Partial Order Reduction in Symbolic Execution for Object-Oriented Languages

A thesis proposal for extending an object-oriented language with the verification of concurrent programs within reasonable time

Written by
Jesse Vrooman
Utrecht University

Supervised by
Wishnu Prasetya
Gabriele Keller

2024
July

Abstract

The path explosion problem is a significant issue for verification tools that use symbolic execution. Monotonic partial order reduction introduces a sound and complete method of pruning paths irrelevant to the verification process. It does this by finding dependencies within transitions, chaining them, and determining if a computation can be considered quasi-monotonic; if not, the path can be pruned. The algorithm does not clarify possible dependencies between symbolic or concrete references to objects and arrays that must be considered when pruning paths in a concurrent object-oriented language. The contributions made by this research are the extension of the original monotonic partial order reduction method, as well as a performance study of the algorithm when implemented in a symbolic execution verification tool. After a comprehensive, in-depth look at the monotonic partial order reduction algorithm, it was implemented in a verification tool for an intermediate object-oriented language called OOX. When testing the verification tool using many benchmark programs, it is shown that the algorithm remained sound in the pruning of paths and is complete over most scenarios (does not allow for pruning paths using assumptions made over symbolic references). The testing also showed that the algorithm caused a significant increase in performance when compared to not having any form of path reduction, and a simplistic reduction method.

Contents

Chapter 1

Introduction

Malfunctioning software can significantly impact anyone's life these days. Verifying programs is becoming more critical as more fields start adopting software as a primary source of operations. There are many ways of verifying programs. The most common method is testing, automatically or manually, but more intriguing approaches exist. Formal program verification is one method that focuses on verifying programs concerning a given specification or behavior.

The previous definition is still quite vague, and that is by intention since there are also many ways of attempting formal program verification. The most relevant for this paper will be symbolic execution, but explicit state model checking will also be referenced throughout. When using explicit state model checking, the verification tool will look at each state individually to check if it fits the specifications given. Depending on that verification process, it can be determined if the program is valid over all reachable states. When using symbolic execution, all paths of the program are constructed, and then, for each path, a formula is constructed that can be formally proven using a satisfiability solver. Symbolic execution proves a program is correct over sets of states instead of checking individual states.

No task comes without its own sets of problems; for program verification, it comes in the form of computational complexity. Since formal program verification tries to fit a program to a specific specification instead of a singular input and output, many different paths must be explored throughout the programs. Certain programs could have infinitely many states, so verification tools generally are limited to a set depth of states to explore. If the verification tool encounters a violation before reaching the maximum set depth, the tool terminates with the violation; if the maximum depth is reached, the program is considered satisfiable. Since reaching said depth can be very expensive, a preferred method would be to prevent the exploration of states that will not impact the outcome of the verification process.

The computational complexity gets dialed up another notch when trying to prove concurrent programs formally. Concurrent programs consist of multiple threads, and a scheduler is used to schedule the transitions of the threads. Schedulers are non-deterministic, which means that the execution order of a program consisting of multiple threads can not be known beforehand. The threads can access the same heap and read and write the same data. The ability to read and write shared data structure causes the program to behave differently based on how the computations are scheduled. Due to the non-deterministic nature of the scheduler, the program must be verified over all interleaving execution paths possibly executed by the schedular. The need to verify all interleaving contributes to the path explosion problem, a known problem when performing symbolic execution.

Multiple methods exist to deal with the path explosion problem in concurrent programs. One of the

methods that will come to mind is partial order reduction. As previously mentioned, the ability to read and write data from the different threads makes it necessary to check all interleaving for the program. This also means that when operations in the threads do not alter the same section of the heap, evaluating all interleaving paths is unnecessary. Using the independence between operations, it can be determined what paths are unnecessary for the verification process. Pruning the paths that do not necessarily need to be verified within the set of interleaving paths is called partial order reduction.

Many different partial order reduction algorithms have been researched over the years. Some of the noteworthy are the original paper on partial order reduction(*POR*) [Peled, 1993], Peephole partial order reduction(*PPOR*) [Wang et al., 2008], and Monotonic partial order reduction(*MPOR*) [Kahlon et al., 2009]. The paper on *POR* written by Peled introduces the idea of pruning unnecessary paths based on the equivalence between sequences of transitions, although this algorithm was far from optimal. *PPOR* is an implementation that can be used with symbolic execution but is limited to two threads. *MPOR* is a complete and sound algorithm for pruning paths without being limited to a set number of threads. Another benefit is that it can be applied to explicit state model checking and symbolic verification methods.

Within this research, a method is explored to implement the ability to verify concurrent programs while minimizing the path explosion problem using Monotonic Partial Order Reduction (MPOR). This will be implemented in a verification tool for an intermediate object-oriented language called OOX [Koppier, 2020]. OOX has been developed within Utrecht University as an alternative to existing program verification methods. Two separate verification tools were developed within the University, one in a Haskell code base and one in a Rust code base. The Rust-based verification tool will be the target of this research, as it cannot currently handle concurrent programs.

In the original paper on Monotonic Partial Order Reduction, an algorithm is introduced that can prune paths using dependencies between shared variable accesses. The paper only mentions the dependencies between concrete variables, but many variables might not be concrete in object-oriented languages. Objects and arrays can be symbolic and, therefore, cause unexpected conflicts between references when altered. Arrays also have unique behavior since their indexation determines what part of the heap is altered, meaning that dependencies only exist when the alteration happens on the same index.

The primary focus of this research is encapsulated in the following two **research questions**:

- **RQ-1**: *How can the monotonic partial order reduction algorithm be extended to account for dependencies in transitions that involve arrays and objects while preserving its soundness and completeness?*: This question explores how Monotonic Partial Order Reduction can be extended to handle the unique dependencies and interactions present in Object-Oriented Languages. Since the original paper proves that the algorithm is complete and sound, after extending it to handle objects and arrays, the goal is to keep this algorithm complete and sound.

- **RQ-2**: *What is the impact of monotonic partial order reduction on the performance of symbolic execution verification tools for object-oriented languages?*: This question investigates whether Monotonic Partial Order Reduction can be effectively implemented within a symbolic execution engine. Symbolic execution allows for exploring program paths using symbolic values instead of concrete ones, which can significantly aid in verifying concurrent programs by reducing the state space that needs to be explored. It is necessary to research the performances gained and the algorithm's possible overhead when verifying programs to prove the effectiveness of monotonic partial order reduction within OOX.

This research makes the following key **contributions**:

- *Extension of Monotonic Partial Order Reduction*: The MPOR algorithm is extended to handle the specific challenges posed by Object-Oriented Languages. This includes managing dependencies between symbolic references and objects not addressed in the original MPOR paper. These extensions make MPOR applicable to a wider range of programming paradigms, particularly those involving complex object interactions.
- *Implementation in a Symbolic Execution Verification Tool*: The extended MPOR method is implemented within the Rust-based OOX verification tool. This involves integrating MPOR with symbolic execution to efficiently verify concurrent programs written in OOX, thereby enhancing the tool's capabilities and performance.
- *Performance Study*: A comprehensive study is conducted to evaluate the performance of the extended MPOR method within the symbolic execution verification tool. This study assesses the effectiveness of MPOR in reducing the path explosion problem and improving the verification process for concurrent programs. The results provide insights into the practical benefits and limitations of the approach.

By addressing these questions and making these contributions, this research aims to enhance the verification capabilities of the Rust-based OOX tool and provide better insights into the performance and capabilities of monotonic partial order reduction.

Chapter 2

Related Work

2.1 Formal Program Verification Tools

Formal program verification is used to verify that software programs behave correctly and is an alternative to testing. Testing suffers that it is limited to a finite amount of runs for programs with a finite length. Therefore, it is nearly impossible to test all behaviors for a program [van den Bos and Huisman, 2022]. A program is proven to adhere to its formal specification in formal program verification. To construct this formal specification, Hoare logic is often used to reason about the validity of a program. The following notation is a Hoare triple, $\{P\}C\{Q\}$. The notation includes a precondition P and post-condition Q, both in the form of boolean formulas. The precondition P implies that the postcondition Q is valid over a computation C [Hoare, 1978]. Using the basis provided, many different techniques have been used to implement many different verification tools. As example CMBC [Kroening and Tautschnig, 2014], CIVL [Zheng et al., 2015], KLEE [Cadar et al., 2008], VerCors [Blom and Huisman, 2014], DIVINE [Baranová et al., 2017], and many others.

CBMC [Kroening and Tautschnig, 2014] uses a symbolic Bounded model checking technique to verify programs. It does this specifically for C programs, but there exists a Java version that is called JBMC [Cordeiro et al., 2019]. A symbolic program verification method provides the program with symbolic inputs instead of concrete inputs. This means the program is validated over a possible range of inputs instead of a single concrete value for each run. Symbolic bounded model checking encodes the initial states and the transition relations that depict the changes that occur to the state when traversing the program. Combined with a property encoding generally constructed in CBMC by the assertions made, the verification tool builds a formula to determine if the reachable states allow for violating the property encoding. When the property encoding can be violated, the program is considered invalid [Cordeiro et al., 2019]. The soundness of the verification is achieved by incrementing the bounds of the state graph until it finds a witness that concludes the program is considered invalid. Completeness is only possible when the input program is finite. If a program is not finite, the bounds will infinitely be increased to find a possible invalid state. To prevent this behavior, a maximum depth is given that stops the process [Kroening et al., 2023]. Even though a non-finite program does not receive a complete verification, it can verify a program to a given depth, unlike other program verification methods.

Divine [Baranová et al., 2017] is an LLVM-based verification tool designed for verifying C and C++ programs. Divine was originally implemented with a dedicated non-embedded domain-specific language, but

later, a shift was made to verify LLVM-based programs. It uses a parallel form of explicit state model-checking techniques to verify programs. Unlike symbolic verification techniques, Explicit state model checking uses concrete inputs. In explicit model checking, each visited state has to fit a formal specification to determine if a program is valid. It lends well to asynchronous programs since the state graph of interleaving can be easily reduced using partial order algorithms and abstraction [Holzmann, 2018]. Divine is unique since, in general, depth-first algorithms are used to progress through program paths, but Divine works on a parallel algorithm that explores program paths breath-first. A main benefit of this method is the ability to handle cycle detection easily. Divine makes use of multiple algorithms that can detect cycles, when a cycle is found it can helps identifying loops in the system's state space. Recognizing loops and cycles can prevent the program from exploring the same loop indefinitely. This means that Divine can avoid redundant computations and, with that, accelerate the verification process [Baranová et al., 2017].

Klee [Cadar et al., 2008] is a verification tool that uses the Symbolic Execution method for verifying programs. Bounded symbolic execution is similar to symbolic bounded model checking, but one major difference exists. While symbolic bounded model checking concatenates the formulas over all the reachable states, symbolic execution explores paths separately and verifies a specification for the given path, e.g., it could be in the form of assertions or LTL formulas. The main benefit of symbolic execution is the early out possibility when an early explored path is determined invalid. This means that invalid programs can be found faster, but when a program is valid, there is a possible downside of repeatedly calling the SMT/SAT solver for each path. SMT/SAT solvers are notoriously expensive to call, but a front-end simplify-er could significantly decrease the size of the single LTL formula in symbolic model checking.

2.2 Challenges of Symbolic Execution

The paper "A Survey of Symbolic Execution Techniques" by Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi [Baldoni et al., 2018] provides a comprehensive overview of symbolic execution. Symbolic execution is a program analysis technique introduced in the mid-70's. For a verification process to be symbolic execution, it has to satisfy the following properties for each explored control flow path:

1. It tracks a first-order boolean that describes if the current path can be satisfied.
2. A symbolic state is kept that maps variables to values or symbolic expressions.

SMT/SAT solvers are used in symbolic execution to prove that the postcondition is valid for each path given the symbolic state. Generally, SMT/SAT solvers confirm if a specific formula is satisfiable, but by negating the formula, it can be checked if an invalid case can be found. If each outcome is considered valid for every path in the program, then it can be determined that the complete program is valid. There are multiple difficulties when working with symbolic execution. When implementing symbolic execution, an exhaustive search is performed through the program to find all control flow paths. While this theoretically works fine, in practice, we deal with the physical limitations of the hardware used. A full, exhaustive search should be sound and complete, but non-finite programs are not feasible for practical implementation. This means concessions are made when verifying programs, such as limiting the maximum depth that the program can take. With the maximum depth, it is still possible that a path explosion occurs when verifying a program. This can occur due to the program branching over many transitions, causing an explosion of the state graph in breadth. Path explosion is a term used when repeated branching in the program causes the creation of many paths and hurts computational time and memory. Therefore, computational complexity and memory management are critical for creating a usable verification tool that uses symbolic execution.

The paper then describes techniques used to handle practical implementations of symbolic execution. These are the Mixing of Symbolic and concrete execution, Path selection, and Symbolic backward execution. These methods can significantly reduce the program's state space, but they all have their own trade-offs.

Mixing symbolic execution with concrete execution works by running a version of symbolic execution and concrete execution simultaneously. An example is a method where concrete execution drives a symbolic execution. By exploring the state space using concrete inputs driven by a search heuristic while still updating both a concrete and symbolic store, it is possible to verify the symbolic path constraints directly using the concrete execution. This minimizes the need to invoke an SMT/SAT solver for each case and improves the performance. This form of symbolic execution is called Dynamic Symbolic Execution (DSE).

Path selection is a method that prunes paths that likely have a minimal impact on the verification process. The main downside to path selection is the possible loss of relevant paths; even though these methods try to minimize the pruning of important paths, they are not always able to do so. An example target could be maximizing code coverage and, therefore, removing the paths that repeatedly go over the same space in the code. This method can cause a loss of completeness due to not exploring every reachable path.

Backward execution is just an alternative approach to the same problem. This method finds the program's assertions and exceptions and then propagates backward toward input constraints. When a path is not satisfiable, it backtracks and restarts. This method could be beneficial in some cases, but unfortunately, implementation can be difficult.
From these methods, it can be seen that the main target for improving the symbolic execution of a program can be summarized in the following design principles:

1. Progression should be feasible within the physical limitations of the system, the main focus being memory management to keep the required memory below the system's total memory capacity.
2. Same parts of the program should not be executed repeatedly for the same values.
3. The program should reuse information as much as possible to reduce the invocations on the SMT solver.

