SCALE: Source Code Analyzer for Locating Errors

Thesis by
Mihai Florian

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

2010
(Submitted April 16, 2010)
To my family and friends.
Acknowledgements

I would like to show my gratitude to my advisor, Dr. Gerard J. Holzmann, for his distinguished mentorship, for many insightful discussions, and for giving me the great opportunity to work with him. I would also like to thank Prof. K. Mani Chandy, for his valuable advice and support. I want to thank Dr. Klaus Havelund and Dr. Rajeev Joshi for sharing some of their ideas. I am thankful to my parents, Alexandru and Lodovica, for their love and continuous support. This research has been supported by The Lee Center for Advanced Networking.
Abstract

This thesis presents the design and implementation of SCALE, a tool for systematic software testing multi-threaded C applications that use the pthread library. SCALE exhaustively explores the non determinism introduced by thread schedulings and tries to find violations of safety properties. We have designed SCALE to be flexible so that it is easy to add and combine different exploration and state space reduction algorithms. In this thesis we describe the currently implemented reduction algorithms, of which the most important ones are local execution cycle detection and super step partial order reduction. To exemplify how SCALE can be used, we have applied it to a few multi-threaded applications, measured its performance and compared the results to those obtained by other tools. While checking the implementation of a non-blocking queuing algorithm, we were able to find a previously unknown bug that appears only in some unexpected thread inter-leavings.
Contents

Acknowledgements iv

Abstract v

1 Introduction 1

2 Related Work 3

3 Multi-Threaded Applications 5
  3.1 Dynamic Semantics ........................................... 5
  3.2 State Representation ......................................... 8

4 State Space Exploration 16
  4.1 Exploration Algorithms ....................................... 16
  4.2 State Filters .................................................. 19
     4.2.1 The NotSeenBefore Filter ............................... 20
     4.2.2 The DepthBounding Filter ............................... 20
     4.2.3 The LocalCycleDetection Filter ....................... 22
     4.2.4 The SuperStepPOR Filter ............................... 29

5 SCALE - Design and Implementation 35
  5.1 Static Analysis and Code Instrumentation .................. 35
  5.2 The SCALE Scheduling Protocol ............................ 36
     5.2.1 Communication Channels ............................... 36
     5.2.2 Commands and Command Results ....................... 37
  5.3 The SCALE Scheduler ....................................... 38
     5.3.1 Backend ................................................ 39
     5.3.2 Frontend ................................................. 39
     5.3.3 Generic Scheduler ...................................... 40
     5.3.4 Scheduling Strategy .................................... 41
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The memory layout of an application</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Exploration using state filters</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Extending a path with an action $a_n$</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>The super step partial order reduction</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>SCALE’s static analysis and code instrumentation process</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Scheduling protocol overview</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>The SCALE Scheduler</td>
<td>39</td>
</tr>
<tr>
<td>5.4</td>
<td>The SCALE Simulator</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>Number of runs up to the first counterexample for the non-blocking queue implementation</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>Replay time vs. state depth for the non-blocking queue implementation</td>
<td>46</td>
</tr>
<tr>
<td>6.3</td>
<td>Number of distinct state representations for the non-blocking queue implementation</td>
<td>46</td>
</tr>
<tr>
<td>6.4</td>
<td>Verification results for the indexer example</td>
<td>48</td>
</tr>
<tr>
<td>6.5</td>
<td>Verification results for the thread pool example</td>
<td>49</td>
</tr>
<tr>
<td>6.6</td>
<td>Verification results for the reusable barrier example</td>
<td>51</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Error free multi-threaded applications are hard to write. Concurrency related errors, might be revealed only in unexpected and improbable thread inter-leavings. Even if run for a large amount of time, traditional testing might still miss such errors because it cannot control how the operating system schedules the threads. When an error is found, it cannot be analyzed because there is no way to reliably reproduce it. Model checking [6] is one way to address these problems.

A model checker takes as input a finite state model of a system and a specification, and by exhaustively exploring the state space of the model, it verifies whether it meets the specification or not. As all states are visited, an error state cannot be missed. If the specification can be violated then the model checker generates a counter example that reveals the error. By replaying the counter example, the bug can be reproduced and analyzed.

The size of an application’s state space grows rapidly with the number and the size of its concurrent components. This is commonly known as the state space explosion problem. Without some form of state space reduction, even for relatively small systems, model checking is infeasible. Partial order reduction [24, 9, 21] is a technique that was successfully used to reduce the number of states explored. Also, by abstracting the system, its state space can be further reduced. Usually the abstracted system and the original will not have exactly the same properties, but some errors might show up in both. When an error not present in the original system is found in the abstracted system, the abstraction can be refined [2] to account for the difference. Yet another approach [4] is to implicitly represent the state graph as a boolean formula in propositional logic.

Unlike testing, model checkers usually work on a finite state abstract model of the application, and can verify both safety and liveness properties. The abstract model can hide some of the bugs, and it can introduce false positives, errors that can happen in the abstract model but cannot happen in the actual application. Also, every time the application’s source code changes, the abstracted model has to be updated.

Systematic software testing is a hybrid approach which, like testing, checks the actual application, but also, like model checking, exhaustively explores the non determinism introduced by thread
schedulings. Much like model checking, systematic software testing suffers from the state space explosion problem: the number of different thread schedulings that have to be tried increases very rapidly with the number and the size of the concurrent components. One drawback of this approach is that it usually does not check liveness properties, it only checks for common safety violations like absence of deadlock, race conditions, and assertion violations.

In this thesis we describe the design of the SCALE (Source Code Analyzer for Locating Errors), a tool for systematic software testing multi-threaded Unix/Linux applications written in C using the pthread library. By exhaustively exploring the non determinism introduced by thread interleavings, it tries to find and report common concurrency errors such as deadlocks, and assertion failures. SCALE can also be used to check safety properties specified as invariants on the global state of the application. We address the state space explosion problem by chaining various state space reduction algorithms. This thesis describes in detail two of the algorithms: local execution cycle detection and super step partial order reduction.

As SCALE checks the actual application, not an abstracted model of the application, it ensures that no spurious violations are reported. All errors that are found are real errors that can manifest when the application is run. When SCALE detects a property violation, it generates a counter example that can later be replayed to reproduce the bug.

We have successfully applied SCALE to a few multi-threaded applications, and we were able to find concurrency related errors. For example we have found a bug in a non-blocking queuing algorithm. Also, by applying our tool to correct systems, we were able to measure the amount of reduction achieved using the local execution cycle detection and super step partial order reduction algorithms. We compare our results to those obtained by other tools and we show that, in some cases, SCALE requires fewer runs than other tools to verify the same applications.

This thesis is organized as follows: In Chapter 2 we briefly present related work in software model checking and systematic software testing. Chapter 3 introduces the multi-threaded model assumed by SCALE. In Chapter 4 we present the state space exploration and state space reduction algorithms implemented in our tool. Chapter 5 presents the design and the implementation of SCALE. Chapter 6 shows some preliminary results obtained by applying our tool to some multi-threaded applications. Finally, in Chapter 7 we conclude and present possible future directions.
Chapter 2

Related Work

A wide variety of model checking and systematic software testing tools that work on C programs have been implemented. Usually, model checkers can verify a larger class of properties specified in temporal logics. Systematic software testing tools only check for safety properties like absence of deadlock, race conditions, and assertion failures.

Another difference between the various tools is how they represent the explored state space. Some tools explicitly store application states. Others store the state space implicitly as a set of constraints. There are even tools that do not store states at all.

Berkeley Lazy Abstraction Software Verification Tool (BLAST)\cite{2} is a model checker for C programs that does counterexample driven automatic abstraction refinement. It constructs an abstract model of the program, which is then model checked for safety properties. If spurious errors are detected, the abstract model is refined and checked again.

CHESS\cite{19} is the first tool to introduce context switch bounding. It verifies multi-threaded C, C++ and .NET code written as test cases. A state is encoded as the happens-before graph of the partial order equivalent executions that lead to that state, thus achieving some partial order reduction. Backtracking to a previous visited state is done by cleaning up the resources used by the test case, restarting it and following a path that leads to the desired state. CHESS checks safety properties like deadlock and data races, and can also find some non progress cycles. CHESS has been successfully used to verify Windows device drivers and also the kernel of a new operating system.

C Model Checker (CMC)\cite{16} has been successfully used to verify protocol code written in C. CMC stores states explicitly and uses heuristics to prioritize the state space search. It is also able to save and restore previously visited states. CMC can check for memory leaks, assertion failures, and global invariants.

Inspect\cite{25} is a systematic software testing tool for checking multi-threaded programs written in C or C++ using the pthread library. It performs a static analysis to identify shared data and then it instruments the program to enable the verification. During state space exploration, Inspect doesn’t store application states, but it uses dynamic partial order reduction \cite{7} to reduce the number
of application runs it has to try. Inspect can find common errors like deadlocks, data races and assertion violations.

pan_cam[27] is an explicit state model checker that verifies C programs compiled down to LLVM bytecode [15]. It uses SPIN as a backend to drive the state space exploration. pan_cam uses super step partial order reduction and context switch bounding to reduce the number of states explored. Taking advantage of the SPIN backend, pan_cam can verify both safety and liveness properties expressed as Promela never claims or LTL[22] formulas.

SPIN[11] is an explicit state model checker generator which can be used to verify models written in Promela (Process Meta Language). Since version 4.0 it allows C code to be embedded in the Promela model. Spin can be used to verify both safety and liveness properties. The properties are specified using never claims which represent Buchi automata. Spin can also convert an LTL formula to the corresponding never claim. A wide range of techniques are used to reduce the state space and the memory occupied by the states: partial order reduction, state compression and bitstate hashing.

VeriSoft[10] is a systematic software testing tool that verifies C code. It performs a stateless search and uses partial order reduction to reduce the number of times it has to run the application. VeriSoft can be used to check for assertion violations and coordination problems like deadlocks.
Chapter 3

Multi-Threaded Applications

3.1 Dynamic Semantics

A Multi-Threaded Application is a finite set $P$ of threads. We define the state space of the multi-threaded application using a dynamic semantics similar to the ones introduced in [10, 7].

Each thread executes a sequence of statements described by a deterministic sequential piece of C code. The code is deterministic in the sense that presented with the same data, it will execute the same sequence of statements.

Initially, the application has only one active thread, called the main thread. The main thread executes the sequence of statements found in the main function of the application’s C source code. For another thread to become active, it must first be created by executing a "create" statement from an already active thread. A newly created thread $p$ executes the sequence of statements found in the start routine that was specified in the statement that created $p$.

An active thread that has ended and has no more statements to execute becomes a dead thread. Dead threads cannot be created again, and cannot become active ever again. $P$, the set of all threads, is finite and ensures that only a finite number of threads are ever created.

Only active threads can execute statements.

Threads communicate with each other by accessing, in their statements, a finite set of communication variables. Each access to a communication variable is atomic. We distinguish two classes of communication variables: (a) data variables such as global variables and the heap memory, and (b) synchronization variables such as threads, mutexes, and condition variables.

A statement that accesses a communication variable or is an assertion is called a global statement. Any other statement is called a local statement.

At any time the multi-threaded application is said to be in a state called the current state. The execution of a statement in the current state is said to block if it cannot currently be completed. The execution of a local statement is not allowed to block.

The application is in a global state if for every active thread, the next statement to be executed
is a global statement. A thread exit is considered to be a global statement, and this ensures that each active thread has at least one global statement. We assume that there are no infinite sequences of local statements, so each thread, after becoming active, eventually reaches a global statement.

The initial global state is the state reached when the main thread reaches the first global statement. The creation of a new thread is a global statement, so the main thread is the only active thread in the initial global state. Since the code for the main thread is deterministic and no global statements have been executed before the initial global state is reached, the initial global state is unique and from now on we refer to it as $s_0$.

A thread transition $t$ moves the application from one global state $s$ to another global state $s'$ by executing statements from a single thread: a global statement followed by a finite sequence of local statements up to (but not including) the next global statement. Formally, we write $s \xrightarrow{t} s'$. Let $T$ be the set of all transitions.

A transition is disabled in a global state $s$ if the execution of its global statement in $s$ would block. A transition that is not disabled in a global state $s$ is said to be enabled in $s$. Note that the execution of an enabled transition always terminates since the global statement won’t block, the number of local statements is finite, and each local statement can’t block.

A global state $s'$ is said to be reachable from a global state $s$ if there exists a finite sequence of transitions $w$ which, taken in order, lead the multi-threaded application from $s$ to $s'$. Formally, we write $s \xrightarrow{w} s'$.

A multi-threaded application is a closed system evolving from its initial global state $s_0$ by executing enabled transitions from $T$. The environment, if any, is either included in the application, or it doesn’t store any state. The execution terminates when it reaches a global state in which no transitions are enabled. If the execution terminates in a state $s$ in which there are still active threads, then $s$ is considered to be a deadlock state.

The above observations impose the following restrictions on the types of C programs that can be verified by SCALE:

- each thread runs a deterministic sequence of C statements;
- there are no infinite loops that contain only local statements;
- there are no other blocking statements besides the pthread operations;
- if during execution, the application reaches a state $s$, with the environment being in a state $e$, then any execution that leads to state $s$ also ensures that the environment is in state $e$.

SCALE assumes that the application satisfies the above restrictions and has no mechanism to check or enforce them.
Definition 1. A multi-threaded application is represented by a transition system $A_G = (S, \rightarrow, s_0)$ where:

- $S$ is the set of reachable global states of the multi-threaded application;
- $\rightarrow \subseteq S \times S$ is the transition relation, $(s, s') \in \rightarrow$ if and only if $\exists t \in T.s \xrightarrow{t} s'$;
- $s_0$ is the initial global state of the multi-threaded application.

Let us fix a multi-threaded application $A_G$ for the rest of this chapter.

Definition 2. A safety property is considered to be one of the following:

- absence of deadlock, which is violated when the multi-threaded application reaches a deadlock state;
- assertion, which is violated when the assertion fails;
- global invariant, which is a boolean function on the communication variables, and it is violated when the function evaluates to false.

Theorem 1. If there exists a reachable state of the multi-threaded application that violates a safety property, then there exists a global state in $A_G$ where the same safety property is violated.

Proof. Let $s$ be a reachable state where a safety property is violated.

If $s$ is a deadlock state, then for each thread, the next statement is a global statement because only global statements can block. So $s$ must be a global state in $A_G$.

If $s$ is an assertion violation, then let $p_i$ be the active thread whose next statement is the assertion. Assertions are global statements, so $p_i$’s next statement is a global statement. If for every other active thread $p_j$, $j \neq i$, we execute all the statements up to its next global statement, then we end up in a global state $s'$ in $A_G$. Note that such a global statement must exist for every active thread because the last statement of a thread must be a thread exit which is a global statement. The assertion violation in thread $p_i$ is not affected by the execution of local statements in other threads so the assertion is still violated in $s'$.

If $s$ is a state in which an invariant is violated, then by executing the local statements in every thread up to the next global statement we end up in a global state $s'$ in $A_G$. Local statements cannot change communication variables, and invariants only refer to communication variables, so the invariant is still violated in $s'$.

Considering the above result, from now on, when we use the term state we mean a global state in $A_G$. 

\[\square\]
3.2 State Representation

**Definition 3.** States are elements from the set \( \text{State} \) where:

\[
\text{State} = \text{SharedState} \times \text{LocalStates},
\]

and

\[
\text{LocalStates} = P \rightarrow \text{LocalState}^2, \quad \text{where} \quad \text{LocalState}^2 = \text{LocalState} \cup \{\bot\}.
\]

For a state \( s = (g,ls) \)

- \( g \in \text{SharedState} \) contains the values for all communication variables.
- \( ls \in \text{LocalStates} \) is a function that takes a thread as an argument and returns its local state. If \( p \) is an active thread then \( ls(p) \) contains the values for all the local variables of \( p \), and also, the values of all the arguments passed to the functions that are currently on \( p \)'s call stack. If \( p \) is not an active thread (either it was not created, or it is dead) then \( ls(p) = \bot \).

**Definition 4.** A transition from a thread \( p \) in a local state \( l \) is a function:

\[
t_{p,l} : \text{SharedState} \rightarrow \text{LocalState}^2 \times \text{SharedState} \times (P \times \text{LocalState})^2.
\]

If in a state \( s = (g,ls) \) we have an enabled transition \( t_{p,l} \), with \( t_{p,l}(g) = (l', g', n) \), then the state \( s' = (g', ls') \) reached by taking the transition \( t_{p,l} \) in \( s \) is obtained as follows:

- \( g' \) is the shared state returned by the transition.
- If \( l' = \bot \), then thread \( p \) has just exited and now \( ls'(p) = \bot \). In this situation we must have \( n = \bot \) and \( g' \) must contain, in the corresponding communication variable, the fact that thread \( p \) has ended.
- If \( l' \neq \bot \) and \( n = \bot \), then \( ls'(p) = l' \). The local state of thread \( p \) has changed.
- If \( l' \neq \bot \) and \( n = (p', l'') \), then \( ls'(p) = l' \), \( ls'(p') = l'' \) and \( g' \) must contain, in the corresponding communication variable, the fact that thread \( p' \) was created.
- If \( n = \bot \), then for every thread \( i \in P, i \neq p \) we have \( ls'(i) = ls(i) \).
- If \( n = (p', l'') \), then for every thread \( i \in P, i \neq p \land i \neq p' \), we have \( ls'(i) = ls(i) \). A transition from one thread cannot affect the local states of other threads, except the local state of a newly created thread.

To represent a state \( s = (g,ls) \) we would like to store the values of all communication variables and also the values of all local variables of each thread. We consider that such an approach is not feasible because it uses too much much memory per stored state. Instead we have opted for a state representation similar to the one introduced in [18, 19].

**Definition 5.** Two transitions \( t_1 \) and \( t_2 \), in a state \( s \), are independent if:

1. they are both enabled and executing one cannot disable the other one,
2. both the execution of the sequence \( t_1; t_2 \), and the execution of the sequence \( t_2; t_1 \) from \( s \) lead the multi-threaded application to the same state \( s' \), and

3. neither \( t_1 \) nor \( t_2 \) are assertions or change the truth value of any global invariant.

