VERIFICATION OF MULTI-THREADED C PROGRAMS

Matthew Pigram

Bachelor of Engineering
Computer Engineering

Department of Engineering
Macquarie University

November 2016

Supervisor: Assoc. Prof. Franck Cassez
Co-Supervisor: Assoc. Prof. Tony Sloane

ACKNOWLEDGMENTS

Assoc. Prof. Franck Cassez and Assoc. Prof. Tony Sloane for their attentiveness and advice throughout the course of this project, as well as for their contributions to Skink and its related libraries in support of my own implementation work.

The other members of the Skink group, Matt Roberts, Pablo Gonzalez De Aledo and Pongsak Suvanpong, whose prior and ongoing work provides a basis for my own.

David, for his thoughtful feedback and diligent proof-reading.

Marion, for being my favourite distraction.

STATEMENT OF CANDIDATE

I, Matthew Pigram, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment at any academic institution.

Student's Name: Matthew Pigram

Student's Signature: Matthew Pigram

Date: November 7, 2016

ABSTRACT

Verifying the correctness of a program involves providing a guarantee, in the form of a logical proof, that the program is free of bugs for all possible inputs. It is therefore able to provide software developers and users with a much higher degree of confidence in a program's ability to perform its job than traditional testing, due to the fact that it produces certainty rather than empirical evidence of a program's partial correctness, something which is especially important when that program is responsible for large sums of money or human lives. Multi-threaded programs are able to take advantage of modern multi-core architectures more effectively than single-threaded programs but introduce challenges both to the author and the verifier. Concurrent programs are regarded as being more difficult to write and understand than traditional sequential programs by developers due to large number of ways in which the multiple threads may interact throughout a program's execution. As such, the ability to formally verify this type of software is perhaps even more valuable than traditional single-threaded programs [19]. For verification, the combinatorial number of ways in which the operations of each thread may be interleaved by a non-determinstic scheduler introduces the problem of *state-space explosion*. Here, we describe a method for adapting *trace abstraction refinement* [15], an existing technique for verification of single-threaded programs for application to multi-threaded programs in combination with an approach to reduce the state space of the program based on *dynamic partial order reduction* [1]. We apply this method to enable the verification of C programs, building on the work on this topic by Cassez and Zeigler in [8].

Contents

List of Figures

Chapter 1

Introduction

Providing formal proof of a program's correctness is a difficult problem, particularly when it is applied to large or complicated pieces of software which have many inputs and possible paths of execution. As with an increasing number of things, the level of complexity and volume of work involved in providing a formal correctness proof for a non-trivial program mean that this task is better suited to computers than to humans. This gives rise to a desire to produce algorithmic solutions that can automate the process. The benefits of proving the correctness of a program, rather than the more usual procedure of testing on a subset of inputs in order to gain some statistical confidence in a program's behaviour, have become increasingly apparent as bugs that traditional methods failed to find have been uncovered by automated formal verification tools [10].

As multi-core CPU architecture has become more prevalent due to its manifold gains over traditional single core architectures so too have multi-threaded programs, which are best placed to take advantage of the possibility for truly concurrent processing that is provided by this new hardware. Multi-threaded programs are regarded as being more difficult to write and understand than traditional single-threaded programs by developers due to large number of ways in which the multiple threads may interact throughout a program's execution. As such, the ability to formally verify this type of software is perhaps even more valuable than traditional single-threaded programs [19].

However, the challenges that mutli-threaded programs present to human programmers apply also to automated formal verification techniques, as the multitude of ways in which threads may communicate and be executed produces a multiplicative relationship between the set of paths taken by each thread and the set of values taken by the variables used by each thread. This leads to an exponential growth in the number of states or paths that must be explored when seeking to provide a correctness proof for a multi-threaded program, a problem which is typically referred to as *state-space explosion*.

1.1 Aims

Our work aims to adapt a well-studied technique for the verification of single-threaded programs, *trace abstraction refinement* [15], to deal with multi-threaded programs and

to address the problem of state-space explosion through the application of *partial order reduction* [1,11,12]. We seek to provide an implementation of this approach for verification of multi-threaded programs as part of an existing verification tool, Skink[1] in order to demonstrate a working example of how it may be applied in the specific case of C programs using the POSIX thread library. To perform experimental evaluation of our work, the final aim of the project is to compete in the software verification competition, SV-COMP[2], which will allow our implementation to be benchmarked against other verification tools in a controlled environment with a large body of tests.

In order to achieve this, the existing representation of programs by the trace abstraction refinement approach must be extended to represent programs with multiple threads and also to allow for synchronisation between these threads on various events. It is also necessary to adapt the trace abstraction refinement representation of programs with an implementation of a partial order reduction algorithm to allow the fewest possible number of traces to be explored during a program's verification.

Skink forms an ideal platform for this work for a number of reasons. First, the presence of an existing codebase provides a large amount of the trace abstraction refinement machinery which is general to both the single and multi-threaded cases. It also provides all the necessary code and surrounding libraries for translating and transforming inputted C code, which often forms a large and burden some part of the implementation verification tools. Finally, it provides the potential for our implementation to exist as a feature of a general purpose verification tool, rather than as a stand-alone program which is designed solely for verification of properties of multi-threaded programs, as was the case with the three most successful tools that participated the Concurrency category of SV-COMP 2016 [2].

1.2 Our Contributions

In this report we present the advantages of using an intermediate representation language (LLVM IR) rather than the original source language as a basis for verification and describe a scheme for the dynamic construction of an automaton which can be to used represent a synchronised concurrent program under the POSIX threads model. We also discuss the techniques applied for the synchronisation of the automaton which disallows the exploration of execution paths which violate the synchronisation semantics of the program and how existing trace abstraction refinement processes can be adapted to suit our approach. As an extension of our initial work in verifying concurrent programs we present an adapted version of the Source-DPOR algorithm defined in [1] for our application and describe a generic implementation of it. We demonstrate the ability of our automaton to be used transparently for synchronous and concurrent programs, supporting our claim that the adaption of trace abstraction refinement for concurrent programs need not harm its ability to verify synchronous ones. We then provide initial results of our approach

[1] https://bitbucket.org/inkytonik/skink/
[2] https://sv-comp.sosy-lab.org/

being applied successfuly to a subset of the SV-COMP concurrency benchmarks.

