a constraint has to be added which ensures that the guard for the write is true and the read will read the value written by the corresponding write event. In order to do so, the constraint $s_{wr} \rightarrow g(w) \wedge x_w = x_r$ is added to C_{rf}. Since ws is a total order it contains for every pair of writes w, w' with $w \neq w'$ a constraint ensuring that either $c_{ww'}$ or $c_{w'w}$. As mentioned above pairs in fr model an implicit existential quantification of another element in rf. In order to encode this existential quantification for pairs $(r, w) \in$ fr a disjunction over constraints $s_{w'r} \wedge c_{w'w} \rightarrow c_{rw}$ is added to C_{fr} which is equivalent to stating that there exists a w' so that $(w', r) \in$ rf and $(w', w) \in$ ws.

Relation to CONCREST

While it is hard to compare a verification tool with a testing tool there are some similarities between the CBMC extension and CONCREST, respectively the WMM-Support presented in this thesis. CONCREST uses the notion of temporal consistency constraints which exhibit some similarities to the order-structures described above and the relations between the orders. Moreover, both approaches use the idea of relating read-events to write-events in their encodings.

One of the major differences is that due to its nature CBMC reasons about executions upfront while CONCREST uses the information gathered through executions for its analysis of the program under test.

[2] A constraint c_{e_1,e_3} is redundant if there are constraints c_{e_1,e_2} and c_{e_2,e_3}.

5.2 RELAXER

The testing tool RELAXER as proposed in [4] is capable of testing concurrent C programs using weak memory semantics. Burnim et al. propose a method which combines dynamic analysis with software testing in order to verify concurrent software using introduced weak memory model semantics. Their approach works in two phases: a generation phase and a check phase as will be described below.

In the first phase, execution traces of the program under test are examined for potential happens-before-cycles under SC. Each of the gathered execution traces consists of load and store events. Each of these events consists of a label for the event, a processor-identifier, a memory address, a value, an instruction index for the processor and a commit index. RELAXER also defines a commit index for read events. The commit index for a read just refers to the index of the value in the series of values written to the memory address. Similar to the cycles mentioned in [2] a happens-before-cycle is a cycle of self-supporting events. This means that when arguing about the causality of read/write events a cycle is formed. For defining the cycle formally the conflict-order relation is introduced. Two events e, e' are considered to be conflict-related if both access the same memory address and either e is a store, e' a load event, where the commit index of e is lower than the commit index of e', or e is the load event and has a lower commit index than e'. The happens-before relation is the union over all events in program-order relation with all events in conflict-order relation. A trace thus violates sequential consistency if there exists a cycle in the happens-before relation. In order to obtain a trace the program under test needs to be executed using a test harness. The program is then executed using this harness with random schedules. Gathered traces are then analysed for sets of events forming potential happens-before-cycles. For each of these sets the second phase of RELAXER is executed.

In the second phase RELAXER will try to find a valid execution for a selected WMM for each of the potential happens-before-cycles from Phase 1. To do so RELAXER executes the program again while actively controlling the schedule and memory operations of the program under test. During these executions operational semantics are required for the selected WMM. RELAXER defines these semantics for Total Store Order (TSO), Partial Store Order (PSO) and Partial Store Load Order (PSLO). In addition to the modification of the operational semantics, RELAXER also uses strategies tailored for each of the models. Each of these strategies consists of functions like `before_store`. The mentioned function is executed before a store operation in the program under test is executed and decides whether the effect of a store should become visible immediately or whether the effect should be delayed. Delays are implemented via a `time`-function and a `waitUntil`-function. While the former is used to obtain time-codes, the latter is used to pause a thread until the time-code is reached. Common among the strategies is the existence of a buffer-list and the `after_any`-function. When a write occurs then the time-code when it is supposed to become visible and the effect of the write-event are appended to the buffer-list. The `after_any`-function is executed after any intercepted operation of the program under test. Using this function the write-events can then be committed at the desired point of time.

Relation to CONCREST

Similarly to CONCREST, RELAXER uses CIL for instrumenting a C program. The instrumentation is used for intercepting loads, stores and synchronisation primitives within the program under test. However, while CONCREST is able to test a program without the need of test harnesses, RELAXER requires the tester to provide a test harness for the program under test. This test harness is responsible to drive the program through the various paths of the program. Moreover, CONCREST is able to systematically examine the concurrent program while RELAXER relies on random schedules during its first phase.

Another similarity to the work presented in this thesis is that RELAXER also adds WMM semantics to its execution model by simulating the WMM-effects using operational models. However, while RELAXER additionally uses heuristics to test for bugs under WMM-semantics, the work of this thesis only offers the possibility of executing a program while simulating the effects of a WMM.

