Choosing Among Alternative Futures

Steve MacDonald[1], Jun Chen[1], and Diego Novillo[2]

[1] School of Computer Science, University of Waterloo, Waterloo, Ontario, CANADA
{stevem,j2chen}@uwaterloo.ca
http://plg.uwaterloo.ca/∼stevem, http://www.cs.uwaterloo.ca/∼j2chen
[2] Red Hat Inc.
dnovillo@acm.org, http://people.redhat.com/dnovillo

Abstract. Non-determinism is a serious impediment to testing and debugging concurrent programs. Such programs do not execute the same way each time they are run, which can hide the presence of errors. Existing techniques use a variety of mechanisms that attempt to increase the probability of uncovering error conditions by altering the execution sequence of a concurrent program, but do not test for specific errors. This paper presents some preliminary work in deterministically executing a multithreaded program using a combination of an intermediate compiler form that identifies the set of writes of a shared variable by other threads are visible at a given read of that variable and aspect-oriented programming to control program execution. Specifically, the aspects allow a read of a shared variable to return any of the reaching definitions, where the desired definition can be selected before the program is run. As a result, we can deterministically run test cases. This work is preliminary and many issues have yet to be resolved, but we believe this idea shows some promise.

1 Introduction

Testing concurrent programs is difficult because of non-determinism; each execution of the program is different because of changes in thread interleavings. Of the large possible number of interleavings, only a few may cause errors. In particular, we are concerned about *race conditions*, when the interleaving violates assumptions about the order of certain events in the program. For this paper, we define a race condition as two operations that must execute in a specific order for correctness but where there is insufficient synchronization to guarantee this order. An important characteristic of this definition is that mutual exclusion is not necessarily a solution; two updates of a variable in different threads, even if protected by locks, can still cause a race under this definition. This is a more general definition that considers timing-dependent errors.

This paper presents preliminary work on deterministically executing a concurrent program using a combination of two technologies. The first technology is CSSAME [19,20,21,22], an intermediate compiler form for explicitly-parallel shared-memory programs that identifies the set of writes to a shared variable by other threads are visible at a given read of that variable. These writes are called *concurrent reaching definitions* for that particular use of the shared variable. The second is aspect-oriented programming, which allows us to intercept field accesses and method calls in object-oriented programs and inject code at these points [14,13]. Using these technologies, we have created a technique for deterministic execution with three desirable characteristics. First, for a given race condition we can deterministically execute each order, allowing all paths to be tested. In contrast, many other tools are dependent on the timing characteristics of a specific execution of the program which means some orders may not be properly tested. Other tools introduce code into the program and can subtly change its timing, preventing the race condition from appearing during execution. Our deterministic approach does not rely on such timing, and does not suffer from this problem. Second, our method requires no existing execution trace of the program since we are deterministically executing the program, not replaying it. Third, the method works even if accesses to shared variables are not protected by locks.

This deterministic execution can be used in several ways. First, not all race conditions represent errors; in some cases, each order may be acceptable. However, in testing it is important that each order be enumerated to verify correctness. This technique can be used for this enumeration, ensuring each case is handled. Second, other testing methods enumerate over a large number of test cases by generating different interleavings, but cannot guarantee that specific, vital tests are run. This work could be used as an initial sanity test, to explicitly check basic functionality. Third, deterministic execution could be used to support incremental debugging. A user can construct schedules where the order differs in a small number of ways. If one schedule is faulty and another correct, then the difference may be used to locate the error.

This paper is organized as follows. Section 2 presents related research. Section 3 describes the CSSAME form and the compiler that produces it. A brief description of aspect-oriented programming is given in Section 4. In Section 5 we describe how we combine CSSAME and aspects to deterministically execute a concurrent program. Examples of the use of this technique are given in Section 7. Outstanding issues and future work are presented in Section 8, and the paper concludes with Section 9.

2 Related Work

One common testing technique is to introduce noise into multithreaded programs to force different event orders, usually adding conditional sleeps or yields on synchronization or shared variable accesses [25,9]. One research effort added noisemaking using aspect-oriented programming [8]. These techniques rely on properly seeding the program, determining which delays to execute and, in the case of sleep, how long to delay. This technique is not intended to test a specific problem, but rather increases the probability that race conditions will appear if the program is run many times.

Another interesting technique that has been used is based on ordering the execution of the atomic blocks in a multithreaded program [4]. This effort assumes that all accesses to shared data are protected by locking. Under these conditions, the behaviour of a concurrent program can be captured by enumerating over all possible orders of the atomic blocks in the program. This captures the more general idea of a race condition in this work. However, it does not address unsynchronized accesses to data. Also, this work uses a customized JVM that includes checkpointing and replay facilities.

An improvement on these schemes is *value substitution* [1]. Rather than perturb the thread schedule with noise, value substitution tracks reads and writes to shared data. When a read operation is executed, the value that is returned can be taken from any already-executed write operation that is visible to the read operation, simulating different thread schedules. To ensure that the substituted values are consistent, a visibility graph is produced to maintain event ordering. There are two main weaknesses of this algorithm. First, the amount of data needed for the visibility graph is prohibitively large. Second, the substituted values must be from writes that have already executed, which can limit test coverage. The second weakness was addressed through *fidgeting* [2], where the choice of substituted value is delayed until the value is used in a program statement that cannot be re-executed (such as output or conditionals). This delay increases the possible substitutions for a read and allows more schedules to be tested.