Each step of symbolic execution still has areas of improvement, so paths can be pruned based on feasibility to handle the state explosion problem or reduce the boolean formulas to lower the usage of the SMT solver. The usage and implementation of symbolic execution within tools to check programs for violations keep growing. Therefore, many new ideas enter the space to improve implementations that handle the challenges of symbolic execution.

2.3 Partial Order Reduction

When dealing with concurrent programs, a scheduler determines in what order the transitions of threads are scheduled. These transitions are generally noted down as t_i where i is a unique id of the transitions, and the thread id of a transition is denoted as $tid(t_i)$. Let's use the notation $t_i < t_j$ if transition t_i is executed before transition t_j. The non-deterministic nature of the scheduler can impact the execution order of the scheduled transitions. Since the execution order of the program is unknown, generally, all interleaving is verified to guarantee that the program is valid over all paths. Exploring all interleaving means exploring all execution orders of transitions (paths). If two threads exist, each with a transition (t_1, t_2), then a path has to be explored where $t_1 < t_2$, and a path where $t_2 < t_1$. The exploration of all interleaving can lead to an unnecessarily huge state graph. It is unnecessary because transitions can be independent of each other. If two transitions are independent, then the ordering of execution does not alter the outcome, leading to the same state. The state graph can be reduced with Partial Order Reduction methods to combat this issue.

In the paper "All from One, One for All: on Model Checking Using Representatives" [Peled, 1993], model-checking techniques are studied that can generate representative interleaving sequences using a reduced state graph. This paper introduces the idea of using partial order reduction to reduce state graphs for program verification. This is done by finding pairs of operations that affect the global states commutatively. With these pairs, a state graph can be constructed with one representative sequence of operations for each pair. This reduces the amount of sequences of operations that have to be evaluated.

This paper introduces an algorithm that has a relation with algorithms introduced in earlier papers. These include the sleep set method for spanning a reduced state graph introduced by Godefroid [Godefroid, 1991] and the algorithm proposed in [Valmari, 1990] that is based on avoiding the expansion of all the enabled directions of a given node. Using these methods from these algorithms, the author introduces a new algorithm that reduces the state graph. This algorithm has two main goals: (1) constructing a reduced state graph with one representative sequence for all equivalence classes; therefore, the results when using the reduced state graph or the full state graph are the same. (2) existing algorithms that use fairness assumptions efficiently can be directly used on the reduced state graph to check that a property holds under a fairness assumption and that the state graph can be further reduced when using fairness during state graph expansion.

The algorithm benefits from using fairness assumptions to reduce the state graph further because it does not recognize that a program takes an unfair sequence where an operation is left out. When checking for all unfair sequences, the algorithm can reduce the state graph further. A downside to this is that calculating enough sets that can be used to generate minimal-sized state graphs, with representatives for each equivalence class, is NP-hard. This leads to the trade-off between better predictions and computational time.

The algorithm uses static analysis, allowing for reducing state graphs in explicit state model checking, but it does not lend well to symbolic execution. When implementing the algorithm, Peled shows a clear reduction in the nodes and edges between the full state graph and the reduced state graph, thereby decreasing computation time.

2.4 Peephole Partial Order Reduction

In the paper "Peephole Partial Order Reduction" [Wang et al., 2008], Wang, Yang, Kaglon, and Gupta propose a new method for reducing state graphs that works with symbolic execution. Previously, partial order reduction methods have been applied to explicit state model checking by exploiting equivalence relations between sequences of interleaving transitions. The methods before [Peled, 1993, Godefroid, 1996, Valmari, 1991] relied on conservative static analysis, however applying these methods to symbolic execution is hard. Symbolic execution evaluates a larger set of states that have been combined, and reducing states that are equivalent for this form of verification is hard compared to individual states.

Something that previous methods failed to do is to set conditions where specific sequences become equivalent, while this could actually be used to reduce state graphs. Missing out on these partial order reductions can be costly. To combat this issue a new method called *Peephole partial order reduction* is proposed, to exploit the dynamic independence of transitions. A new relation is introduced called the guarded independence relation (GIR), which takes the form $\langle t_1, t_2, c_G \rangle$ where the transitions t_1 and t_2 are independent in a state S when condition c_G holds on S. The benefit of this type of relation is that they can easily be constructed with a single traversal of the program.

After defining the properties of an independence relation and a guarded independence relation, multiple scenarios can be presented that would require the use of the guarded independence relation. These scenarios are within a C-like program.

- When two transitions (t_1, t_2) do not contain heap accesses to the same variable, the GIR is $\langle t_1, t_2, true \rangle$.

- When two transitions (t_1, t_2) both contain an array access with $\{a[i], a[j]\}$, the GIR would be $\langle t_1, t_2, i \neq j \rangle$.
- When two transitions (t_1, t_2) contain a reference access with $\{*p_i, *p_j\}$, the GIR would be $\langle t_1, t_2, p_i \neq p_j \rangle$.
- When two transitions (t_1, t_2) contain a global variable with $\{x\}$, the GIR would depend on the following cases.
  - **RD-WR**, where one transition reads and the other writes, with the assignment $x := e$ the GIR would be $\langle t_1, t_2, x = e \rangle$.
  - **WR-WR**, where both transitions write to the variable, with the statements $x := e_1$ and $x := e_2$, the GIR would be $\langle t_1, t_2, e_1 = e_2 \rangle$.
  - **WR-C**, where one transition writes, and the other uses the variable in a condition *cond*. The difference between RD-WR and WR-C is that the assignment does not necessarily change the outcome of the condition. Therefore the GIR for the assignment $x := e$ would be $\langle t_1, t_2, cond = cond[x \rightarrow e] \rangle$

The proposed algorithm works as follows: the scheduler originally consists of all possible interleaving transitions of the threads. If multiple sequences fall within the same equivalence class, then ideally, only one has to be checked for a violation of properties. With the constructed GIR relations, the program can be constrained from exploring paths that would be equivalent. This is done in the order of the thread ids, where the smaller id would be the preferred execution. If two transitions are independent, the transition is explored with the lowest thread id; both transitions will be explored when the transitions are dependent. This method guarantees that all the removed interleaving are redundant, meaning that they wouldn't have altered the outcome of the verification. When a program consists of only two threads, the algorithm guarantees that all the redundant interleaving paths are removed.

GIR constraints have an overhead that can be significant for SMT/SAT solvers; therefore, reducing the overhead would be beneficial. There are some methods to lower the overhead, (1) is to merge GIR constraints, when a transition t_1 is guarded independent to all transitions t_2, then it is not necessary to have separate constraints for all pairs of $\langle t_1, t_2 \rangle$. (2) if two threads are completely independent, the threads do not have to be interleaved at all, allowing for the full execution to be in order of thread id.

Concluding the paper, the authors mention that the *Peephole partial Order Reduction* method leads to significantly more reduction than existing methods at the time and that the method is well suited for SMT/SAT solvers. However, the method is unsuited for systems with more than two concurrent threads.

2.5 Separation Logic

Separation logic [Reynolds, 2002] is an extension of Hoare logic [Hoare, 1978], designed to provide reasoning over programs that manipulate pointer data structures. Separation logic makes clear distinctions between parts of the heap and introduces new operations to the existing syntax of Hoare logic. To define the allocation of cells in the heap, the notations $x \mapsto 1$ and $x \mapsto 1, 2$ are added. In the first case, x is a pointer referencing a cell with the value 1, whereas in the second case, x is a pointer that points to a cell with the value 1 with a neighboring cell with the value 2. By defining neighboring cells, the notation allows using pointer arithmetic where $[x + 1]$ retrieves the neighboring cell of x. This allows for defining cells, but in the case of $x \mapsto 1 \wedge y \mapsto 1$ it can not be guaranteed that x and y do not point at the same cell. This is where separation logic introduces the $P_1 * P_2$ notation to show a clear separation between P_1 and P_2, indicating that both parts of the condition point to different parts of the heap. In the case of $x \mapsto 1 * y \mapsto 1$, it can be guaranteed that x and y point to different cells in the heap. The separation implication operator enables the user to define

that a condition holds after extending the heap with a separated disjoint part that satisfies a condition. This means in the case of $P -\!\!* Q$ if the heap is extended with the disjoint part where P holds, then Q holds in the modified heap after the operation. As an example, $\{x \mapsto 9 * ((x \mapsto 16) -\!\!* p)\}\ [x] := 16\ \{p\}$ can be proven as correct. Separation Logic can inspire ideas that could be transferred over to other verification techniques but are generally implemented in deductive program verification systems. Deductive systems split up parts of a program and require assertions of properties over these sections. This is where separation logic can be used to reason about pointers by determining ownership of the references within these assertions.

The heap is often shared between the different threads when the program allows for the execution of concurrent programs. Reasoning about the heap could allow more proper reasoning about concurrency. Hearn developed an extension called Concurrent Separation Logic [O'Hearn, 2007] that can be used for reasoning independence between threads. The following proof rule is considered the key to this method:

$$\frac{\{P_1\}\ C_1\ \{Q_1\}\ \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1\ ||\ C_2\ \{Q_1 * Q_2\}}$$

The rule states that for a parallel composition to be proven, each process must be given a separate piece of state, and the postconditions for each process must be combined. The goal of the paper is to simplify logical notation for concurrent proofs. In the case of a merge sort algorithm, the array is separated in the middle, and either half of the array is merge sorted. This can be a clear case of an algorithm that can be executed in parallel without conflicts due to its ability to adjust separate parts of the heap.

$$\frac{\{array(a,i,m)\}\ ms(a,i,m)\ \{sorted(a,i,m)\} \quad \{array(a,m+1,j)\}\ ms(a,m+1,j)\ \{sorted(a,m+1,j)\}}{\{array(a,i,m) * array(a,m+1,j)\}\ ms(a,i,m)\ ||\ ms(a,m+1,j)\ \{sorted(a,i,m) * sorted(a,m+1,j)\}}$$

In the example proof above, it can be seen that if we prove that the left side of the array ($array(a,i,m)$) results in a sorted version of the sub-array ($sorted(a,i,m)$), after executing mergesort over that sub-array ($ms(a,i,m)$), and similarly for the right side of the array; it can be proven that if the two sub-arrays are separated in a heap, and the merge sort of the sub-arrays are executed in parallel, that it results in the sorted sub-arrays in separated sections of the heap. Writing these kinds of proofs allows for better reasoning over concurrent programs while being trivial. Although the parallel composition rule mostly captures the ease of proving independently operating programs, it struggles to write proof rules over non-disjoint concurrent programs. This is where another proof rule is introduced for critical regions.

$$\frac{\{(P * RI_r) \wedge B\} C \{Q_1 * RI_r\}}{\{P\}\ with\ r\ when\ B\ do\ C\ \{Q\}}$$

This rule requires a given "resource invariant" RI_r, for each resource r present in the program. This allows for reasoning over programs that are not disjoined; however, it's important to note that the logic is unsound unless resource invariants are precise, i.e., unambiguously carve out an area of the heap [Gotsman et al., 2011]. This precision is crucial to ensure the soundness of the logic when dealing with concurrent threads manipulating shared memory. The additions of the parallel composition and critical region rule proofs can be written over concurrent programs using separation logic.

Chapter 3

Concurrent programs in the OOX verification tool

OOX was previously defined with concurrency in mind by Stephan Koppier in the thesis "The Path Explosion Problem in Symbolic Execution: An Approach to the Effects of Concurrency and Aliasing" [Koppier, 2020]. OOX was designed with other object-oriented languages in mind, like Java and C#, so that programs written in those languages could be easily represented in OOX. This excluded the inheritance and polymorphism features since it was not within the project's scope. When the Rust implementation of the OOX verification tool was implemented, it was chosen to focus on other aspects of the code, for example, inheritance and exceptions. With the necessity of verifying concurrent programs, we opted to introduce concurrency into the Rust base OOX verification tool.

3.1 Statements

Multiple statements have to be added to allow verification of concurrent programs. Most of the statements introduced are similar to the implementation of Koppier [Koppier, 2020], but there are some critical differences in locking and unlocking. The introduced statements are fork, lock, unlock, and join.

$$S \in Statement ::= Statement \mid fork\ I \mid lock\ ref \mid unlock\ ref \mid join$$

3.1.1 Fork

The fork is used to spawn a new thread. The syntax $fork\ I$ shows that an invocation I is given to the fork statement; the invocation is seen as the entry point for the thread. Any program state consists of a set of threads. A fork spawns a new thread using the given invocation. When this happens, the invocation parameters are pushed onto the stack. Besides variables, it also allows references to the heap to be passed into threads, making it a shared variable. When the thread is spawned, a unique tid (shorthand for thread id) is generated to allow distinction between the different threads/processes.

The invocation has to satisfy a set of properties to allow it to be forked:

- The invocation has to be a method or static method; constructors cannot be forked.

- The invocation can only have the return type of void. A fork can never return a value, so allowing it could confuse the user.
- During symbolic execution, invocations are only considered if the invocation target is non-symbolic. An invocation target is symbolic when called from a symbolic reference to an object. The symbolic nature of the reference could cause major branching over its aliases. While this is possible to implement, it has not been attempted as part of this research.

In the OOX example given below, the fork statement is used to spawn a thread. The fork in line 10 adheres to the before-given rules. The invocation given is a method call with a return type of void, and the method is not invoked from a symbolic reference to an object. The assertion in line 12 has to consider that the spawned thread may not execute before the assertion due to the non-deterministic nature of threads.

```
1: class Example {
2:     int val;
3:     Example() {}
4:     void f() {
5:         this.val := 5;
6:     }
7:     static void main() {
8:         Example x := new Example();
9:         x.val := 0;
10:        fork x.f();
11:        int out := x.val;
12:        assert(out == 0 || out == 5);
13:    }
14: }
```

3.1.2 Lock and Unlock

Locking and unlocking is performed on a reference to the heap. When an object is created, the reference of the object can be locked or unlocked. When another thread already locks a reference, the current thread cannot lock on that same reference. The thread is considered disabled until the reference is unlocked using an unlock statement.