CHAPTER 6

Conclusion & Future Work

Conclusion. This thesis presented CONCRESTWMM, a concolic testing tool for concurrent software that is able to simulate the effects of weak memory models during an execution of a program under test. CONCRESTWMM was implemented as an extension of the tool CONCREST. As a central component, CONCRESTWMM adds a WMM-Scheduler. By using WMM-schedules it is possible to select a WMM and to trigger effects of this model at specific points during the execution of a test case. Thus, the tool CONCRESTWMM offers the possibility to discover bugs that can occur on real-world processor architectures that do not adhere to Sequential Consistency but to another WMM.

As a part of the thesis two WMMs were implemented. First, the WMM Sequential Consistency which can be used to simulate the behaviour of CONCREST. This model does not need any additional schedules. Secondly, the PSO WMM was implemented. This model allows to delay the effect of a write event.

Memory barriers are a common means to limit the deviation of executions from Sequential Consistency. Support for memory barriers was also implemented in CONCRESTWMM. By using memory barriers it is possible to force the effect of writes to become visible, i.e., memory barriers can be used to guarantee certain commits to happen. CONCRESTWMM offers three types of memory barriers which have to implemented by the WMM.

Since many concurrent applications and data structures make use of a Compare and Swap (CAS) operation, a CAS operation was also implemented as part of CONCRESTWMM. The CAS operation updates a memory cell with a new value if the cell contains an expected old value. The built-in operation offers the same semantics as the Boolean CAS operation offered by the compiler GCC and is atomic. Moreover, the built-in CAS operation is equipped with symbolic information. This enables CONCRESTWMM to search for test cases covering both branches of the CAS-operation, i.e., finding test cases where the value at the memory location is changed and test cases where the value is not changed.

To test the implementation of CONCRESTWMM and to show its capabilities, several experiments were performed. These examples are taken from the literature. The experiments showed

that on average CONCRESTWMM is only a constant amount of ~ 0.02 seconds slower than CONCREST when comparing both tools.

Future Work. As mentioned in Chapter 3, there are several possibilities for extending CONCRESTWMM in order to be able to examine concurrent programs while identifying WMM related bugs systematically. A possibility for solving this problem is to equip CONCRESTWMM with a search facility identifying WMM related bugs for a given schedule. Another possibility for integrating the full support of WMM-effects into CONCREST is to encode the semantics of WMM-effects into the constraint systems of CONCRESTWMM. These modified constraint systems then produce schedules for both the symbolic scheduler and the WMM-Scheduler of CONCRESTWMM. Additionally, there is also the possibility of reformulating the constraint systems of CONCRESTWMM in a way so that solutions for them yield interferences where a relaxation of the write-consistency would violate an assertion. See also the discussion in Chapter 3.7.

A compiler inserts memory barriers into a program and uses CAS operations at specific points. However, this is compiler dependent, i.e., the program generated from the same source code can be different for two compilers. As a result of this two programs generated from the same source code but with different compilers can behave differently. CONCRESTWMM can be extended to simulate the behaviour of different compilers. For example, the instrumentation process can be changed in order to add memory barriers to the instrumented program under test reflecting the behaviour of the selected compiler. Moreover, the instrumentation process could be changed in such a way that it adds CAS operations where a compiler would add them. Alternatively, memory barriers can also be added to the program under test using WMM-schedules.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 7. ACM, 2014.

[4] Jacob Burnim, Koushik Sen, and Christos Stergiou. Testing concurrent programs on relaxed memory models. In Matthew B. Dwyer and Frank Tip, editors, *ISSTA*, pages 122–132. ACM, 2011.

[5] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In Patrice Godefroid, editor, *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23. Springer, 2005.

[6] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[7] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *ESEC/SIGSOFT FSE*, pages 37–47. ACM, 2013.

[8] Patrice Godefroid. Compositional dynamic test generation. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 47–54. ACM, 2007.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[10] Free Software Foundation Inc. Gnu compiler manual. https://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc/Atomic-Builtins.html. last-seen: 2014-09-04.

[11] Intel. Pentium processor family user's manual: Vol 3, architecture and programming manual, 1994.

[12] Luc Maranget, Susmit Sarkar, and Peter Sewell. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models.

[13] Moir Mark and Nir Shavit. *Concurrent data stuctures*, pages 47–14. Handbook of Data Structures and Applications, 2007.

[14] Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. CHESS: A Systematic Testing Tool for Concurrent Software, 2007.

[15] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[16] Shaz Qadeer. Poirot - a concurrency sleuth. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, volume 6991 of *Lecture Notes in Computer Science*, page 15. Springer, 2011.

[17] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

[18] Nishant Sinha and Chao Wang. On interference abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 423–434, New York, NY, USA, 2011. ACM.

[19] Chao Wang, Sudipta Kundu, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. *Formal Aspects of Computing*, 23(6):781–805, 2011.

[20] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *EDCC*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.