Two transitions that are not independent are dependent. Let \( D \subseteq T \times T \) be the dependency relation between transitions.

A similar definition of independence was originally introduced in [13].

**Definition 6.** Two sequences of transitions are partial order equivalent with respect to an independence relation, if they can be obtained from each other by successively exchanging adjacent independent transitions.

**Lemma 1.** Let \( w \) be a finite sequence of transitions, and let \( w' \) be another sequence of transitions obtained from \( w \) by exchanging two adjacent independent transitions. If \( s \xrightarrow{w} s' \), then \( s \equiv w' \xrightarrow{s'} \).

**Proof.** Let \( w = t_1 t_2 \ldots t_i t_{i+1} \ldots t_n \), and let \( w' = t_1 t_2 \ldots t_{i+1} t_i \ldots t_n \) be obtained by swapping the adjacent independent transitions \( t_i \) and \( t_{i+1} \).

We split \( w \) in the following three parts: \( w_1 = t_1 \ldots t_{i-1} \), \( w_2 = t_i t_{i+1} \) and \( w_3 = t_{i+2} \ldots t_n \).

We also split \( w' \) in the following three parts: \( w'_1 = t_1 \ldots t_{i-1} \), \( w'_2 = t_i t_{i+1} \) and \( w'_3 = t_{i+2} \ldots t_n \).

Note that \( w_1 = w'_1 \) and \( w_3 = w'_3 \).

Let \( s \xrightarrow{w_1} s'' \), we also have \( s \xrightarrow{w'_1} s'' \).

\( t_i \) and \( t_{i+1} \) are independent so we must have \( s'' \xrightarrow{w_2} s''' \) and \( s'' \xrightarrow{w'_2} s''' \).

We know that \( s \xrightarrow{w} s' \) and we have \( \exists s \xrightarrow{w''} s''' \) so we must have \( s''' \xrightarrow{w} s' \).

Because \( w_3 = w'_3 \) we also have \( s''' \xrightarrow{w_3} s' \).

Combining the parts we have: \( s \xrightarrow{w'_1} s'' \xrightarrow{w'_2} s''' \xrightarrow{w_3} s' \), so \( s \equiv w' \xrightarrow{s} s' \).

**Lemma 2.** If \( w \) and \( w' \) are two partial order equivalent sequences of transitions and \( s \equiv w \xrightarrow{s} s' \), then \( s \equiv w' \xrightarrow{s} s' \).

**Proof.** From the definition of partial order equivalence we know there exists a finite sequence of sequences of transitions: \(<w = w_1, w_2, \ldots, w_n = w'> \) such that for each \( 0 < i < n \), \( w_{i+1} \) is obtained from \( w_i \) by exchanging two independent transitions.

We repeatedly apply Lemma 1 for each pair \((w_i, w_{i+1})\), starting with \((w_1, w_2)\), and at the end we get that \( s \xrightarrow{w'} s' \).

**Definition 7.** A happens-before graph[14], corresponding to a finite sequence of transitions \( w = t_1 t_2 \ldots t_n \), starting from \( s_0 \), and a dependency relation \( D \subseteq T \times T \), is a graph \( HB(w) = (A, H) \) where:

- \( A = \{ t_i \mid 0 < i \leq n \} \) is the set of nodes, one node per each transition in \( w \). If it happens that the same transition appears twice in \( w \), then we have two nodes for it, and we distinguish between them considering the order in which they appear.
Proof. Let \( H = \{(t_i, t_j) \mid 0 < i < j \leq n \land (t_i, t_j) \in D^+\} \), the set of edges, is the transitive closure of the dependency relation \( D \) on the set \( A \).

**Lemma 3.** Two sequences of transitions \( w_1 \) and \( w_2 \), starting from \( s_0 \), are partial order equivalent if and only if \( HB(w_1) = HB(w_2) \).

Proof. Let \( HB(w_1) = (A_1, H_1) \) and \( HB(w_2) = (A_2, H_2) \). If \( w_1 \) is partial order equivalent to \( w_2 \), then they must contain the same transitions so we must have \( A_1 = A_2 \).

For any two independent transitions \( t_i \) and \( t_{i+1} \) from \( w_1 \), we have that \( (t_i, t_{i+1}) \notin H_1 \). If we swap them and form a new sequence \( w'_1 \), with \( HB(w'_1) = (A'_1, H'_1) \), then \( H'_1 = H_1 \).

\( w_1 \) and \( w_2 \) are partial order equivalent, so there exists a sequence of sequences of transitions \(<w_1 = v_1, v_2, ..., v_n = w_2>\) such that for each \( 0 < i < n \), \( v_{i+1} \) is obtained from \( v_i \) by exchanging two independent transitions. Repeatedly applying the previous argument for adjacent pairs \((v_i, v_{i+1})\), \( 0 < i < n \), we get that \( H_1 = H_2 \).

So we must have \( HB(w_1) = HB(w_2) \).

We prove by induction on the length of \( w_1 \) that if \( HB(w_1) = HB(w_2) \) then \( w_1 \) is partial order equivalent to \( w_2 \).

If \( HB(w_1) = HB(w_2) \), then \( A_1 = A_2 \), and so \( w_1 \) and \( w_2 \) contain the same transitions, maybe in a different order.

For a \( w_1 \) of length 0, \( w_2 \) must also have length 0 and so they are partial order equivalent.

For a \( w_1 = t_1 \) of length 1, we must also have \( w_2 = t_1 \) since they have the same transitions.

Assuming that for any \( w'_1 \) and \( w'_2 \), of length \( n - 1, n > 0 \), if \( HB(w'_1) = HB(w'_2) \), then \( w'_1 \) and \( w'_2 \) are partial order equivalent, we proceed to prove that for any \( w_1 \) and \( w_2 \) of length \( n \), if \( HB(w_1) = HB(w_2) \) then \( w_1 \) and \( w_2 \) are partial order equivalent.

We have two cases:

(a) \( w_1 = t_1...t_n \) and \( w_2 = t'_1...t'_n \) with \( t_n = t'_n \). We can remove the node that corresponds to \( t_n \) from both of \( HB(w_1) \) and \( HB(w_2) \) and we are left with the happens before graphs for \( t_1...t_{n-1} \) and \( t'_1...t'_{n-1} \). These graphs must be identical (because we've removed the same node from both) and so we can apply the induction hypothesis and get that \( t_1...t_{n-1} \) is partial order equivalent to \( t'_1...t'_{n-1} \).

Since \( t_n = t'_n \) we can conclude that \( w_1 \) and \( w_2 \) are partial order equivalent.

(b) \( w_1 = t_1...t_n \) and \( w_2 = t'_1...t'_n \) with \( t_n \neq t'_n \).

Because \( w_1 \) and \( w_2 \) have the same transitions, there exists a transition \( t'_i = t_n \). \( t_n \) is the last transition in \( w_1 \), meaning that it cannot have outgoing edges in \( H_1 \). Since \( H_1 = H_2 \), for any \( t'_j, j > i \) we have \( (t'_i, t'_j) \notin H_2 \land (t'_j, t'_i) \notin H_2 \). In other words, \( t'_i \) is independent with all the transitions that occur in \( w_2 \) from its place up to the end of \( w_2 \). We can repeatedly swap \( t'_i \) with these transitions until \( t'_i \) becomes the last transition.

We have constructed a new sequence, \( w'_2 \), which is partial order equivalent to \( w_2 \). Using the first
part of this proof, we know that $HB(w'_2) = HB(w_2)$, so $HB(w'_2) = HB(w_1)$, and moreover, $w'_2$ and $w_1$ have the same last transition. We can use case (a) to prove that $w_1$ and $w'_2$ are partial order equivalent, so $w_1$ and $w_2$ are partial order equivalent.

**Definition 8.** For a state $s$ and a finite sequence of transitions $w$ such that $s_0 \xrightarrow{w} s$, we say that $HB(w)$ is a representation of $s$.

Note that only reachable states can be represented. Also, there might be more than one representation for a single state, corresponding to different transition sequences that lead to it.

**Theorem 2.** Let’s consider a state $s_1$ that has a representation $(A_1, H_1)$ and a state $s_2$ that has a representation $(A_2, H_2)$. If $(A_1, H_1) = (A_2, H_2)$, then $s_1 = s_2$.

**Proof.** $s_1$ has a representation implies that there exists a sequence of transitions $w_1$ such that $s_0 \xrightarrow{w_1} s_1$ and $(A_1, H_1) = HB(w_1)$.

$s_2$ has a representation implies that there exists a sequence of transitions $w_2$ such that $s_0 \xrightarrow{w_2} s_2$ and $(A_2, H_2) = HB(w_2)$.

We have $HB(w_1) = HB(w_2)$ which implies, using Lemma 3, that $w_1$ and $w_2$ are partial order equivalent.

From Lemma 2 we know that two partial order equivalent sequences of transitions, if executed from the same state $s_0$, lead to the same state $s$, so we must have $s_1 = s_2$.

**Theorem 3.** Each state $s$ from $A_G$ has at least one representation.

**Proof.** $s$ is in $A_G$, and $A_G$ contains only the reachable states, so there must exist a $w$ such that $s_0 \xrightarrow{w} s$.

Then $HB(w)$ is a representation of $s$.

**Theorem 4.** If $A_G$ is acyclic, then each state $s$ from $A_G$ has only a finite number of representations.

**Proof.** From Graph Theory we know that if a graph is acyclic then it contains only a finite number of different paths.

For our case, there are a finite number of paths from $s_0$ to $s$, so there are a finite number of transition sequences $w$ such that $s_0 \xrightarrow{w} s$. Each such transition sequence $w$ can generate at most one different representation $HB(w)$ for $s$.

So $s$ can only have a finite number of representations.

In all the previous arguments we’ve assumed that we know the transitions and that we can store them. For a C program this would not be feasible. For example, a transition might involve some library calls, and it would be complicated to come up with a function that represents what the call does and compose it with all the other statements in the same transition.
Luckily, if we take a closer look at our arguments, the only thing that we have assumed is that we can identify which thread takes each transition, which transitions are enabled in the current state, and which transitions are dependent. Our definition of the state space $A_G$ requires us to also be able to apply a transition $t$ to a state $s$ and get the new state $s'$. We can do that by running the actual application. Assuming that the application is in the state $s$, we can schedule the thread that corresponds to transition $t$ and let it run up to the next global statement. The application is guaranteed now to be in state $s'$ because we have assumed that all transitions are deterministic.

Whenever we want to get to a certain state $s$, reachable through a sequence of transitions $w$, we can restart the application from its initial state $s_0$ and schedule, in order, the threads that correspond to the transitions in $w$. Because all the transitions are deterministic, we are guaranteed to end up in $s$. So what we only need in order to recreate $s$, is the sequence of threads that have to be scheduled from $s_0$. The same sequence of threads will always lead, when scheduled from $s_0$, to the same state $s$.

We also have to identify which transitions create new threads, and which transitions exit active threads.

We can over-approximate the dependency relation as follows.

**Definition 9.** Two transitions are dependent if any of the following is true:

- they access the same communication variable, and at least one of the transitions writes to the variable;
- at least one of the transitions is an assertion;
- at least one of the transitions accesses a communication variable used to calculate the value of at least one global invariant.

Any access to a synchronization variable is considered to be a write access.

We abstract all the information we need to know about a transition in an action.

**Definition 10.** For a transition $t$, the corresponding action is a triple $(p, op, vl)$ where:

- $p$ is the thread taking the transition $t$;
- $op$ is the operation performed in the transition $t$. For example, it can be a read, a write, a thread_create, a thread_exit, mutex_lock, etc.;
- $vl$ is a list of communication variables accessed by the transition $t$. For example, if the transition creates a new thread, then the variable list specifies which thread is created, if the transition reads a variable, then the variable list specifies which variable is read, etc.
An action $a$, which abstracts a transition $t$, is \textit{enabled} in a state $s$ if $t$ is enabled in $s$. An action that is not enabled is \textit{disabled}.

We can extend the dependency relation and the partial order equivalence, Definitions 9 and 6, from transitions to actions.

If we have $s_0 \xRightarrow{w} s$, then we can use a scheduling mechanism and the sequence of actions that corresponds to $w$ to recreate $s$. Also, every time we schedule the same sequence of actions from $s_0$, we are guaranteed to reach the same state $s$.

From what we have said before, we know that we could use a sequence of threads, or a sequence of actions, to represent a state $s$ that is reachable from $s_0$. But, by doing so, we would get different representations of $s$ even for executions that are partial order equivalent. As it was the case with transitions, we can use a happens-before graph to create a single representation for all partial order equivalent executions.

**Definition 11.** A \textit{happens-before graph for actions}, corresponding to a finite sequence of actions $q = a_1a_2...a_n$, is a graph $HBA(q) = (A,H)$ where:

- $A = \{a_i \mid 0 < i \leq n\}$ is the set of nodes, one node for each action. If the same action appears multiple times in the sequence we create a new node for each time it appears and we distinguish between them considering their order in the sequence.

- $H = \{(a_i,a_j) \mid 0 < i < j \leq n \land (a_i,a_j) \in D^+\}$, where $D^+$ is the transitive closure of the dependency relation $D$ for actions from Definition 9.

Definition 11 is similar to Definition 7, but it uses actions instead of transitions.

From now on, to represent a state $s$, reachable from $s_0$ by scheduling a sequence $q$ of actions, we will use the happens-before graph for actions $HBA(q)$.

Lemmas 1, 2, 3, and Theorems 2, 3, 4 also hold for sequences of actions and happens-before graphs for actions. The reason why this is true is that using actions and a scheduling mechanism, we can reconstruct both the effect of transitions and the dependency relation between transitions, assuming that transitions are deterministic.

Before we can present the state space exploration algorithms, we have to make some assumptions about the scheduling mechanism. In Chapter 5 we will show how such a mechanism can be implemented.

Remember that a state $s$ was a tuple $(g,ls)$ where $g$ is the shared state and $ls$ is the local state for each thread. We assume that $g$ is a tuple $(dv,sv)$ where $dv$ holds the values for all the data variables (global variables and the heap), and $sv$ holds the values for all the synchronization variables. Considering the memory layout of an application, presented in Figure 3.1, we can identify the components of the state. For each thread, the local state is stored in the thread’s stack. The
Figure 3.1: The memory layout of an application

data variables are stored in the global variables and the heap. The synchronization variables are
stored in the OS kernel.

We assume that the scheduling mechanism gives us read access to the stacks of each thread and
also to the data variables that correspond to the current state of the application. The scheduling
mechanism also allows us to evaluate invariants in the current state.

We assume that the scheduling mechanism allows us to get the application to its initial state $s_0$
and from that state, allows us to schedule one thread at a time. For each thread scheduling $p$, the
mechanism atomically executes the transition $t$ that corresponds to thread $p$, and the current state
of the application becomes $s'$, assuming that we had $s \xrightarrow{t} s'$.

We assume that the scheduling mechanism allows us to get what is the configuration correspond-
ing to the current state.

**Definition 12.** A configuration that corresponds to a state $s$ is a set of tuples $(a, enabled, new)$,
one tuple for each active thread $p$, where:

- $a$ is the next action of thread $p$ in state $s$;
- $enabled$ is true if $a$ is enabled in $s$, and false otherwise.
- $new$ is true if $a$ is from a thread $p$ that was just created, and false otherwise. Only one tuple
can have $new = true$.

**Definition 13.** We make the following notations for a configuration $conf$:

- $actions(conf)$ is the set of all available next actions including both the enabled and the
disabled ones;
- $enabled\_actions(conf)$ is the set of enabled next actions;
- $enabled\_threads(conf)$ is the set of threads that have an enabled next action;
Lemma 4. Let depth be defined as the number of nodes of the graph: depth((A, H)) = |A|.

Definition 14. For a happens-before graph (A, H) that represents a state s, and an action a' enabled in s, extend((A, H), a') is a happens-before graph (A', H') with:

- A' = A ∪ {a'}. Add a new node for the action a';
- H' = H ∪ {(a, a') | a ∈ A ∧ (a, a') ∈ D^+}, is the transitive closure of the dependency relation D on A'.

Lemma 4. Given a happens-before graph (A, H) and an action a, extend((A, H), a) is a happens-before graph (A', H') with:

- A' = A ∪ {a}. Add a new node for the action a;
- H' = H ∪ {(a, a') | (a, a') ∈ D^+}, is the transitive closure of the dependency relation D on A'.

Proof. Let extend((A, H), a) = (A', H').

(A, H) is a representation of s implies that there exists a sequence of actions q = a_1...a_n that abstracts a sequence of transitions w = t_1...t_n such that s_0 w s.

We construct a new sequence of transitions w' = t_1...t_n t_{n+1}, with t_{n+1} = t', and from s w' s' we have that s_0 w' s'. Let q' = a_1...a_n a_{n+1}, with a_{n+1} = a', the sequence of actions that abstracts w'.

HBA(q') = (A'', H'') is a representation for s'.

What is left to show is that (A'', H'') = (A', H').

From Definition 11 we have A'' = \{a_i|0 < i ≤ n + 1\} = \{a_i|0 < i ≤ n\} ∪ \{a'\} = A ∪ \{a'\} which is the same as A' in Definition 14. So A'' = A'.

Also, from Definition 11 we have H'' = \{(a_i, a_j) | 0 < i < j ≤ n + 1 ∧ (a_i, a_j) ∈ D^+\} = \{(a_i, a_j) | 0 < i < j ≤ n ∧ (a_i, a_j) ∈ D^+\} = H ∪ \{(a, a') | a ∈ A ∧ (a, a') ∈ D^+\} which is the same as H'. So H'' = H'.

We have extend((A, H), a) = (A'', H''), so extend((A, H), a) is a representation of s'.