A record of my interactions with my supervisor Franck Cassez and the other members of the Skink group throughout the progress of my work on this topic is provided in Appendix A.

Chapter 2

Background and Related Work

2.1 SV-COMP

The International Competition on Software Verification (SV-COMP) is a competition which allows developers of verification tools for a variety of applications to measure their success against others, and to provide an opportunity for those involved in software verification to observe the development of differing approaches as the techniques and their implementation become more mature.

All the programs used as part of the competition are open source, available on Github[1], and consist of programs which have been contributed by participants in the competition. At present, the only programs applied as part of the competition are written in C, with different categories consisting of programs which involve the use of different language features and libraries, including dynamic memory allocation with malloc, loops, floating point calculations and also large, real world programs such as device drivers.

Programs within the SV-COMP benchmark corpus are annotated with special purpose functions[2] which provide verification tools with a means to recognise error locations within a program that are being tested for reachability, or to simulate other properties on the program such as atomicity and non-determinism. These functions are members of the `__VERIFIER` family, with a `__VERIFIER_error` function corresponding to an error location within the control flow of the program.

The concurrency category of SV-COMP contains five sub-categories, all contributed by different SV-COMP competitors and all relying on the POSIX thread library for their implementation of concurrent benchmarks. The primary sub-category targeted as part of the work described in this report is the base "pthread" category, which includes mostly simple benchmarks which do not require support for a dynamic memory model (something which is not currently handled by Skink) but which allow all the basic operations necessary for dealing with concurrent program verification to be correctly implemented.

In recent years of the SV-COMP concurrency category, the most successful tools have been those based on sequentialisation [2], which is the process of converting a concurrent

[1] https://github.com/sosy-lab/sv-benchmarks
[2] https://sv-comp.sosy-lab.org/2017/rules.php

program into an equivalent sequential program and then simulating concurrency by modeling a non-deterministic scheduler in the main function of the program [18, 22]. From here the program can be verified by a sequential bounded model checker, which explores error paths through the target program with a number of steps up to a bound k and computes their feasibility in a similar manner to trace abstraction refinement. This is achieved by mapping the execution to a a logical representation of the trace which can be analysed by an SMT solver (further background for which can be found in Section 2.6). The fundamental difference of bounded model checking approaches to our own is that they do not attempt to verify all possible paths of execution in a program but rather a reasonable subset (up to an arbitrary bound). If no feasible error path is found that fits within the bound, then the program is assumed to be correct. This means that bounded model checking cannot provide a proof of total correctness for a program, but only that the program is correct within the bounds that were explored.

2.2 POSIX Threads

The POSIX threads (Pthreads) API is a language agnostic framework for the implementation of concurrent programs defined as part of the POSIX ISO standard and which provides a simple but powerful set of functions to allow the creation, synchronisation and destruction of threads. At a higher level POSIX threads define an execution model for concurrent programs which is the defacto standard for all Unix-like operating systems and for C systems programming, providing the motivation for their use in the SV-COMP concurrency category.

In the Pthread model, each new thread within a concurrent program is associated with a unique `pthread_t` identifier and is passed a pointer to a function which it is to execute and may begin to execute any time after the `pthread_create` call which initalises it has returned. Arguments can be passed to the thread's function via a block of untyped memory and return values collected by the thread creator by the same means. A single function may be run by multiple threads within a program and any local variables within the function will be distinct for each thread. Communication between threads while they are running is achieved via global variables, which can be of any type admitted in C, with additional types defined by the Pthread API available for synchronisation.

The Pthreads API provides two types that can be used for synchronisation and a set of accompanying functions. The `pthread_mutex_t` type provides a mutual exclusion (mutex) token, which can be used to synchronise access to global variables, and which is managed with the `pthread_mutex_lock` and `pthread_mutex_unlock` functions. Alongside mutexes are conditions, which can be used as a means for a thread to explicitly "wake" another thread in the program by signalling a change to a condition. This is achieved with the use of the `pthread_cond_t` type and the accompanying `pthread_cond_wait` and `pthread_cond_signal` functions. As is typical for wait operations in concurrency models, the `pthread_cond_wait` function requires the calling thread to be holding a mutex, which is unlocked when the calling thread is blocked and re-locked when the condition is signaled

and the thread unblocks. This is necessary in order to prevent undefined behaviour when multiple threads are waiting on the same condition.

The final synchronisation method available in the Pthreads API is `pthread_join` which allows for a thread to wait until another thread is completed before resuming, this is most typically by the main thread to ensure that its child threads are able to complete their work before the main thread resumes. A thread is eligible to be joined after a call to `pthread_exit` has been made on that thread, with the exit status of the thread passed back to the parent via an untyped block of memory, as with arguments and return values. A parent thread may also choose to kill a child thread using the `pthread_cancel` function, which will cause the thread to exit if and when it enters a cancellable state.

A number of extensions exist to the standard Pthread API as part of the common implementation, GNU's libpthread, but which we choose not to model as part of our work as they are not used within the SV-COMP benchmark corpus and because their behaviour cannot be modeled in a platform agnostic fashion.

Although the choice to target Pthreads as the first concurrency model for our verification was made for us by their use in SV-COMP, the small interface and relatively well defined semantics of the model makes it well suited for our approach and allows us to reason staticially about the behaviour of the program quite easily. This is particularly useful when constructing automata which represent concurrent programs using the Pthread API, something which is discussed in detail in Section 3.4.

2.3 Trace Abstraction Refinement

Trace abstraction refinement, as introduced by Heizmann et al. in [15] and expounded upon in [16], is a development of previous *counterexample-guided abstraction refinement* (CEGAR) techniques, involving an iterative process by which the feasibility of error traces (ie. paths through a program which lead to an error state) is examined. In this process, each trace is a word the language accepted by a finite automaton which represents the control flow graph (CFG) of the program. Like many verification techniques, trace abstraction refinement relies on an automated theorem prover (ATP) to decide the feasibility of a particular trace but seeks to minimise the number of queries to the ATP. The existence of ATPs as crucial element of CEGAR verification strategies presents a significant obstacle to the scalability of these techniques, which trace abstraction refinement seeks to address by reducing the number of calls made to the ATP and re-using as much of the ATP work as possible during the verification of a program. Through the exhaustion of all error traces within a program, trace abstraction refinement produces sound verification of a program, meaning that if the program is asserted to be correct, then it is must be correct. This property comes at the expense of completeness, the lack of which means that trace abstraction refinement is not guaranteed to return a result for all correct input programs.