In the original specifications of OOX created by Koppier [Koppier, 2020], the lock is performed on a reference and a code block. As long as the code block is not finished executing, the reference remains locked, and no other thread can lock the same reference. Within the rust implementation, we chose a separate lock and unlock statement. In many scenarios, the original version would suffice, but the separation of the statement comes with some utility. For example, a thread can now unlock a reference that was locked within another thread (this requires proper use to ensure a deadlock does not occur). This is done to keep OOX more in line with other object-oriented languages that also allow separate locking and unlocking actions.

In the OOX example below, locking and unlocking statements allow the user to guarantee a specific control flow. The assertion does not have to consider that the spawned thread might execute after the assertion since the lock statement at line 13 prevents continuing the main thread until the child thread unlocks the reference. This type of guaranteeing a specific control flow would not be available when the lock and unlock statements are fused together. Note that the locking of object x does not prevent the ability to execute the forking of x or its ability to continue its spawned thread.

```
1: class Example {
2:     int val;
3:     Example() {}
4:     void f() {
5:         this.val := 5;
6:         unlock this;
7:     }
8:     static void main() {
9:         Example x := new Example();
10:        x.val := 0;
11:        lock x;
12:        fork x.f();
13:        lock x;
14:        int out := x.val;
15:        assert(out == 5);
16:    }
17: }
```

3.1.3 Join

The *join* statement can be used to ensure that all child processes are finished executing before continuing. When the join statement is called, the thread becomes disabled. When a child of the joining thread finishes, it will check if all child processes of the thread are finished. If not, the thread will stay disabled. The thread state will update to enabled if all child processes are finished.

In the previous example, locks are used to guarantee the control flow by locking and unlocking references within threads. In some instances, like the example, the *join* statement can simplify forcing a specific control flow. The *join* statement at line 11 enforces that all child processes are finished. This means that the spawned thread will always execute before the join statement and assertion.

```
1: class Example {
2:     int val;
3:     Example() {}
4:     void f() {
5:         this.val := 5;
6:     }
7:     static void main() {
8:         Example x := new Example();
9:         x.val := 0;
10:        fork x.f();
11:        join;
12:        int out := x.val;
13:        assert(out == 5);
14:    }
15: }
```

3.2 Threads and Scheduling

A program always consists of at least one thread with the thread id 0. Generally, this is called the main thread. The program can spawn in more processes using the fork statement. All threads are by default children of the main thread; this is the case because if a thread $T1$ spawns a thread $T2$, and $T2$ spawns a thread $T3$, then $T3$ is still considered a child of $T1$. The program can be synchronized using the locking and join statements, enforcing specific control flows as shown in the examples above. The main challenge of verifying concurrent programs comes from the non-deterministic schedule. The schedule determines the order in which the transitions should be scheduled. This means that the control flow is not guaranteed in some instances when multiple threads exist in a program. When formally proving the correctness of a program, all possible execution orders must be verified. This means all interleaving of the threads has to be checked.

Case 1 Program with two independent threads

1: **procedure** T1	1: **procedure** T2
2: t_1: x := 1;	2: t_2: y := x;
3: **end procedure**	3: **end procedure**

In Case 1, two threads exist, both with a single transition t_1 and t_2. Since the scheduler is non-deterministic when scheduling these transitions, to verify this program, two paths have to be explored, where t_1 is executed before t_2 ($t_1 < t_2$), and where t_2 is executed before t_1 ($t_2 < t_1$). This means that threads will force branching to verify all possible control flows when the number of transitions or threads, the state graph, will scale exponentially by having to explore all permutations of the transitions.

3.2.1 Deadlocks and Exceptions

A deadlock occurs when no thread can continue because locking methods block them. Note that if the main thread is finished, it is not considered deadlock. Deadlocks should not happen in correctly written programs; therefore, this is not considered strictly valid or invalid. The verification process is terminated and returned with a deadlock error message to indicate that verifying the program was not possible. The Rust-based implementation of OOX offers the functionality of exception handling. This allows a program to throw exceptions and catch these exceptions. When an exception is not caught in the main thread, the execution path that found this exception is ended, and an exceptional clause is checked. The exceptional clause is used when the user wants to guarantee specific program behavior when an exception is not caught. When a spawned thread throws an exception without catching it, it also terminates the path and checks the exceptional clause provided at the start of the main thread. A decision had to be made on whether the thread should terminate and that the execution path should continue throughout the other threads, or that the entire execution path should be terminated. The behavior of other languages can vary based on the specifics of the language, runtime environment, and how the program is structured. Due to the undefined nature of this specific case, it was chosen to go for the more simplistic implementation that terminates the whole execution path. This means that the further execution of the program is left unverified.

3.3 A complete and sound verification

The implementation of verifying concurrent programs must be complete and sound. This means that if the previously defined concurrent statements are implemented and the program correctly handles threads, then any concurrent program using these statements should verify with the correct output. The output of the verification tool is *valid* or *invalid*. The following definitions determine whether the verification tool is sound and complete.

Definition 1 (Sound verification of concurrent programs)**.** *The verification process of a program is considered sound when it always considers a valid program valid, but it may classify invalid programs as valid. This means it allows for false positives but never false negatives.*

Definition 2 (Complete verification of concurrent programs)**.** *The verification process of a program is considered complete when it always considers a invalid program invalid, but it may classify valid programs as invalid. This means it allows for false negatives but never false positives.*

When the verification process is considered complete and sound, the tool always returns the right output for every program. Classifying all *valid* programs as *valid*, and all *invalid* programs as *invalid*.

3.4 Implementation

A simple algorithm can be implemented to explore all interleaving of a concurrent program. The algorithm is a recursive program that explores until the maximum depth k is reached or until there aren't any available states to progress. Let S be a state of the program. S contains the list of threads $\mathcal{T}$, the *threadCounter* that generates unique thread ids, the active thread id *atid*, and the heap H. Let CFG be the control flow graph that contains nodes (N) and edges (E). N_{pc} returns the primitive statement for program counter *pc*. E_{pc} returns all transitions to other nodes as a list of program counters. In $\mathcal{T}$, let the individual thread be accessed with $\mathcal{T}_{tid}$. A thread contains its unique id *tid*, the program counter *pc*, its parents as a vector of thread ids, and its local stack.

The algorithm that explores all interleaving can be seen in algorithm 1. The algorithm loops over all the remaining states, first it executes the transitions (the primitive statements that occur at the transition is executed) in the active thread. The next step is to find all transitions from the current node in the edges E. For each found transition, a new state is cloned from the current state, and then the active thread and the pc of the active thread are updated. Over the iterations of the program finished states are collected, these are states that have reached a point where there were not possible transitions to be taken even if there are not any disabled threads in the program.
In line 8, it is checked if there is at least a single enabled thread. If this is not the case, this is now considered a Deadlock, and the whole verification process gets terminated. A program that possibly leads to a deadlock is considered faulty in all scenarios.
The algorithm given already shows the explosive nature of this method very well since a new state is generated for each possible transition that could be taken from any of the threads in the program at the current time step. This requires a significant amount of computational complexity and memory.

Algorithm 1 Exploration of all interleaving

```
1: procedure ExecuteStates(remaining_states, k, (N, E) ∈ CFG)
2:     if k ≤ 0 then
3:         return remaining_states
4:     end if
5:     resulting_states ← ∅
6:     finished_states ← ∅
7:     for S in remaining_states do
8:         if not ∃_T ∈ 𝒯 : T.enabled then
9:             throw DEADLOCK
10:        end if
11:        S ← ExecuteTransition(S)          ▷ Current transition is executed, and S is updated
12:        continues ← false
13:        for T in S.𝒯 do
14:            if T.enabled then
15:                from ← T.tid
16:                for to in CFG_from do
17:                    next ← clone S
18:                    next.atid ← from
19:                    next.(𝒯_tid).pc ← to
20:                    push next onto resulting_states
21:                    continues ← true
22:                end for
23:            end if
24:        end for
25:        if not continues then
26:            push S onto finished_states
27:        end if
28:    end for
29:    if |resulting_states| > 0 then
30:        append ExecuteStates(resulting_states, k − 1, CFG) onto finished_states
31:    end if
32:    return finished_states
33: end procedure
```

The current transition is executed in line 11 of the algorithm. The addition of the statements fork, lock, unlock, and join requires a change in the *ExecuteTransition* method.

In algorithm 2, the implementation can be found for the *ExecFork* method. This method is executed by the *ExecuteTransition* method in algorithm 1 when the transition found contains a fork statement. The method requires the current state of the program, as well as the invocation given as part of the fork statement. The entry point is initially found and verified using the *GetEntryPoint* method. This entry point can only have a return type of void; otherwise, the program recognizes a typing error at the point of the fork. A new thread is then initialized and given a thread id based on the *threadCounter* within the state. This is done to make sure each thread has a unique id. The processing of the invocation ensures that the invocation arguments are correctly stored within the stack of the new thread and that the program

Algorithm 2 Executing a fork primitive statement

```
1: procedure ExecFork(S, invocation)
2:     entry ← GetEntryPoint(invocation)
3:     if not entry then
4:         throw InvalidEntryPoint
5:     end if
6:     thread ← new Thread()
7:     thread.tid ← S.threadCounter
8:     push thread.tid onto S.T_atid.children
9:     S.threadCounter ← S.threadCounter + 1
10:    ExecInvocation(thread, entry)
11:    push thread onto S.threads
12: end procedure
```

counter is set to the invocations entry point.

Within the earlier definition of locking and unlocking statements, a specific reference is locked. So, generally, an object is created and then passed into multiple threads to make it a shared reference. In the case of two threads, T1 and T2, when the thread T1 locks this object, it acquires the lock. When T2 attempts to lock on this object, it is refused until the reference is unlocked.

Algorithm 3 Executing a lock primitive statement

```
1: procedure ExecLock(S, T, ref)
2:     if ref ∈ S.L then
3:         push T.tid onto S.L_ref
4:         T.enabled ← false
5:     else
6:         S.L_ref ← ∅
7:     end if
8: end procedure
```

In algorithm 3, the *ExecLock* method is introduced and is executed when the *ExecuteTransition* method finds a lock statement. Locks are saved inside a map located at $S.\mathcal{L}$. This structure maps references to an array of thread ids. When ref does not exist within $S.\mathcal{L}$, then there is no lock on the current reference. When the reference does exist, it maps to a set of thread id's. These thread ids give the threads currently disabled due to them having attempted to lock the reference.

Algorithm 4 Executing an unlock primitive statement

1: **procedure** ExecUnlock(S, T, ref)
2: $tids \leftarrow S.\mathcal{L}_{ref}$
3: **for** tid **in** $tids$ **do**
4: $S.T_{tid}.enabled \leftarrow true$
5: **end for**
6: **remove** ref **from** $S.\mathcal{L}$
7: **end procedure**

In algorithm 4, the *ExecUnlock* method is introduced, which is executed when the $ExecuteTransition$ method finds an unlock statement. When an unlock is found on a reference, all disabled threads that have attempted to lock this reference are enabled. This causes the locking and unlocking of statements to be handled non-deterministic. Then, the reference is entirely removed from the locks map to ensure a new lock can be acquired.

Algorithm 5 Executing a join primitive statement

1: **procedure** ExecJoin(S, T, ref)
2: $T.enabled \leftarrow$ **not** $\exists_{tid} \in T.children$: **not** $S.T_{tid}.finished$
3: $T.joining \leftarrow$ **not** $T.enabled$
4: **end procedure**

In algorithm 5, the *ExecJoin* method is introduced, which is executed when the $ExecuteTransition$ method finds a *join* statement. When a join statement is found, it is checked if all children have already finished executing. If this is not the case, then the thread is disabled, and the joining flag is set to true on the thread. When this flag is enabled, and a child of the joining thread is finished, the *ExecJoin* is executed again on the joining thread. When all children have finished joining, the thread can continue.

Chapter 4

Monotonic Partial Order Reduction

An attempt has been made to reduce the full state graph of all interleaving executions of a concurrent program. This is done to combat the path explosion problem. Monotonic Partial Order reduction is proposed in the paper "An Optimal Symbolic Partial Order Technique"[Kahlon et al., 2009]. Using dependencies between transitions, the algorithm can determine if two program paths are Mazurkiewicz equivalent. This indicates that the program's behavior is the same even though the ordering of transitions in the program paths is different. Since the behavior is the same, the algorithm considers them representatives of each other, so only one program path has to be executed. The algorithm is considered sound since it only removes unnecessary paths and complete because it removes all unnecessary paths. Note that this differs from the definition of a sound and complete verification process; the following definitions are given in perspective of the pruning of paths.

Definition 3 (Soundness of a Partial Order Reduction method)**.** *A partial order reduction method is sound when an equivalent representative path is not pruned by the algorithm for every path pruned by the algorithms.*

Definition 4 (Completeness of a Partial Order Reduction method)**.** *A partial order reduction method is complete when it prunes every path in a program where an equivalent representative path exists that is not pruned by the algorithm.*

This differs from the definitions given by the original paper since they consider the algorithm to be optimal. Their definition of monotonic partial order reduction being optimal is that "No two computations explored are Mazurkiewicz equivalent" [Kahlon et al., 2009]. This aligns with our definition of completeness given in definition 4. Generally, algorithmic complexity is reduced to its minimal point when referring to an algorithm as optimal. Since this is not necessarily the case for the algorithm, a wording change was chosen for this research.

4.1 Dependent transitions

A dependency between transitions exists when the order of execution for these transitions determines the outcome of a verification. This means that for two transitions t_1 and t_2 that are part of a computation x, a dependency exists when executing t_1 before t_2 results in a different state than when executing t_2 before t_1. From now on, the notation $t_1 <_x t_2$ indicates that transition t_1 is executed before t_2 along computation x.

When applying partial order reduction on concurrent programs, it is assumed that this dependency is only relevant if the transitions t_1 and t_2 are part of different threads.