Chapter 4

State Space Exploration

4.1 Exploration Algorithms

Definition 16. For a state representation \( n \) that corresponds to a state \( s \), a call of \( \text{replay}(n) \) ensures that the current state of the application becomes \( s \), and also, the call returns the configuration that corresponds to \( s \).

Definition 17. A state \( s \) is considered to be explored if, during the exploration, \( s \) was at least once the implicit current state of the application.

Lemma 6. A state \( s \) is explored by an exploration algorithm, if the algorithm calls replay at least once with a representation \((A,H)\) of \( s \) as an argument.

Proof. Calling \( \text{replay} \) with a representation of \( s \) as an argument ensures that the current state of the application becomes \( s \). Using Definition 17 we can conclude that \( s \) is explored.

If \( \text{replay} \) is the only way the current state can be changed, then the implication is actually an equivalence.

Definition 18. Assuming that \( A_G = (S, \rightarrow, s_0) \) is acyclic, \( A_R = (V_R, E_R, u_0) \) is a graph of representations in which:

- \( V_R = \{HBA(q) \mid \exists s \exists w. s_0 \xrightarrow{w} s \land q = \text{abstract}(w)\} \). Any representation of a state from \( A_G \) is a node in \( A_R \).
- \( E_R = \{(HBA(q), HBA(q; a)) \mid \exists s \exists w \exists s'. s_0 \xrightarrow{w} s \xrightarrow{t} s' \land q = \text{abstract}(w) \land a = \text{abstract}(t)\} \).
  If there exists a transition between two states \( s \) and \( s' \) in \( A_G \), then there exists an edge between any representation of \( s \) and any representation of \( s' \) that extends the representation of \( s \). \( HBA(q; a) = \text{extend}(HBA(q), a) \).
- \( u_0 = HBA(\epsilon) \) is a node that corresponds to the initial state.
Lemma 7. If \( A_G \) is acyclic, then the corresponding \( A_R \) has a finite number of nodes (\( A_R \) is well defined).

Proof. \( A_G \) has a finite number of states.

\( A_G \) is acyclic, and using Theorem 4 we know that each state has only a finite number of representations.

The nodes of \( A_R \) are all the representations for all the states, we have a finite number of representations for each state, and a finite number of states, so \( A_R \) has a finite number of nodes.

Lemma 8. \( A_R \) is acyclic.

Proof. Let’s assume that \( A_R = (V_R, E_R, u_0) \) contains a cycle: \((v_0, v_1, ..., v_n)\), with \((\forall i.0 \leq i \leq n. v_i \in V_R) \land (v_0 = v_n) \land (\forall i.0 \leq i < n. (v_i, v_{i+1}) \in E_R)\). From Definition 18 we know that for any \( i \), \( 0 \leq i < n \), there exists an action \( a_{i+1} \) such that \( v_{i+1} = \text{extend}(v_i, a_{i+1}) \). From Lemma 5 we have that \( \text{depth}(v_{i+1}) = \text{depth}(v_i) + 1 \), so \( \text{depth}(v_i) < \text{depth}(v_{i+1}) \). Repeatedly applying this observation, we can conclude that \( \text{depth}(v_0) < \text{depth}(v_n) \), but \( v_0 = v_n \) and we should have \( \text{depth}(v_0) = \text{depth}(v_n) \).

We have reached a contradiction, so our assumption that \( A_R \) is not acyclic is wrong.

Algorithm 1 Basic stateless exploration algorithm.

1: \( \text{Frontier} = \{HBA(\epsilon)\} \)
2: \textbf{while} \( \text{Frontier} \neq \emptyset \) \textbf{do}
3: \( n = \text{pick}(\text{Frontier}) \)
4: \( \text{Frontier} = \text{Frontier} \setminus \{n\} \)
5: \( \text{config} = \text{replay}(n) \)
6: \( \text{Successors} = \emptyset \)
7: \textbf{for all} \( a \in \text{enabled\_actions}(\text{config}) \) \textbf{do}
8: \( \text{Successors} = \text{Successors} \cup \{\text{extend}(n, a)\} \)
9: \textbf{end for}
10: \( \text{Frontier} = \text{Frontier} \cup \text{Successors} \)
11: \textbf{end while}

Lemma 9. If \( A_G \) is acyclic, and \( A_R \) is the graph of representations that corresponds to \( A_G \), then Algorithm 1 terminates and every node in \( A_R \) is eventually picked in line 3 of the algorithm.

Proof. First we prove that \( \text{Frontier} \) holds only nodes from \( A_R \). This is an invariant for the while loop.

The invariant holds at the beginning of the loop:

After the execution of line 1, right before the while loop, \( \text{Frontier} \) contains \( u_0 \) which is a node from \( A_R \).

The invariant is maintained by the loop body:

In line 3 of the algorithm we know that \( \text{Frontier} \) is not empty (from the guard), and assuming the invariant holds at the beginning of the loop, \( \text{Frontier} \) only contains nodes from \( A_R \).
At the end of line 3, \( n \) is an element from \( \text{Frontier} \) so it is a node from \( A_R \), a representation of a state \( s \) in \( A_G \).

In line 4, we remove \( n \) from \( \text{Frontier} \), so it still contains only nodes from \( A_R \), or it is empty.

\( n \) is a representation of some state \( s \) from \( A_G \), and by calling \( \text{replay} \) on \( n \) we ensure that, after line 5 is executed, the implicit current state of the application becomes \( s \) and \( \text{config} \) is the configuration that corresponds to \( s \).

In line 7, \( \text{enabled\_actions} \left( \text{config} \right) \) is a set that contains all actions that are enabled in the current state \( s \) (Definition 13).

In line 8, \( n \) is a representation of \( s \) and \( a \) is enabled. Using Lemma 4 we get that \( \text{extend} \left( n, a \right) \) is a representation of a state \( s' \) that is a successor of \( s \) in \( A_G \). From Definition 18 we know that \( \text{extend} \left( n, a \right) \) is a successor of \( n \) in \( A_R \). So after line 9, \( \text{Successors} \) is the set of all successors of \( n \) in \( A_R \).

In line 10, \( \text{Frontier} \) is extended with the set \( \text{Successors} \), which contains only nodes in \( A_R \). So at the end of line 10, \( \text{Frontier} \) contains only nodes from \( A_R \).

So far we have proved that \( \text{Frontier} \) only holds nodes from \( A_R \), \( n \) is a node from \( A_R \), and at the end of line 9, \( \text{Successors} \) contains all the successors of \( n \).

We can now regard Algorithm 1 as a generic graph traversal algorithm that visits nodes from \( A_R \) without storing them.

From Graph Theory we know that if \( A_R \) is acyclic, then such a traversal terminates and visits all the nodes.

From Lemma 8 we know that \( A_R \) is acyclic, so Algorithm 1 will visit every node of \( A_R \), meaning that every node is eventually picked in line 3.

\textbf{Lemma 10.} If for every node \( n \) in \( A_R \), an exploration algorithm calls \( \text{replay} \left( n \right) \) at least once, then it necessarily explores all states in \( A_G \).

\textit{Proof.} From Theorem 3 we know that every state from \( A_G \) has a representation. Definition 18 ensures that \( A_R \) contains all the representations for all the states in \( A_G \). So for any state \( s \) from \( A_G \), \( A_R \) contains at least one node \( n \) that is a representation of \( s \). The exploration algorithm calls \( \text{replay} \left( n \right) \), and using Lemma 6 we get that \( s \) is explored. \( \square \)

\textbf{Theorem 5.} If \( A_G \) is acyclic, then Algorithm 1 terminates and explores all the states in \( A_G \).

\textit{Proof.} By applying Lemma 9 we get that Algorithm 1 terminates.

Lemma 9 ensures that Algorithm 1 calls \( \text{replay} \left( n \right) \) for every node \( n \) in \( A_R \). Using Lemma 10 we get that all the states in \( A_G \) are explored. \( \square \)

\textbf{Lemma 11.} If \( \text{Filters} = \emptyset \), then Algorithm 2 behaves exactly like Algorithm 1.
Algorithm 2 Exploration algorithm with state filtering.

1: `Frontier = \{HBA(\epsilon)\}
2: while `Frontier \neq \emptyset` do
3: `n = pick(Frontier)`
4: `Frontier = Frontier \setminus \{n\}`
5: `config = replay(n)`
6: `Successors = \emptyset`
7: for all `a \in enabled\_actions(config)` do
8: `Successors = Successors \cup \{extend(n, a)\}`
9: end for
10: for all `f \in Filters` do
11: `Successors = f(Successors)`
12: end for
13: `Frontier = Frontier \cup Successors`
14: end while

Proof. If `Filters = \emptyset`, then line 11 of Algorithm 2 will never be executed and the set `Successors`, at the end of line 12 will be the same as at the end of line 9. The algorithm has the same effect as if lines 10 - 12 were removed. Removing lines 10 - 12 leaves us with the code for Algorithm 1.

4.2 State Filters

![Diagram](image)

Figure 4.1: Exploration using state filters

Each state filter can tag extra information to each state representation. The extra information can be retrieved in a later call of the filter function. Also, a state filter can add some global
4.2.1 The NotSeenBefore Filter

Algorithm 3 NotSeenBeforeFilter(Successors)

1: \( H = H \cup \{n\} \)
2: for all \( n' \in \text{Successors} \) do
3: \[ \text{if } n' \in H \text{ then} \]
4: \( \text{Successors} = \text{Successors} \setminus \{n'\} \)
5: \[ \text{end if} \]
6: \[ \text{end for} \]
7: return \( \text{Successors} \)

The extra information used by the NotSeenBeforeFilter in Algorithm 3 is a set (a cache) of already visited state representations.

**Theorem 6.** If \( A_G \) is a directed acyclic graph and the set Filters in Algorithm 2 contains only the function presented in Algorithm 3, then Algorithm 2 terminates and explores all the states in \( A_G \).

**Proof.** Algorithm 3 is called for each iteration of the while loop in Algorithm 2. In each iteration, Algorithm 3 can only remove elements from the set \( \text{Successors} \) (line 4).

Line 1 of Algorithm 3 gets called for each visited state representation \( n \), so the set \( H \) stores the representations visited so far.

The combination of the two algorithms is a restricted traversal of the graph \( A_R \) in which previously visited nodes are not revisited. From Graph Theory we know that such a traversal terminates and visits all the nodes in \( A_R \). So for any node \( n \) in \( A_R \), Algorithm 2 calls, in line 5, \( \text{replay}(n) \). Using Lemma 10 we get that all the states in \( A_G \) are explored. \( \square \)

The main difference between Algorithm 1 and Algorithm 2 with the NotSeenBefore filter is that the first one doesn’t store any state representations, while the second one stores all the states representations. It is a trade-off between memory consumption and doing the same work multiple times. The two algorithms are at the extremes of this trade-off. Algorithm 1 uses little memory, but cannot avoid doing extra work when it encounters a node that was previously visited. Algorithm 2 with the NotSeenBefore filter avoids exploring nodes more than once, but it uses a lot more memory because it has to store all visited nodes. By modifying \( H \) in Algorithm 3 from a set that stores all the nodes, to a cache that can store only a limited number of nodes, we could control the trade-off between memory consumption and duplicate work.

4.2.2 The DepthBounding Filter

**Lemma 12.** If \( A_G \) contains at least one cycle, and if we construct \( A_R \) according to Definition 18, then \( A_R \) has an infinite number of nodes.
Proof. Let’s pick a cycle from $A_G$: $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n$, with $n > 1$ and $s_1 = s_n$.

Let $a_i$ be the action that abstracts transition $t_i$ for all $0 < i < n$.

Let $n$ be a representation of $s_1$, and let us extend $n$ with $a_1$, the resulting state with $a_2$, and so on, until we have extended $n$ with the sequence $a_1 \ldots a_{n-1}$ and we reach a representation $n^{(1)}$ that is also a representation of $s_1$ (we have executed the cycle once). $\text{depth}(n^{(1)}) = \text{depth}(n) + n - 1$, the depth is increased by the size of the cycle.

We can now repeat the argument and extend $n^{(1)}$ to get $n^{(2)}$, another representation for $s_1$ that has $\text{depth}(n^{(2)}) = \text{depth}(n) + 2(n - 1)$. In fact for any natural number $k$, we can extend $n$ with the actions in the cycle $k$ times and we get $\text{depth}(n^{(k)}) = \text{depth}(n) + k(n - 1)$.

We have an infinite number of representations for $s_1$, so $A_R$ has an infinite number of nodes. Note that any state from the cycle has an infinite number of representations, not just $s_1$. 

Considering the result presented in Lemma 12, the exploration algorithms presented so far (Algorithm 1 and Algorithm 2 with the NotSeenBefore filter), do not terminate if they are run for a multi-threaded application $A_G$ that has a cycle.

The common solution for such a problem is to use a technique called depth bounding[5], the model checker verifies finite executions that are not longer than a certain bound. The usual approach is to set a large depth bound, and whenever this bound is reached, declare that a live-lock was found [10, 19]. Algorithm 4 shows how depth bounding is implemented in SCALE as a state filter.

\begin{algorithm}
\caption{DepthBoundingFilter(Successors)}
\begin{algorithmic}[1]
\FORALL{$n' \in \text{Successors}$} 
\IF{$\text{depth}(n') > \text{DEPTH\_BOUND}$} 
$\text{Successors} = \text{Successors} \setminus \{n'\}$ \ENDIF 
\ENDFOR 
\RETURN $\text{Successors}$
\end{algorithmic}
\end{algorithm}

**Theorem 7.** If Algorithm 2 has in the Filters list Algorithms 3 and 4, then it explores all the states from $A_G$ that are reachable through a sequence of transitions of length at most $\text{DEPTH\_BOUND}$.

**Proof.** Let $A'_R = (V'_R, E'_R)$ be a sub-graph of $A_R = (V_R, E_R)$ such that:

- $V'_R = \{v \in V_R \mid \text{depth}(v) \leq \text{DEPTH\_BOUND}\}$, the nodes that do not exceed the depth bound;

- $E'_R = \{(u, v) \in E_R \mid u \in V'_R \land v \in V'_R\}$, the restriction of $E_R$ to the nodes in $V'_R$.

From the definition of $A_R$ (Definition 18), and the definition of $\text{depth}$ (Definition 15), we know that if $(u, v) \in E_R$ then $\text{depth}(v) = \text{depth}(u) + 1$. So for any two nodes $u, v \in V_R$ with $\text{depth}(v) < \text{depth}(u)$ we have that $(u, v) \notin E_R$. $A_R$ does not contain edges from representations with a higher
depth to representations with a lower depth. So there is no edge in $E_R$ from a node that is in $V_R \setminus V'_R$ to a node that is in $V'_R$.

The DepthBounding filter limits the traversal of $A_R$ to that of $A'_R$. Applying a similar argument to the one used in Theorem 6, we get that the exploration algorithm visits all the nodes in $A'_R$.

For any state $s$ from $A_G$ that is reachable through a sequence of transitions $w$ that has a length of at most $\text{DEPTH}_\text{BOUND}$, there exists a representation $n$ in $A'_R$ that corresponds to the sequence of actions $q$ that abstracts $w$. $n$ is visited by the exploration algorithm, so the corresponding state $s$ is explored (because $\text{replay}(n)$ is called in line 5 of Algorithm 2).

4.2.3 The LocalCycleDetection Filter

From now on we assume that the implementation of $\text{extend}$ and the way a happens-before graph for actions is stored allows us to recover, for any state representation, the parent state representation and the action used to extend the parent representation.

**Definition 19.** For a state representation $n' = \text{extend}(n, a)$ we define:

- $\text{parent}(n') = n$, the parent state representation from which $n'$ was extended;
- $\text{last\_action}(n') = a$, the action used to extend the parent state representation to get the new state representation $n'$;
- $\text{last\_thread}(n') = p$, if $a = (p, op, vl)$. $p$ is the thread that executed the last action;

For $u_0$, the representation of $s_0$ we define:

- $\text{parent}(u_0) = \bot$, $u_0$ was not obtained by extending an already existent state representation;
- $\text{last\_action}(u_0) = \bot$;
- $\text{last\_thread}(u_0) = m$, where $m$ is the main thread.

In Chapter 5 we will present how we can efficiently store and extend happens-before graphs, and at the same time preserve the parent and last\_action information. We will also show how we can efficiently hash state representations. Storing the hash of a representation instead of the representation can reduce the memory used by the NotSeenBefore filter, and also decrease the time required to test the equality of two representations. There is a drawback to using hashing: we no longer have a guarantee that all state representations will be explored.

**Definition 20.** A local execution cycle from $A_G$ is a cycle $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n$, with $s_1 = s_n$ in which all the transitions $t_i, 0 < i < n$ are taken by the same thread $t$ and none of them change a communication variable.
Algorithm 5 \textit{LocalCycleDetection}(Successors)

1: $A = \emptyset$
2: $t = \text{last\_thread}(n)$
3: $\text{LocalState}(n) = \text{get\_stack}(t)$
4: $a = \text{last\_action}(n)$
5: $p = \text{parent}(n)$
6: $\text{progress} = \text{false}$
7: while $p \neq \bot \land \lnot \text{progress} \land \text{Successors} \neq \emptyset$ do
8: \hspace{1em} assert $a \neq \bot$
9: \hspace{1em} if $\text{MadeProgress}(a, t) \lor \text{conflicts}(a, A, t)$ then
10: \hspace{1em} \hspace{1em} $\text{progress} = \text{true}$
11: \hspace{1em} else
12: \hspace{1em} \hspace{1em} if $\text{last\_thread}(p) = t \land \text{LocalState}(p) = \text{LocalState}(n)$ then
13: \hspace{1em} \hspace{1em} \hspace{1em} $\text{Successors} = \emptyset$
14: \hspace{1em} \hspace{1em} else
15: \hspace{1em} \hspace{1em} \hspace{1em} $A = A \cup \{a\}$
16: \hspace{1em} \hspace{1em} \hspace{1em} $a = \text{last\_action}(p)$
17: \hspace{1em} \hspace{1em} \hspace{1em} $p = \text{parent}(p)$
18: \hspace{1em} \hspace{1em} end if
19: \hspace{1em} end if
20: end while
21: return $\text{Successors}$

Definition 21. For each visited state representation $n \in A_R$, Algorithm 5 tags $\text{LocalState}(n)$ as extra information.