The primary technique for achieving a reduction in ATP calls is through the introduction of interpolant automata, which are produced from the automata representing an

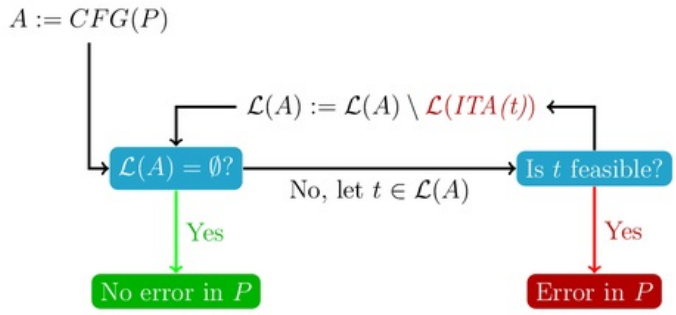


Figure 2.1: Synchronous Trace Abstraction Refinement Loop Adapted from [5]

individual trace to provide using interpolants which are returned as part of the ATP's infeasibility proof for a trace. This provides the potential for a single call to the ATP to remove multiple traces from the trace abstraction of program as part of each refinement step. The use of trace abstraction also allows for ATP expressions to be produced directly from program semantics, without any need for translations in terms of the abstraction used to represent the program. The use of interpolant automata also provides a degree of generality to the abstractions used as part of the verification of an individual program, allowing caching of previously computed results from the ATP and further reducing reliance on ATP calls for common program constructs.

In comparison to predicate abstraction, trace abstraction provides a higher level and arguably simpler view of a particular execution path in a program. Trace abstraction also represents the parent case of predicate abstractions (proven in [15]) as any program state represented by such an abstraction can be re-constrcuted with the appropriate trace whose execution will result in that particular set of predicates.

Fig. 2.1 shows an example of the trace abstraction refinement process for the single threaded case described by Heizmann et al., which further exemplifies the intuitive nature of this approach to verification. Having produced a regular language which accepts traces that reach an error location in the program's CFG, we check the feasibility of words in the language, corresponding to traces in the program, using an ATP. If the trace is infeasible, we seek to construct an interpolant automaton, which accepts the infeasible trace and potentially other equivalent traces, and then remove the regular language generated by this automaton from our original abstraction, completing one iteration of the refinement process. If we reach a point where our original language is empty, we have proved that there exists no feasible trace through the program which leads to an error location, and thus that the program is correct (in the sense that it does not violate any of the predicates that have been annotated within the program). A more complete example of the refinement process for a simple case is given in [15], with an example more directly related to our work given later in this thesis.

The fact that trace abstraction requires that each trace to be represented with an individual automaton allows for simple and well defined set theoretic and graph oper-

ations. This allows transformations to be applied on the program without the need to determine its semantics. This is distinct in comparison to predicate based abstraction, which requires additional ATP calls in order to produce a combination of abstractions.

The benefits of the system of abstraction described in [16] also has advantages over techniques that require the construction of a single monolithic automaton which relates directly to the structure of the CFG of the program being verified. As such, such an approach requires that the structure of the automata representing each trace should map directly to the structure of the CFG in order to ensure all traces are covered by the resulting automaton. By doing away with the requirement that an automaton representing a trace must directly reflect the structure of the CFG, it is possible to increase the size of the set of traces which are eliminated with a single ATP call.

By building on the trace abstraction refinement techniques described in [15], we seek to supplement a state of the art technique for CEGAR based software verification and allow concurrent programs to be verified as part of a more general verification tool, rather than one specifically written for that type of program.

2.4 Partial Order Reduction

The foundational work in partial order reduction for verification of concurrent programs is [12], in which Godefroid specifies a technique for addressing the state-space explosion problem by introducing several techniques to reduce the number of interleavings of operations between threads which need to be explored. The term partial order reduction comes from the notion that it is possible to identify operations on each thread of a multi-threaded program which are dependent and operations which are independent.

For dependent operations, the order in which these operations are executed is important when considering the overall correctness of the program, as different interleavings of these operations (the scheduling of which is non-deterministic) may cause the program to behave in different ways. In contrast to this, if two threads are performing purely independent operations (for example each has its own completely exclusive set of variables) then all possible interleavings of these operations will result in the same effective execution of the program.

In [12], Godefroid describes programs in terms of states and traces as the paths for reaching a particular state in a program. Partial order reduction is applied by first identifying operations which are regarded as being dependent and then applies a `SelectiveSearch` algorithm to remove equivalent interleavings of independent operations and generate an automaton representing a reduced set of paths of execution through the threads of the program.

`SelectiveSearch` as described by Godefroid can be applied with one or both of *persistent sets* and *sleep sets*, algorithms for the computation of which are described in [12]. These two sets represent the operations from a particular state which are used to identify which will produce a non-equivalent interleaving in the case of persistent sets or which do not need to be explored further, in the case of sleep sets. The primary difference

between persistent sets and sleep sets is in how they are computed, with persistent sets being found statically from the structure of the program being verified, while sleep sets are computed dynamically based on the path of the search as it is applied. Godefroid also describes how these two sets can be combined to achieve the most effective reduction in interleavings explored.

This technique for partial order reduction is described as static partial order reduction, owing to the fact that it is applied as a pre-computation on an input program before model checking is applied and has as its output a trace automaton which is essentially a transformation of the input program upon which the verification algorithm may then compute. More recently, a dynamic method for applying partial order reduction to reduce the state-space of concurrent programs while a model checking algorithm is being applied is defined in [11] and improved upon in [1].