Definition 5 (Independence relation)**.** *Two transitions t_1 and t_2 from different threads are independent if the following properties hold over any state $s \in S$.*

- *If t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s', and*
- *If t_1 and t_2 are enabled, then there exists a unique state s' such that $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$.*

Case 2 Program with three independent threads

```
1: procedure T1          1: procedure T2          1: procedure T3
2:     t1: x := 1;       2:     t2: y := x;       2:     t3: z := x + 1;
3: end procedure         3: end procedure         3: end procedure
```

When a program consists of multiple threads, these threads execute transitions within their own local scope; this means that dependencies only occur when a variable is shared between different threads. In Case 2, an example is given of a program that consists of three different threads, T1, T2, and T3, each with a single statement that corresponds with a transition t_1, t_2 and t_3. Let x, y, and z be shared variables with an initial value of 0. In this case, an explicit dependency exists between the transitions t_1 and t_2. For the execution order $t_1 <_x t_2$, the variable y will have the value of 1 at the end. In comparison, the execution order $t_2 <_x t_1$ will cause the variable y to be assigned 0 at the end. A very similar dependency can be found between t_1 and t_3. There is no dependency between t_2 and t_3 since the execution order does not impact the resulting state since both variables only read from x.

When two transitions are independent, the execution order does not matter for the resulting state. To choose which execution order is explored, we generally prefer in-order execution, meaning that the transition with the lowest thread id is chosen in a set of independent transitions.

Definition 6 (Out of Order execution)**.** *An execution is **out-of-order** when two transitions t_i and t_j are executed in the order $t_i < t_j$, where $tid(t_i) > tid(t_j)$. A computation x is fully **in-order** when any pair of transitions along computation x where $t_i <_x t_j$, the following holds: $tid(t_i) < tid(t_j)$.*

The original paper on Monotonic Partial Order Reduction implements the algorithm only for a language consisting of concrete references to shared variables. These can be easily determined, as shown in the previous example. The goal is to implement the algorithm for an object-oriented language that allows for symbolic references and array indexations. This means that the original monotonic partial order reduction algorithm must be extended to consider dependencies between these data structures.

Symbolic references differ from concrete references since multiple symbolic references can point to the same object. When two symbolic references a and b that point to an object of the same type exist, we have to consider the possibility that b points to object a and vice versa. When a possibly points to b, we consider b an alias of a. These aliases are stored in an alias table, denoted by A.

Case 3 Program with three independent threads

```
1: procedure T1          1: procedure T2
2:     t1: a.x := 1;     2:     t2: b.x := 2;
3: end procedure         3: end procedure
```

In Case 3, two threads exist with a single transition, t_1 and t_2, respectively. Let's assume that a and b are symbolic references pointing to an object with the same type and that A contains an alias $a \longrightarrow b$. When checking for a dependency between t_1 and t_2, we find two writes with different symbolic references, a and b. Since the references are symbolic, a and b must be checked for possible aliases. Since A contains the alias b for the symbolic reference a a, we consider that t_1 can possibly write to $a.x$ or $b.x$. Since t_2 writes to $b.x$, we consider t_1 and t_2 to have a possible dependency. There is no way of determining a definitive dependence or independence in this situation. To keep the algorithm sound, we consider a possible dependency as an actual dependency.

To determine dependencies between transitions that write to specific indices of arrays, the indices must be compared to determine if the writes alter the same value. Suppose two transitions t_1 and t_2 write to array a with assignments to $a[i]$ and $a[j]$ respectively, where i and j are expressions. Then, it can be said that a dependency exists when $i = j$. In the OOX language, expressions cannot contain any references. If the user wants to use the value of a reference, it first has to assign it to a local variable. When an assignment is made from a symbolic reference to a local variable, the state is split for each alias. This means it is unnecessary to consider symbolic references inside the array indexation. When implementing this in other languages that allow for this behavior, it must be considered that array indexation can possibly contain symbolic references.

For a full recap of all possible dependencies that can occur in the program, labels are given to all possible heap accesses that can occur:

- **IndexedWrite** $\{a, i\}$**:** The writing of an indexed element of an array to the heap, where a is a (symbolic/concrete) reference to the array, and i is the index.
- **IndexedRead** $\{a, i\}$**:** The reading of an indexed element of an array of the heap, where a is a (symbolic/concrete) reference to the array, and i is the index.
- **FieldWrite** $\{obj, f\}$**:** The writing of an object field to the heap, where obj is a (symbolic/concrete) reference to the object and f is the field label.
- **FieldRead** $\{obj, f\}$**:** The reading of an object field of the heap, where obj is a (symbolic/concrete) reference to the object and f is the field label.

Locks can also cause dependencies since a lock occurs on a reference shared between threads. Since the value linked to the reference is not important for locking threads, they never cause dependency with statements other than with locks. Joins can only have dependencies with a FunctionExit, where the joining thread is a parent of the thread that exits. When a thread exits, it can cause a join to be able to continue, always causing an **out-of-order** execution. This **out-of-order** execution should be allowed since the join enforces this control flow, which is considered a dependency.

- **LockAccess** $\{ref\}$**:** The locking and unlocking of the reference ref.
- **JoinAccess** $\{tid\}$**:** The joining of a thread.
- **FunctionExit** $\{parents\}$**:** The joining of a thread.

Using the above labels, a complete map can be given for all possible dependencies between transitions. A dependency condition is given for each pair of accesses, and when the dependency condition is evaluated to be true, the pair is considered dependent on each other. The function $aliases(x)$ is used when dealing with symbolic references. The $aliases(x)$ function returns all aliases in alias map A for reference x. When x is already concrete, the method returns just the concrete reference. This means that if A contains the aliases $x \longrightarrow y$, and $x \longrightarrow z$, the resulting set of $aliases(x)$ is $\{x, y, z\}$.

Access 1	**Access 2**	**Dependency Condition**
IndexedWrite $\{a, i\}$	**IndexedWrite** $\{b, j\}$	$aliases(a) \cap aliases(b) \neq \emptyset \wedge i = j)$
IndexedWrite $\{a, i\}$	**IndexedRead** $\{b, j\}$	$aliases(a) \cap aliases(b) \neq \emptyset \wedge i = j$
IndexedWrite $\{a, i\}$	**FieldWrite** $\{obj, f\}$	$false$
IndexedWrite $\{a, i\}$	**FieldRead** $\{obj, f\}$	$false$
FieldWrite $\{obj, f\}$	**IndexedRead** $\{a, i\}$	$false$
FieldWrite $\{obj1, f1\}$	**FieldWrite** $\{obj2, f2\}$	$aliases(obj1) \cap aliases(obj2) \neq \emptyset \wedge f1 = f2$
FieldWrite $\{obj1, f1\}$	**FieldRead** $\{obj2, f2\}$	$aliases(obj1) \cap aliases(obj2) \neq \emptyset \wedge f1 = f2$
LockAccess $\{ref1\}$	**LockAccess** $\{ref2\}$	$ref1 = ref2$
LockAccess $\{ref\}$	**JoinAccess** $\{tid\}$	$false$
LockAccess $\{ref\}$	**FunctionExit** $\{tid\}$	$false$
JoinAccess $\{tid\}$	**FunctionExit** $\{parents\}$	$\{tid\} \cap parents \neq \emptyset$

Table 4.1: Dependency conditions for the possible accesses of a program

As previously mentioned, no pairs of reads are given in the table since two reads are never considered a dependency. In the table, three different pairs result in a false dependency condition. The pair consists of an object field and an indexed access in all three cases. In many languages, there could be a case for a possible dependency since the object field could be equal to an array accessed in another thread. Still, for this research, we consider each statement to consist of at most one heap access. As an example, the statement $obj.f[i] = 0$ would be split into two separate statements $int[]a = obj.f; a[i] = 0$. Therefore, the first statement would cause the dependency with another $obj.f$ write. When comparing expressions like the indexations of arrays, it is also important that the expressions are fully evaluated in their found state. All variables present in the expression must be retrieved from their current state to ensure that the value of the expressions cannot change when executing future transitions.

For future reference, let's define the following notation $DEP(t_1, t_2)$ to show a dependency between the accesses in t_1 and those in t_2. To determine the dependence between transitions, at least one dependency should be found when using Table 4.1.

4.2 Dependency Chains

An important concept to understand is dependency chains before getting into the reduction of the state graph. A program consists of transitions that alter the state. This means that each program path can be seen as a linear order of transitions. Dependency chains are connected pairs of dependent transitions, meaning a single pair of dependent transitions is also considered a dependency chain. Let us use $t_1 \Rightarrow t_2$ to denote a dependency chain between t_1 and t_2.

$$t_1 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow t_4 \longrightarrow t_5$$

Figure 4.1: Path of transitions

In the above example, the base case of a dependency chain would be if there exists a dependency $DEP(t_1, t_2)$, then automatically there exits a dependency chain $t_1 \Rightarrow t_2$. When there exist two dependencies

$DEP(t_1, t_2)$ and $DEP(t_2, t_3)$, then the following chains exist, $t_1 \Rightarrow t_2$, $t_2 \Rightarrow t_3$, and $t_1 \Rightarrow t_3$. This is the case since the dependency chains chain together, allowing for a larger chain. This is possible for any dependency chain present in the path of transitions, so if there exist two dependencies $DEP(t_1, t_3)$ and $DEP(t_3, t_5)$, the following chains exist $t_1 \Rightarrow t_3$, $t_3 \Rightarrow t_5$, and $t_1 \Rightarrow t_5$. This example is given to indicate that there can exist a dependency chain from $t_3 \Rightarrow t_5$ without there being any dependencies for t_4. This is because dependency chains are present not only on consecutive transitions but can exist for any sub-sequence of transitions within a computation. Note that dependency chains can only exist in the order of transitions; therefore, it can never be considered that a dependency chain $t_5 \Rightarrow t_3$ exists since, in this case, t_5 and t_3 are not in the order of scheduled transitions.

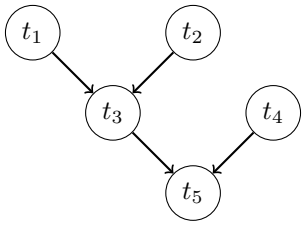

Figure 4.2: Dependency graph of transitions

The above graph visualizes a dependency graph for the following dependencies: $DEP(t_1, t_3)$, $DEP(t_2, t_3)$, $DEP(t_3, t_5)$, and $DEP(t_4, t_5)$. The graph is just for visualization to show what dependency chains exist in the execution. All transitions have a dependency chain to t_5 in the graph since all paths eventually lead to t_5. A more formal definition taken from the original "Monotonic Partial Order Reduction" paper:

Definition 7 (Dependency chain)**.** *Let t and t' be transitions executed along a computation x such that $t <_x t'$. A dependency chain along x starting at t is a (sub-)sequence of transitions $tr_{i_0}, ..., tr_{i_k}$ executed along x, where (a) $i_0 < i_1 < ... < i_k$, (b) for each $j \in [0..k-1]$, tr_{i_j} is dependent with tri_{j+1}, and (c) there does not exist a transition executed along x between tr_{i_j} and tri_{j+1} that is dependent with tr_{i_j}.*

4.3 Quasi Monotonic Computation

With the concepts of determining dependencies between transitions and dependency chains along a computation, it is possible to start reducing paths with quasi-monotonic computations. When a path/computation is not considered quasi-monotonic, removing this as a path from the verification process is possible without altering the program's outcome. This rule ensures that all unnecessary paths are removed and none of the necessary paths are removed.

For a path to be a quasi-monotonic computation, one of the following two rules has to be satisfied for each pair of transitions (t_i, t_j) along computation x, where $t_i <_x t_j$ and $tid(t_j) < tid(t_i)$:

- **Rule 1:** There exists a dependency chain $t_i \Rightarrow t_j$, or
- **Rule 2:** There exists a transition t_l where there exists a dependency chain $t_i \Rightarrow t_l$, with the execution order $t_i <_x t_l <_x t_j$, and $tid(t_l) < tid(t_j)$.

This concept can be difficult to grasp; therefore, the following section provides a complete example and a step-by-step guide for determining whether an execution path is quasi-monotonic.

4.4 Validating Paths to be Quasi Monotonic

Case 4

1: **procedure** T1	1: **procedure** T2
2: t_1: int y = 3;	2: t_4: int i = x.val;
3: t_2: x.val := y * 2;	3: t_5: a[i] = 2;
4: t_3: a[y] = 3;	4: t_6: int z = a[i];
5: **end procedure**	5: **end procedure**

In case 4, two threads are given that both contain a couple of statements that modify local and shared variables. x and a are the only variables shared in this program. x and a are symbolic references with no aliases. The accesses to shared variables can be extracted for each transition in the above program.

Transitions	**Accesses**
t_1	$\emptyset$
t_2	**FieldWrite** $\{x, val\}$
t_3	**IndexedWrite** $\{a, y\}$

Table 4.2: Accesses T1

Transitions	**Accesses**
t_4	**FieldRead** $\{x, val\}$
t_5	**IndexedWrite** $\{a, i\}$
t_6	**IndexedRead** $\{a, i\}$

Table 4.3: Accesses T2

The tables 4.2 and 4.3 show the accesses that can be used to check if there exist dependencies between the transitions using table 4.1. Below, multiple execution paths are given. Then, for each path, what dependency chains occur and if the computations are quasi-monotonic according to the rules. Note that there are no possible paths, like $t_2 \longrightarrow t_1$, since out-of-order execution is not allowed inside a single thread.

Example 1

$$t_1 \longrightarrow t_4 \longrightarrow t_2 \longrightarrow t_5 \longrightarrow t_3 \longrightarrow t_6$$

In the definition of quasi-monotonic computations, it can be found that only transitions that are executed **out-of-order** (according to their thread ids) are relevant to determining if an execution path is quasi-monotonic. This means that in the above path, the only relevant execution orders are $t_4 < t_2$, $t_5 < t_3$, and $t_4 < t_3$.

- $t_4 < t_2$: Both t_4 and t_2 contain a single access, a **FieldWrite**$\{x, val\}$ and **FieldRead**$\{x, val\}$ respectively. According to table 4.1, the dependency condition is $aliases(x) \cap aliases(x) \neq \emptyset \wedge val == val$. This is true, and even though this is an **out-of-order** execution, it follows **Rule 1** of quasi-monotonic computations due to its dependency. Therefore, this execution order is allowed.
- $t_5 < t_3$: Both t_5 and t_3 contain a single access, a **IndexedWrite**$\{a, i\}$ and **IndexedWrite**$\{a, y\}$ respectively. According to table 4.1 the dependency condition is $aliases(a) \cap aliases(a) \neq \emptyset \wedge i = y$, both a reference the same array so that part is considered true. $i = y$ can be evaluated down to $x = 3$, and then x was assigned $y * 2$ in t_3, resulting in the final expression $6 = 3$. This means that this out-of-order execution does not allow this path to be quasi-monotonic.