Another common strategy is program replay, in systems like DejaVu [5] among others. These systems capture the state of a program execution and use it to reproduce the execution. If the captured execution is an erroneous one, then it can be rerun multiple times to debug the problem. However, it may take many executions of the program to produce an erroneous schedule. Further, capturing the state can perturb the execution, reducing the probability of the error (or even removing it altogether).

A variation of replay called *alternative replay* uses a visibility graph to produce alternate schedules from a saved program execution [2]. This work is similar is style to *reachability testing* [11]. Alternative replay takes a partial execution state of a program and runs it up to some event e. Before e is run, the replay algorithm can look ahead in the visibility graph and may reorder events to produce an interleaving that is different from the recorded one. For example, if e is a read operation, alternative replay may trace the visibility graph to find a write operation w that executed later in the program but could have run before e, making w a potential reaching definition for the read. In baseline value substitution, because w happened later, its value cannot be substituted in the read. Alternative replay rewrites the visibility graph so w happens first. From this point, the program is run normally. Reachability testing systematically enumerates over all possible thread interleavings by tracing every execution and generating alternative schedules from those as well. Both of these systems rely on being able to establish a total order of the events in a program (at least for those that are replayed at the start of the program execution). In contrast, the properties of the CSSAME form permits us to achieve the same effects without imposing this total order, by having a history of each write to a variable. The necessary properties are discussed in Section 5. Further, we can control the execution of the complete program or any portion, not just starting from the beginning of the execution.

Other systems detect race conditions at runtime, such as Eraser [24]. These tools check that accesses to shared data are performed while a lock is held. However, these tools can only detect problems for a specific execution of the program. Since data races may not happen on each execution, these tools may miss race conditions. Some of these tools may also produce false alarms. Furthermore, these tools only check that shared data accesses are protected by locks, and do not address the more general notion of a race condition that is used in this paper.

```
$a$ = 0
if (condition)
    $a$ = 1
print($a$)
```

(a) Original source code.

```
$a_1$ = 0
if (condition)
    $a_2$ = 1
$a_3$ = $\phi(a_1, a_2)$
print($a_3$)
```

(b) SSA form.

Fig. 1. An example of ϕ functions in SSA.

The idea of incremental debugging by schedule comparison has been explored using *Delta Debugging* [6]. The technique starts with a failing schedule and then tries to construct a correct execution using a combination of inserting noise and splicing scheduling information from successful runs. The correct and incorrect schedules are compared to determine their differences and locate the source of the error. The process is repeated until the most likely source of the error is located.

Another approach to locating errors in concurrent code is model checking, implemented by tools like Java Pathfinder [] and many others. Most model checkers do not execute the application, but rather create an internal representation (usually a finite state machine) and analyze the representation to verify properties of the original program. This internal representation may be in a modeling language rather than the original source code, and it generally elides much of the detail in the program to reduce the size of the representation and make analysis tractable. JPF-2, the second generation of JPF, implements its own Java virtual machine to run the application to locate errors. JPF-2 uses the Eraser algorithm described above to locate simple data races. Since they implement their own JVM, they also control the thread scheduler and use this to enumerate over the different set of thread interleavings, using backtracking to save re-executing a program from the beginning for each test. The enumeration is similar to that used in reachability testing. Our aspect-oriented approach cannot implement backtracking. However, our approach is considerably simpler and does not require a new JVM.

3 CSSAME - Concurrent Single Static Assignment with Mutual Exclusion

CSSAME (Concurrent Static Single Assignment with Mutual Exclusion, pronounced *sesame*) [19,20,21,22] is an intermediate compiler form used to analyze explicitly parallel shared-memory programs (programs where parallelism is explicitly expressed by introducing library calls or language constructs into the application code). CSSAME is a variant of the Concurrent Static Single Assignment form [17] that includes additional analysis based on the synchronization structure of the program. Both forms extend the Static Single Assignment form.

The SSA form has the property that each variable is assigned only once in the program, and that every use of a variable is reached by one definition. However, control flow operators can result in multiple reaching definitions. To resolve this problem, SSA includes *merge operators* or ϕ *functions*. Figure 1 shows an example of the SSA form.

CSSAME extends SSA to include π functions that merge concurrent reaching definitions from other threads in an explicitly parallel program. A concurrent reaching

```
cobegin
    $T_1$:  Lock($L$)
            $a$ = 1
            $a$ = 2
            Unlock($L$)
    $T_2$:  Lock($L$)
            $a$ = 3
            Unlock($L$)
    $T_3$:  Lock($L$)
            print($a$)
            Unlock($L$)
coend
```

(a) Original Odyssey source code.

```
cobegin
    $T_1$:  Lock($L$)
            $a_1$ = 1
            $a_2$ = 2
            Unlock($L$)
    $T_2$:  Lock($L$)
            $a_3$ = 3
            Unlock($L$)
    $T_3$:  Lock($L$)
            $a_4$ = $\pi(a_2, a_3)$
            print($a_4$)
            Unlock($L$)
coend
```

(b) CSSAME form.

Fig. 2. An example of π functions in CSSAME.

definition is a write to a shared variable by one thread that may be read by a particular use of that variable. The value that is read will be one of these concurrent reaching definitions. One important contribution of the CSSAME form is that it prunes the set of reaching definitions based on the synchronization structure of the program. This pruning is based on two observations regarding the behaviour of shared variable accesses inside critical sections [22]. First, a definition can only be observed by other threads if it reaches the exit of a critical section. Second, if a definition and use occur within the same critical section, then concurrent definitions from other threads cannot be observed. This pruning process reduces the number of dependencies and provides more opportunities for compiler optimization.