In [11], an algorithm for dynamic partial order reduction is described, introducing the notion of *backtracking points* which provide a means for the program to retrospectively compute possible traces which maybe have been taken to reach a particular program state as these states are encountered during the verification of a program. Flanagan and Godefroid [11] also describe the newly defined dynamic partial order reduction algorithm as being easier to implement in comparison to static partial order reduction as it does not require static analysis of the program prior to execution. As a byproduct of this lack of pre-computation, dynamic partial order reduction is able to handle dynamic changes in the structure of the program (such as the creation or joining of threads). Dynamic partial order reduction is also described by Flangan and Godefroid as being complementatry to the techniques described by Godefroid in [12], including persistent sets and sleep sets.

Abdulla et al. build upon this work in [1], supplementing the original dynamic partial order reduction algorithm with a new class of sets known as *source sets*, which effectively replace the persistent sets and provide a provably minimal set of representative non-equivalent traces within the program. Additionally, the backtrack points introduced by Flanagan and Godefroid for the dynamic partial order reduction algorithm in [11] are replaced by a *wake-up tree*, which is introduced to ensure that traces produced as part of a backtrack triggered exploration are not redundant (meaning that they will be blocked by the sleep set). This improved dynamic partial order reduction algorithm is provided with a correctness and optimality proof, making it the first partial order reduction algorithm to provide an optimal reduction on the set of representative traces within a concurrent program.

The implementation of the algorithm described by Abdulla et al. is limited in scope to specific types of bugs in Erlang programs, and so lacks the generality of our implementation and also is unable to compete in SV-COMP (which is limited to C programs). It is also designed with targeted test generation in mind and must be adapted to allow it to be used for the full exploration of all representative traces within a program containing multi-way branches within each thread.

Aside from the improvements to the reduction provided by dynamic partial order reduction over static partial order reduction, this class of algorithm can be more neatly adapted for trace abstraction refinement as it removes the need for a large pre-computation

(and thus for a translation between the trace automaton produced by partial order reduction and the language of trace abstractions used by the trace abstraction refinement loop) and may allow it to be easily composed as part of the trace exploration phase of the abstraction refinement process.

2.5 Verification of Concurrent Programs

Implementations of partial order reduction techniques for state-space reduction in model checking based verification tools were initially in tools based on simulated execution of programs [13, 17]. However, abstraction based techniques as have become more popular in recent years. However developments in abstraction based model checking techniques has given rise to a number of attempts to combine partial order reduction with these techniques, including in [8], in which Cassez and Ziegler provide a method for combining partial order reduction and trace abstraction refinement and which forms the basis for our approach.

In [8], Cassez and Ziegler provide a theoretical basis for the extension of trace abstraction refinement to concurrent programs (described and proven with two threads, but general for n threads), essentially by representing each individual thread of the program with its own program automaton. The authors then go on to provide an explanation of partial order reduction in the context of trace abstraction refinement and develop an algorithm for trace abstraction refinement of concurrent programs with partial order reduction. This involves performing the usual trace abstraction refinement loop on a language produced from the reduction of the program threads.

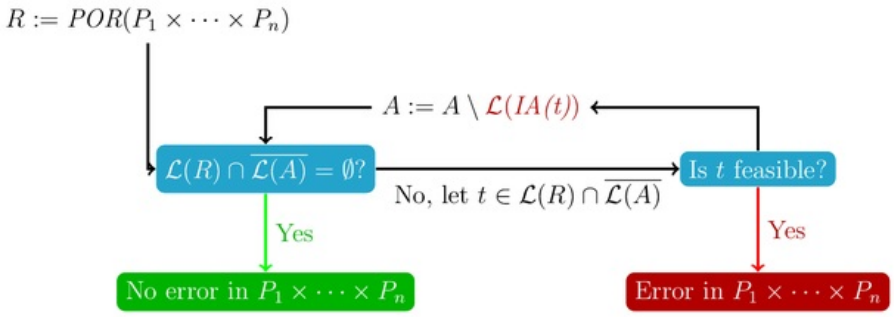


Figure 2.2: Concurrent Trace Abstraction Refinement Loop Adapted from [8]

Fig. 2.2 shows the adapted trace abstraction loop as it is adapted by Cassez and Ziegler, with the new input to the refinement process being a reduced automaton, R, for the product of the input threads, as generated by partial order reduction. At each iteration of the refinment loop, the regular language generated by R is refined by generating a second automaton, A, which captures the class of infeasible traces which are accepted by the interpolant automata produced at each iteration of the loop. A correct program is

then one for which the intersection of accepted traces in R and $\overline{A}$, the complement of the refinement automaton, is empty, meaning that there are no error traces remaining in the reduced automaton which are not accepted by the combined automaton A, accepting infeasible traces in the input program.

Cassez and Ziegler also provide a simple technique for adding the ability to check global reachability by introducing a third monitor thread which has a single operation to an error state which represents a data race. This produces a transformation from global reachability to local reachability by adding a thread for which the program's global variables can be regarded as local. This approach is one which provides the ability of the verifier to detect some forms of bugs in multi-threaded programs (most notably, data races) without the need for annotations but, as described by Ziegler in [25], also may require the user to disable this functionality for certain variables which are intended to be accessed by multiple threads at once (for example a variable which contains a lock or mutex).

The implementation provided alongside the techniques described in [8] does not allow for the verification of arbitrary C programs, as it relies on a custom C-like language which provides the necessary concurrency primitives. The tool described in [8] also only provides an implementation of the static partial order reduction algorithm described by Godefroid in [12], and so does not apply a provably optimal reduction to the concurrent programs it is seeking to verify. As part of the work conducted in this project, an implementation of this technique that accepts arbitrary C programs is be provided, as well as potentially an implementation of Abdulla et al.'s dynamic partial order reduction algorithm.

An example of how this approach can be applied to a simple concurrent program build in the Conc-Apron language is given in [25]. Our own example, applying the results of Cassez and Ziegler to verify C programs, is described in Chapter 3 of this thesis.

Other approaches to concurrent program verification with partial order reduction include the one described in [23] by Wachter et al. The technique described in [23] is based on predicate abstraction refinement, rather than trace abstraction which has a number of potential drawbacks as described in Section 2.3. There are also a number of implementation challenges in implementing partial order reduction in concert with predicate abstraction, which mean that this approach lacks the potential for extension with other forms of state space reduction.