Example 2

$$t_1 \longrightarrow t_4 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow t_5 \longrightarrow t_6$$

As in the previous example, it's only necessary to validate the out-of-order executions in a computation to determine if it is quasi-monotonic. For the above computation, the out-of-order computations are $t_4 < t_2$ and $t_4 < t_3$.

- $t_4 < t_2$: Transitions t_4 and t_2 both contain a single access, **FieldWrite**$\{x, val\}$ and **FieldRead**$\{x, val\}$ respectively. According to table 4.1 the dependency condition for these accesses would be $concrete(x) \cap concrete(x) \neq \emptyset \wedge val == val$. The dependency condition is true since $concrete(x)$ always contains x as an alias. This means a dependency chain $t_4 \Rightarrow t_2$ exists, and according to **Rule 1** of quasi-monotonic computations, this out-of-order transition is allowed.
- $t_4 < t_3$: This execution order, according to **Rule 2** of quasi monotonic computations is valid. This is because there exists a dependency chain $t_4 \Rightarrow t_2$ and a dependency chain between $t_2 \Rightarrow t_3$. This means a direct dependency chain $t_4 \Rightarrow t_2$ also exists.

Chapter 5

Algorithm for Dynamic Pruning of Non Quasi Monotonic Paths

In the previous section, a method was discussed on how it can be determined if an execution path is quasi-monotonic. In this section, the method is applied as an algorithm that dynamically prunes paths that do not follow the rules for quasi-monotonic computations. In certain cases, some choices had to be made about the result of the verification process. When a possible dependency is mentioned, it means that a symbolic reference has an alias that causes a conflict. If just one alias causes a dependency with another thread, then a pessimistic approach is taken, considering the possible dependency as a dependency.

5.1 Simplification of a complex concept

The concept of quasi-monotonic computations is somewhat difficult to understand, while the algorithm can be simplified to a point where it is easy to implement. This is possible because keeping track of all the full dependency chains is unnecessary. Lets take the following threads with their transitions, $T1\{t_1, t_2\}$, $T2\{t_3, t_4, t_7\}$ and $T3\{t_5, t_6\}$.

$$t_5 \longrightarrow t_1 \longrightarrow t_6 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow t_4$$

With the above path, the dependency chains are, by default, $t_5 \Rightarrow t_6$, $t_1 \Rightarrow t_2$ and $t_3 \Rightarrow t_4$ due to the dependencies of transitions within the same thread, let's ignore the other dependencies that would make this path quasi-monotonic. When attempting to schedule t_7 that is part of $T2$, then it can be said that there exists two **out-of-order** executions, $t_5 < t_7$ and $t_6 < t_7$. From the perspective of t_7, if t_6 satisfies **Rule 1** of quasi-monotonic computations, then so does t_5. If t_6 satisfies **Rule 2** of quasi-monotonic computations, then so does t_5. It can be said that t_5 is never relevant for scheduling t_7 due to its natural dependency chain to t_6. This means that the $from$ transitions in dependency chains between transitions of the same thread can be considered irrelevant by both rules of quasi-monotonic computations. This means only the last executed transition of each thread is relevant to determine the ability to schedule the next transition.

Instead of tracking the full dependency chains, each thread only has to store the accesses that occurred in the last executed transition.

5.2 Extracting Accesses from Statements

The OOX language was designed not to contain heap accesses in expressions. This means the only statement that actually contains heap accesses is the assignment statement. The lock, unlock, join, and function exit primitive statements also allow for possible dependencies; therefore, they are included within the accesses of a statement.

5.2.1 Assignment

The primitive statement of assignment $\textbf{\textit{Assign}}\ Lhs\ Rhs$ contains a left-hand side (Lhs) and a right-hand side (Rhs). Both the Lhs and Rhs can only contain a single heap access; with that, at most, two accesses are present in any primitive statement.

$$\begin{aligned} Lhs &::= lhs_{var}(x) \mid lhs_{field}(x,f) \mid lhs_{elem}(x,i) \\ Rhs &::= rhs_{expr}(E) \mid rhs_{var}(x) \mid rhs_{field}(x,f) \mid rhs_{call}(I) \mid rhs_{elem}(x,i) \mid rhs_{array}(x) \end{aligned} \tag{5.1}$$

$$\begin{aligned} &Accesses(\textbf{\textit{Assign}}\ Lhs\ Rhs) = Accesses(Lhs) \cup Accesses(Rhs) \\ &Accesses(Lhs) = \begin{cases} \{IndexedWrite\{x,i\}\} & lhs_{elem}(x,i) \\ \{FieldWrite\{x,f\}\} & lhs_{field}(x,f) \end{cases} \\ &Accesses(Rhs) = \begin{cases} \emptyset & rhs_{expr}(x) \\ \{FieldRead\{x,f\}\} & rhs_{field}(x,f) \\ \emptyset & rhs_{call}(I) \\ \{IndexedRead\{x,i\}\} & rhs_{elem}(x,i) \\ \emptyset & rhs_{array}(x) \end{cases} \end{aligned} \tag{5.2}$$

5.2.2 Locks and Joins

The primitive statements of locks ($\textbf{\textit{Lock}}\,x$, $\textbf{\textit{Unlock}}\,x$) contain both a singular access. The Join ($\textbf{\textit{Join}}\,tid$) statement results in an access that only contains a thread id. The $FunctionExit$ statement results in an access containing its parents' thread ids. The function $getRef(x)$ is used to get the reference of the variable x from the stack.

$$\begin{aligned} &Accesses(\textbf{\textit{Lock}}\,x) = \{LockAccess\{getRef(x)\}\} \\ &Accesses(\textbf{\textit{UnLock}}\,x) = \{LockAccess\{getRef(x)\}\} \\ &Accesses(\textbf{\textit{Join}}) = \{JoinAccess\{atid\}\} \\ &Accesses(\textbf{\textit{FuncionExit}}) = \{FunctionExit\{parents\}\} \end{aligned} \tag{5.3}$$

5.3 Reducing Non Quasi Monotonic Paths

To start reducing non-quasi-monotonic paths, every transition has to be checked to see if it conforms to the quasi-monotonic rules. The algorithm is implemented in symbolic execution, so it would also be possible to prune the paths after the full exploration, but a large benefit of this algorithm is the ability to dynamically remove paths during the path exploration step.

For the purposes of this research, each transition consists of a single primitive statement. The previously explained method to extract accesses from primitive statements is used after each execution of a transition, and these accesses are stored in a part of the thread. The value of the thread's previous accesses can be *null* or a list of accesses. When the value of the thread's previous accesses is *null*, a dependency chain exists with another thread's last executed statement. In this scenario, the thread's last executed transition does not have to be checked for an **out-of-order** execution with the currently scheduled transition, as explained in section 5.1. When the value of a thread's previous accesses is a list of accesses, it has to be checked if the current transition t_{curr} is **out-of-order** with the thread's last executed transition t_{prev}. t_{curr} is considered **out-of-order** if $tid(t_{prev}) > tid(t_{curr})$. When the scheduling of t_{curr} is **out-of-order**, it has to be checked if there exists a dependency between t_{curr} and t_{prev}. This is the case if $tid(t_{curr}) = tid(t_{prev})$, or when $DEP(t_{curr}, t_{prev}) = true$ when using table 4.1. When there exists a dependency, it can be said that there exists a dependency chain $t_{prev} \Rightarrow t_{curr}$, and therefore the previous accesses in t_{prev}'s thread can be set to *null*. If no dependency exists, the MPOR algorithm should prune the path. After each iteration, the *prev_accesses* of the active thread is always set for the current scheduled transition.

These methods can be applied to pseudo-code easily. The data structures from the implementation at section 3.4 must be extended. The previous accesses are stored in each thread with the notation $\mathcal{T}.\chi$. The $GetAccesses(n)$ method takes a node n and then checks for accesses within that node using the $Accesses(Node)$ method denoted in section 5.2. The $Dep(x, y)$ method takes two different sets of accesses and uses table 4.1 to determine if there exists a dependency between the accesses.

Algorithm 6 Monotonic Partial order reduction

```
1:  procedure MPOR((T, atid, h) ∈ S, (N, E) ∈ CFG)
2:      curr ← T_atid.pc
3:      accesses ← GetAccesses(N_curr)
4:      out ← true
5:      for thread in T do
6:          if thread.χ ≠ null then
7:              if thread.tid = atid or Dep(accesses, thread.χ) then
8:                  thread.χ ← null
9:              else if thread.tid > atid then
10:                 out ← false
11:                 break
12:             end if
13:         end if
14:     end for
15:     T_atid.χ ← accesses
16:     return out
17: end procedure
```

The method $MPOR$ of algorithm 6 is called for each remaining state in $ExecuteStates$ method in algorithm 1. Due to the state being cloned beforehand in the execute states method, no side effects can happen during the $MPOR$ method. The currently scheduled transitions are allowed when the $MPOR$ method returns true. If it returns false, the current execution path is considered not quasi-monotonic, and with that, the execution path gets pruned (removing the state from the remaining states before continuing).

Chapter 6

Benchmarks and Data Collection

Multiple important steps must be taken to determine if this implementation of MPOR benefits the concurrent form of the OOX verification tool. The goal of the performed benchmarks is to confirm the soundness and completeness of the verification process and to answer the following research question:

RQ-2: *What is the impact of monotonic partial order reduction on the performance of symbolic execution verification tools for object-oriented languages?*

The verification tool has to be tested to see if it retains its soundness with the implementation that allows the verification of concurrent programs, meaning that for all valid programs, valid is the outcome of the verification process. It has to be tested if the verification tool retains its completeness over the given depth with the implementation that allows the verification of concurrent programs, meaning that invalid programs can be determined invalid. The verification tool's ability to verify concurrent programs has to be tested to see if it retains its completeness and soundness when using monotonic partial order reduction, and therefore, monotonic partial order reduction being sound and complete itself according to the definitions 3 and 4. The verification tools' performance has to be measured with and without monotonic partial order reduction to determine if there is a performance benefit.

All the above points will be tested throughout experiments, challenging the ability of verification tools to verify programs. In table 6.1, it can be seen what experiments target which of the above-mentioned targets.

targets	*EXP-1*	*EXP-2*	*EXP-3*	*EXP-4*	*EXP-5*	*EXP-6*
Soundness	$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$		$\mathcal{X}$	
Completeness	$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$		$\mathcal{X}$	
Performance	$\mathcal{X}$			$\mathcal{X}$		$\mathcal{X}$
Soundness of MPOR	$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$		$\mathcal{X}$	
Completeness of MPOR			$\mathcal{X}$			
Performance with MPOR	$\mathcal{X}$			$\mathcal{X}$		$\mathcal{X}$

Table 6.1: Targets of the performed experiments

During the experiments, NOPOR will be used to indicate that no form of partial order reduction is used.

MPOR will indicate that monotonic partial order reduction is used. When performing *EXP-4* and *EXP-6*, a straight comparison between NOPOR and MPOR could be insufficient since it is likely that any form of reduction will perform better than no form of reduction. SR (Simplistic reduction) is introduced for these experiments to see the impact of MPOR when compared to a minimal form of reduction. SR considers a transition dependent on all other transitions when it contains any access from section 4.1. This means many paths with equivalent representatives remain, but this method still prunes out many interleaving over statements that affect the local process.

For all experiments, a laptop was used with the following environment and specifications:
- Apple Silicon M2 (8 cores)
- 16 GB Ram (DDR4)
- Minimal background processes
- Power cord attached

6.1 *EXP-1*: Soundness of verifying concurrent programs

It has to be determined if the verification tool can verify concurrent programs with and without the MPOR algorithm to indicate if both the tool and algorithm are sound. Multiple programs that make use of concurrency are tested. Some of these programs should be returned as invalid. When showing the verification tool can correctly determine if a program is valid or invalid, it can be determined that the verification tool behaves as expected. The programs that are to be tested are available in Appendix A.1, and they will have the following targets:

- *EXP-1.1* (Invalid): Showing that threads behave non-deterministic and therefore not guaranteeing that the threads execute before the assertion.
- *EXP-1.2* (Valid): Showing that a join can be used to enforce that the threads are executed before continuing the main thread.
- *EXP-1.3* (Invalid): Showing that the non-deterministic nature of threads causes the program to execute reads and writes of shared variables in random orders.
- *EXP-1.4* (Valid): Showing that locks can be used to guarantee a wanted control flow to prevent the random reads and writes of shared variables.
- *EXP-1.5* (Invalid): Shows that arrays are also considered shared variables and that the verification tool can determine that the non-deterministic flow causes the program to be considered Invalid.
- *EXP-1.6* (Valid): Showing that the execution order is non-deterministic, causing either thread to be executed last.
- *EXP-1.7* (Deadlock): This shows that the verification tool can detect deadlocks caused by improper usage of locks.
- *EXP-1.8* (Valid): Shows that unlocking the shared variable prevents the deadlock from EXP-1.7 from occurring.
- *EXP-1.9* (Invalid): Showing that the verification tool can correctly handle conditional branches on a shared variable.

- *EXP-1.10* (Invalid): Showing that the verification tool can correctly handle loops and being able to identify an invalid control flow.
- *EXP-1.11* (Valid): This shows that locking can guarantee a control flow inside a loop, ensuring shared variables are read and written properly.
- *EXP-1.12* (Invalid): Showing that the verification tool can find undesired behavior when branching occurs, causing a shared variable to be assigned a non-determinist value.

		NOPOR		MPOR		Expected
Experiment	Max Depth k	Output	Time(S)	Output	Time(S)	Output
EXP-1.1	40	Invalid	0.17	Invalid	0.18	Invalid
EXP-1.2	40	Valid	0.14	Valid	0.16	Valid
EXP-1.3	40	Invalid	0.14	Invalid	0.15	Invalid
EXP-1.4	40	Valid	29.37	Valid	0.18	Valid
EXP-1.5	40	Invalid	0.14	Invalid	0.14	Invalid
EXP-1.6	40	Valid	1.83	Valid	0.15	Valid
EXP-1.7	40	Deadlock	0.01	Deadlock	0.01	Deadlock
EXP-1.8	40	Valid	1.06	Valid	0.13	Valid
EXP-1.9	40	Invalid	174.20	Invalid	0.18	Invalid
EXP-1.10	120			Invalid	0.20	Invalid
EXP-1.11	120			Valid	1.17	Valid
EXP-1.12	40	Invalid	0.17	Invalid	0.15	Invalid
EXP-1.13	120			Valid	28.38	Valid

Figure 6.1: Experiment results that show NOPOR output, MPOR output, and expected output

In figure 6.1, the results can be found for all the programs that were run for *EXP-1*. Some values are left out for the NOPOR settings since the program could not produce a result within a reasonable time. It can be seen that the NOPOR (when the program is finished) and MPOR both return the correct output for the given programs. This confirms that the earlier setup goals can be verified.