An example of the CSSAME form is shown in Figure 2. In T_3, the use of the variable a can only be reached by one definition, which requires all concurrent reaching definitions be merged using a π function. When generating the arguments for the π function, note that the definition a_1 in T_1 cannot reach this use because of synchronization; a_1 does not reach the exit of the critical section because of definition a_2.

The set of reaching definitions for unprotected accesses to shared memory are dictated by the underlying memory model. Odyssey, the compiler that implemented the CSSAME form, assumes a sequentially consistent memory store. However, different memory models can be accommodated by changing the placement of and arguments to the π functions [19].

The Odyssey compiler works with a superset of C that includes explicit parallelism in the form of cobegin/coend constructs and parallel loops. The cobegin/coend construct is used in this paper. In this construct, the body of each thread is indicated using a switch-statement-like syntax, as shown in Figure 2. Each thread must be finished at the coend statement. In addition, Odyssey supports locks for mutual exclusion. Event variables and barriers are also available, but Odyssey does not take this synchronization into account when adding terms to π functions.

```
aspect LoggingAspect {
    Logger logger = new Logger();

    pointcut publicMethods(): call(public * *(..));

    before() : publicMethods() {
        logger.entry(thisJoinPoint.getSignature().toString());
    }
    after() : publicMethods() {
        logger.exit(thisJoinPoint.getSignature().toString());
    }
}
```

Fig. 3. Logging aspect code in AspectJ.

4 Aspect-Oriented Programming

Aspect-oriented programming was created to deal with *cross-cutting concerns* in source code [14]. A cross-cutting concern is functionality in a system that cannot be encapsulated in a procedure or method. Instead, this functionality must be distributed across the code in a system, resulting in tangled code that is difficult to maintain.

A common example of such a concern is method logging, where every public method logs its entry and exit times. Such code must be placed at the start and end of every public method; there is no way to write this code once and have it applied to every method. Worse, every developer of the application must be aware of this requirement when adding new methods or changing protection modifiers on existing ones, and changes to the logging interface may require many changes throughout the code.

Aspect-oriented programming was created to address these cross-cutting concerns. Such tangled code is encapsulated into an *aspect*, which describes both the functionality of the concern and the *join points*, which indicates where the aspect code should appear in the original source code. The aspect code and application code are merged in a process called *aspect weaving* at compile time to produce a complete program.

Figure 3 shows the aspect code for the logging example using AspectJ [13], which supports aspect-oriented programming in Java. Like a class, an aspect can have instance variables, in this case an instance of the logging class. The pointcut represents a set of join points. In this example, the pointcut represents any call to any public method with any return type through the use of wildcards. Following that are two pieces of *advice*, which is the aspect code to be woven into the application code. The first piece of advice represents code that should be run before the call is made, which logs the entry. The argument is a string of the signature of the method, obtained through the join point in the aspect. The second piece of advice is run after the method returns, logging its exit.

There is a wide variety of point cuts other than method calls that can be intercepted, including constructor executions, class initialization, and object initialization. For this paper, the most important of these is the ability to define a pointcut that intercepts accesses to fields, both reads and writes to instance and static variables.

5 Controlling the Future: Controlling the Execution of the CSSAME Form using Aspects

The key observation is to note that the π functions in the CSSAME form shows, for a given use of a variable, all possible concurrent reaching definitions from other threads contained within the code that the compiler analyzes. Concurrent reaching definitions hidden from the compiler (*i.e.*, contained in library code) cannot be analyzed and limit our ability to properly detect race conditions. Assuming that no accesses are hidden, the CSSAME form identifies all potential race conditions in the program. Further, the terms in the π function show all possible values that can reach a given variable use. If one of these values leads to an error, then it forms a race condition. Thus, a necessary condition for a race condition is a term in a π function that, if selected for a particular use, causes an error. Finding race conditions requires these terms be identified, and removing race conditions requires these terms be removed. In some cases, simply detecting an extra term in the π function may be enough to find a race condition if the race is the error [19].

However, there are legitimate reasons for multiple definitions to appear in a π function that do not represent concurrency errors. If the race is not an error, then the terms in the π function show the possible execution interleavings for a given variable use. In such cases, it is important to be able to test multiple paths through the code, including paths caused by concurrent reaching definitions. Unfortunately, the π functions also represent the non-deterministic parts of a concurrent program; the exact reaching definition (or, rather, the term whose value is returned in the π function) is determined at runtime based on the order of events during the execution of the program.

For testing purposes, it is desirable to be able to deterministically execute a concurrent program. For a given π function with n terms, we would like to run the program exactly n times to check that each reaching definition results in a correct execution. This determinism has three main benefits. First, it reduces the number of times the program must be run to test it. If non-determinism is still present, the program may need to be run many times to ensure all orderings are covered. Worse, some orderings may not occur during testing because of the deterministic nature of some thread schedulers, instead appearing only after the application has been deployed on a system with a different scheduler [1]. Second, determinism provides better support for debugging. Once a bug is detected, it must usually be replicated several times before the source of the error can be located and fixed. Once a fix is applied, the program must be retested to ensure the fix is correct. If the error appears infrequently, this debugging process is difficult. Third, determinism can eliminate the possibility of hiding the bug when trying to locate it. Adding code to monitor the execution can alter the order of events and mask the problem, only for it to reappear once the monitoring code is removed.

The first part of our approach to removing the non-determinism in the execution of a concurrent program is to execute the code from the CSSAME form. That is, for the code in Figure 2(a) we generate the code in Figure 2(b), where each variable is assigned once and has only one reaching definition. Also included in the transformed code are the π functions for concurrent reaching definitions and ϕ functions for control flow.