In [1] Abdulla et al. apply their results in dynamic partial order reduction to provide a tool for the verification of concurrent Erlang programs. This is the only tool that we are aware of that has implemented the optimal version of this algorithm as part of a verification tool, but differs in that it uses partial order reduction in order to generate minimal interleavings of a dynamic execution of the program for directed testing rather than for verification as our own application.

2.6 Satisfiability Modulo Theories and Verification

The process of a converting sequence of statements in an imperative program to be converted into a satisfiability problem is at the root of many verification techniques [3, 4, 14] as a means for determining the feasibility of a path of execution through a program. Satisfiability modulo theories (SMT) have become the most common means of formulating program traces as a satisfiability problem, due to the manifold advantages that SMT solvers have at their disposal versus traditional SAT solvers. While the decidability of SMT is still in NP, ongoing work on improving SMT solvers has made verification techniques which rely on checking satisfiability (sat) or unsatisfiability (unsat) on large SMT terms the dominant approach in software verification. This development is supported a yearly competition similar to SV-COMP (the appropriately named SMT-COMP[3]) in which SMT solver implementations are benchmarked against one another.

Trace abstraction refinement relies particularly on SMT solvers to provide not only a sat decision but also to construct interpolants for the terms that capture the effect of each step in a trace. This is in order to allow the construction of interpolant automata, which enable the refinement of multiple equivalent traces in a single iteration of the analysis. As part of the process of converting a trace in a program into an equivalent SMT term, we consider the effect of each statement on the state of the program and the statements in the trace to static single assignment (SSA) form so that each time a variable is assigned a new name is created for it and that name is used until the variable's value changes.

Fig. 2.3 shows an example of conversion from a trace in a simple program, highlighted in green, to the SSA form of the same program, with the index of `k`, given by the number after the `@` symbol, updating after the initial store and the increment. Fig. 2.3c shows the complete SMT logical representative for this trace, with the effect of each statement in the sample program converted to an equivalent SMT statement which captures its effect. Each assignment becomes an equality assertion, with the conditional statements `k > 0` becoming a conditional assertion. With this representation we are able to determine the feasibility of the candidate error trace by checking the satisifiability of our representation using an SMT solver such as Z3[4].

[3] http://smtcomp.sourceforge.net/2016/
[4] https://github.com/Z3Prover/z3

```
1  int i = 1;
2  k = i;
3  if (k > 0)
4    i--;
5  else
6    i++;
7  if (i > 0)
8    __VERIFIER_error();
```

(a) Sample C Program For SMT Term Conversion

```
1  int i@1 = 1;
2  k@1 = i@1;
3  if (k@1 > 0)
4    i@2 = i@1 - 1;
5  else
6    i@2 = i@1 + 1;
7  if (i@2 > 0)
8    __VERIFIER_error();
```

(b) Sample C Program Converted To SSA Form

$$
\begin{aligned}
&(i@1 = 1) \wedge \\
&(k@1 = i@1) \wedge \\
&(k@1 > 0) \wedge \\
&(i@2 = i@1 + 1) \wedge \\
&(i@2 > 0)
\end{aligned}
$$

(c) SMT Term For Feasibility of Trace in Sample C Program

Figure 2.3: Program Trace To SMT Term Transformation

Chapter 3

From C Programs to Formal Automata Based Abstraction

The sequence of operations applied to the input program in order to produce our final representation as a deterministic synchronised thread-product automaton is shown in Fig. 3.1, beginning with the process of applying the Clang C compiler[1] and opt optimisation tool[2] to produce an LLVM intermediate representation[3] program. The program is then parsed and an abstract syntax tree (AST) representation of it is produced by the ScalaLLVM library before it is transformed and our formal automaton abstraction is constructed within Skink.

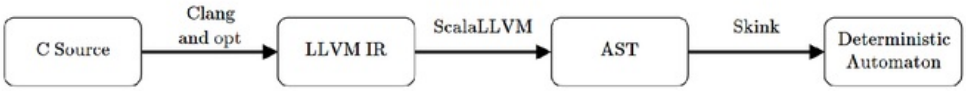


Figure 3.1: C Source to Formal Automaton Representation

In this chapter, we extend the typical examples of trace abstraction refinement and partial order reduction as given in [8, 25] to capture some of the advantages of our work. We also extend the typically more abstract examples with some concrete details – in keeping with our intention to provide a practicable implementation of this approach. We begin with an example program, given in Fig. 3.2. This is a program which provides a simple but non-trivial verification task, as the lack of synchronisation means that there are a large number of possible interleavings of all operations within the program, many of which are non-equivalent.

[1] http://clang.llvm.org/
[2] http://llvm.org/docs/CommandGuide/opt.html
[3] http://llvm.org/docs/LangRef.html

```
1  extern void __VERIFIER_error() __attribute__ ((__noreturn__));
2
3  #include <pthread.h>
4
5  int i=1, j=1;
6
7  void* f1(void* arg)
8  {
9    j += i;
10
11   pthread_exit(NULL);
12 }
13
14 void* f2(void* arg)
15 {
16   i += j;
17
18   pthread_exit(NULL);
19 }
20
21 int main(int argc, char **argv)
22 {
23   pthread_t t1, t2;
24
25   pthread_create(&t1, NULL, f1, NULL);
26   pthread_create(&t2, NULL, f2, NULL);
27
28   if (i + j != 5)
29   {
30     ERROR: __VERIFIER_error();
31   }
32
33   return 0;
34 }
```

Figure 3.2: Example Concurrent C Program Using POSIX Threads

In this program, two threads are spawned using the POSIX `pthread_create` library call, subsequent to which the threads are able to begin execution at any time. It is worth noting that due to the simplicity of this program and the lack of synchronisation, it is possible for `main` to return (and hence for the program to terminate) before either thread has executed. This potential path does not represent an accepted trace, as it does not involve reaching the error state which has been annotated in this program. Each thread accesses two global variables, `i` and `j` and adds one to the other, progressing from the first and second Fibonacci numbers (1 and 1) and computing the third and fourth into one of `i` and `j` depending on the order of execution, with the final assertion in main checking that their sum is the fifth fibonacci number. It is worth noting that when considering permutations of these two statements it appears that all paths of execution in this program will produce the same result, however this is not the case, as we will explore in the Section 3.1.

```
1   %3 = load i32, i32* @i, align 4, !dbg !26
2   %4 = load i32, i32* @j, align 4, !dbg !27
3   %5 = add nsw i32 %4, %3, !dbg !27
4   store i32 %5, i32* @j, align 4, !dbg !27
5   call void @pthread_exit(i8* null) #5, !dbg !28
```