Even though this experiment is not about performance, the above execution times show that MPOR is not always faster than NOPOR. This is the case when the overhead of the pruning is larger than the benefit gained from the pruned branches. When the complexity and explosive nature of the code increase, MPOR shifts to being faster.

6.2 *EXP-2*: Symbolic References when using MPOR

When determining dependencies to ensure no unnecessary paths are pruned, symbolic references are somewhat of a challenge. This is due to multiple symbolic objects of the same type that could refer to the same reference, meaning that a dependency is possible. The implementation has a pessimistic approach, meaning that if a possible dependency exists, it is considered a dependency, therefore not allowing the pruning of some paths. As example in the following algorithm 11 can be seen that two objects of type Foo are passed to a separate fork. The outcome of this program should be invalid since there is a possibility that the two symbolic references point to the same object, possibly causing the out variable to have a value of 1.

Algorithm 11 Usage of symbolic references in a concurrent program

```
1:  class I2 {
2:    static void SetZero(Foo x) {
3:      x.val := 0
4:    }
5:    static void SetOne(Foo y) {
6:      y.val := 1
7:    }
8:    static void main(Foo x, Foo y) {
9:      fork I2.SetZero(x)
10:     fork I2.SetOne(y)
11:     join
12:     int out := x.val
13:     assert(out == 0)
14:   }
15: }
```

To determine that these scenarios are handled properly, multiple tests are run on different concurrent programs that include the usage of symbolic references. The programs that are to be tested are available in Appendix A.2, and they will have the following targets:

- *EXP-2.1* (Valid): Writing to an array using two different symbolic references in separate threads. Including an assumption that the symbolic references are not equal.
- *EXP-2.2* (Valid): Writing to an array using two different symbolic references in separate threads. Invalid due to a possible equality between the references.
- *EXP-2.3* (Valid): Updating a field on two symbolic references that point to an object with the same type in separate threads. Including an assumption that the symbolic references are not equal.
- *EXP-2.4* (Invalid): Updating a field on two symbolic references that point to an object with the same type in separate threads. Invalid due to a possible equality between the references.
- *EXP-2.5* (Valid): The locking of two symbolic references that point to the same typed object in separate threads. Including an assumption that the symbolic references are not equal.
- *EXP-2.6* (Deadlock): The locking of two symbolic references that point to the same typed object in separate threads. Deadlocks are due to the possibility of having the same reference.

Experiment	NOPOR	MPOR	Expected
EXP-2.1	Valid	Valid	Valid
EXP-2.2	Invalid	Invalid	Invalid
EXP-2.3	Valid	Valid	Valid
EXP-2.4	Invalid	Invalid	Invalid
EXP-2.5	Deadlock	Deadlock	Valid
EXP-2.6	Deadlock	Deadlock	Deadlock

Table 6.2: Experiment results that show NOPOR output, MPOR output, and expected output

As seen in table 6.2, it can be seen that for all except one, the expected output is given. In *EXP-2.5*, it can be seen that the program still deadlocks even when an assumption is made that the two symbolic references are separate. Locking looks at the alias table to identify whether the two symbolic references overlap. An assumption does not alter the alias table; it just clarifies that the two values of the references are unequal. This is a case that is not caught with the current implementation.

6.3 *EXP-3*: MPOR path pruning using traces

This shows that the implementation of monotonic partial order reduction prunes all paths that do not affect the outcome of the verification while keeping all the necessary paths sound when verifying the program.

Algorithm 12 SharedVar and T1 classes

```
1:  class SharedVar {
2:      int val
3:      SharedVar() { }
4:  }
5:  class T1 {
6:      static void a(SharedVar var) {
7:          var.val := 5;
8:      }
9:      static void main() {
10:         SharedVar var := new SharedVar();
11:         fork T1.a(var);
12:         var.val := 3;
13:     }
14: }
```

The algorithm shows that the outcome of *var.val* is non-deterministic because lines 7 and 12 do not have a guaranteed control flow while editing a shared variable. Even though, through our eyes, only these two transitions are necessary to interleave, the program contains many more primitive statements that occur between these transitions. In this case, these statements include if-else statements to guarantee that classes are not *null* and entering and exiting methods. This means that the amount of paths generated using interleaving heavily increases, in the case of this program 128 path traces are recorded during the verification process.

Program Counter	Code Line
0, 1, 2, 3	3
4	6
5, 11, 13	7
14	8
15	9
17, 20	10
23, 25, 31	11
33	12
34	13

Table 6.3: Program counters in relation to lines of code from algorithm 12

1: $15 \to 17 \to 20 \to 0 \to 1 \to 2 \to 3 \to 23 \to 25 \to 31 \to 4 \to 5 \to 11 \to 13 \to 33 \to 34 \to 14$
2: $15 \to 17 \to 20 \to 0 \to 1 \to 2 \to 3 \to 23 \to 25 \to 31 \to 33 \to 34 \to 4 \to 5 \to 11 \to 13 \to 14$
3: $15 \to 17 \to 20 \to 0 \to 1 \to 2 \to 3 \to 23 \to 25 \to 31 \to 4 \to 5 \to 11 \to 13 \to 33 \to 14 \to 34$
4: $15 \to 17 \to 20 \to 0 \to 1 \to 2 \to 3 \to 23 \to 25 \to 31 \to 33 \to 4 \to 5 \to 11 \to 13 \to 14 \to 34$

Figure 6.2: path traces found using MPOR on algorithm 12 represented with program counters

The traces in figure 6.2 are the only traces found when using MPOR. When using NOPOR, the program found 128 traces. This means that MPOR was able to prune 124 traces from the program. One hundred twenty-eight traces sound like a lot for a program containing four statements and some declarations, but many primitive statements can take place for a single statement in code. In table 6.3, the program counters from the traces in figure 6.2 are translated to code lines. This shows that, for example, line 3 contains four separate primitive statements (entering the method, the skip statement, returning the created object, and exiting the method).

1: $9 \to 10 \to 3 \to 11 \to 6 \to 7 \to 12 \to 13 \to 8$
2: $9 \to 10 \to 3 \to 11 \to 12 \to 13 \to 6 \to 7 \to 8$
3: $9 \to 10 \to 3 \to 11 \to 6 \to 7 \to 12 \to 8 \to 13$
4: $9 \to 10 \to 3 \to 11 \to 12 \to 6 \to 7 \to 8 \to 13$

Figure 6.3: path traces found using MPOR on algorithm 12 represented in lines of code

With the traces shown using lines of code in figure 6.3, it is possible to reason about the paths that were retained when using MPOR. It is important to note that two traces (3 and 4) here are not actually quasi-monotonic; this is due to this implementation of MPOR pruning all non-quasi-monotonic chains in the following iteration. Trace 3 and 4 are both not quasi-monotonic due to line 13 being executed after line 8. There is no dependency between these transitions; therefore, the paths would always be pruned in the next

iteration, retaining the completeness of the algorithm. Since line 6 and line 12 have a dependency, there should be a representative trace where line 6 is to be executed before 12 and vice versa. Since trace 1 and trace 2 are representative of these scenarios, it can be determined that, for this example, MPOR retains the soundness of the verification tool.

6.4 *EXP-4*: The Dining Philosophers Problem

The dining philosopher's problem is well known as a challenge to write a concurrent program that cannot result in a deadlock. The Dining Philosophers problem can cause a large path explosion by spawning N threads for N amount of philosophers. This generates many interleaving paths, making it a good benchmark for testing the performance of the Monotonic Partial Order Reduction method. The dining philosopher's problem allows for many dependent transitions between different threads, and in that scenario, MPOR is challenged because dependencies limit the pruning of paths. This allows us to get more evidence to be able to answer **RQ-2**. The program used is available in Appendix A.3, and the results of the benchmarks can be found in Appendix A.6.

Multiple runs are performed with the maximum depth increased to test MPOR's performance. Each run tests the verification tool using three different settings: without any state graph reduction, using a simple reduction method that only takes the interleaving if no heap access is found, and monotonic partial order reduction. The program was run with $N = 4$ so that there are four philosophers/threads.

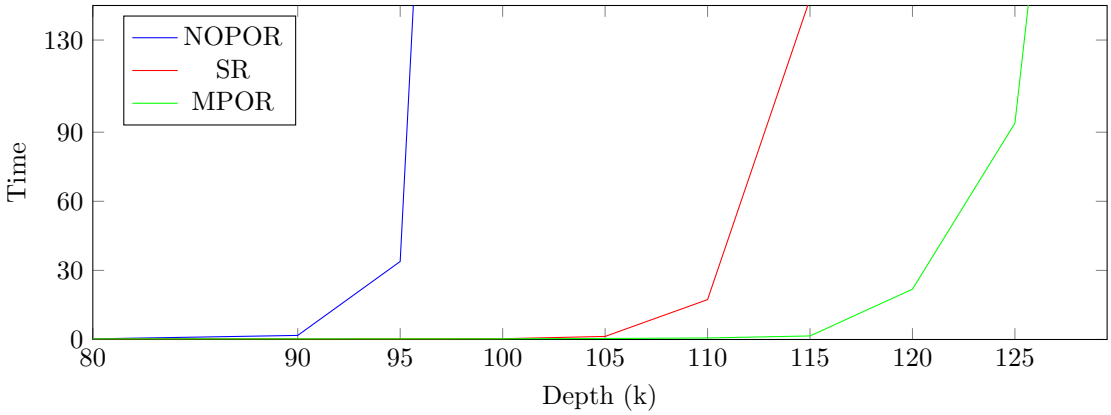

Figure 6.4: Performance in time for a range of given depths

In figure 6.4, the performance using the three different verification tool settings is measured against multiple depths. In all cases, clear exponential growth happens, but for each method, there is a different depth where the time for validation heavily increases. For NOPOR, the extreme growth starts at a depth of 90; for SR, the growth starts at 105; and for MPOR, the growth starts at 115. This shows that MPOR can handle a significantly higher depth than the other programs.

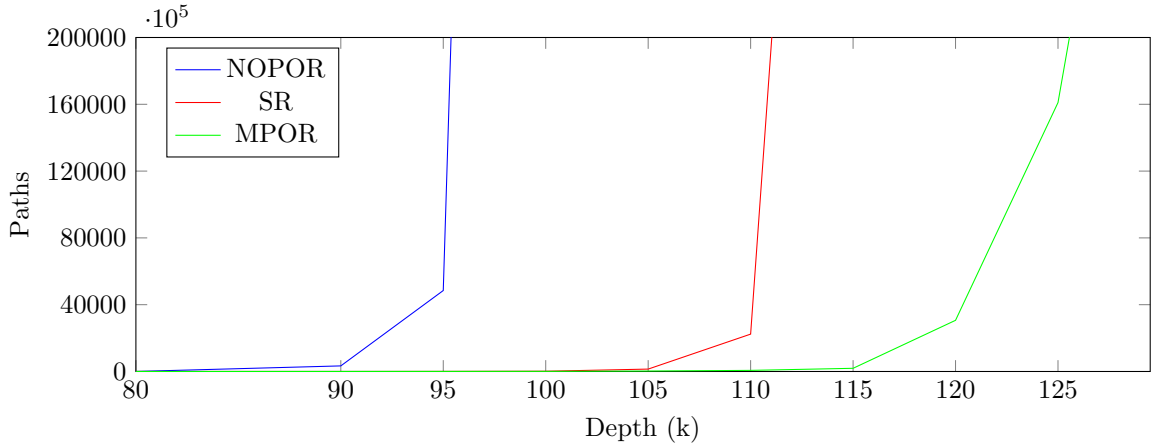

Figure 6.5: Completed paths over a range of depths

The number of completed paths is plotted in figure 6.5. When comparing this graph with the graph in figure 6.4, it can be easily determined that SR and MPOR perform better than NOPOR because paths are being pruned using the reduction methods when the verification occurs. MPOR prunes more paths in the SR since it can detect more accurate dependencies between transitions. It does this with a cost by verifying possible dependency chains for each transition. In regards to **RQ-2**, it can be said that for the dining philosophers, the pruning of paths performed by the MPOR algorithm outways the possible overhead of the algorithm; it can be said that the algorithm is beneficial for this case.

6.5 *EXP-5*: Deadlocked Dining Philosophers Problem

As previously mentioned, the Dining Philosophers Problem is prone to deadlocking if implemented incorrectly. A program was written for the Dining Philosophers problem that deadlocks to verify that the MPOR algorithm can find a deadlock within a faulty program. The program used is available in Appendix A.4.

The program is faulty in that it allows all philosophers to take the left fork, causing no philosopher to be able to take the right fork. Since the program can be evaluated manually to show there exists a deadlock, the faulty program is only verified using the monotonic partial order reduction method. After ten runs with $N = 4$ philosophers, the average time to find the deadlock was 51.50 seconds. For ten runs of $N = 3$, the average time to deadlock was 2.23 seconds. This again shows that exponential growth occurs in paths when the number of threads is increased.

6.6 *EXP-6*: Concurrent Merge Sort

Merge sort is easily convertible to a concurrent program due to its nature to compartmentalize sections of an array. This is a program that shows the strength of monotonic partial order reduction. This is due to the compartmentalization where, in no case, the same section of an array is modified by different threads. All updates to an array still cause SR to find interleaving, whereas MPOR can distinguish the different sections of the array and, therefore, prune more paths. To answer **RQ-2**, in $EXP - 4$, a benchmark is done on a

program that contains many dependencies, causing MPOR to be limited in the pruning of paths. In this experiment, merge sort spawns concurrent threads over separate parts of an array. This should be a best-case scenario for MPOR. The program used is available in Appendix A.5, and the results of the benchmarks can be found in Appendix A.7.