To generate different test cases for Figure 2(b), we need to control the value returned by the π function. Again, the arguments to the π function are the set of potentially reaching definitions at this point in the execution of the program. The return value must be one of the arguments, a_2 or a_3, which is assigned to a_4 and printed in the next line.

This leads to the second part of our approach, which is to control the π functions using aspects. To produce different test cases, the user selects the desired reaching definition from the terms in the π function. Different π functions are distinguished by adding the source line number to the function name. The call to this function is intercepted by an aspect, which can override the return value to return the desired definition.

The benefit of using an aspect to control the execution of the CSSAME form is the same benefit as aspects in general: separation of concerns. Using this approach, we can generate the CSSAME form once, independently of any specific test case. Each test case is written as a separate aspect. Keeping the test cases separate from the code makes the development of each simpler.

However, we need to ensure that the reaching definition has been written before the read can take place. This task is accomplished by having our test aspect intercept all writes to a field. This is sufficient in Java because only fields can be shared between threads. Here, we benefit from the single assignment nature of the CSSAME form. Each instance variable is represented by a set of individual subscripted variables. We can save the values for all writes to a field since they are now separate variables. Furthermore, we treat each variable as a latch. This latch is easily implemented as a class that maintains a boolean variable indicating if the value has been set. A read of the field must block until the latch has been set. A write sets the latch value and unblocks readers attempting to obtain the value for the π function.

As an example, consider the code in Figure 2(b). The print statement in T_3 has two possible outcomes: 2 (from a_2) or 3 (from a_3). The aspect code for the test case where T_3 reads the value from a_2, with additional support code, is shown in Figure 4. In addition, the source code includes implementations for the π functions which are subsumed by the aspect. To get the π function to return the value for a_3, we need only change the value of the choice variable in the advice for the π function to latch_a_3.

An important characteristic of this approach is that it is not sensitive to any timing in the program execution. If the desired term in a π function has not yet been written, the reading thread blocks. We also exploit the single assignment property of CSSAME. Each write is to a separate instance of a variable, meaning we have a complete history of each write. Thus, we can return any already-written term in a π function, even if the same variable has been updated many times. Further, the read value is assigned to a specific instance of a variable, meaning that it cannot be subsequently overwritten by another thread, which removes race conditions on variables involved in test cases. This facilitates testing and debugging of concurrent programs.

The CSSAME form also includes ϕ functions to merge multiple definitions resulting from control flow operators. The ϕ functions cannot be removed as they may be used as terms in a π function. Thus, they must also be executed. Again, we use our aspect. The aspect captures all writes to a variable and includes advice for the ϕ functions. For this case, it is important to note that a ϕ function only includes terms from the current thread and not concurrent definitions. Thus, the aspect maintains a single thread-local variable that represents the last update of the variable by the current thread. The advice for a ϕ function returns this value.

```
public aspect TestCase1 {
    // The args() clause lets advice use values of the arguments.
    // Pointcut for the $\pi$ function.
    pointcut piFunc(int a, int b) :
        (call private int $\pi$(int, int) && args(a, b));

    // Pointcuts for setting the field $a_2$ and $a_3$.
    // Since $a_1$ is not visible in a $\pi$ function, don't capture it.
    pointcut set_$a_2$(int n) :
        set(protected int Example.$a_2$) && args(n);
    pointcut set_$a_3$(int n) :
        set(protected int Example.$a_3$) && args(n);

    // Advice that wraps around execution of $\pi$ function.
    // The body of that function is replaced by this code.
    // Return the selected argument after it has been written.
    int around(int $a_2$, int $a_3$) : piFunc(int, int) && args($a_2$, $a_3$) {
            ItemLatch choice = latch_$a_2$;
            synchronized(choice) {
                while(!choice.ready) {
                    choice.wait();
                }
            }
            return(choice.value);
    }

    // Trip the latch when the field is set.
    after(int n) : set_$a_2$(n) {
        synchronized(latch_$a_2$) {
            latch_$a_2$.ready = true;
            latch_$a_2$.value = n;
            latch_$a_2$.notifyAll();
        }
    }
    after(int n) : set_$a_3$(n) {
        synchronized(latch_$a_3$) {
            latch_$a_3$.ready = true;
            latch_$a_3$.value = n;
            latch_$a_3$.notifyAll();
        }
    }
    ItemLatch latch_$a_2$ = new ItemLatch();
    ItemLatch latch_$a_3$ = new ItemLatch();
}
```

(a) Aspect that selects a_2 for π function.

```
public class ItemLatch {
    public ItemLatch() {
        ready = false;
    }
    public boolean ready;
    public int value;
}
```

(b) ItemLatch helper class.

Fig. 4. Aspect and helper code for a test case in Figure 2(b).

6 Current Prototype

Our goal is to produce a Java version of this technique, including tool support for producing and running test cases selected by the user. However, to demonstrate the idea, we are currently prototyping the idea by manually translating the explicitly-parallel superset of C supported by Odyssey into Java, and using AspectJ to write aspects.

Currently, we start with an Odyssey program using the cobegin/coend parallel construct. The compiler produces the CSSAME form for the program, which looks like the code in Figure 2(b). Once we have this code, it is translated into its Java equivalent.