Definition 22. In Algorithm 5, $\text{get\_stack}(t)$ returns the local state associated with thread $t$.

Definition 23. In Algorithm 5, for an action $a$ and a thread $t$:

- $\text{MadeProgress}(a, t)$ is true if an only if $\text{thread}(a) = t$, and the operation corresponding to $a$ changes a communication variable;

- for a set of actions $A$, $\text{conflicts}(a, A, t)$ is true if and only if there exists an action $a' \in A$ such that $(a, a') \in D \land ((\text{thread}(a) = t \land \text{thread}(a') \neq t) \lor (\text{thread}(a) \neq t \land \text{thread}(a') = t))$, where $D$ is the dependency relation form Definition 9.

Definition 24. A sequence of state representations from $A_R$, $q = u_1 \xrightarrow{a_1} u_2 \xrightarrow{a_2} \ldots \xrightarrow{a_k} u_{k+1}$ is said to exhibit a local execution from $A_G$ if and only if there exists a sequence $q' = u'_1 \xrightarrow{a'_1} \ldots \xrightarrow{a'_k} u'_{k+1}$ that is partial order equivalent to $q$ such that:

- $a'_k = a_k$,

- there exists $i$, $0 < i \leq k$, such that $\text{last\_thread}(u'_i) = \text{last\_thread}(u_{k+1})$,

- for all $l$, $0 < l \leq i$, $u'_l = u_l$,

- for all $l$, $0 < l < i$, $a'_l = a_l$, 

- $\text{progress} = \text{false}$

- while $p \neq \bot \land \lnot \text{progress} \land \text{Successors} \neq \emptyset$ do

  - assert $a \neq \bot$

  - if $\text{MadeProgress}(a, t) \lor \text{conflicts}(a, A, t)$ then

    - $\text{progress} = \text{true}$

  - else

    - if $\text{last\_thread}(p) = t \land \text{LocalState}(p) = \text{LocalState}(n)$ then

      - $\text{Successors} = \emptyset$

    - else

      - $A = A \cup \{a\}$

      - $a = \text{last\_action}(p)$

      - $p = \text{parent}(p)$

    - end if

  - end if

end while

return $\text{Successors}$

21: return $\text{Successors}$

• there exists \( j, i \leq j \leq k \), such that

- for all \( l, j \leq l \leq k \), \( \text{thread}(u'_l) = \text{thread}(u_{k+1}) \),
- for all \( l, i < l < j \), \( \text{thread}(u'_l) \neq \text{thread}(u_{k+1}) \),
- for all \( l, i < l < j \), and for all \( m, j \leq m \leq k \), \((a_l, a_m) \notin D\),
- \( u'_j \xrightarrow{a'_i} \cdots \xrightarrow{a'_k} u'_{k+1} \) corresponds to a local execution cycle in \( A_G \).

**Lemma 13.** For any node \( n \in A_R \) which was obtained by extending the initial node \( u_0 \) with a sequence of actions \( q = a_1a_2a_k \), \( k \geq 0 \), and for any set \( \text{Successors} \), Algorithm 5 terminates. Moreover, if \( q \) exhibits a local execution cycle starting from \( u_0 \) then, at the end of the algorithm, \( \text{Successors} \) becomes \( \emptyset \), otherwise \( \text{Successors} \) is not changed.

**Proof.** In each iteration of the loop, either \( \text{depth}(p) \) is decreased by 1 or \( p \) becomes \( \perp \) (line 17), or \( \text{progress} \) is set to \( \text{true} \) (line 10), or \( \text{Successors} \) is set to \( \emptyset \) (line 13). If \( \text{progress} \) is set to \( \text{true} \) or \( \text{Successors} \) is set to \( \emptyset \), then the loop is exited because of the loop guard. Also, \( \text{depth}(p) \) cannot decrease forever because it has a lower bound of 0 reached when \( p = u_0 \). \( \text{parent}(u_0) = \perp \). So eventually the loop is exited, and Algorithm 5 terminates.

Let us define the property \( \text{Cycle} \) to be true if and only if \( q = a_1a_2a_k \) exhibits a local execution cycle (Definition 24).

Let \( \text{Successors}^0 \) be the value of \( \text{Successors} \) at the beginning of Algorithm 5.

What we have to prove is that at the end of Algorithm 5:

\[
(Cycle \Rightarrow \text{Successors} = \emptyset) \land (\neg Cycle \Rightarrow \text{Successors} = \text{Successors}^0)
\]

Let’s call this property \( \text{Post} \). We have:

\[
\text{Post} \\
\equiv \{\text{definition of Post}\}
\]

\[
(Cycle \Rightarrow \text{Successors} = \emptyset) \land (\neg Cycle \Rightarrow \text{Successors} = \text{Successors}^0)
\]

\[
\equiv \{(P \Rightarrow Q) \equiv (\neg P \lor Q)\}
\]

\[
(\neg Cycle \lor \text{Successors} = \emptyset) \land (Cycle \lor \text{Successors} = \text{Successors}^0)
\]

\[
\equiv \{\text{distributivity of } \land \text{ over } \lor\}
\]

\[
(\neg Cycle \land Cycle) \lor (\neg Cycle \land \text{Successors} = \text{Successors}^0)
\]

\[
\lor ((Cycle \lor \text{Successors} = \text{Successors}^0) \land \text{Successors} = \emptyset)
\]

\[
\equiv \{ P \land \neg P \equiv \text{false} ; \text{false} \lor P \equiv P \}
\]

\[
(\neg Cycle \land \text{Successors} = \text{Successors}^0) \lor ((Cycle \lor \text{Successors} = \text{Successors}^0) \land \text{Successors} = \emptyset)
\]

Assuming that \( J \equiv J_1 \land J_2 \) is a loop invariant, with:

\[
J_1 \equiv \text{progress} \Rightarrow (\neg Cycle \land \text{Successors} = \text{Successors}^0), \text{ which is the same as } J_1 \equiv \neg \text{progress} \lor (\neg Cycle \land \text{Successors} = \text{Successors}^0), \text{ and }
\]
\(J_2 \equiv Successors = \emptyset \Rightarrow (Cycle \lor Successors = Successors^0)\), which is the same as
\(J_2 \equiv Successors \neq \emptyset \lor Cycle \lor Successors = Successors^0\),
we can prove that Pre is satisfied at the end of the while loop.

Let \(G \equiv \neg progress \land p \neq \bot \land Successors \neq \emptyset\) be a notation for the while loop guard.

At the end of the while loop we have \(\neg G \land J\) and we have to prove that \((\neg G \land J) \Rightarrow Post\).

\((\neg G \land J) \Rightarrow Post\)
\[\equiv \{\text{definitions of } G \text{ and } J, \text{ properties of } \neg \}\]
\(\equiv \{\text{distributivity of } \land \text{ over } \lor; \text{ definitions of } J_1 \text{ and } J_2\}\)
\(\equiv \{P \land \neg P \equiv \text{false}; \text{ definition of } Post\}\)
\(\equiv \{\text{true}\}\)

What it is left to prove is that invariants \(J_1\) and \(J_2\) hold at the beginning of the while loop, and
that they are preserved by the loop.

At the beginning of the loop (between line 6 and line 7) we have:

\(J_1\)
\[\equiv \{\text{definition of } J_1\}\]
\(\neg progress \lor (\neg Cycle \land Successors = Successors^0)\)
\[\equiv \{\neg progress \text{ because } progress \text{ was set to } false \text{ in line 7}\}\]
\[\text{true}\]

We also have:

\(J_2\)
\[\equiv \{\text{definition of } J_2\}\]
\(Successors \neq \emptyset \lor Cycle \lor Successors = Successors^0\)
\[\equiv \{Successors = Successors^0 \text{ because } Successors \text{ was never changed}\}\]
\[\text{true}\]

So both \(J_1\) and \(J_2\) hold at the beginning of the while loop, and we are left to prove that they
are maintained by the loop body.

First, we notice that lines 8, 15, 16, and 17 from the loop body cannot change the values of \(J_1\)
and \(J_2\), if they hold before executing those lines they have to hold after executing those lines. We
only have to look at lines 10 and 13.
Let’s assume that right before line 10 $P_1$ holds, with $P_1 \equiv \neg\text{Cycle} \land \text{Successors} = \text{Successors}^0$.

We know that $\text{progress} = \text{true}$ cannot change the value of $P_1$, so $P_1$ is also true after executing line 10. We know that $P_1 \Rightarrow J_1 \land J_2$, so line 10 preserves $J_1$ and $J_2$.

Let’s assume that right before line 13 $P_2$ holds, with $P_2 = \neg\text{progress} \land \text{Cycle}$. Executing line 13, does not change $P_2$. We know that $P_2 \Rightarrow J_1 \land J_2$, so line 13 preserves $J_1$ and $J_2$.

We are left to prove that $P_1$ holds right before line 10 and that $P_2$ holds right before line 13.

Let’s define $J_3 \equiv \text{Successors} = \text{Successors}^0 \lor \text{Successors} = \emptyset$. $J_3$ holds before the while loop because the variable $\text{Successors}$ is not changed. $J_3$ is maintained by the loop body because the only line that changes $\text{Successors}$, changes it to $\emptyset$, keeping $J_3$ true. So $J_3$ is also a loop invariant.

Right before line 10 we know that the loop guard holds and also $J_3$ holds. We prove that $G \land J_3 \Rightarrow \text{Successors} = \text{Successors}^0$.

$G \land J_3 \Rightarrow \text{Successors} = \text{Successors}^0$
$\equiv \{\text{definitions of } G \text{ and } J_3\}$
$(\neg\text{progress} \land p \neq \bot \land \text{Successors} \neq \emptyset \land (\text{Successors} = \text{Successors}^0 \lor \text{Successors} = \emptyset)) \Rightarrow \text{Successors} = \text{Successors}^0$
$\equiv \{\text{distributivity of } \land \text{ over } \lor; P \land \neg P \equiv \text{false}\}$
$(\neg\text{progress} \land p \neq \bot \land \text{Successors} \neq \emptyset \land \text{Successors} = \text{Successors}^0) \Rightarrow \text{Successors} = \text{Successors}^0$
$\equiv \{P \land Q \Rightarrow P\}$
$\text{true}$

Right before line 13 we know that the loop guard holds, and so we have $\neg\text{progress}$.

We are left to prove that right before line 10 we have $\neg\text{Cycle}$, and that right before line 13 we have $\text{Cycle}$.

$n$ was obtained by extending $u_0$ with the sequence $q = a_1a_2...a_k$. Let $q[i : j] = a_i...a_j$ if $0 < i \leq j \leq k$, and $q[i : j] = \epsilon$ otherwise. Let $a_i$ be the current action $a$ in the loop body. Initially $a_i = a_k$.

We define a new invariant $J_4$ stating that $q[i + 1 : k]$ does not exhibit a local execution cycle (Definition 24).

$J_4$ trivially holds at the beginning of the while loop since $q[k + 1 : k] = \epsilon$.

To prove that $J_4$ is maintained by the loop body, we only have to look at line 16, since it is the only line that changes the current action $a$.

Right before line 16 we have (from the conditions in the if statements):
$\neg\text{MadeProgress}(a, t) \land \neg\text{conflicts}(a, A, t) \land (\text{last_thread}(p) \neq t \lor \text{LocalState}(p) \neq \text{LocalState}(n))$

Assuming that $q[i + 1 : k]$ does not exhibit a local execution cycle, we prove that $q[i : k]$ does not exhibit a local execution cycle either.

To exhibit a local execution cycle, $p$ would have to correspond to $u_i$ in Definition 24, otherwise $q[i + 1 : k]$ would have exhibited a local execution cycle.
If $\text{last\_thread}(p) \neq \text{last\_thread}(n)$, then $p$ cannot correspond to $u_i$ in Definition 24 and $q[i:k]$ does not exhibit a local execution cycle.

If $\text{last\_thread}(p) = \text{last\_thread}(n) \land \text{LocalState}(p) \neq \text{LocalState}(n)$, then we try to repeatedly swap, according to the dependency relation, actions from other threads so they end up before $a_i$. If the swapping fails for at least one action, we are done since we cannot form a local execution cycle.

Let's assume that the swapping succeeds for all the actions from the other threads. We are left with a tail that starts at a state representation $u$ with $a_i$ the next action and contains only actions from $\text{last\_thread}(n)$.

Because all the actions between $p$ and $u$ are from other threads than $\text{last\_thread}(n)$ we know that the local state for this thread has not changed between $p$ and $u$. We have $\text{LocalState}(p) \neq \text{LocalState}(n)$, so there cannot be a local execution cycle in $A_G$ between states corresponding to $u$ and $n$. If there were such a local execution cycle then we would have had $\text{LocalState}(p) = \text{LocalState}(n)$.

We have established that $J_4$ is maintained by the loop body.

We need another invariant: $J_5 \equiv \forall a_j \in A. \neg \text{MadeProgress}(a,t) \land \neg \text{conflicts}(a_j, A, t)$.

$J_5$ trivially holds at the beginning of the loop since $A = \emptyset$.

$A$ is changed only in line 15, by adding a new action $a$. From the guards of the if statements, right before line 15, we have:

$\neg \text{MadeProgress}(a,t) \land \neg \text{conflicts}(a, A, t)$ holds. From Definition 23 we get that $\neg \text{conflicts}(a, A \cup \{a\}, t)$, so the invariant $J_5$ is maintained by the loop body.

We need another invariant: $J_6 \equiv \forall j. i < j \leq k. a_j \in A$.

$J_6$ trivially holds at the beginning of the loop since $i = k$.

$A$ is changed only in line 15, by adding the new action $a = a_i$. We get that $\forall j. i - 1 < j \leq k. a_j \in A$.

Right after line 16, $i$ has decreased by 1 (we have moved up one action), but $J_6$ still holds since we have added $a$ to $A$ one line above.

We are now ready to prove that right before line 10 $\neg \text{Cycle}$ holds.

Right before line 10, from the guard of the if statement we have:

$\text{MadeProgress}(a,t) \lor \text{conflicts}(a, A, t)$.

If $\text{MadeProgress}(a,t)$ is true then action $a$ is from thread $t$ and has changed a global variable. No local execution cycle can include action $a$ and so now we can declare that $\neg \text{Cycle}$ holds.

If $\text{conflicts}(a, A, t)$ is true and $\text{thread}(a) = t$, then there exists $a' \in A$, $\text{thread}(a') \neq t$, such that $(a, a') \in D$. From invariant $J_4$ we know that if there exists a local execution cycle, then $a$ has to be part of it. To exhibit this local execution cycle, we should be able to swap $a$ with any action that follows it and is from another thread. From invariant $J_6$ we know that $A$ contains all the actions that follow $a$, so $a'$ follows $a$, but we cannot swap them because $(a, a') \in D$. So in this case $\neg \text{Cycle}$ holds.
If \( \text{conflicts}(a,A,t) \) is true and \( \text{thread}(a) \neq t \), then there exists \( a' \in A, \text{thread}(a') = t \) such that \( (a,a') \in D \). From invariant \( J_6 \) we know that \( A \) contains all the actions that follow \( a \), so \( a' \) follows \( a \). Any local execution cycle that ends in the state represented by \( n \) would have to include \( a \) (from invariant \( J_4 \)). The Definition 24 requires that \( (a,a') \notin D \) which is not possible in this case.

We can now prove that right before line 13 Cycle holds.

Right before line 13, from the guard of the if statements we have:

\[
\neg \text{MadeProgress}(a,t) \land \neg \text{conflicts}(a,A,t) \land \text{last_thread}(p) = t \land \text{LocalState}(p) = \text{LocalState}(n).
\]

From last_thread\((p) = \text{last_thread}(n)\) and invariants \( J_5 \) and \( J_6 \) we get that we can reorder the actions from \( p \) to \( n \) in such a way that Definition 24 is satisfied except for the part with the local execution cycle in \( A_G \).

We know there exists an \( u_j \) between \( p \) and \( n \) such that all the actions between \( p \) and \( u_j \) are from other threads than \( t \), and also all the actions between \( u_j \) and \( n \) are from \( t \). Also, we know that these groups of actions are independent with respect to each other.

From \( p \) to \( u_j \) the local state of thread \( t \) is not changed since no action is from \( t \). From \( u_j \) to \( n \) we have actions only from thread \( t \) so the local state of the other threads cannot be changed. We also have that between \( u_j \) and \( n \) no communication variable was modified. It follows from Definition 3 that the state \( s \in A_G \) that corresponds to \( u_j \) is the same as the state \( s' \in A_G \) that corresponds to \( n \). All the actions between \( s \) and \( s' \) are from thread \( t \) and none of them change communication variables, so according to Definition 20 we have found a local execution cycle.

Now, using Definition 24, we get that the sequence of actions that led to \( n \) exhibits a local execution cycle.

\( \blacksquare \)

**Theorem 8.** If all the execution paths in \( A_G \) are either finite (no cycles), or partial order equivalent to some other executions that involve only local execution cycles, then Algorithm 2, with the list Filters containing only the functions presented in Algorithms 3 and 5 (in this order), terminates and explores all the states in \( A_G \).

**Proof.** Algorithm 5 does not allow the exploration of infinite paths. Any infinite path has a finite length prefix that exhibits a local execution cycle. According to Lemma 13, such a local execution cycle would be identified by Algorithm 5 and Successors would be set to \( \emptyset \) effectively stopping the exploration of that path at that point. Since only finite paths are explored, and \( A_R \) contains no cycles, Algorithm 2 must terminate.