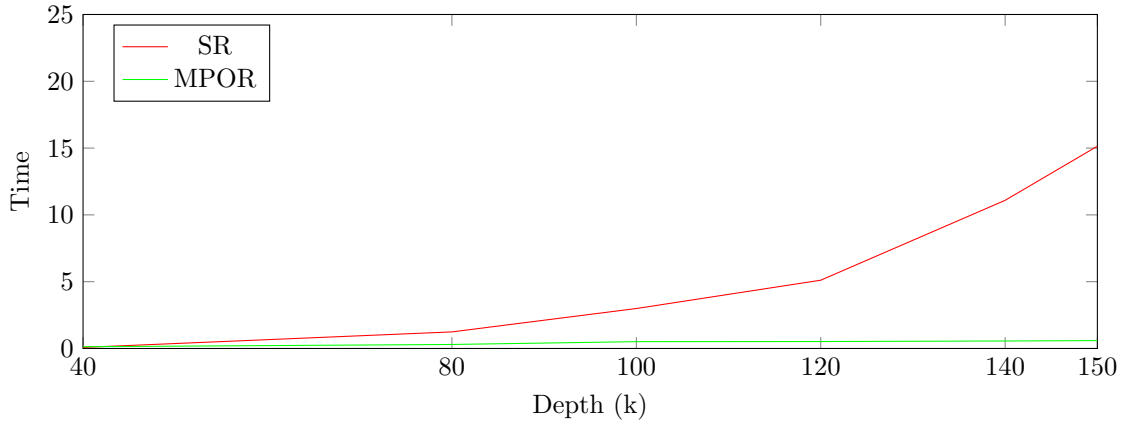

Figure 6.6: Performance in time for a range of given depths

In figure 6.6, the performance is measured of the verification tool using the SR and MPOR method. The reason for not including NPOR in the above graph is that it causes a memory overload from 40 depth onward. This means it can clearly be said that without any form of state graph reduction, the verification tool cannot verify the concurrent merge sort algorithm reasonably.

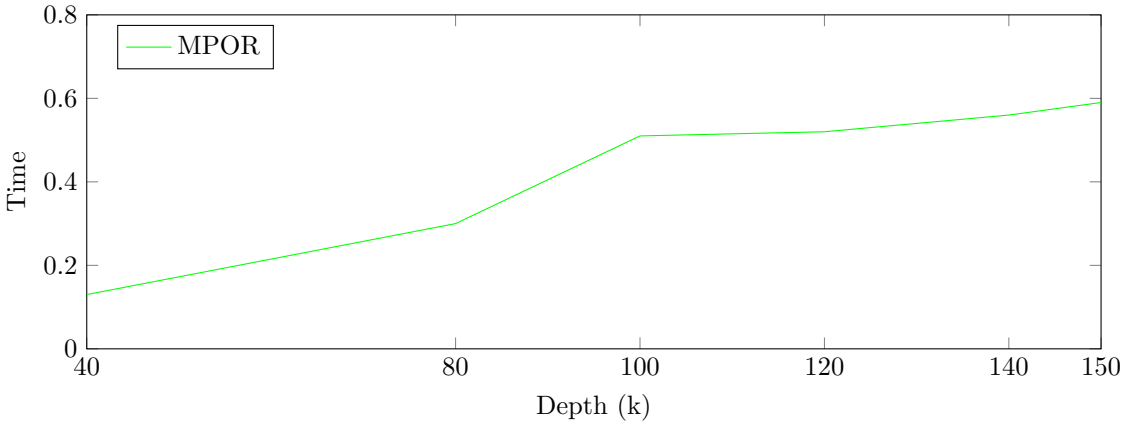

Figure 6.7: Performance in time for a range of given depths zoomed in

When looking at figure 6.7, the same results are graphed as in figure 6.6, but zoomed in along the y-axis. It is shown that unlike in the other experiments, after a depth of 100, the time required to verify

stops exponentially growing. This is due to the merge sort algorithm finishing in most paths. Due to the algorithm being recursive with a symbolic array as input, not every path can be pruned afterward. SR also has this decrease in growth, but due to its inability to prune the same number of paths, it will continue its exponential growth after a slight stall.

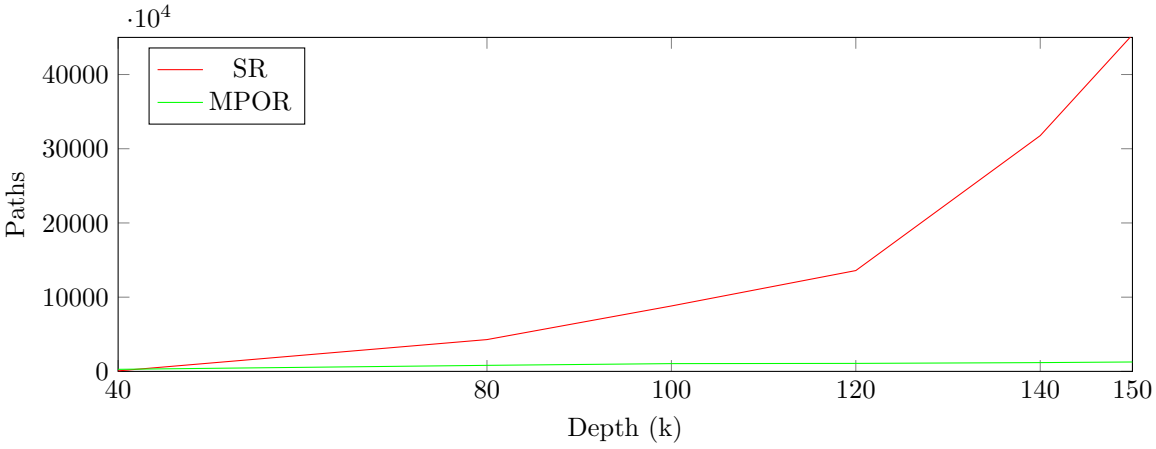

Figure 6.8: Completed paths for a range of given depths

In figure 6.8, it can be seen that the number of paths explored. At a depth of 150, the simple method explored 45355, while MPOR only had to verify 1267 paths to determine if the program was valid at this depth. This confirms the previous statement, in which it was identified that MPOR can reduce the number of paths significantly due to the nature of the concurrent merge sort algorithm. Regarding **RQ-2**, just like with the dining philosophers problem, the benefits of the MPOR algorithm are significant by heavily outperforming NOPOR and SR.

Chapter 7

Conclusion & Discussion

The research aimed to try to manage the path explosion problem within the OOX verification tool. After reviewing the results from the experiments, it can be concluded that the implementation of Monotonic Partial Order Reduction is feasible within a symbolic execution verification tool for an object-oriented language. It can also be said that the performance of the OOX verification tool using monotonic partial order reduction is heavily increased, with no form of reduction or a simplistic form of reduction.

The original paper on Monotonic Partial Order Reduction[Kahlon et al., 2009] explores the algorithm within a bounded symbolic model checking system. Due to the partial order reduction focusing on reducing paths dynamically during the exploration step, this method lends well to Symbolic Execution. As far as known, this is the first implementation of Monotonic Partial Order Reduction in a full-scale symbolic execution verification tool for an object-oriented language.

The performance is measured for multiple large-scale concurrent programs like the dining philosophers problem in *EXP-4* and *EXP-5* as well as the merge sort algorithm in *EXP-6*. These experiments show that the use of the monotonic partial order reduction results in a significant gain in performance. A clear relation is found between the number of pruned paths and the program's performance. This performance increase allows the verification tool to explore the program in a larger depth within the same amount of time. For the merge sort in *EXP-4*, the verification tool, without any form of state graph reduction, could not handle a depth higher than 40, not allowing it to traverse the program fully. This means that the algorithm significantly improves the OOX verification tool.

There is a possibility that the overhead of checking dependencies between two different indexations of an array could cause a lower performance when using Monotonic Partial Order Reduction. Since the reduction has such a high-performance gain, this is unlikely. Still, in very complex programs that require large invocation of the formula prover, this overhead might become large enough to be a detriment. This can not be verified currently due to the depth required for programs that are this complex. The depth causes the program to not verify within a reasonable time.

Two cases are currently not handled properly. The first case is shown in *EXP-2.5*, where the locking of two threads using symbolic references causes a deadlock, while an assumption is made that the two symbolic references are not equal. The cause is the assumption not affecting the alias table, causing the lock and unlock statements to always check for a collision between aliases. The second case is not being able to guarantee that the Monotonic Partial Order Reduction is complete for this implementation. This is because assumptions are not considered when tracking dependencies, causing false-positive dependencies. This means the algorithm remains sound but does not prune every possible path. As with the first case, the algorithm

also does not handle assumptions over symbolic references. These cases will be explored within the future work 8, and a possible solution will be given.

Chapter 8

Future Work

As mentioned in the discussion section of this thesis, two cases are currently not being handled properly by the OOX verification tool. Both cases have very similar origins since they are caused by assumptions over symbolic references not being considered. There are multiple ways to tackle this problem, including checking the assumptions over each iteration where a dependency has to be checked. This method would cause a significant increase in invocations of the theorem prover and is unlikely to have a performance benefit.

Another method is introducing a new statement or operator that allows us to reason about symbolic references. This is a very similar idea to the operations that were introduced in the paper Separation logic [Reynolds, 2002]. A separation operator guarantees that two symbolic references do not point to the same part of the heap and, with that, alters the alias table. This could allow the monotonic partial order reduction to ignore possible conflicts in aliases, and with that, more paths could be pruned. The same would go for locking and unlocking statements, not causing deadlocks when two symbolic references that are being locked have aliases left. This method also has its own downsides since if an assumption is made over the value of symbolic references with the not equals operator, it could be said that automatically, the two symbolic references do not point to the same object.

There are many more possibilities, and therefore, some future work can be done on handling assumptions and symbolic references within symbolic execution over concurrent programs as well as monotonic partial order reduction.

Due to the extension and implementation of monotonic partial order reduction being quite complex, there was not much time to research other methods of partial order reduction as well as completely different techniques like deductive systems using concurrent separation logic [O'Hearn, 2007]. Future work can be done towards a more inclusive comparison of monotonic partial order reduction compared to other verification methods and state graph reduction.

Bibliography

[Baldoni et al., 2018] Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., and Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3).

[Baranová et al., 2017] Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., and Štill, V. (2017). Model checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis*, volume 10482 of *LNCS*, pages 201–207. Springer.

[Blom and Huisman, 2014] Blom, S. and Huisman, M. (2014). The vercors tool for verification of concurrent programs. In Jones, C., Pihlajasaari, P., and Sun, J., editors, *FM 2014: Formal Methods*, pages 127–131, Cham. Springer International Publishing.

[Cadar et al., 2008] Cadar, C., Dunbar, D., and Engler, D. (2008). KLEE: Unassisted and automatic generation of High-Coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA. USENIX Association.

[Cordeiro et al., 2019] Cordeiro, L., Kroening, D., and Schrammel, P. (2019). Jbmc: Bounded model checking for java bytecode. In Beyer, D., Huisman, M., Kordon, F., and Steffen, B., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 219–223, Cham. Springer International Publishing.

[Godefroid, 1991] Godefroid, P. (1991). Using partial orders to improve automatic verification methods. In Clarke, E. M. and Kurshan, R. P., editors, *Computer-Aided Verification*, pages 176–185, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Godefroid, 1996] Godefroid, P., editor (1996). *Persistent sets*, pages 41–73. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Gotsman et al., 2011] Gotsman, A., Berdine, J., and Cook, B. (2011). Precision and the conjunction rule in concurrent separation logic. *Electr. Notes Theor. Comput. Sci.*, 276:171–190.

[Hoare, 1978] Hoare, C. A. R. (1978). *An Axiomatic Basis for Computer Programming*, pages 89–100. Springer New York, New York, NY.

[Holzmann, 2018] Holzmann, G. J. (2018). *Explicit-State Model Checking*, pages 153–171. Springer International Publishing, Cham.

[Kahlon et al., 2009] Kahlon, V., Wang, C., and Gupta, A. (2009). Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In Bouajjani, A. and Maler, O., editors, *Computer Aided Verification*, pages 398–413, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Koppier, 2020] Koppier, S. (2020). The path explosion problem in symbolic execution.

[Kroening et al., 2023] Kroening, D., Schrammel, P., and Tautschnig, M. (2023). Cbmc: The c bounded model checker.

[Kroening and Tautschnig, 2014] Kroening, D. and Tautschnig, M. (2014). Cbmc – c bounded model checker. In Ábrahám, E. and Havelund, K., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg. Springer Berlin Heidelberg.

[O'Hearn, 2007] O'Hearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307. Festschrift for John C. Reynolds's 70th birthday.

[Peled, 1993] Peled, D. (1993). All from one, one for all: on model checking using representatives. In Courcoubetis, C., editor, *Computer Aided Verification*, pages 409–423, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Reynolds, 2002] Reynolds, J. (2002). Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74.

[Valmari, 1990] Valmari, A. (1990). A stubborn attack on state explosion. In Clarke, E. M. and Kurshan, R. P., editors, *Computer Aided Verification, 2nd International Workshop, CAV 90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer.

[Valmari, 1991] Valmari, A. (1991). Stubborn sets for reduced state space generation. In *Applications and Theory of Petri Nets*.

[van den Bos and Huisman, 2022] van den Bos, P. and Huisman, M. (2022). *The Integration of Testing and Program Verification*, pages 524–538. Springer Nature Switzerland, Cham.

[Wang et al., 2008] Wang, C., Yang, Z., Kahlon, V., and Gupta, A. (2008). Peephole partial order reduction. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 382–396, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Zheng et al., 2015] Zheng, M., Rogers, M. S., Luo, Z., Dwyer, M. B., and Siegel, S. F. (2015). Civl: Formal verification of parallel programs. *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 830–835.