Translating Odyssey programs to Java is straightforward. First, we create a class for the complete Odyssey program. For the cobegin/coend construct, each thread body in the construct is translated to a different inner class implementing the `Runnable` interface, making it a valid thread body. Finally, all shared variables in the construct are converted to instance variables, since local variables are not shared in Java threads. By using inner classes for the thread bodies, the threads are able to share the instance variables without the need for accessor methods or public instance variables. Locks become synchronized blocks. Finally, we add dummy methods for the π and ϕ functions.

This translation is currently done by hand. Later, we hope to have an implementation of the CSSAME form for Java, possibly using the SSA support in Shimple [27], part of the Soot framework for Java [28].

Once we have translated Odyssey code into Java, we construct the aspect to control the execution of the program. At this time, this aspect is also written manually. However, the aspect code is straightforward to write. Individual test cases are selected as explained in Section 5, by manually changing the aspect. In the future, we expect to build tool support to generate the aspects for selected test cases.

7 Examples of Alternative Futures

In this section, we examine several of the example programs based on examples from [1]. The objective is to show how to use the π functions to highlight errors in concurrent code. In these cases, these errors can be located by examining the terms in the π function and noting race conditions. Using our aspect-oriented approach, we can also execute the applications to produce the different answers.

7.1 Example 1

Our first example, in Figure 5, is based on Figure 7 from [1]. In the original, a short thread sets a connection variable to null. A second thread starts by sleeping then processing a long method call before setting the connection variable to some non-null value. A third thread waits for the second to complete and then uses connection, relying on a non-null value. We simulate this example using integer arithmetic, where the first thread sets a shared integer to zero, a second does some work before setting the variable to some non-zero value, and the third performs division using the shared variable. The waiting is performed with a barrier that only synchronizes the last two threads.

```
int $n$ = 1
cobegin
    $T_1$:  Lock(L)
            $n$ = 0
            Unlock(L)
    $T_2$:  Lock(L)
            sleep(10)
            $n$ = 1
            Unlock(L)
            barrier(2)
    $T_3$:  int result
            barrier(2)
            Lock(L)
            result = 21/$n$
            print(result)
            Unlock(L)
coend
```

(a) Odyssey code.

```
int $n_0$ = 1
cobegin
    $T_1$:  Lock(L)
            $n_1$ = 0
            Unlock(L)
    $T_2$:  Lock(L)
            sleep(10)
            $n_2$ = 3
            Unlock(L)
            barrier(2)
    $T_3$:  int result
            barrier(2)
            Lock(L)
            $n_3$ = $\pi(n_1, n_2)$
            result = 21/$n_3$
            print(result)
            Unlock(L)
coend
```

(b) CSSAME form.

Fig. 5. An example based on Figure 7 from [1].

When a Java version of this code is run, the value of n seen by T_3 is the write by T_2, so the output is 7. The barrier, combined with the long execution time of T_2 compared to T_1, hides the race condition at runtime. However, the π function in T_3 exposes this condition, noting that the value of n in the division can be the zero value set in T_1.

In [1], the race condition is considered to be the error. In that work, the authors can select either n_1 or n_2 in T_3 with a 0.50 probability, assuming both writes have run. In most cases, the error will present itself in two executions. However, this probability decreases as the number of concurrent writes increases; for w writes in different threads, the probability decreases to $\frac{1}{w}$. This will likely require more than w runs. In contrast, we need exactly w executions to capture all possible orders for such a race.

Furthermore, the assumption here is that n_1 is not a legitimate reaching definition for n_3 in T_3. Another possibility is that this race condition is not an error, but assigning the value of zero (which causes the division by zero error) is the mistake. In this example, the consequences of this mistake are obvious and happen immediately. However, more subtle mistakes may require the user to rerun this particular error case many times to locate and correct the problem. Our aspect can deterministically return n_1 in the π function, allowing us to rerun this test as many times as needed to fix the error.

However, a small change in this example makes it difficult for value substitution to produce an incorrect execution. Value substitution allows a read operation to take a value from any already-executed write operation that is visible [1]. In this example, all writes to n generally happen before its use in T_3. If we instead move the sleep from T_2 to T_1, the race condition is still unlikely to occur yet value substitution is also unlikely to produce a read of n with a value of zero. Alternative replay is one way to solve this problem [2]. In our approach, the CSSAME form may be enough to spot the problem,

```
sum = $n0$ = $n1$ = $n2$ = $n3$ = 0
cobegin
    $T_1$:  $n0$ += 1
    $T_2$:  $n1$ += 1
    $T_3$:  $n2$ += 1
    $T_4$:  $n3$ += 1
    $T_5$:  sleep(10)
            sum = $n0$ + $n1$ + $n2$ + $n3$
            print(sum);
coend
```

(a) Odyssey code.

```
sum = $n0_1$ = $n1_1$ = $n2_1$ = $n3_1$ = 0
cobegin
    $T_1$:  $n0_2$ += 1
    $T_2$:  $n1_2$ += 1
    $T_3$:  $n2_2$ += 1
    $T_4$:  $n3_2$ += 1
    $T_5$:  sleep(10)
            $n0_3$ = $\pi_0(n0_1, n0_2)$
            $n1_3$ = $\pi_1(n1_1, n1_2)$
            $n2_3$ = $\pi_2(n2_1, n2_2)$
            $n3_3$ = $\pi_3(n3_1, n3_2)$
            sum = $n0_3$ + $n1_3$ + $n2_3$ + $n3_3$
            print(sum);
coend
```

(b) CSSAME form.

Fig. 6. An example based on Figure 9 from [1].

by noting that n_1 is a term in the π function. If this term is expected, then we can use an aspect to test the cases for both n_1 and n_2 deterministically rather than probabilistically.