Figure 3.3: Body Of Function `f1` in LLVM IR

3.1 Source Program Representation

In our approach, the representation of the input program on which we compute is provided by Clang, a C frontend for the LLVM Compiler, which produces LLVM intermediate representation (IR) code representing the program. This step in the process takes care of a number of pre-processing steps which are necessary in the current version of Skink (for example attempting to inline of all function calls which might have been included in the program) but also provides a number of advantages during the verification process.

The first advantage of dealing with the program in this form is that it allows us to make safer assumptions about the atomicity of each operation within our concurrent program, allowing correct reasoning about the variety of possible interleavings of operations belonging to the program's constituent threads. In LLVM IR each access to a piece of global state within the program is discrete, meaning we are able to consider all the possible interleavings of these accesses within non-atomic statements allowing the discovery of data races. An example of how this may take effect can be seen in Fig. 3.3, which shows the LLVM IR code for the body of the function `f1`. These lines of IR code are equivalent to the statement `j += i`, but the number of steps involved are demonstrative of the possibility for interleaving to affect the result of these statements' execution.

Consider the case where lines 1-3 of Fig. 3.3 are executed, before the body of the second thread (contained in function `f2`) begins and is executed in full. In this case, the values of global variable `i` will be modified but the local copy of it, used for computing the sum `j += i` is unchanged, leading to confusing behaviour. In the case of our example, this trace will lead to the error location and is an example of a data race which cannot be captured solely by considering interleavings of the source language statements (ie. by permuting the ordering of `i += j` and `j += i` between the two threads).

Another advantage of this approach to representing the input program is the inclusion in the LLVM IR grammar of the `Local` and `Global` non-terminals which are required when declaring a name in an LLVM IR program. This can be seen in Fig. 3.3, with global variables `i` and `j` prefixed with an `@` character, while local variables (or in this case registers) are prefixed with `%`. The ability to detect the use of global variables syntactically is important for the verification process, both in terms of how the program is represented for generating traces and also later for the detection of dependent operations when attempting to reduce the state-space of the program.

Also useful in the construction of a representation of the input program is the presence of the `Block` non-terminal in the LLVM IR grammar, which is a means for grouping a series of instructions within the program which contain no conditional control flow. This

```
1   int i = 0;
2   int j = 1;
3   int k = 0;
4   while (k < 5)
5   {
6     i += j;
7     j += i;
8     k++;
9   }
10  if (j != 34)
11  {
12    ERROR: __VERIFIER_error(); // SV-COMP error location annotation
13  }
```

Figure 3.4: Annotated C Program With Structure Appropriate For Block Trace Description

means that instead of viewing the program as a sequence of instructions or even (as is often done when describing traces from higher level source languages) as a sequence of statements, we can regard our program as a sequence of blocks which are linked by conditional or unconditional branches. While this view of a program is not one that is particularly difficult to generate from any input language, its explicit presence in LLVM IR means that it can be taken advantage of without introducing any additional complexity in the translation between our input program's code and our internal representation.

Viewing our input program as a sequence of blocks first of all allows for a trace in the program to be described more succinctly than is usually possible examples of trace abstraction refinement. Take, for example, the program in Fig. 3.4, written in the style of an SV-COMP benchmark for consistency. Considered at statement level, an example error trace through this program is given by

`i := 0, j := 0, k := 0, i += j, j += i, k++, ERROR.`

This is the shortest possible trace which this program's CFG can generate (although it is obviously infeasible), and each time we wish to describe a different trace through this program we add three new statements for each iteration through the loop. In our representation, this program would comprise of three blocks, B_1 from lines 1-3, B_2 for the conditional branch on line 4, B_3 for lines 6-8 (the loop body), B_4 for the conditional branch on line 10 and E for line 12, the error location. An equivalent trace to the one previously described can then be given by

$$B_1,\ B_2,\ B_3,\ B_4,\ E.$$

This representation scales especially well in comparison to statement level representations as the size of the program (and subsequently the traces) or complexity of the program grows. The simplicity lent by this representaton extends not just to how we describe traces but also the number of necessary states in the automaton which describes the behaviour of the program, and therefore in the computation time required to perform the various operations on the automata which form part of the trace abstraction refinement

process. It also means that although we have very good resolution in terms of the actual execution of the program (as we may consider interleavings of individual instructions), our internal representation of the program can still be quite terse even in comparison to those which only consider the program at the statement level.

3.2 Source Program Transformation

While the block structure used by the LLVM IR grammar is sufficient to allow exploration of all possible paths of execution through a single-threaded program, it does not allow sufficient resolution when considering how individual instructions from multiple threads may be interleaved when a concurrent program is running. An example of a case where this becomes an issue is shown in Fig. 3.3, the LLVM IR code for line 12 of Fig. 3.2. In this block of IR code there are three accesses to the global state of the program, in the form of two loads (from the two global variables `i` and `j`) and one store (to global variable `i`). The fact that all of these instructions occur inside the body of the loop with no branches between them means that within AST of our source program these nine lines of assembly (along with all of the LLVM debug information which has been elided) exist inside the same block.

This means that, due to the use of blocks as the primitive element in our internal representation of the program, it is not possible to insert any instructions from other threads between the loads and store instructions within this block, leaving us unable to explore some potentially error creating interleavings within our example program. In order to allow us to explore these interleavings without the need to consider the entire program at the instruction level and hence massively increase the size of our CFG, we transform the program and generate new blocks surrounding every instruction which may form part of a data race.

For the same reason, it is also necessary to apply the same process to allow interleavings to occur between thread primitives such as creation, exit and synchronisation primitives including wait, join, lock and unlock.