Appendix A

Benchmarks and Results

A.1 EXP-1: Benchmarks for Soundness

A.1.1 EXP-1.1: Invalid

```
1: class E1 {
2:   static void a(SharedVar var) {
3:     var.val := 5;
4:   }
5:   static void main() {
6:     SharedVar var := new SharedVar(0);
7:     fork E1.a(var);
8:     int x := var.val;
9:     assert(x == 5);
10:  }
11: }
```

A.1.2 EXP-1.2: Valid

```
1: class E2 {
2:   static void a(SharedVar var) {
3:     var.val := 5;
4:   }
5:   static void b(SharedVar var) {
6:     var.val := 3;
7:   }
8:   static void main() {
9:     SharedVar var := new SharedVar(0);
10:    fork E2.a(var);
11:    fork E2.b(var);
12:    join;
13:    int x := var.val;
```

```
14:       assert(x == 5 || x == 3);
15:    }
16: }
```

A.1.3 EXP-1.3: Invalid

```
 1: class E3 {
 2:    static void a(SharedVar var) {
 3:       int val := var.val;
 4:       var.val := val + 1;
 5:    }
 6:    static void main() {
 7:       SharedVar var := new SharedVar(0);
 8:       fork E3.a(var);
 9:       fork E3.a(var);
10:       join;
11:       int x := var.val;
12:       assert(x == 2);
13:    }
14: }
```

A.1.4 EXP-1.4: Valid

```
 1: class E4 {
 2:    static void a(SharedVar var) {
 3:       lock var;
 4:       int val := var.val;
 5:       var.val := val + 1;
 6:       unlock var;
 7:    }
 8:    static void main() {
 9:       SharedVar var := new SharedVar(0);
10:       fork E4.a(var);
11:       fork E4.a(var);
12:       join;
13:       int x := var.val;
14:       assert(x == 2);
15:    }
16: }
```

A.1.5 EXP-1.5: Invalid

```
1: class E5 {
2:    static void a(int[] var) {
3:       arr[2] := 5;
4:    }
```

```
 5:    static void b(int[] var) {
 6:      arr[2] := 6;
 7:    }
 8:    static void main() {
 9:      int[] arr := new int[5];
10:      arr[2] := 0;
11:      fork E5.a(arr);
12:      fork E5.b(arr);
13:      join;
14:      int x := arr[2]
15:      assert(x == 5);
16:    }
17: }
```

A.1.6 EXP-1.6: Valid

```
 1: class E6 {
 2:    static void a(int[] var) {
 3:      arr[2] := 5;
 4:    }
 5:    static void b(int[] var) {
 6:      arr[2] := 6;
 7:    }
 8:    static void main() {
 9:      int[] arr := new int[5];
10:      arr[2] := 0;
11:      fork E6.a(arr);
12:      fork E6.b(arr);
13:      join;
14:      int x := arr[2]
15:      assert(x == 5 || x == 6);
16:    }
17: }
```

A.1.7 EXP-1.7: Deadlock

```
 1: class E7 {
 2:    static void a(SharedVar var) {
 3:      lock var;
 4:    }
 5:    static void main() {
 6:      SharedVar dummy := new SharedVar(0);
 7:      fork E7.a(dummy);
 8:      fork E7.a(dummy);
 9:      join;
10:      assert(true);
```

```
11:   }
12: }
```

A.1.8 EXP-1.8: Valid

```
 1: class E8 {
 2:   static void a(SharedVar var) {
 3:     lock var;
 4:     unlock var;
 5:   }
 6:   static void main() {
 7:     SharedVar dummy := new SharedVar(0);
 8:     fork E8.a(dummy);
 9:     fork E8.a(dummy);
10:     join;
11:     assert(true);
12:   }
13: }
```

A.1.9 EXP-1.9: Invalid

```
 1: class E9 {
 2:   static void a(SharedVar var) {
 3:     var.val := 1;
 4:   }
 5:   static void b(SharedVar var, SharedVar var2) {
 6:     int x := var.val;
 7:     if (x == 0) {
 8:       var2.val = 1;
 9:     }
10:   }
11:   static void main() {
12:     SharedVar var1 := new SharedVar(0);
13:     SharedVar var2 := new SharedVar(0);
14:     fork E9.a(var1);
15:     fork E9.b(var2);
16:     join;
17:     int val2 := var2.val;
18:     assert(val2 == 1);
19:   }
20: }
```

A.1.10 EXP-1.10: Invalid

```
1: class E10 {
2:   static void a(SharedVar var) {
```

```
3:       int k := 0;
4:       while (k < 2) {
5:         int x := d.val;
6:         d.val := x + 1;
7:         k := k + 1;
8:       }
9:     }
10:    static void main() {
11:      SharedVar var := new SharedVar(0);
12:      fork E10.a(var);
13:      fork E10.a(var);
14:      join;
15:      int val := var.val;
16:      assert(val == 4);
17:    }
18: }
```

A.1.11 EXP-1.11: Valid

```
1: class E11 {
2:   static void a(SharedVar var) {
3:     int k := 0;
4:     while (k < 2) {
5:       lock var;
6:       int x := var.val;
7:       var.val := x + 1;
8:       unlock var;
9:       k := k + 1;
10:    }
11:  }
12:  static void main() {
13:    SharedVar var := new SharedVar(0);
14:    fork E11.a(var);
15:    fork E11.a(var);
16:    join;
17:    int val := var.val;
18:    assert(val == 4);
19:  }
20: }
```

A.1.12 EXP-1.12: Invalid

```
1: class E12 {
2:   static void a(SharedVar var) {
3:     int val := var.val;
4:     if (val == 0) {
```

```
5:          var.val := 10;
6:       }
7:    }
8:    static void main() {
9:       SharedVar var := new SharedVar(0);
10:      fork E12.a(var);
11:      var.val := 1;
12:      join;
13:      int val := var.val;
14:      assert(val1 == 10);
15:   }
16: }
```

A.1.13 EXP-1.13: Valid

```
1: class E13 {
2:    V10() {}
3:    static void a(int[] var) {
4:       int k := 0;
5:       while (k < #a) {
6:          lock this;
7:          int x := a[i];
8:          a[i] := x + 1;
9:          unlock this;
10:         k := k + 1;
11:      }
12:   }
13:   static void b(int[] var) {
14:      int k := 0;
15:      while (k < #a) {
16:         lock this;
17:         int x := a[i];
18:         a[i] := x + 2;
19:         unlock this;
20:         k := k + 1;
21:      }
22:   }
23:   void main(int[] a, int k) requires(a ≠ null && #a > 1 && #a < 10 && 0 <= k && k < #a) {
24:      a[k] := 10;
25:      E13 g = new V10();
26:      fork g.a(var);
27:      fork g.b(var);
28:      join;
29:      int val := a[k];
30:      assert(val == 13);
31:   }
```

```
32: }
```

A.2 EXP-2: Benchmarks for Symbolic References

A.2.1 EXP-2.1: Valid

```
1: class E1 {
2:   static void SetZero(Foo x) {
3:     x.val := 0;
4:   }
5:   static void SetOne(Foo y) {
6:     y.val := 1;
7:   }
8:   static void main(Foo x, Foo y) {
9:     assume(x != y);
10:    fork E1.SetZero(x);
11:    fork E1.SetOne(y);
12:    join;
13:    int out := x.val;
14:    assert(out == 0);
15:  }
16: }
```

A.2.2 EXP-2.2: Invalid

```
1: class E2 {
2:   static void SetZero(Foo x) {
3:     x.val := 0;
4:   }
5:   static void SetOne(Foo y) {
6:     y.val := 1;
7:   }
8:   static void main(Foo x, Foo y) {
9:     fork E2.SetZero(x);
10:    fork E2.SetOne(y);
11:    join;
12:    int out := x.val;
13:    assert(out == 0);
14:  }
15: }
```

A.2.3 EXP-2.3: Valid

```
1: class E3 {
2:   static void SetZero(int[] arr, int key) {
3:     arr[key] := 0;
4:   }
5:   static void SetOne(int[] arr, int key) {
```

```
6:      arr[key] := 1;
7:    }
8:    static void main(int x, int k) requires(x ≤ 0&&x < 5&&k ≤ 0&&k < 5) {
9:      assume(x != k);
10:     int[] arr := new int[5];
11:     fork E3.SetOne(arr, x);
12:     fork E3.SetOne(arr, y);
13:     join;
14:     int out := arr[x];
15:     assert(out == 0);
16:   }
17: }
```

A.2.4 EXP-2.4: Invalid

```
1: class E4 {
2:    static void SetZero(int[] arr, int key) {
3:      arr[key] := 0;
4:    }
5:    static void SetOne(int[] arr, int key) {
6:      arr[key] := 1;
7:    }
8:    static void main(int x, int k) requires(x ≤ 0&&x < 5&&k ≤ 0&&k < 5) {
9:      int[] arr := new int[5];
10:     fork E4.SetOne(arr, x);
11:     fork E4.SetOne(arr, y);
12:     join;
13:     int out := arr[x];
14:     assert(out == 0);
15:   }
16: }
```

A.2.5 EXP-2.5: Valid

```
1: class E5 {
2:    static void Lock(Foo x) {
3:      lock x;
4:    }
5:    static void main(Foo x, Foo y) {
6:      assume(x != y);
7:      fork E5.Lock(x);
8:      lock y;
9:      join;
10:     assert(true);
11:   }
12: }
```

A.2.6 EXP-2.6: Deadlock

```
1: class E6 {
2:    static void Lock(Foo x) {
3:       lock x;
4:    }
5:    static void main(Foo x, Foo y) {
6:       fork E6.Lock(x);
7:       lock y;
8:       join;
9:       assert(true);
10:   }
11: }
```

A.3 EXP-4: Benchmark, Valid Philosophers Dining Problem

```
 1: class Philosophers {
 2:   static void main() {
 3:     Mutex mainLock := new Mutex();
 4:     int n := 4;
 5:     Fork[] forks := new Fork[n];
 6:     int i := 0;
 7:     while (i < n) {
 8:       Fork f := new Fork();
 9:       forks[i] := f;
10:       i := i + 1;
11:     }
12:     i := 0;
13:     while (i < n) {
14:       fork Philosophers.eat(mainLock, left, right, i);
15:       i := i + 1;
16:     }
17:     join;
18:   }
19:   static void eat(Mutex mainLock, Fork[] forks, int i) {
20:     Fork left := forks[i];
21:     Fork right := forks[(i + 1) % n];
22:     while(true) {
23:       // Philosopher is Thinking
24:       lock mainLock;
25:       lock left;
26:       lock right;
27:       // Philosopher is Eating
28:       unlock mainLock;
29:       unlock left;
30:       unlock right;
31:     }
32:   }
33: }
```

A.4 EXP-5: Benchmark, Invalid Philosophers Dining Problem

```
 1: class Philosophers {
 2:   static void main() {
 3:     int n := 4;
 4:     Fork[] forks := new Fork[n];
 5:     int i := 0;
 6:     while (i < n) {
 7:       Fork f := new Fork();
 8:       forks[i] := f;
 9:       i := i + 1;
10:     }
11:     i := 0;
12:     while (i < n) {
13:       fork Philosophers.eat(left, right, i);
14:       i := i + 1;
15:     }
16:     join;
17:   }
18:   static void eat(Fork[] forks, int i) {
19:     Fork left := forks[i];
20:     Fork right := forks[(i + 1) % n];
21:     while(true) {
22:       // Philosopher is Thinking
23:       lock left;
24:       lock right;
25:       // Philosopher is Eating
26:       unlock left;
27:       unlock right;
28:     }
29:   }
30: }
```

A.5 EXP-6: Benchmark, Valid Concurrent Merge Sort

```
1: class Main {
2:   static int[] sort(int[] array) requires(!(array == null))
3:   ensures(forall v, i : retval : forall w, j : retval : i < j ==> v ≤ w) exceptional(false)
4:   {
5:     Main.mergesort(array, 0, #array - 1);
6:     assert(false);
7:     return array;
8:   }
9:   static void mergesort(int[] array, int left, int right) exceptional(false)
10:  {
11:    if (left < right) {
12:      int middle := (left + right) / 2;
13:      fork Main.mergesort(array, left, middle);
14:      Main.mergesort(array, middle + 1, right);
15:      join;
16:      Main.merge(array, left, middle, right);
17:    }
18:  }
19:  static void merge(int[] array, int left, int middle, int right) exceptional(false)
20:  {
21:    int[] temp := new int[right - left + 1];
22:    int i := left;
23:    int j := middle + 1;
24:    int k := 0;
25:    while (i ≤ middle  &&  j ≤ right) {
26:      int arrayI := array[i];
27:      int arrayJ := array[j];
28:      if (arrayI ≤ arrayJ) {
29:        temp[k] := array[i];
30:        k := k + 1;
31:        i := i + 1;
32:      } else {
33:        temp[k] := array[j];
34:        k := k + 1;
35:        j := j + 1;
36:      }
37:    }
38:    while (i ≤ middle) {
39:      temp[k] := array[i];
40:      k := k + 1;
41:      i := i + 1;
42:    }
43:    while (j ≤ right) {
44:      temp[k] := array[j];
```

```
45:           k := k + 1;
46:           j := j + 1;
47:       }
48:       i := left;
49:       while (i ≤ right) {
50:           array[i] := temp[i - left];
51:           i := i + 1;
52:       }
53:   }
54: }
```

A.6 EXP-4: Results Valid Philosophers Dining Problem

Philosophers	NOPOR		SPOR		MPOR	
Depth (k)	Avg. Time (s)	Paths	Avg. Time (s)	Paths	Avg. Time (s)	Paths
20	0.26	1	0.35	1	0.27	1
40	0.27	5	0.26	5	0.26	5
80	0.27	18	0.27	8	0.26	8
90	1.77	3347	0.26	13	0.27	15
95	33.87	48479	0.28	48	0.28	37
100	900.26	2041193	0.33	146	0.30	77
105			1.33	1436	0.43	264
110			17.31	22375	0.66	665
115			147.87	882280	1.54	1908
120					21.85	30639
125					93.92	161025
130					487.03	510520

A.7 EXP-6: Results Valid Philosophers Dining Problem

Mergesort	NOPOR		SPOR		MPOR	
Depth (k)	Avg. Time (s)	Paths	Avg. Time (s)	Paths	Avg. Time (s)	Paths
20	0.07	112	0.02	18	0.06	40
30	13.45	69187	0.07	28	0.08	155
31	38.43	198795				
32	107.11	547295				
40			0.08	70	0.13	277
80			1.24	4285	0.30	813
100			2.99	8815	0.51	1047
120			5.11	13585	0.52	1077
140			11.09	31765	0.56	1183
150			15.14	45355	0.59	1267
160			20.63	56226	0.60	1345
170			25.17	69655	0.61	1411