It is worth noting that the current version of Odyssey does not correctly handle barrier synchronization. In this example, it is clear that T_3 should not be able to see the initial value of n set before the cobegin statement, as it must execute after T_2. However, Odyssey adds n_0 as another term in the π function. When we implement the CSSAME form for Java, we will augment its set of supported synchronization mechanisms to include barriers and event synchronization.

7.2 Example 2

The second example is shown in Figure 6, which is based on Figure 9 from [1]. In the original, an integer array of length 100 is created, with all elements initialized to 0. The main thread launches 100 other threads, where thread i increments the value at index i. After launching the threads, the main thread sleeps for some time and then sums up the elements in the array, without joining with the threads. Our example simplifies this example by using four separate variables and uses a fifth thread to perform the summation. We cannot use the main thread here because of the semantics of the cobegin/coend construct; when the coend statement is reached, all of the threads have completed execution, which removes the concurrency error that we wish to introduce.

The intended result of this code is a sum of 4. With a large enough sleep value, this is the obtained result because threads T_1 through T_4 finish before T_5 reads the values. In fact, any non-zero sleep suffices. If we remove the sleep, then the order in which threads are launched plays the largest role in determining the outcome. Because the threads are so short, they generally run to completion once launched. Without the sleep, T_5 usually sees the updates from all threads launched before it. If launched in program order, T_5 almost always sees the updates from all earlier threads. In a simple test running the program 1000 times, the program reported a sum of 3 only 5 times.

However, from the π functions in Figure 6(b), the race condition in the code becomes clear. Each read of the four summed variables in T_5 may obtain either the incremented value from one of the other threads or the initial value of 0. From this, we can see that the sum may be any value between 0 and 4. That the sum returns 4 is a function of the thread scheduler and not because the program has correct synchronization.

Note that this race condition is not solved by adding mutual exclusion to the program to protect the accesses to the four variables. The race condition results because T_5 can run at any time with respect to the other threads; the solution is to add an event variable or a barrier to control T_5 and ensure it runs after T_1 through T_4.

The above argument assumes that the intended result is 4, in which case static analysis and user observation is sufficient to detect the problem. However, it may be the case that the intent of the code is that the sum must be between 0 and 4, where any value is acceptable. In this case, one potential mistake is that a variable is decremented in one of threads T_1 through T_4. This error can be detected by controlled execution of the program. We can create an aspect that returns the initial value for each of the summed variables except one, where we use the value set by the updating thread. If any thread mistakenly decrements the variable, the sum will be negative. We can test this by running the program only four times, once for each of the setting threads, by returning the appropriate value for each π function in our aspects.

In the most general case, we may need to know what the set of possible values for sum to verify that the program works correctly regardless of the computed value. In this case, there are 16 possible combinations of the four summed values, which we can enumerate over with 16 executions by again controlling the returned values for each π function.

8 Issues and Future Research

Inconsistent Futures Our approach does have a significant drawback in that it is possible to select an inconsistent execution of a program. Figure 7 shows an example of this problem. In the second mutex region in thread T_2, a and b must have the same value, either 1 or 2. As a result, the print statement can only produce the output “11” or “20”, based on the control flow statements in the program.

The problem is that the π functions do not capture the dependency between a and b. Instead, the π functions make it appear that the choice between a_1 and a_2 in π_a is independent of the choice between b_1 and b_2 in π_b. A user who is trying to control the execution of this program could select a_1 in the former case and b_2 in the latter. In this particular example, b_2 is always assigned, so it is possible to construct an aspect that results in the output “12” which is invalid for this program.

Making this case more difficult is that this particular execution is invalid because of the synchronization code. Different synchronization code (say, where the threads obtain different locks to update a and b) or a lack of synchronization code with a sequentially consistent memory could legitimately produce this result.

This case is difficult to detect because both b_1 and b_2 are assigned in this case. The problem is easier to detect when the user-selected return value for a π function is not assigned because of other control flow statements. For example, if b_2 was not assigned

```
cobegin
    $T_1$:  Lock($L$)
            $a$ = 1
            $b$ = 1
            Unlock($L$)
    $T_2$:  int $z$ = 0
            Lock($L$)
            $a$ = 2
            $b$ = 2
            Unlock($L$)
            Lock($L$)
            if ($a$ == 1)
                $z$ = $b$
            print($a$, $z$)
            Unlock($L$)
coend
```

(a) Original code.

```
cobegin
    $T_1$:  Lock($L$)
            $a_1$ = 1
            $b_1$ = 1
            Unlock($L$)
    $T_2$:  int $z_0$ = 0
            Lock($L$)
            $a_2$ = 2
            $b_2$ = 2
            Unlock($L$)
            Lock($L$)
            $a_3$ = $\pi_a(a_1, a_2)$
            if ($a_3$ == 1)
                $b_3$ = $\pi_b(b_1, b_2)$
                $z_1$ = $b_3$
            $z_2$ = $\phi(z_0, z_1)$
            print($a_3$, $z_2$)
            Unlock($L$)
coend
```

(b) CSSAME form.

Fig. 7. A program where a user can select an inconsistent history.

in T_2, the latch for it would never open and T_2 would block in the aspect, trying to return that value in π_b. Eventually all running threads would be blocked, which would indicate that an invalid execution had been selected.

One possible solution is to construct the visibility graph from [1] as the program executes to verify the integrity of the selected test case at run-time. A second option may be to use a *program slice* [26], which can be extended to concurrent programs [18]. A program slice shows the subset of statements in a program that may affect the final value of a particular variable at some specific point in a program. We may be able to use the slice to help detect inconsistent test cases during test case construction.