For any state \( s \in A_G \) with \( s_0 \models s \), let \( w' \) be a sequence of transitions that is partial order equivalent to \( w \) and in which all the unrolls of a local execution cycle are grouped together and don’t have any transitions from other threads (Definition 20). By removing from \( w' \) each complete unrolling of a local execution cycle we are left with a sequence \( w'' \). We have \( s_0 \models w' \models s \) and \( w'' \) does not contain any local execution cycles. By taking \( q = \text{abstract}(w) \), the sequence of actions that
leads from $s_0$ to $s$ in $A_G$, we have that $q$ does not exhibit a local cycle, and so it will not be cut by Algorithm 5. If $q$ is not cut by Algorithm 3, then a state $n$, that corresponds to $u_0$ being extended with $q$, is explored. If $q$ is cut by Algorithm 3, then we can construct $q'$ partial order equivalent to $q$ such that $q'$ also leads to $s$ (see Theorem 6).

4.2.4 The SuperStepPOR Filter

The SuperStepPOR filter implements the partial order reduction introduced in [27, 26].

**Definition 25.** A sub-graph $A_S = (V_S, E_S)$ of a state representation graph $A_R = (V_R, E_R)$ is called a super step reduced graph of $A_R$ if it has the following properties:

- $V_S \subseteq V_R$ is the set of nodes from $A_S$.
- $V_S = V_{SN} \cup V_{IN}$, where $V_{SN}$ is the set of super nodes from $A_S$, and $V_{IN}$ is the set of intermediate nodes from $A_S$. $V_{SN}$ and $V_{IN}$ are not necessarily disjoint.
- $u_0 \in V_{SN}$, the representation of the initial state $s_0$, is a super node.
- a super step $ss$ is a path $u_1 \xrightarrow{a_1} u_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} u_n$ from $A_S$ such that $u_1 \in V_{SN}$, $u_n \in V_{SN}$, for all $u_i$, $1 < i < n$, $u_i \in V_{IN}$, and there exists a thread $t$ such that for all $a_i$, $0 < i < n$, $\text{thread}(a_i) = t$.
- for a super step $ss = u_1 \xrightarrow{a_1} u_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} u_n$, we call $\{a_i \mid 0 < i < n-1\}$ the set of intermediate actions of the super step $ss$.
- for a super step $ss = u_1 \xrightarrow{a_1} u_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} u_n$, we call $a_{n-1}$ the last action of the super step $ss$.
- any super node $u \in V_{SN}$ has an associated set $\text{SuperSteps}(u)$ that contains the outgoing super steps from $u$.
- for any super node $u \in V_{SN}$, that corresponds to a state $s$ in $A_G$, and for each action $a \in \text{enabled}(s)$, there exists a super step $ss(\text{thread}(a)) \in \text{SuperSteps}(u)$ that is taken by $\text{thread}(a)$.
- for any super node $u$, and for any super step $ss \in \text{SuperSteps}(u)$, all the actions of $ss$ are independent with all the intermediate actions of all the other super steps in $\text{SuperSteps}(u)$.
- for any super step $ss$, none of its intermediate actions enables a previously disabled action from another thread.

**Lemma 14.** Let $A_R$ be the graph of representations corresponding to $A_G$, and let $A_S$ be a super step reduced graph of $A_R$. Assuming that $A_G$ is acyclic, for any state representation $u$ from $A_R$ reachable from $u_0$ by a sequence of actions $q = a_1a_2\ldots a_n$, there exists a sequence of actions $q' = a'_1a'_2\ldots a'_n$ that
is partial order equivalent to \( q \), and there exists an \( i, 0 \leq i \leq n \), such that the following properties hold:

- \( u_0 \xrightarrow{a'_1} u_1 \xrightarrow{a'_2} ... \xrightarrow{a'_i} u_i \) is a path from \( A_S \) and \( u_i \) is a super node in \( A_S \);

- for any thread \( t \), if \( b'_1b'_2...b'_k \) is the sequence of those actions taken by \( t \) from the sequence \( a'_{i+1}...a'_n \), then for all \( l, 0 < l \leq k \), the path \( u_i \xrightarrow{b'_{i}} ... \xrightarrow{b'_{i}} u_{i+l} \) does not end in a super node from \( A_S \) (\( u_{i+l} \not\in V_{SN} \)).

**Proof.** We prove the lemma by induction on the depth of \( u \), which is the same as the length of \( q \).

For \( n = 1 \), \( q = \epsilon \) we have \( q' = \epsilon \) and \( i = 0 \). The two properties are trivially satisfied since \( u_0 \in V_{SN} \).

For \( n = 1 \), \( q = a_1 \) we have \( q' = a_1 \). Because \( u_0 \) is a super step end, and \( a_1 \) is enabled in \( u_0 \), there exists a super step starting from \( u_0 \) that contains \( a_1 \). We have two cases:

a) \( u_1 \) is a super node, and in this case \( i = 1 \). The two properties hold since \( u_0 \xrightarrow{a'_1} a_1 \) is a path from \( A_S \) ending in a super node, and the suffix \( a_2...a_n \) is empty.

b) \( u_1 \) is not a super node, and in this case \( i = 0 \). The first property trivially holds since \( u_0 \) is a super node, and the second property holds because \( u_1 \) is not a super node.

We have proved the base case, and now we proceed to the induction step.

Let \( q = a_1...a_{n-1}a_n \). From the induction step we know there exists a path \( a'_1...a'_{n-1} \) and an \( i \) such that the two properties hold for \( a_1...a_{n-1} \).

Let \( q' \) be the sequence of actions taken by \( thread(a_n) \) starting from \( u_i \). From the induction hypothesis we know that \( q' \) doesn’t pass through a super node in \( A_S \). Again, we have two cases:

a) The sequence \( q'a_n \) leads to a super node in \( A_S \) (the case depicted in Figure 4.2a). From the induction hypothesis we know that for any thread \( p \neq t \), \( q^p \) doesn’t pass through a super node, and so we must have that \( q'a_n \) is independent with all the actions in \( q^p \). If this were not true, then considering Definition 25, either \( q' \) or \( q^p \) would have to pass through a super node, thus violating the induction hypothesis.

Because for any thread \( p \neq t \) we have that \( q'a_n \) is independent with \( q^p \), we can reorder \( q \) to get a partial order equivalent sequence \( q' \) with \( q' = a'_1...a'_iq'q'a_n...a''_n \) such that \( a'a_n \) starts from \( u_i \) and ends in a super node \( u_j \). We now have that \( a'_1...a'_iq'q'a_n \) is a path from \( A_S \). We also know from the induction hypothesis that for any two threads \( t_1 \) and \( t_2 \), \( q^{t_1} \) is independent with \( q_{t_2} \) (otherwise one of them would have to pass through a super node starting from \( u_i \)). If started from \( u_j \), any \( q^p \) passes through a super node, we can reorder the suffix such that the part of \( q^p \) that forms the super step appears right after \( u_j \). We repeat this process until the two properties hold.

b) The sequence \( q'a_n \) does not lead to a super node in \( A_S \) (the case depicted in Figure 4.2b). \( q' = a'_1...a'_{l-1}a_n \) and \( i \) satisfy the two properties. Using the same argument as in case a) we have
that for any two threads $t_1$ and $t_2$, $q^{t_1}$ and $q^{t_2}$ are independent starting from $u_i$. $a_n$ has to be independent with all the actions from other threads, since otherwise it would imply that either $q^t a_n$ or another $q^p$ passes through a super node.

**Theorem 9.** Let $A_R$ be the graph of representations corresponding to $A_G$, and let $A_S$ be a super step reduced graph of $A_R$. Assuming that $A_G$ is acyclic, if a state $s$ in $A_G$, having a representation $u$ in $A_R$, violates a safety property then there exists a state representation $u'$ in $A_S$ that corresponds to a state $s'$ in $A_G$ such that $s'$ violates the same safety property.

**Proof.** By applying Lemma 14 we can construct a sequence of actions $q = a_1...a_i...a_n$ that starting from $u_0$ leads to $u$ such that $a_1...a_i$ is a path in $A_S$ starting from $u_0$ and none of the actions $a_j$, $i < j \leq n$ is the end of a super step.

If $u$ corresponds to a deadlock state then all the threads are disabled in $u$. Any action from a thread $t$, that leads to a state in which $t$ is disabled has to be the last action of a super step (we can’t continue the super step). So $u$ is reached by the last action of a super step, so $u$ has to be a super node, and in this case $u' = u$.

If $u$ corresponds to an assertion violation or a global invariant violation then none of the actions $a_j$, $i < j \leq n$ can change the value of the assertion or the value of the global invariant because otherwise $a_j$ would conflict with all the other actions and so it would have to be the end of a super step. So the same assertion violation or global invariant violation is present in $u_i$ reached by taking the sequence of actions $a_1...a_i$ from $u_0$. So in this case we have $u = u_i$.

The *SuperStepPOR* filter tags to each state representation $u$ the super step $SuperStep(u)$. This means that last_action($u$) is part of $SuperStep(u)$ as either an intermediate action or as the last
Algorithm 6 SuperStepPOR(Successors)

1:  ss_end = true
2:  NextRep = \{ n' ∈ Successors | last_thread(n') = last_thread(n) \}
3:  if n ≠ u₀ ∧ NextRep ≠ ∅ then
4:      ss = SuperStep(n)
5:      sn = SuperNode(ss)
6:      next = choose(NextRep)
7:      if IndepLast(ss, sn) ∧ IndepInter(next, sn) ∀
8:          ¬ Enables(last_action(n), Successors \ NextRep) then
9:          ss_end = false
10:         Intermediate(ss) = Intermediate(ss) ∪ \{ Last(ss) \}
11:         Last(ss) = last_action(next)
12:         SuperStep(next) = ss
13:         Successors = NextRep
14:      end if
15:  end if
16:  if ss_end then
17:      sn = new SuperNode
18:      SuperSteps(sn) = ∅
19:      for all n' ∈ Successors do
20:         ss = new SuperStep
21:         SuperNode(ss) = sn
22:         Intermediate(ss) = ∅
23:         Last(ss) = last_action(n')
24:         SuperSteps(sn) = SuperSteps(sn) ∪ \{ ss \}
25:         SuperStep(n') = ss
26:      end for
27:  end if
28:  return Successors

action.

With each super step ss we associate its set of intermediate actions Intermediate(ss), its last action Last(ss), and the parent super node SuperNode(ss).

A super node is represented by the set of outgoing super nodes, and when it is created, it marks the fact that the current state representation n is in V_SN.

The main idea of the SuperStepPOR filter is to create new super steps for each outgoing action from a super node u. Initially these new super steps contain only the corresponding action that was enabled in u. Each super step is repeatedly extended as long as it can be extended. When the super step cannot be extended any longer, a new super node is created at the end of the super step, and the same process is applied to the new super node.

IndepLast(ss, sn), in line 7 of the filter, tests if the last action from the super step ss is independent with all the last actions from other sibling super steps taken from the super node sn.

IndepInter(next, sn), in line 7 of the filter, tests if last_action(next) is independent with all the intermediate actions from super steps taken from the super node sn, except the super step that corresponds to last_thread(next).

action.
Enables(last_action(n), Successors \ N extRep) determines if the last taken action enabled actions from threads that were not previously enabled.

**Theorem 10.** Algorithm 2 with Filters holding only Algorithm 6 terminates and traverses $A_S$, a super step reduced graph of $A_R$.

**Proof.** The filter does not contain any loops so it has to terminate. The variable Successors is modified only on line 12 where it becomes a subset of the initial value for Successors. Algorithm 2 is a restricted graph exploration, and because $A_G$ is acyclic, it calls the SuperStepPOR filter only a finite number of times. So Algorithm 2 terminates.

For $u_0$, Algorithm 6 creates a new super node. The guard of the if statement in line 3 is false when $n = u_0$ and this ensures that the variable ss_end is true in line 15. So for $n = u_0$, lines 16 to 25 of the algorithm are executed. These lines create a new super node that corresponds to $u_0$ and also create new super steps for each enabled action in $u_0$.

![Figure 4.3: The super step partial order reduction](image)

Figure 4.3 presents the data structures used by the SuperStepPOR filter, and what are their connections to each other.

The current state representation $n$ is part of a super step $ss = SuperStep(n)$ taken by a thread $i = last_thread(n)$ from the parent super node $sn = SuperNode(ss)$.

The algorithm maintains the following invariant $J_1$: for any thread $t \neq last_thread(n)$, Intermediate(ss) is independent with Intermediate($ss^t$), and also Last(ss) is independent with Intermediate($ss^t$), where $ss^t$ is the super node taken from $sn$ by thread $t$.

The algorithm also maintains the invariant $J_2$ none of the actions in Intermediate(ss) enable actions from other threads that were previously disabled.

Together, these two invariants ensure that the conditions presented in the super step definition (Definition 25) are not violated during the exploration.
The SuperStepPOR filter tries to extend $ss = \text{SuperStep}(n)$ by adding the next action from $\text{last\_thread}(n)$ to the super step. First, line 3, ensures that $n$ is not $u_0$ so there exists a last action. Second, line 3, also ensures that there exists a next action from the same thread ($\text{NextRep} \neq \emptyset$, there exists a successors state representation reached by taking an action from the same thread).

$\text{IndepLast}(ss, sn)$ and $\text{IndepInter}(next, sn)$, if true, ensure that the invariant $J_1$ is maintained by extending the super step with the action $\text{last\_action}(next)$.

$\neg \text{Enables}(\text{last\_action}(n), \text{Successors} \setminus \text{NextRep})$, if true, ensures that the invariant $J_2$ is maintained by extending the super step with the action $\text{last\_action}(next)$.

If the super step can be extended, then we can remove from Successors (line 12) all the successor state representations that are not reached by taking an action from the current thread.

If it is determined that the super step cannot be extended ($ss\_end = \text{true}$), then to maintain the invariants $J_1$ and $J_2$, $n$ has to be the end of the super step $ss$ and become a super node. This is exactly what happens in lines 16-25: a new super node corresponding to $n$ is created, and also new outgoing super steps are created for all the actions that are enabled in $n$ (for all the successors state representations).

The conditions presented in Definition 25 are not violated during the algorithm execution meaning that the algorithm explores $A_S$, a super step reduced graph of $A_R$. \hfill $\square$

**Theorem 11.** If a state $s$ in $A_G$ violates a safety property, then Algorithm 2 with Filters holding only Algorithm 6 terminates and explores a state $s'$ from $A_G$, with $s'$ violating the same safety property as $s$.

**Proof.** We apply Theorem 10 and we get that Algorithm 2 terminates and explores $A_S$, a super step reduced graph of $A_R$.

We apply Theorem 9 and we get that any safety violation that would be exposed by exploring $A_R$ is also exposed by just exploring $A_S$. \hfill $\square$
Chapter 5

SCALE - Design and Implementation

SCALE is divided into three major components. First, a Static Analysis And Code Instrumentation component finds, by analyzing the source code, the global statements and modifies the code around them to allow an external scheduler to control the way threads are interleaved. Second, a Scheduling component, explores the application’s state space by trying many different thread schedulings. During exploration, when stepped, a thread atomically executes the code between two consecutive global statements. SCALE checks if every visited state satisfies the specified properties. If a violation is found, then a trace of how to get to the error state is generated. Finally, a third Simulation component replays error traces allowing the bugs to be reproduced and analyzed at a later time.

SCALE’s static analysis and code instrumentation is written in OCaml, has about 300 lines of code, and was developed as a CIL [20] code transformation. The other components were developed in C and have around 11000 lines of code, most of which is for the scheduling component. SCALE was tested on x86, 32 bit and 64 bit, Linux platforms, and it should be easily ported to any platform that has: a C compiler, an OCaml compiler, the pthread library, and a mechanism for turning off Address Space Layout Randomization[8, 1, 3].

5.1 Static Analysis and Code Instrumentation

The purpose of SCALE’s static analysis and code instrumentation is to modify the verified program such that an external Scheduler process can control how the program’s threads are interleaved. Using CIL’s[20] points-to analysis and then a CIL transformation, we identify and replace the global statements described in section 3 with wrapped versions. Besides having the same functionality the original ones, the wrapped versions of the global statements, also implement the instrumented executable’s side of the Scheduling Protocol that will be presented in Section 5.2. If finer control is needed, the wrappers can also be inserted manually. There are wrappers for each kind of global statement: creating a new thread, locking and unlocking a mutex, reading and writing to a variable, etc. The wrappers are grouped together in a wrapper library so they can be easily reused. The
whole process is presented in Figure 5.1. At the end, we get an *instrumented executable* that uses the *Scheduling Protocol* to communicate with a *Scheduler*.

![Diagram](image)

Figure 5.1: SCALE’s static analysis and code instrumentation process

5.2 The SCALE Scheduling Protocol

The *Scheduling Protocol* describes the communication between the *instrumented executable* and the *Scheduler*. As a consequence of this communication, the *Scheduler* is able to control the way the *instrumented executable*’s threads are interleaved and also to inspect parts of the *instrumented executable*’s state.

5.2.1 Communication Channels

For each thread of the instrumented executable, there exists a set of two communication channels between the scheduler and that thread. One channel is used to send *commands* from the scheduler to the thread, and the other channel is used to send *command results* from the thread back to the scheduler. The interaction between the scheduler and the instrumented executable is depicted in Figure 5.2.

In each of the instrumented executable’s threads, each wrapped global statement contains, right before the original statement, a loop that reads commands sent by the scheduler. During execution, the application can be seen as a collection of threads, each blocked right before a global statement, waiting for commands. A thread step happens when the scheduler unblocks one of the threads. The unblocked thread exits the current loop and executes up to the next wrapped global statement where
it blocks again in a new loop. The process is repeated until the application ends or it is canceled by the scheduler.

5.2.2 Commands and Command Results

There are two types of commands: query commands and control commands. The scheduler uses query commands to get relevant information about the current state of the application, and control commands to either move the application to a successor state by scheduling a thread in the current state, or to exit the application. To allow multiple queries for the current state, commands are received in a loop. While query commands leave the thread who receives them in the same loop, control commands exit the current loop, and cause the execution to be either continued or aborted. Sending a command and receiving the result is done using synchronous communication and resembles a synchronous remote procedure call.