The necessary transformation is achieved, fairly simply, by processing each block in each function of a given concurrent program and, wherever we find an instruction which is dependent on some global state, inserting an unconditional branch immediately following that instruction to a new block containing the next group of instructions. When an access to some global state has been performed as the result of taking a branch from such a block it is then possible for the program automaton to transition to a block belonging to another thread where a potentially dependent operation is executed. This allows the instruction level interleaving of these accesses without the need for every instruction in the program to be represented within our automaton.

An example of a transformed version of the original LLVM IR source code from Fig. 3.3 is given in Fig. 3.5. In the transformed IR, the initial loads from `i` and `j` are separated into their own blocks, with the store into `j` grouped with the add instruction which does not depend on any global state, but only local registers. The grouping of the global

```
1      %3 = load i32, i32* @i, align 4, !dbg !26
2      br label %__threading.1.nolabel
3
4    __threading.1.nolabel:
5      %4 = load i32, i32* @j, align 4, !dbg !27
6      br label %__threading.0.nolabel
7
8    __threading.0.nolabel:
9      %5 = add nsw i32 %4, %3, !dbg !27
10     store i32 %5, i32* @j, align 4, !dbg !27
11     br label %__threading.nolabel
12
13   __threading.nolabel:
14     call void @pthread_exit(i8* null) #5 , !dbg !28
```

Figure 3.5: Transformed Body of Function `f1`

access instructions with local effect instructions is arbitrary, and we could equivalently have included the add instruction from lines 9 with the load instruction on line 5.

The use of static single assignment (SSA) form for all registers in LLVM IR, combined with the syntatic reflection of variable scoping (by way of the `Global` and `Local` non-terminals) provides us not only with the ability to reason very easily about which instructions (or subsequent to this transformation, blocks) within the input program are dependent. It also provides a substantial and completely free of charge initial reduction on the interleavings that must be explored. As we begin with the assumption that any sequence of instructions which are not separated by branches (a block) can be considered as one element in a trace of the input program's execution, we immediately discount all interleavings of instructions which are contained in separate blocks. As blocks can sometimes contain instructions which when interleaved do change the result of the program's execution, we are required to transform the input program in order to preserve the correctness of our assumption, but are still left a substantial reduction.

An example of how this reduction applies can be seen when considering the transformed function body in Fig. 3.5. We know that the thread function `f2` may run concurrently with `f1` and also that the `load` instruction performed by `f2` on `i` is dependent on whether that `load` occurs before or after the `store` to that variable which occurs in `f1`. However, we also know (in this case by inspection but in a normal case from analysing the syntax of the IR) that line 9 is not dependent on any other instruction which might be executed in the program as it refers only to registers which are always local. As such we can discount any interleaving between this load and all of the instructions on those lines, and instead consider only the two interleavings where that `load` precedes or succeeds the `store` to `i`.

As we observed before in the case of the ability of blocks to reduce the size of our internal representation of the program, this reduction is not a particularly difficult one to spot or to compute. With other approaches where the internal representation of the tool produces traces in terms of individual instructions or statements, this reduction would not occur until after the program automaton or thread automata have been constructed. This would be the responsibility of a dedicated reduction algorithm (the role filled by

partial order reduction in our approach). In contrast, our approach is able to achieve this result without any extra computational burden, as a by-product of how we use the LLVM IR representation of our input program.

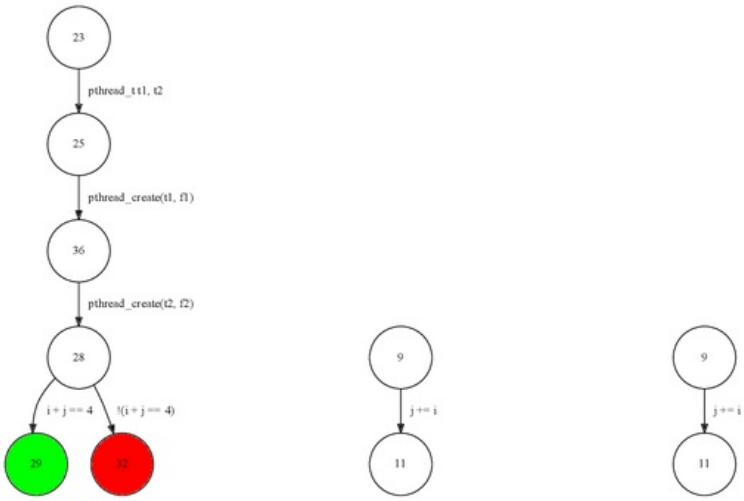
3.3 Automaton Construction

The statement level control flow graph (CFG) for each of the three threads in the example program from Fig. 3.2 are given in Fig. 3.6, with the error location (shown in Fig. 3.2 as the `ERROR` label) highlighted in red, and the error-free terminating location (the return statement in `main`) highlighted in green. Each location in the CFG is annotated with the line number of the code corresponding to that point in the execution, with some simplification done to avoid overcomplicating the diagram. In order to consider the execution of the example program, we must consider concurrent paths through each of these CFGs.with the synchronisation requirement that `f1` and `f2` may not begin execution until their corresponding `pthread_create` calls have been made. The thread CFGs in Fig. 3.6 can also, with some small modifications, be regarded as thread automata, as is more common in the literature [25].

For the sake of readability, the CFGs have been presented with each transition representing a single statement in the program, rather than branches between blocks, as it is in our approach. In this section, we consider the construction of an automaton which allows the exploration of interleavings of the transtions within these CFGs from a statement level. However, our approach is general for application to blocks and, as described in in the previous section, we can consider the blocks in our representation of the program to be equivalent to the statements that exist between control flow within the CFGs in Fig 3.6.

With the individual CFGs for each thread available by traversing the AST of the translated LLVM IR input program, we seek to provide an automaton which can be used to construct a language accepting traces that reach an error location. In the most general case, a program consisting of two threads running concurrently can be represented by the product of the two automata representing the threads. This representation however lacks two features. First, the ability to manage synchronised transitions on each of the threads (for example one thread waiting to join another or one thread waiting on a lock held by another), which is fundamental to verification of many of even the simplest benchmark programs used in SV-COMP. Secondary to this is the fact that a product of thread automata is an over-approximation for the true behaviour of concurrent threads, and still requires some special cases to manage when each thread in the program starts. This is often represented by the state for each child thread starting in a special inactive state and then beginning to run when a particular transition is reached in the parent thread's automaton.