Other SSA/CSSAME Limitations In addition to the inconsistent futures problem, the SSA form has other limitations (which are inherited by CSSAME and other derivatives of SSA). Some of these have been considered by other research, and others are still open questions. The three problems we will consider are issues with arrays, handling loops, and synchronization analysis.

The SSA form was created to analyze programs with scalars. A simple approach for dealing with arrays is to consider the array as one object. However, the resulting analyses are too coarse-grained to be useful for many programs. Instead, the Array SSA form allows a more precise element-by-element analysis of arrays, where the ϕ functions perform an element-level merge of its arguments [15]. This merge is based on a timestamp associated with each element. A concurrent version of Array SSA that includes π terms for merging writes by different threads is proposed in [7]. These analyses could be added to our compiler to handle arrays.

Another problem with SSA is loops. The *static* part of static single assignment means that each statement that is an assignment uses a separate variable, but it is pos-

```
$a$ = 1
$b$ = 1
cobegin
    $T_1$: Lock($L$)
           while(condition) {
               $b$ = $a$ + $b$
               $a$ = $a$ * 2
           }
           Unlock($L$)

    $T_2$: Lock($L$)
           print($a$)
           print($b$)
           Unlock($L$)
coend
```

(a) Original code.

```
$a_1$ = 1
$b_1$ = 1
cobegin
    $T_1$: Lock($L$)
           while(condition) {
               $a_2$ = $\phi(a_1, a_3)$
               $b_2$ = $\phi(b_1, b_3)$
               $b_3$ = $a_2$ + $b_2$
               $a_3$ = $a_2$ * 2
           }
           Unlock($L$)

    $T_2$: Lock($L$)
           $a_4$ = $\pi(a_1, a_2)$
           print($a_4$)
           $b_4$ = $\pi(b_1, b_2)$
           print($b_4$)
           Unlock($L$)
coend
```

(b) CSSAME form.

Fig. 8. An CSSAME example with a loop, based on an example from [3].

sible for that statement to be executed several times in a loop. An example is shown in Figure 8. At the end of the loop in T_1 in the CSSAME form (Figure 8(b)), the values in a_2 and b_2 represent the current values for a and b that should be used after the loop terminates. (For simplicity, we assume the loop executes at least once in this example.)

If the same lock protects both the loop in T_1 and all other accesses to a and b, as is the case in this example, then this analysis is sufficient even though the loop body may be executed many times. Only the last write of a_2 and b_2 will reach the exit of the critical section. Any earlier writes cannot reach the uses of the two variables in T_2.

However, if the synchronization in T_1 is removed (or if the two writes to a and b are protected individually rather than the complete loop), the CSSAME form is insufficient. The scheduler may interrupt T_1 at any time while it is executing in the loop and run T_2, which will see values for a_2 and b_2 from the current iteration. It is now possible for the values written by any iteration of the loop to be visible in T_2, not just the final ones. We must instead capture each such value (or *write instance*) to enumerate over the set of values that could be printed in T_2. These instances could be captured in an array rather than a single variable, a technique used in *dynamic single assignment* (DSA) [29]. However, the DSA form is applicable to only a small subset of programs, such as multimedia applications [29], suggesting DSA will be inappropriate for this work.

One added difficulty is that there may be dependencies between the write instances that are visible to a thread because of the memory consistency model. For example, in Figure 8(b), if the value for a_2 is taken from iteration i in the loop in T_1, then the value for b_2 must be taken from iteration $i - 1$ or later (since we have assumed a sequentially consistent memory - other consistency models may differ). Event variables or barrier synchronization may introduce similar dependencies.

The final issue we want to address is limitations in analyzing synchronization. This analysis is used to remove terms in π functions where it can be proven that a definition cannot reach a particular use of a shared variable. Where Odyssey cannot prove that a statement is synchronized, it must assume the access is unprotected. This assumption can lead to extraneous terms in pi functions. Since the aspects use these terms to determine the set of interleavings to be tested, it can result in extra test cases being run and may lead to more cases of inconsistent futures.

With the `synchronized` block in Java, such analysis is relatively straightforward. However, with the introduction of `java.util.concurrent` in Java 1.5 (or the use of Lea's `util.concurrent` library [16], on which this new package is built), other synchronization primitives like locks and semaphores have been introduced. With these primitives, it is possible to construct irregular synchronization code that are used in real concurrent program but that can be difficult to analyze. Odyssey includes some novel techniques for identifying these locking patterns based on adding and analyzing reaching definitions for the locks used in the code [21].

Of course, if additional synchronization primitives are added, Odyssey must be extended to recognize the primitives and apply their semantics in its analysis.

Tool Support The goal of this project is to provide tool support for creating and running test cases for concurrent programs. There are several features that a tool should have.

First, we would like to work with the original source code. Although the CSSAME form is useful for a compiler, the transformation may not be as clear to programmers. As a result, we will need to map the statements in the CSSAME form into original source code statements. This is already done by the Odyssey compiler.

Second, tool support should help the user construct test cases, particularly in the presence of control flow statements. Given a write to a shared variable, it may be useful for the user to view the path that must be taken through the code to execute that statement. This will allow the user to specify input to the program that causes the program to run as desired. This problem may be addressed by the program slices discussed earlier. Furthermore, we could automatically generate the aspect code corresponding to each concurrent test. The idea would be to allow users to select the appropriate concurrent reaching definition for a subset of the reads in the program to create the conditions in which they are interested. Ideally, the user can specify these conditions and the program input, and a complete test case can be constructed from this information.