GetNextAction is a query command that asks a thread to report its next action (Definition 10) in the current state. The result structure that is sent back from the thread to the scheduler contains the following information:

- the identifier of the blocked thread;
- the operation performed by the global statement. It can be one of the following:
  - program start: the main thread is created and the execution starts in the initial state;
  - program exit: the program ends, and there are no more statements that can be executed;
  - thread creation: a new thread is created, the scheduler must know how to communicate with it;
  - thread start: a thread starts executing its statements, it is now active in the application;
  - thread exit: a thread is exited so its statements can no longer be executed;
  - thread join: wait for another thread to exit before proceeding;
  - read or write a global variable: access to a global variable;
  - synchronization primitive call: one for each primitive, for example locking or unlocking a mutex, waiting on a condition variable, etc.
– assertion: tests on the current state of the application that were specified in the source code;

• the list of communication variables that will being accessed by the global statement if the thread is scheduled in the current state. As described in Section 3, in SCALE, a communication variable is one of the following: a thread identifier, the memory location of a shared variable, a mutex handle, or a condition variable handle.

GetCurrentBackTrace is a query command that asks a thread to report its current function call stack. The result contains the following information:

• number of stack frames: how many functions are in the call stack, and also how many control points does the result contain;

• array of control points: one control point for each entry in the call stack. A control point is an abstraction of a stack frame and contains the following information:
  – level: the stack frame number, 0 for the top of the stack, 1 for the second one, and so on;
  – address: the address of the current instruction in the function’s code;

• stack hash: a hash of the current contents of the stack.

EvaluateFunctionCall is a query command that asks a thread to call a function in its current context and send the result back to the scheduler. The command contains the name of the function to be evaluated. The function takes no arguments and returns an integer. The result of the function call is sent back to the scheduler as a command reply. The Evaluate Function Call command is used for checking whether invariants are satisfied or not in the current state.

Step is a control command that asks a thread to exit the current command loop and start executing statements until the next global statement. The returned result is the next action corresponding to the newly reached global statement. The command spans between two consecutive global statements since its result can only be sent after the new global statement is reached. SCALE solves this problem by requiring the thread, right before entering a command loop, to send the scheduler a reply containing its next action.

Exit is a control command used by the scheduler to abort the current execution of the application. The receiving thread sends an acknowledgment back to the scheduler and then forcefully causes the whole application to exit.

5.3 The SCALE Scheduler

This section describes how SCALE’s Scheduler implements the algorithms presented in Chapter 4. These algorithms guide the state space exploration and check if any properties are violated.
As presented in figure 5.3, the scheduler has a layered implementation. At the lowest layer it has a component that communicates with the instrumented executable and allows the higher layers to schedule individual threads. The higher layers keep track of what states have been visited and decide what new states should be explored next. The following subsections describe these layers in more detail.

### 5.3.1 Backend

The **Backend** is a thin layer that completely abstracts away the communication with the instrumented executable’s threads. Together with the *wrapper library*, it implements the scheduling protocol described in Section 5.2. The sending and the receiving of scheduling commands and results are wrapped and presented to the layers above as API functions.

Next is a description of the **Backend** API:

- `bk_init()`: spawns a new process with the instrumented executable (which should block before the first global statement), and returns the next action for the main thread;

- `bk_exit()`: causes the application to be exited;

- `bk_restart()`: aborts the current run of the application, restarts its execution (which should block before the first global statement), and returns the next action for the main thread;

- `bk_step_thread(t)`: schedules thread `t`, runs it up to its next global statement, and returns the next action for thread `t`;

- `bk_get_back_trace(t)`: returns the current back trace of thread `t`;

- `bk_eval_function(t,func)`: evaluates the function `func` in thread `t`’s current context and returns the function’s result.

### 5.3.2 Frontend

The **Frontend** is a layer built on top of the backend which keeps track of the current configuration of the application (Definition 12).
A configuration contains the following information:

- the state of each thread: keeps track of each thread’s next action, and next step number.
- the state of each communication variable: each such variable is given a unique identifier. For every variable, the frontend keeps track of how many times it was written so far. We call this number the current version of the variable, and it is used to differentiate accesses to the same variable.
- the state of each synchronization primitive: for each mutex it stores the current holder thread, and for each condition variable, it keeps track of the set of waiting threads.
- the thread that was scheduled in the last step: this is the current thread.
- the new thread that was created during the last step (if one was created): the new thread is now active, was added to the current configuration, and from now on it can be scheduled.

Given the current configuration, the set of enabled threads and their next actions can be easily computed.

The frontend exposes a higher level API than the backend.

- \textit{fr\_init}(): starts the execution of the application and returns the initial configuration;
- \textit{fr\_exit}(): aborts the current run of the application;
- \textit{fr\_restart}: aborts the current run of the application, restarts the application, and returns the initial configuration;
- \textit{fr\_step\_thread}(t): steps thread \( t \) up to the next global statement, updates the current configuration accordingly, and returns the current configuration;
- \textit{fr\_get\_back\_trace}(): returns the result of the same backend API function, called for the last scheduled thread;
- \textit{fr\_check\_invariant(func)}: takes an invariant name as argument and uses the backend’s \textit{bk\_eval\_function} API call to determine if the given invariant is satisfied or not in the current thread’s context.

### 5.3.3 Generic Scheduler

Algorithm 7 presents how the \textit{Generic Scheduler} glues together the other parts of the state space exploration to implement the algorithms described in Chapter 4.
Algorithm 7 The Generic Scheduler

1: \texttt{config} = \texttt{fr\_init()}
2: \texttt{while true do}
3: \hspace{1em} \texttt{if violates\_properties(config) then}
4: \hspace{2em} \texttt{report\_error()}
5: \hspace{1em} \texttt{break}
6: \texttt{end if}
7: \hspace{1em} \texttt{dec} = \texttt{strategy\_decide(config)}
8: \hspace{2em} \texttt{if dec = Schedule(t) then}
9: \hspace{3em} \texttt{config} = \texttt{fr\_step\_thread(t)}
10: \hspace{2em} \texttt{else}
11: \hspace{3em} \texttt{if dec = Restart then}
12: \hspace{4em} \texttt{config} = \texttt{fr\_restart()}
13: \hspace{3em} \texttt{else}
14: \hspace{4em} \texttt{break}
15: \hspace{3em} \texttt{end if}
16: \hspace{2em} \texttt{end if}
17: \texttt{end while}
18: \texttt{fr\_exit()}

All scheduling decisions are deferred to a scheduling strategy by calling the \texttt{strategy\_decide} function. The strategy can decide to schedule a thread from the current state, restart the verified application from the initial state, or end the exploration. All verifications are deferred to the property checker by calling the \texttt{violates\_properties} function.

At any moment, the generic scheduling algorithm stores only the current configuration and has no mechanism for back tracking to a previously visited state.

5.3.4 Scheduling Strategy

The Scheduling Strategy is the component responsible for representing and storing visited states as happens-before graphs for actions (Definition 11).

Similarly to [19], we represent a happens before graph as a set of actions that have been extended with information about the edges.

Each such extended action is a tuple: \(< tid, step, op, al >\) where:

- \(tid\): is the identifier of the thread taking the action;
- \(step\): counts how many actions were taken by thread \(tid\) so far;
- \(op\): the operation performed by the action;
- \(al\): is a list of accesses to global variables. Each access is a tuple \(< v, at, w >\), where \(v\) is the variable identifier, \(at\) is the type of access and can be either \(READ\) or \(WRITE\), and \(w\) is a counter that gets incremented with each \(WRITE\). A \(READ\) refers to the last \(w\) value.
There is an edge from an extended action \(<tid_1, step_1, op_1, al_1>\) to an extended action \(<tid_2, step_2, op_2, al_2>\) if any of the following is true:

- \((tid_1 = tid_2) \land (step_1 < step_2)\);
- \(\exists (v_1, at_1, w_1) \in al_1 \land \exists (w_2, at_2, w_2) \in al_2 \text{ such that } (v_1 = v_2) \land (at_1 = WRITE) \land (w_1 \leq w_2)\);
- \(\exists (v_1, at_1, w_1) \in al_1 \land \exists (w_2, at_2, w_2) \in al_2 \text{ such that } v_1 = v_2 \land (at_1 = READ) \land (w_1 < w_2)\).

An advantage of this state representation is that the state space is stored efficiently, by constructing states incrementally, a child state is stored as the last action plus a link to the parent state. Moreover, as presented in [19] we can compute a hash of a state incrementally in the same way. The hash of a child state is the hash of the parent state xor'ed with the hash of the last action. Since xor is commutative, the same set of actions will have the same hash no matter in which order they are xor'ed. By comparing the hashes, we get a fast, but unsound, comparison between states. Also, storing the states incrementally makes some reduction algorithms easier to write.

When the exploration algorithm requires to replay a state \(s\), there are two situations:

- \(s\) is a successor of the current state reachable by executing an action \(a\), and in this case the \textit{strategy\_decide} function returns \textit{Schedule}(t), where \(t = \text{thread}(a)\);
- \(s\) is not a successor of the current state, and in this case the \textit{strategy\_decide} function returns \textit{Restart}. Assuming that \(a_1...a_n\) is a sequence of actions that leads from \(s_0\) to \(s\), subsequent calls of \textit{strategy\_decide} return in order \textit{Schedule}(t_i), where \(t_i = \text{thread}(a_i)\). When \(s\) is reached the scheduling strategy resumes normal operation.

When the state space exploration is done, the \textit{strategy\_decide} function returns \textit{Exit}.

### 5.3.5 Property Checker

The property checking is done by delegating the verification to specialized checkers, each looking for one of the following classes of errors: deadlocks, assertion failures, and invariant violations. The checkers are provided with the current configuration of the application, and are also allowed to evaluate expressions in the current application context. By default, SCALE searches for all of the above errors, except invariant violations. To also look for invariant violations, the user has to provide a property file that contains the names of the functions implementing the invariant tests. The functions should be callable from each thread, at any point and should return a boolean value, true if the invariant holds, false otherwise. SCALE reports all errors that it finds to the user, and also generates a trace file that contains all the scheduling decisions that were made, starting from the initial state, up to the error state.
5.4 Error Replay

The Simulation component runs the application and guides it on a path that is described in an input trace file. This component is used to replay errors discovered during verification.

The simulator is a simple loop that reads lines, one by one, from the trace file and steps the corresponding threads. The trace file is a list of thread identifiers, separated by white spaces. When replaying the trace, the threads are scheduled in the order in which they appear in the file. The simulator provides detailed information about what happened in each step: the line of code where the step started, what operation was performed, and what variables were accessed.

Before stepping a thread some checks are done. First, the thread must be active in the application, and second, it must be enabled in the current state. The simulator reports an error if these two conditions are not met.

Figure 5.4: The SCALE Simulator
Chapter 6

Experiments and Preliminary Results

6.1 Non-Blocking Queue Algorithm

We have implemented a C version of the non-blocking queuing algorithm presented in [23] and we have managed to discover a previously unknown bug in it. The full source code of our implementation is listed in Appendix A.1. We have limited the system to a producer and a consumer communicating via a shared non-blocking queue of size $N$. The producer enqueues the integers from 0 up to $N - 1$, and the consumer dequeues $N$ integers.

The queue is represented as a circular array of elements. Two indexes are stored, $FRONT$, with $FRONT \mod N$ being the position from where an item should be dequeued, and $REAR$, with $REAR \mod N$ being the position where an item should be enqueued. The queue is empty when $FRONT = REAR$, and full when $REAR = FRONT + N$. With each enqueue $REAR$ should increase by 1, and with each dequeue, $FRONT$ should increase by 1. Each queue element, if it is in use then it stores a pointer to the item, and if it is not in use then stores a NULL pointer.

Since it is a queue implementation, we expect the consumer to dequeue the same numbers that were enqueued by the producer and in the same order. We have found that this is not always true. Using SCALE to check the correctness of the algorithm’s implementation, we have determined scenarios in which a queue element is lost, it is enqueued by the producer, but it is skipped and never dequeued by the consumer. For example, with a queue of size $N = 2$, there is an execution in which the consumer first dequeues 1 instead of 0. The correct behavior of the consumer would be to dequeue 0, then 1, and then terminate.

For any $N \geq 2$, SCALE finds the following counter example, or a similar one: The consumer starts a dequeue operation on an empty queue ($REAR = FRONT$), reads and stores the queue element $x$ at $Q[FRONT \mod N]$. Since the queue is empty, the element at that position contains a NULL pointer. The consumer is preempted right before checking if $REAR = FRONT$. Next the consumer enqueues two items $I_1$ and $I_2$. The $REAR$ is updated and now we have $REAR = FRONT + 2$. The consumer is scheduled, and checks if the queue is empty ($REAR \not\equiv FRONT$) and notices that the
queue is not empty. It then compares the item pointer in its stale version of the queue element at \( REAR \ mod \ N \) \( (x) \) and finds that this pointer is \( NULL \). If this is the case, the consumer considers that there must be another consumer dequeuing that the item at that position at the same time. \( REAR \) is incremented in an attempt to help this other (nonexistent) consumer. So the consumer skips the item \( I_1 \) and dequeues item \( I_2 \) instead.

![Number of runs for NBQ up to the first counterexample](image)

**Figure 6.1:** Number of runs up to the first counterexample for the *non-blocking queue* implementation

In figure 6.1 we plot how many runs were necessary to find a counter example for various queue sizes \( N \). To show that the exploration order affects how fast the counterexample is found, we have used SCALE with 3 exploration algorithms:

(a) SCALE: the default DFS exploration which picks next actions in increasing order of the corresponding thread identifiers.

(b) SCALE(54): a DFS exploration that randomly picks the order in which the next actions are explored. The pseudo random number generator was seeded with the value 54.

(c) SCALE(29): same as (b), but with a seed of 29.

Note that not all runs are complete. SCALE restarts the program when it decides that running it further would not reach any unexplored states.

We have disabled the local cycle detection filter, and we have used only depth bounding, and we have measured the time required to restore a state as a function of the state’s depth. Figure 6.2 presents the results that were obtained on an Ubuntu 9.04 Linux system, running as a virtual machine in Mac OS X, on an Intel Core 2 Duo 2.4GHz processor. As expected, the time required to restore a state increases with the state’s depth. For large state depth values, the restore time becomes too large and the exploration is unfeasible.

We have configured SCALE to ignore the assertion violations in the non-blocking queue implementation, and we have measured the number of distinct state representations explored using different combinations of state filters. The results are presented in Figure 6.3. The non-blocking queue implementation contains busy-waiting loops, so if the local cycle detection filter is not used,
then the depth bounding filter must be used to ensure that the verification terminates. In Figure 6.3, NSB stands for the not seen before filter, LCD for the local cycle detection filter, SS for the super step partial order reduction filter, and DB for the depth bounding filter.

Figure 6.3 shows that using the super step partial order reduction filter in combination with the local cycle detection filter achieves a significant reduction compared to just using the not seen before filter and depth bounding.

Because the non-blocking queue implementation contains infinite executions we were unable to use Inspect to verify it. Inspect only works on terminating programs, and although it could bound the depth of the search, it doesn’t have this feature implemented.

CHESS bounds the depth of the search, but when the depth bound is reached it reports a livelock. The execution that exhibits a livelock is unfair and would not be explored by a fair scheduler.
6.2 Indexer

We have implemented the *indexer* example presented in [7] and compared SCALE’s results to those of Inspect and CHESS. SCALE and Inspect use the same Linux version of the source code, while CHESS uses the Windows version. The source code for the example (both the Linux and Windows versions) is presented in Appendix A.2. The *indexer* example was introduced to show the benefits of using Dynamic Partial Order Reduction versus other partial order reductions where the independence relation between transitions is precomputed by a static analysis of the source code. In the example, each thread reads and writes an element of an array. The index of the element is computed dynamically, and depends on the thread identifier. A static analysis would fail to determine which threads would access the same element. The example is so constructed that no conflicts occur up to 11 threads.

CHESS, Inspect and SCALE compute the independence relation between transitions dynamically while running the program. If necessary, all three tools restart the program from the initial state to explore additional paths.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Threads</th>
<th>Runs</th>
<th>Time(s)</th>
<th>Tool</th>
<th>Threads</th>
<th>Runs</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALE</td>
<td>1</td>
<td>1</td>
<td>0.041</td>
<td>SCALE</td>
<td>5</td>
<td>146</td>
<td>5.148</td>
</tr>
<tr>
<td>Inspect</td>
<td>1</td>
<td>1</td>
<td>0.182</td>
<td>Inspect</td>
<td>5</td>
<td>1</td>
<td>0.224</td>
</tr>
<tr>
<td>CHESS(2)</td>
<td>1</td>
<td>1</td>
<td>0.160</td>
<td>CHESS(2)</td>
<td>5</td>
<td>37927</td>
<td>88.110</td>
</tr>
<tr>
<td>SCALE*</td>
<td>1</td>
<td>1</td>
<td>0.078</td>
<td>SCALE*</td>
<td>5</td>
<td>89363</td>
<td>4863.917</td>
</tr>
<tr>
<td>SCALE</td>
<td>2</td>
<td>2</td>
<td>0.056</td>
<td>SCALE</td>
<td>6</td>
<td>818</td>
<td>37.881</td>
</tr>
<tr>
<td>Inspect</td>
<td>2</td>
<td>1</td>
<td>0.197</td>
<td>Inspect</td>
<td>6</td>
<td>1</td>
<td>0.207</td>
</tr>
<tr>
<td>CHESS(2)</td>
<td>2</td>
<td>73</td>
<td>0.156</td>
<td>CHESS(2)</td>
<td>6</td>
<td>257265</td>
<td>677.390</td>
</tr>
<tr>
<td>SCALE*</td>
<td>2</td>
<td>18</td>
<td>0.301</td>
<td>SCALE*</td>
<td>6</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SCALE</td>
<td>3</td>
<td>8</td>
<td>0.147</td>
<td>SCALE</td>
<td>7</td>
<td>8593</td>
<td>478.146</td>
</tr>
<tr>
<td>Inspect</td>
<td>3</td>
<td>1</td>
<td>0.180</td>
<td>Inspect</td>
<td>7</td>
<td>1</td>
<td>0.294</td>
</tr>
<tr>
<td>CHESS(2)</td>
<td>3</td>
<td>715</td>
<td>1.234</td>
<td>CHESS(2)</td>
<td>7</td>
<td>1885380</td>
<td>5663.672</td>
</tr>
<tr>
<td>SCALE*</td>
<td>3</td>
<td>309</td>
<td>11.229</td>
<td>SCALE*</td>
<td>7</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SCALE</td>
<td>4</td>
<td>32</td>
<td>0.728</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inspect</td>
<td>4</td>
<td>1</td>
<td>0.197</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHESS(2)</td>
<td>4</td>
<td>5448</td>
<td>10.937</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCALE*</td>
<td>4</td>
<td>5256</td>
<td>228.827</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Verification results for the *indexer* example

Table 6.1 presents the results obtained for different number of threads. We have measured the number of runs performed by each tool and the time necessary to complete the search. CHESS was run with a context switch bound of 2, so its search is not guaranteed to be complete. SCALE was run with the not seen before and super step partial order reduction filters enabled. There are no cycles in the example, so there is no need to use the local cycle detection filter. In SCALE* we have also disabled the super step partial order reduction filter. Inspect and SCALE were run on 32bit Ubuntu Linux in a virtual machine on top of Mac OS X on a MacBook Pro with an Intel Core 2 Duo 2.4GHz CPU. CHESS was run on 32bit Windows XP in a virtual machine on the same MacBook.
Pro. The results are also presented graphically in Figure 6.4.