Third, an important consideration is that we clearly cannot require the user to specify the complete program execution by determining the return value for every π function. It is important that the user be able to specify only those terms needed to produce the desired test case. A typical program may well have thousands of these functions, far more than a user can reasonably manage. As well, only data needed to produce the test case should be maintained by the aspect executing the code, to reduce the memory footprint. The CSSAME form introduces multiple copies of a variable, which may require substantial memory. Clearly, we will need a way to reduce the volume of information that must be maintained. This is the topic of ongoing research.

One option is to construct this project as a plugin in the Eclipse IDE for Java [23], and model it after the JUnit framework [12] for constructing unit tests. This would allow a user to construct a complete test suite and run it easily.

Benchmarks Another direction for future work is to use this technique on a variety of concurrent programs. One source of such programs is a concurrent benchmark effort currently underway [10]. This effort also includes the creation of a concurrent testing framework in which different static and dynamic techniques are combined into more complete and powerful tools. This research may find a place within this framework.

Correcting Concurrent Bugs Since a race condition appears as an undesired term in a π function, the CSSAME form can be useful in helping a user determine when the problem has been corrected. After the program has been changed, the terms of the relevant π function can be examined once again to ensure the problematic reaching definition is no longer visible.

9 Conclusions

This paper presented some preliminary work in deterministically executing a concurrent program. The method is based on using aspect-oriented programming to control the execution of the CSSAME intermediate compiler form. The main advantages of this approach are that we can deterministically execute the program covering all potential orders for a given race condition, that we require no execution trace, and that we can execute the program even if accesses to shared variables are not synchronized.

Although this work is preliminary and there are still many issues to be resolved, we believe it holds some promise in the field of testing and debugging concurrent programs.

Acknowledgements

This research was supported by the Natural Science and Engineering Research Council of Canada and the University of Waterloo. We would also like to thank the anonymous referees for their comments and suggestions.

References

1. M. Biberstein, E. Farchi, and S. Ur. Choosing among alternative pasts. In *Proc. 2003 Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2003.
2. M. Biberstein, E. Farchi, and S. Ur. Fidgeting to the point of no return. In *Proc. 2004 Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2004.
3. M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684-1698, 1994.
4. D. Bruening. Systematic testing of multithreaded java programs. Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1999.
5. J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proc. SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.
6. J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *Proc. 2002 International Symposium on Software Testing and Analysis*, pages 210-220, 20023
7. J.-F. Collard. Array SSA for explicitly parallel programs. In *Proc. 5th International Euro-Par Conference*, *LNCS* vol. 1685, pages 383-390. Spring-Verlag, 2005.

8. S. Copty and S. Ur. Multi-threaded testing with AOP is easy, and it finds bugs! In *Proc. 11th International Euro-Par Conference*, *LNCS* vol. 3648, pages 740-749. Springer-Verlag, 2005.
9. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.
10. Y. Eytani, K. Havelund, S. Stoller, and S. Ur. Toward a benchmark for multi-threaded testing tools. *Concurrency and Computation: Practice and Experience*, 2005. To appear.
11. G.-H. Hwang, K.-C. Tai, and T.-L. Huang. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5(4):493-510, 1995.
12. JUnit. *JUnit: Testing Resources for Extreme Programming*. http://www.junit.org.
13. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. Fifteenth European Conference on Object–Oriented Programming*, *LNCS* vol. 2072, pages 327–353. Springer–Verlag, 2001.
14. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object–Oriented Programming*, *LNCS* vol. 1241, pages 220–242. Springer-Verlag, 1997.
15. K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *Proc. 25th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 107-120, 1998.
16. D. Lea. *Overview of package* `util.concurrent` *Release 1.3.4*, 2004. Available at http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html.
17. J. Lee, S. Midkiff, and D. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proc. 10th Workshop on Languages and Compilers for Parallel Computing*, 1997.
18. M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proc. 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 180–190, 2000.
19. D. Novillo. *Analysis and Optimization of Explicitly Parallel Programs*. PhD thesis, Department of Computing Science, University of Alberta, 2000.
20. D. Novillo, R. Unrau, and J. Schaeffer. Concurrent ssa form in the presence of mutual exclusion. In *Proc. 1998 International Conf. on Parallel Programming*, pages 356-364, 1998.
21. D. Novillo, R. Unrau, and J. Schaeffer. Identifying and validating irregular mutual exclusion synchronization in explicitly parallel programs. In *Proc. 6th International Euro-Par Conference*, *LNCS* vol. 1900, pages 389-394. Springer-Verlag, 2000.
22. D. Novillo, R. Unrau, and J. Schaeffer. Optimizing mutual exclusion synchronization in explicitly parallel programs. In *Proc. Fifth Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, pages 128–142, 2000.
23. Object Technology International, Inc. *Eclipse Platform Technical Overview*, 2003. Available at: http://www.eclipse.org/whitepapers/eclipse-overview.pdf.
24. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391-411, 1997.
25. S. Stoller. Testing concurrent java programs using randomized scheduling. *Electronic Notes in Theoretical Computer Science*, 70(4), 2002.
26. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121-189, 1995.
27. N. Umanee. *A Brief Overview of Shimple*, 2003. http://www.sable.mcgill.ca/soot/tutorial.
28. R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Proc. 9th International Conference on Compiler Construction*, pages 18–34, 2000.
29. P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. Advanced copy propagation for arrays. In *Proc. 2003 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 24-33, 2003.