Inspect uses the algorithm presented in [7] and so it manages to infer that a single run of the example is enough. CHESS can detect when the same state has been previously visited and so it is able to reduce the number of explored transitions, but not the number of explored states. As a consequence, it requires a lot more runs, even when context switch bounding is used. SCALE, using Super Step Partial Order Reduction, is able to infer that the threads are independent and can be considered as having just one super step, but is unable to deduce that executing any permutation of these threads has the same effect. We are currently investigating ways to include the Dynamic Partial Order Reduction algorithm presented in [7] in our framework and combine it with the rest of the reductions.

SCALE uses more memory than both Inspect and CHESS. While Inspect is stateless, it only keeps a stack of states for the current program run and does not store previously visited states, both SCALE and CHESS keep track of previously visited states. Because SCALE uses state filters it has to store more information for each state.

From Table 6.1 we can see that CHESS runs more tests per second than both Inspect and SCALE. The reason is that tests in CHESS are not separate programs, they are test cases run in the same process as the CHESS scheduler. Because consecutive runs of a test case happen in the same address space, state might be carried from one run to the other. The user must clean up everything at the end of the test case to ensure that any further runs have the same initial conditions. SCALE and Inspect run the scheduler and the verified program in two different processes, and use IPC mechanisms to communicate between them. While this approach adds communication overhead, it has the advantage that no cleanup code is necessary. The choice of IPC mechanism influences the performance, and for SCALE, we might get some improvement by switching from named pipes to shared memory.
6.3 Thread Pool

We have implemented a thread pool and verified it for absence of deadlock. The main thread creates \( n \) pool threads, enqueue a job and then signals the thread pool to stop. There are many possible scenarios: the job can be run by any of the threads or it can be ignored if the pool threads don’t get a chance to run before the main thread decides to stop them. We have implemented two versions: a Linux one, verified with SCALE and Inspect, and a Windows one, verified with CHESS. For the Linux implementation we used mutexes and condition variables to synchronize the threads. Conditions variables were not available in Windows, so we used semaphores and we tried to keep the implementation as close as possible to the Linux one. The source code of both versions is presented in Appendix A.3.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Tool} & \text{Size} & \text{Runs} & \text{Time(s)} \\
\hline
\text{SCALE} & 1 & 6 & 0.077 \\
\text{Inspect} & 1 & 3 & 0.169 \\
\text{CHESS}(2) & 1 & 15 & 0.630 \\
\text{SCALE*} & 1 & 6 & 0.077 \\
\hline
\text{SCALE} & 2 & 124 & 0.980 \\
\text{Inspect} & 2 & 102 & 2.788 \\
\text{CHESS}(2) & 2 & 367 & 0.672 \\
\text{SCALE*} & 2 & 136 & 1.020 \\
\hline
\end{array}
\]

Table 6.2: Verification results for the thread pool example

![Number of runs for the threadpool example](image)

![Verification time for the threadpool example](image)

Table 6.2 presents the results obtained for different thread pool sizes. CHESS was run with a context switch bound of 2, and so it does not guarantee the completeness of the search. SCALE has the not seen before and super step partial order reduction filters enabled. SCALE* has only the not seen before filter enabled. We can see that for larger thread pool sizes, 3 and 4 in the table, SCALE needs fewer runs to complete the verification, but this comes at the price of using more memory than both Inspect and CHESS. The same results are also presented in Figure 6.5.
6.4 Reusable Barrier

We have implemented a reusable barrier and verified it for various number of threads \((N)\) and steps \((STEPS)\). In each step each thread blocks at the barrier and waits for the other threads to reach the same point. When all the threads have reached the barrier, they all continue. This example turned out to be one in which the super step partial order reduction does not help much. We have implemented two versions of the barrier example, one for Linux and one for Windows. The Linux version uses mutexes and condition variables, while the Windows version uses critical sections and semaphores. The code is listed in Appendix A.4.

<table>
<thead>
<tr>
<th>Tool</th>
<th>N</th>
<th>STEPS</th>
<th>Runs</th>
<th>Time(s)</th>
<th>Tool</th>
<th>N</th>
<th>STEPS</th>
<th>Runs</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALE</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>0.068</td>
<td>SCALE</td>
<td>2</td>
<td>5</td>
<td>39365</td>
<td>430.832</td>
</tr>
<tr>
<td>Inspect</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0.107</td>
<td>Inspect</td>
<td>2</td>
<td>5</td>
<td>19683</td>
<td>1196.423</td>
</tr>
<tr>
<td>CHESS(2)</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>0.470</td>
<td>CHESS(2)</td>
<td>2</td>
<td>5</td>
<td>195</td>
<td>0.375</td>
</tr>
<tr>
<td>CHESS(5)</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>0.470</td>
<td>CHESS(5)</td>
<td>2</td>
<td>5</td>
<td>3066</td>
<td>9.150</td>
</tr>
<tr>
<td>SCALE*</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>0.076</td>
<td>SCALE*</td>
<td>2</td>
<td>5</td>
<td>39365</td>
<td>435.793</td>
</tr>
<tr>
<td>SCALE</td>
<td>2</td>
<td>2</td>
<td>54</td>
<td>0.428</td>
<td>SCALE</td>
<td>2</td>
<td>3</td>
<td>481</td>
<td>3.603</td>
</tr>
<tr>
<td>Inspect</td>
<td>2</td>
<td>2</td>
<td>27</td>
<td>0.857</td>
<td>Inspect</td>
<td>2</td>
<td>3</td>
<td>390</td>
<td>12.792</td>
</tr>
<tr>
<td>CHESS(2)</td>
<td>2</td>
<td>2</td>
<td>42</td>
<td>0.125</td>
<td>CHESS(2)</td>
<td>3</td>
<td>1</td>
<td>658</td>
<td>1.218</td>
</tr>
<tr>
<td>CHESS(5)</td>
<td>2</td>
<td>2</td>
<td>90</td>
<td>0.188</td>
<td>CHESS(5)</td>
<td>3</td>
<td>1</td>
<td>2685</td>
<td>4.656</td>
</tr>
<tr>
<td>SCALE*</td>
<td>2</td>
<td>2</td>
<td>54</td>
<td>0.432</td>
<td>SCALE*</td>
<td>3</td>
<td>1</td>
<td>481</td>
<td>3.663</td>
</tr>
<tr>
<td>SCALE</td>
<td>2</td>
<td>3</td>
<td>486</td>
<td>4.576</td>
<td>SCALE</td>
<td>3</td>
<td>2</td>
<td>431904</td>
<td>4119.880</td>
</tr>
<tr>
<td>Inspect</td>
<td>2</td>
<td>3</td>
<td>243</td>
<td>12.806</td>
<td>Inspect</td>
<td>3</td>
<td>2</td>
<td>683100</td>
<td>34764.863</td>
</tr>
<tr>
<td>CHESS(2)</td>
<td>2</td>
<td>3</td>
<td>81</td>
<td>0.204</td>
<td>CHESS(2)</td>
<td>3</td>
<td>2</td>
<td>12182</td>
<td>21.641</td>
</tr>
<tr>
<td>CHESS(5)</td>
<td>2</td>
<td>3</td>
<td>378</td>
<td>0.609</td>
<td>CHESS(5)</td>
<td>3</td>
<td>2</td>
<td>712912</td>
<td>1276.906</td>
</tr>
<tr>
<td>SCALE*</td>
<td>2</td>
<td>3</td>
<td>486</td>
<td>4.467</td>
<td>SCALE*</td>
<td>3</td>
<td>2</td>
<td>431904</td>
<td>4132.043</td>
</tr>
<tr>
<td>SCALE</td>
<td>2</td>
<td>4</td>
<td>4374</td>
<td>43.542</td>
<td>SCALE</td>
<td>4</td>
<td>1</td>
<td>143639</td>
<td>1245.443</td>
</tr>
<tr>
<td>Inspect</td>
<td>2</td>
<td>4</td>
<td>2187</td>
<td>85.904</td>
<td>Inspect</td>
<td>4</td>
<td>1</td>
<td>321008</td>
<td>9562.527</td>
</tr>
<tr>
<td>CHESS(2)</td>
<td>2</td>
<td>4</td>
<td>132</td>
<td>0.282</td>
<td>CHESS(2)</td>
<td>4</td>
<td>1</td>
<td>107307</td>
<td>210.641</td>
</tr>
<tr>
<td>CHESS(5)</td>
<td>2</td>
<td>4</td>
<td>1140</td>
<td>1.906</td>
<td>CHESS(5)</td>
<td>4</td>
<td>1</td>
<td>1668069</td>
<td>3239.891</td>
</tr>
<tr>
<td>SCALE*</td>
<td>2</td>
<td>4</td>
<td>4374</td>
<td>44.073</td>
<td>SCALE*</td>
<td>4</td>
<td>1</td>
<td>158758</td>
<td>1466.807</td>
</tr>
</tbody>
</table>

Table 6.3: Verification results for the reusable barrier example

In Table 6.3 SCALE has the not seen before and the super step partial order reduction filters enabled, while SCALE* has only the not seen before filter enabled. To observe the reduction power of context switch bounding, we have used CHESS with two context switch bounds: 2 and 5. From Table 6.3, we can see that an exploration with a context switch bound of 5 needs a lot more runs than one with a context switch bound of 2. Empirically it was observed that most bugs are found with a low context switch bound [17], so it would be useful to add this reduction to SCALE. The same results are also presented in Figure 6.6.

We have reasons to believe that Inspect reduces the state space too much so the search is no longer complete. For the two threads and one iteration experiment, after creating the additional thread, Inspect always continues to run the main thread and never tries to schedule the new thread first. The order in which the two threads are run is important since they have dependent transitions.
Figure 6.6: Verification results for the reusable barrier example

In all the runs checked by Inspect, the second thread is the last one that arrives at the barrier.

### 6.5 Other Interesting Examples

We have run SCALE on some buggy programs that pass either Inspect’s or CHESS’s checks and we were able to detect the errors. One such example is the producer-consumer problem in which we assert that when the consumer tries to take an element from the buffer, the buffer is empty and the consumer must wait for the producer to put an element in the buffer. It is easy to see that this assertion is violated if the producer runs first and manages to put an element in the buffer before the consumer tries to get an element from the buffer. Both CHESS and Inspect do not detect the assertion failure when the producer puts only one element in the buffer and the consumer takes only one element from the buffer. This only happens when the consumer is run in the main thread and the producer is run in another thread, if run the other way, the consumer in the new thread and the producer in the main thread, the error is detected by both CHESS and Inspect. The problem is that the producer does not get a chance to run immediately after it is created, instead the consumer is always scheduled first. The source code for this example (both the Linux and Windows versions)
is presented in Appendix A.5.

We can construct even simpler examples where the other tools miss some assertion failures. In Appendix A.6 we present such an example that has an execution leading to an assertion violation that is found by SCALE, but missed by Inspect. The lock is used to ensure that there are no race conditions.

The assertion failure happens when \texttt{thread\_b} runs before \texttt{thread\_a}. This is very likely to occur in practice and it can be easily exposed by compiling the program and running the resulting executable a few times.
Chapter 7

Conclusions and Future Directions

In this thesis we have presented the design and the implementation of the SCALE systematic software testing tool including its two major state space reduction algorithms: local execution cycle detection and super step partial order reduction.

We have applied SCALE to a C implementation of a non-blocking queuing algorithm and we were able to find a previously unknown bug. Using the algorithms presented in this thesis, the state space was reduced considerably, and this allowed us to find the error faster by taking fewer steps and exploring fewer distinct states. The local execution cycle detection allowed us to find the error even when we didn’t bound the depth of the search.

We have also applied SCALE to other applications and compared the results to those obtained by two other tools: CHESS and Inspect. In some cases SCALE’s algorithms prove to be useful in reducing the number of runs necessary to verify the applications. There are also cases where the partial order reduction is not that helpful. One way to improve SCALE’s performance is to reorganize the way states and the information associated with the state filters are stored so that they would take less space. One other way to improve SCALE’s performance is to use shared memory instead of named pipes as the IPC mechanism.

For large applications, SCALE’s backtracking mechanism, restarting the program from the initial state and following the same scheduling decisions, proves to be too expensive. Revisiting a state that is deep down in the state space graph, requires the execution of the application again up to that point. SCALE would benefit greatly from a check-pointing mechanism that would allow us to restart the application, not from the initial state, but from a closer ancestor state.

SCALE’s design allows us to easily add, enable and disable various reduction algorithms. For the future, we plan to add even more state space reductions. Preemptive context switch bounding is a promising reduction, and it appears that it would integrate well with the already existing ones. Dynamic partial order reduction is another algorithm that might prove to be useful. We also plan to add support for a scripting language that would allow users to write and test their own reduction and exploration algorithms.
As we have presented in this thesis, SCALE decouples the reduction algorithms from the exploration algorithms. So far we have been using only depth first search as our exploration algorithm, but we plan to implement and try others too. It also seems that it would be easy to adapt SCALE for distributed exploration and verification [12].

SCALE only deals with the non determinism introduced by thread scheduling. For the future we plan to also handle non determinism caused by user input and system calls.

SCALE was implemented to work on Unix/Linux applications that use the pthread API, but we see no reason why it could not be ported to other platforms, and be extended to support other multi-threading models.
Bibliography


Appendix A

A.1 Non-Blocking Queue Source Code

```c
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

#define N 10

struct element {
    int *val;
    unsigned int ref;
};

pthread_mutex_t cas_lock;
struct element Q[N];
unsigned int FRONT = 0;
unsigned int REAR = 0;

int CAS(struct element *shared, struct element *old, struct element *new) {
    int result = 0;
    pthread_mutex_lock (&cas_lock);
    if (shared->val == old->val && shared->ref == old->ref) {
        shared->val = new->val;
        shared->ref = new->ref;
        result = 1;
    } else {
        result = 0;
    }
    pthread_mutex_unlock (&cas_lock);
    return result;
}

int CASi (unsigned int *shared, unsigned int old, unsigned int new) {
    int result = 0;
    pthread_mutex_lock (&cas_lock);
    if (*shared == old) {
        *shared = new;
        result = 1;
    } else {
        result = 0;
    }
}
```
pthread_mutex_unlock(&cas_lock);
return result;
}

void enqueue(int *item) {
    unsigned int rear = 0;
    struct element x, tmp;
enq_try_again:
    rear = REAR;
    x.val = Q[rear % N].val;
    x.ref = Q[rear % N].ref;
    if (rear != REAR) {
        goto enq_try_again;
    }
    if (rear == FRONT + N) {
        goto enq_try_again;
    }
    if (x.val == NULL) {
        tmp.val = item;
        tmp.ref = x.ref + 1;
        if (CAS(&Q[rear%N], &x, &tmp)) {
            CASi(&REAR, rear , rear + 1);
            return;
        }
    } else {
        CASi(&REAR, rear , rear + 1);
    }
    goto enq_try_again;
}

int * dequeue () {
    unsigned int front = 0;
    struct element x, tmp;
deq_try_again:
    front = FRONT;
    x.val = Q[front % N].val;
    x.ref = Q[front % N].ref;
    if (front != FRONT) {
        goto dqq_try_again;
    }
    if (front == REAR) {
        goto dqq_try_again;
    }
    if (x.val != NULL) {
        tmp.val = NULL;
        tmp.ref = x.ref + 1;
        if (CAS(&Q[front%N], &x, &tmp)) {
            CASi(&FRONT, front , front + 1);
            return x.val;
        }
    } else {
        CASi(&FRONT, front , front + 1);
    }
    goto dqq_try_again;
}
void * producer_main (void *arg) {
    static int items[N];
    int i = 0;
    for (i = 0; i < N; i++) {
        items[i] = i;
        enqueue (&items[i]);
    }
    return NULL;
}

void * consumer_main (void *arg) {
    int *item = NULL;
    int i = 0;
    for (i = 0; i < N; i++) {
        item = dequeue ();
        assert (*item == i);
    }
    return NULL;
}

int main (int argc, char **argv) {
    pthread_t tid;
    pthread_mutex_init (&cas_lock, NULL);
    pthread_create (&tid, NULL, producer_main, NULL);
    consumer_main (NULL);
    pthread_join (tid, NULL);
    pthread_mutex_destroy (&cas_lock);
    return 0;
}

A.2 Indexer Source Code

A.2.1 Linux Version

#include <pthread.h>

#define SIZE 128
#define MAX 4
#define N 8

int table[SIZE];
pthread_mutex_t cas_mutex[SIZE];
pthread_t tids[N-1];

int cas (int *tab, int h, int val, int new_val) {
    int result = 0;
    pthread_mutex_lock(&cas_mutex[h]);
    if (tab[h] == val) {
        tab[h] = new_val;
        result = 1;
    }
    pthread_mutex_unlock(&cas_mutex[h]);
}
return result;
}

void thread_routine (int tid) {
    int m = 0, w, h;
    while(1){
        if ( m < MAX ) {
            w = (++m) * 11 + tid;
        } else {
            break;
        }
        h = (w * 7) % SIZE;
        while ( cas(table, h, 0, w) == 0) {
            h = (h+1) % SIZE;
        }
    }
}

void * thread_main (void *arg) {
    thread_routine ((int)(unsigned long)arg);
    return NULL;
}

int main (int argc, char **argv) {
    int i;
    for ( i = 0; i < SIZE; i++ ) {
        pthread_mutex_init(&cas_mutex[i], NULL);
    }
    for ( i = 0; i < N-1; i++ ) {
        pthread_create(&tids[i], NULL, thread_main, (void*)(unsigned long)(i));
    }
    thread_routine(N-1);
    for ( i = 0; i < N-1; i++ ) {
        pthread_join(tids[i], NULL);
    }
    for ( i = 0; i < SIZE; i++ ) {
        pthread_mutex_destroy(&cas_mutex[i]);
    }
    return 0;
}

A.2.2 Windows Version

/* indexer.cpp */
#include <windows.h>
#include <iostream>
#include <cassert.h>

#define SIZE 128
#define MAX 4
#define N 7
int table[SIZE];
CRITICAL_SECTION cas_mutex[SIZE];
HANDLE threads[N-1];
DWORD tids[N-1];

int cas (int *tab, int h, int val, int new_val) {
    int result = 0;
    EnterCriticalSection (&cas_mutex[h]);
    if ( tab[h] == val ) {
        tab[h] = new_val;
        result = 1;
    }
    LeaveCriticalSection (&cas_mutex[h]);
    return result;
}

void thread_routine (int tid) {
    int m = 0, w, h;
    while(1) {
        if ( m < MAX ) {
            w = (++m) * 11 + tid;
        } else {
            break;
        }
        h = (w * 7) % SIZE;
        while ( cas(table, h, 0, w) == 0) {
            h = (h+1) % SIZE;
        }
    }
}

DWORD WINAPI thread_main (LPVOID arg) {
    thread_routine ((int)(unsigned long)arg);
    return 0;
}

extern "C"
__declspec(dllexport) int ChessTestRun () {
    int i = 0;
    for (i = 0; i < SIZE; i++) {
        table[i] = 0;
        InitializeCriticalSection (&cas_mutex[i]);
    }
    for (i = 0; i < N-1; i++) {
        threads[i] = CreateThread (NULL, 0, thread_main,
                                  (LPVOID)(unsigned long)i, 0, &tids[i]);
    }
    thread_routine (N-1);
    for (i = 0; i < N-1; i++) {
        WaitForSingleObject (threads[i], INFINITE);
        CloseHandle (threads[i]);
    }
for (i = 0; i < SIZE; i++) {
    DeleteCriticalSection(&cas_mutex[i]);
}
return 0;

A.3 Thread Pool Source Code

A.3.1 Linux Version

#include <pthread.h>
#include <assert.h>
#include <stdio.h>

#define N 4

typedef void (*job)(void);

struct node {
    job job;
    struct node *next;
};

struct queue {
    struct node *head;
    struct node *tail;
};

struct queue jobs;
pthread_mutex_t lock;
pthread_cond_t wakeup_cond;
int exiting = 0;
int idle = 0;

job dequeue_job () {
    struct node *v = NULL;
    job job;
    v = jobs.head;
    job = v->job;
    jobs.head = v->next;
    free (v);
    if (jobs.head == NULL) {
        jobs.tail = NULL;
    }
    return job;
}

void enqueue_job (job job) {
    struct node *n = NULL;
    pthread_mutex_lock (&lock);
    n = (struct node *) malloc (sizeof (struct node));
    assert (n != NULL);
    n->job = job;
    P

n->next = NULL;
if (jobs.tail == NULL) {
    assert (jobs.head == NULL);
    jobs.head = n;
}
else {
    jobs.tail->next = n;
}
jobs.tail = n;
if (idle > 0) {
    pthread_cond_signal (&wakeup_cond);
}
pthread_mutex_unlock (&lock);

void * thread_main (void *arg) {
    job job;
    while (1) {
        pthread_mutex_lock (&lock);
        while ((jobs.head == NULL) && !exiting) {
            idle++;
            pthread_cond_wait (&wakeup_cond, &lock);
            idle--;
        }
        if (exiting) {
            pthread_mutex_unlock (&lock);
            break;
        } else {
            job = dequeue_job ();
            pthread_mutex_unlock (&lock);
            job ();
        }
    }
    return NULL;
}

void print_hello () {
    printf ("%s", "Hello!");
}

int main (int argc, char **argv) {
    pthread_t threads[N];
    int i = 0;
    pthread_mutex_init (&lock, NULL);
    pthread_cond_init (&wakeup_cond, NULL);
    for (i = 0; i < N; i++) {
        pthread_create (&threads[i], NULL, thread_main, NULL);
    }
    enqueue_job (print_hello);
    pthread_mutex_lock (&lock);
    exiting = 1;
    pthread_cond_broadcast (&wakeup_cond);
    pthread_mutex_unlock (&lock);
for (i = 0; i < N; i++) {
    pthread_join (threads[i], NULL);
}
pthread_mutex_destroy (&lock);
pthread_cond_destroy (&wakeup_cond);
return 0;
}

A.3.2 Windows Version

#include <windows.h>
#include <iostream>
#include <stdio.h>
#include <assert.h>
#define N 4

typedef void (*job)(void);

struct node {
    job job;
    struct node *next;
};

struct queue {
    struct node *head;
    struct node *tail;
};

struct queue jobs={NULL, NULL};
CRITICAL_SECTION lock;
HANDLE wakeup_sem;
int exiting = 0;
int idle = 0;

job dequeue_job () {
    struct node *v = NULL;
    job job;
    v = jobs.head;
    job = v->job;
    jobs.head = v->next;
    free (v);
    if (jobs.head == NULL) {
        jobs.tail = NULL;
    }
    return job;
}

void enqueue_job (job job) {
    struct node *n = NULL;
    EnterCriticalSection (&lock);
    n = (struct node *) malloc (sizeof (struct node));
    assert (n != NULL);
    n->job = job;
}
n->next = NULL;
if (jobs.tail == NULL) {
    assert (jobs.head == NULL);
    jobs.head = n;
} else {
    jobs.tail->next = n;
}
jobs.tail = n;
if (idle > 0) {
    ReleaseSemaphore (wakeup_sem, 1, NULL);
}
LeaveCriticalSection (&lock);
}

DWORD WINAPI thread_main (LPVOID param)
{
    job job;
    while (1) {
        EnterCriticalSection (&lock);
        while (((jobs.head == NULL) && !exiting) {
            idle++;
            LeaveCriticalSection (&lock);
            WaitForSingleObject (wakeup_sem, INFINITE);
            EnterCriticalSection (&lock);
            idle --;
        }
        if (exiting) {
            LeaveCriticalSection (&lock);
            break;
        } else {
            job = dequeue_job ();
            LeaveCriticalSection (&lock);
            job ();
        }
    }
    return 0;
}

void print_hello () {
    printf ("%s", "Hello!\n");
}

using namespace std;

void cleanup () {
    node *v = NULL;
    exiting = 0;
    idle = 0;
    while (jobs.head != NULL) {
        v = jobs.head;
        jobs.head = v->next;
        free (v);
    }
    jobs.head = NULL;
jobs.tail = NULL;
}

extern "C"
__declspec( dllexport ) int ChessTestRun() {
    DWORD tid[N];
    HANDLE hThread[N];
    unsigned int i = 0;
    cleanup();
    InitializeCriticalSection (&lock);
    wakeup_sem = CreateSemaphore (NULL, 0, N+1, NULL);
    for (i = 0; i < N; i++) {
        hThread[i] = CreateThread (NULL, 0, thread_main,
                                   (LPVOID)(unsigned long)i, 0, &tid[i]);
    }
    enqueue_job (print_hello);
    EnterCriticalSection (&lock);
    exiting = 1;
    ReleaseSemaphore (wakeup_sem, idle, NULL);
    LeaveCriticalSection (&lock);
    for (i = 0; i < N; i++) {
        WaitForSingleObject (hThread[i], INFINITE);
        CloseHandle(hThread[i]);
    }
    DeleteCriticalSection (&lock);
    CloseHandle (wakeup_sem);
    return 0;
}

A.4 Barrier Source Code

A.4.1 Linux Version

#include <pthread.h>

#define N 4
#define STEPS 1

pthread_mutex_t lock;
pthread_cond_t cond;
int count = 0;
int phase = 0;

void phase_0 () {
    pthread_mutex_lock (&lock);
    count++;
    if (count == N) {
        phase = 1;
        pthread_cond_broadcast (&cond);
    } else {
        while (phase == 0) {
            pthread_cond_wait (&cond, &lock);
        }
    }
}
void phase_1 () {
    pthread_mutex_lock (&lock);
    count--;
    if (count == 0) {
        phase = 0;
        pthread_cond_broadcast (&cond);
    } else {
        while (phase == 1) {
            pthread_cond_wait (&cond, &lock);
        }
    }
    pthread_mutex_unlock (&lock);
}

void barrier () {
    phase_0 ();
    phase_1 ();
}

void process (void *arg) {
    int i = 0;
    for (i = 0; i < STEPS; i++) {
        barrier ();
    }
}

void * thread_main (void *arg) {
    process (arg);
    return NULL;
}

int main (int argc, char **argv) {
    pthread_t threads[N-1];
    int i = 0;
    pthread_mutex_init (&lock, NULL);
    pthread_cond_init (&cond, NULL);
    for (i = 0; i < N-1; i++) {
        pthread_create (&threads[i], NULL, thread_main, NULL);
    }
    process (NULL);
    for (i = 0; i < N-1; i++) {
        pthread_join (threads[i], NULL);
    }
    pthread_mutex_destroy (&lock);
    pthread_cond_destroy (&cond);
    return 0;
}

A.4.2 Windows Version

#include <windows.h>
#include <iostream>
#include <cassert.h>

#define N 3
#define STEPS 2

CRITICAL_SECTION lock;
HANDLE sem1, sem2;
unsigned int count = 0;

void phase_1 () {
    int signaller = 0;
    EnterCriticalSection (&lock);
    count++;
    if (count == N) {
        signaller = 1;
        ReleaseSemaphore (sem1, N-1, NULL);
    }
    LeaveCriticalSection (&lock);
    if (!signaller) {
        WaitForSingleObject (sem1, INFINITE);
    }
}

void phase_2 () {
    int signaller = 0;
    EnterCriticalSection (&lock);
    count--;
    if (count == 0) {
        signaller = 1;
        ReleaseSemaphore (sem2, N-1, NULL);
    }
    LeaveCriticalSection (&lock);
    if (!signaller) {
        WaitForSingleObject (sem2, INFINITE);
    }
}

void barrier () {
    phase_1 ();
    phase_2 ();
}

DWORD WINAPI thread_main(LPVOID param) {
    int i = 0;
    for (i = 0; i < STEPS; i++) {
        barrier ();
    }
    return 0;
}

using namespace std;

extern "C"
__declspec(dllexport) int ChessTestRun() {

DWORD tid[N-1];
HANDLE hThread[N-1];
unsigned int i = 0;
count = 0;
InitializeCriticalSection (&lock);
sem1 = CreateSemaphore (NULL, 0, N-1, NULL);
sem2 = CreateSemaphore (NULL, 0, N-1, NULL);
for (i = 0; i < N-1; i++) {
    hThread[i] = CreateThread (NULL, 0, thread_main,
        (LPVOID)(unsigned long)i, 0, &tid[i]);
}
thread_main ((LPVOID)(unsigned long)(N-1));
for (i = 0; i < N-1; i++) {
    WaitForSingleObject (hThread[i], INFINITE);
    CloseHandle(hThread[i]);
}
DeleteCriticalSection (&lock);
CloseHandle (sem1);
CloseHandle (sem2);
return 0;

A.5  Producer-Consumer Source Code

A.5.1 Linux Version

#include <assert.h>
#include <pthread.h>

#define SIZE 2
#define N 1

struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t cond;
    int value;
    int flag;
};

void semaphore_init (struct semaphore *sem, int value) {
    pthread_mutex_init (&sem->lock, NULL);
    pthread_cond_init (&sem->cond, NULL);
    sem->value = value;
    sem->flag = 0;
}

void semaphore_destroy (struct semaphore *sem) {
    pthread_mutex_destroy (&sem->lock);
    pthread_cond_destroy (&sem->cond);
}

void semaphore_wait (struct semaphore *sem) {
    int tvalue;
    pthread_mutex_lock (&sem->lock);

tvalue = sem->value;
tvalue--;
sem->value = tvalue;
if (tvalue < 0) {
    while (sem->flag == 0) {
        pthread_cond_wait (&sem->cond, &sem->lock);
    }
    sem->flag = 0;
}
pthread_mutex_unlock (&sem->lock);

void semaphore_post (struct semaphore *sem) {
    int tvalue;
    pthread_mutex_lock (&sem->lock);
    tvalue = sem->value;
tvalue++;
    sem->value = tvalue;
    if (tvalue <= 0) {
        sem->flag = 1;
        pthread_cond_signal (&sem->cond);
    }
    pthread_mutex_unlock (&sem->lock);
}

pthread_mutex_t mutex;
struct semaphore items, spaces;
unsigned int fill = 0;
unsigned char buffer[SIZE];

void * producer (void *arg) {
    int i;
    unsigned char item;
    for (i = 0; i < N; i++) {
        item = (unsigned char)(i%256);
        semaphore_wait (&spaces);
        pthread_mutex_lock (&mutex);
        buffer[i%SIZE] = item;
        fill++;
        pthread_mutex_unlock (&mutex);
        semaphore_post (&items);
    }
    return NULL;
}

void * consumer (void *arg) {
    int i = 0;
    unsigned char item;
    assert (fill == 0);
    for (i = 0; i < N; i++) {
        semaphore_wait (&items);
        pthread_mutex_lock (&mutex);
        item = buffer[i%SIZE];
assert (item == (unsigned char)(i%256));

pthread_mutex_unlock (&mutex);

semaphore_post (&spaces);

} return NULL;

} int main (int argc, char **argv) {

pthread_t t;
pthread_mutex_init (&mutex, NULL);

semaphore_init (&items, 0);

semaphore_init (&spaces, SIZE);

pthread_create (&t, NULL, producer, NULL);

consumer (NULL);

pthread_join (t, NULL);

pthread_mutex_destroy (&mutex);

semaphore_destroy (&items);

semaphore_destroy (&spaces);

return 0;

} A.5.2 Windows Version

#include <windows.h>
#include <iostream>
#include <assert.h>

#define SIZE 2
#define N 1

CRITICAL_SECTION mutex;
HANDLE items, spaces;

unsigned int fill = 0;

unsigned char buffer[SIZE];

DWORD WINAPI producer (LPVOID param) {

int i;

unsigned char item;

for (i = 0; i < N; i++) {

item = (unsigned char)(i%256);

WaitForSingleObject (spaces, INFINITE);

EnterCriticalSection (&mutex);

buffer[i%SIZE] = item;

fill++;

LeaveCriticalSection (&mutex);

ReleaseSemaphore (items, 1, NULL);

}

return 0;

}

DWORD WINAPI consumer(LPVOID param) {

int i = 0;

unsigned char item;
for (i = 0; i < N; i++) {
    assert (fill == 0);
    WaitForSingleObject (items, INFINITE);
    EnterCriticalSection (&mutex);
    item = buffer[i%SIZE];
    fill --;
    assert (item == (unsigned char)(i%256));
    LeaveCriticalSection (&mutex);
    ReleaseSemaphore (spaces, 1, NULL);
}
return 0;
}

using namespace std;

extern "C"
__declspec(dllimport) int ChessTestRun() {
    DWORD tp;
    HANDLE prod;
    unsigned int i = 0;
    fill = 0;
    InitializeCriticalSection(&mutex);
    items = CreateSemaphore (NULL, 0, SIZE, NULL);
    spaces = CreateSemaphore (NULL, SIZE, SIZE, NULL);
    prod = CreateThread (NULL, 0, producer, NULL, 0, &tp);
    consumer (NULL);
    WaitForSingleObject (prod, INFINITE);
    DeleteCriticalSection(&mutex);
    CloseHandle (prod);
    CloseHandle (items);
    CloseHandle (spaces);
    return 0;
}

A.6 Bug Inspect

#include <pthread.h>
#include <assert.h>

int x = 2;
pthread_mutex_t lock;

void * thread_a (void *arg) {
    int tmp;
    pthread_mutex_lock (&lock);
    tmp = x;
    pthread_mutex_unlock (&lock);
    tmp++;
    pthread_mutex_lock (&lock);
    x = tmp;
    pthread_mutex_unlock (&lock);
    return NULL;
}
void * thread_b (void *arg) {
    int tmp;
    pthread_mutex_lock (&lock);
    tmp = x;
    pthread_mutex_unlock (&lock);
    tmp *= 2;
    pthread_mutex_lock (&lock);
    x = tmp;
    pthread_mutex_unlock (&lock);
    return NULL;
}

int main (int argc, char **argv) {
    pthread_t p;
    pthread_mutex_init (&lock, NULL);
    pthread_create (&p, NULL, thread_b, NULL);
    thread_a (NULL);
    pthread_join (p, NULL);
    pthread_mutex_destroy (&lock);
    assert (x != 5);
    return 0;
}