The state for our automaton consists of two components; the first is a set of thread locations with each thread identified by a unique integer and its location by the entry label of the block within that thread's CFG. The second component of the state is a set



(a) CFG For `main` thread **(b)** CFG for `t1` thread **(c)** CFG for `t2` thread

Figure 3.6: Control Flow Graphs for each thread in Fig 3.2

of synchronisation tokens which are tracked during construction of the automaton. How these tokens are discoverd and applied is described in Section 3.4. A transition in our automaton corresponds to a block in one of the program's threads having been scheduled in a possible execution, with the label consisting of the unique identifier of the thread being scheduled and the exit branch which was taken from the block that being exited.

Our approach reduces most of the redundancy in the more general product approach, constructing the automaton by walking the AST of the source program and applying a small sub-set of the semantics of a given block to the structure of our automata. By constructing a single unified automaton from scratch, rather than building individual thread automata and then taking a product, we are also able to build our automaton "on the fly", which provides a number of benefits.

The primary advantage of this dynamic approach for construction of the automaton is that for large programs we avoid the potentially quite expensive process of exploring the entire CFG of the program in order to find all the necessary states and labels for a fully specified automata. Instead, our automata is effectively stateless and refers back to the source program's AST to apply transitions. As we explore the accepted traces in our automata, we will begin to explore the CFG and memoise the transitions which have been previously computed, but are able to avoid exploring the whole CFG in the case where a feasible error trace is found during the trace abstraction refinement process.

As the structure of the CFG is discovered dynamically we have no foreknowledge of how many threads exist in a program or where they begin, instead we begin with an initial

assumption that our program consists of only a single thread with an initial location at the start of the program's main function. Each time we apply a label in our automata, we inspect the contents of the block that has been scheduled to discover if a thread has been created at this point (by looking for calls to the POSIX `pthread_create` function). If we find a thread creation we add a new entry into the successor state to signify that a new thread has been spawned, with its location at the start of the function which it will run.

As a consequence of the lazy construction of our automaton, if a particular trace does not encounter creation of a thread (for example because an error location is reached before the thread is created) then all the states of our automaton for that particular trace will not contain any reference to the unstarted thread. This property means that while we must carry a small amount of additional information in our state to account for the possibility of new threads being created, the additional overhead of our approach when applied on single-threaded programs is minimal and does not require any pre-computation to discover that a program is single or multi-threaded – we simply traverse the AST and expand the number of locations that are recorded as necessary.

Computing the enabled labels and successor for a given state in our machine also involves a simple inspection of the program source at each of the locations recorded in the state to discover which branches are available and which new block they will lead to, which allows us to select the appropriate branch and update our state in order to apply the selected label.

An example of part of an automaton constructed using our approach is given in Fig. 3.7, based on the three CFGs shown in Fig 3.6. In this example, we begin from the same starting point as the `main` CFG in Fig 3.6 and complete the first transition as normal. After passing the first `pthread_create` transition, we reach a state with two locations in the program, line 26 in the middle of `main` and line 9 at the start of `f1`. From here, we apply the labels of `main` in order to follow the shortest path to an accepting state or a sink state. Following the second `pthread_create` call, a third element is introduced into the automaton state, accounting for the position of execution in `f2`.

The complete transformed LLVM IR program source and the full concurrent automaton for the example program in Fig. 3.2 is given in Appendix B.

3.4 Synchronisation

Having applied the semantics of thread creation to the structure of our automaton we find that we can generalise this idea of applying the semantics of thread synchronisation operations to our automaton. This allows us to avoid the need to explore and refine traces which violate the semantics of synchronisation operations or implementing special synchronising transitions in our automaton. This is achieved by extending the process used for detecting thread creation to consider not just the possibility of thread creation but of all the possible effects of this block on the structure of our automaton.

When applying a label in our automaton, we can compute the effects of the block which

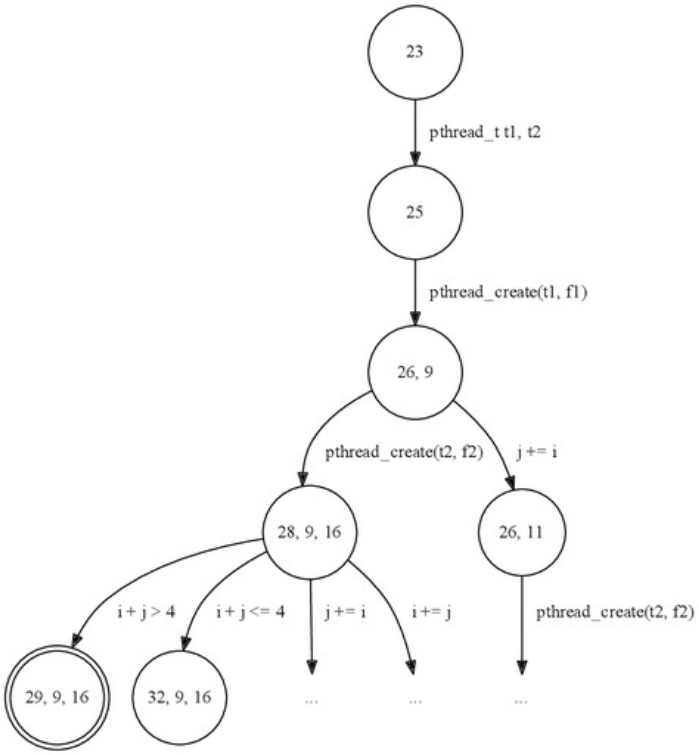


Figure 3.7: Concurrent Automaton For Example Program

has been scheduled (which we regard as having "run" in our trace), which produces the **out effects** for this transition. Similarly, the new block whose location has been added to the successor state gives us the **in effects**. The effects of the transition are recorded in the state by storing a set of synchronisation tokens, corresponding to the value of a token within our program (a mutex or a condition). For the semantics of the POSIX thread library, most synchronisation effects occur as out effects, for example if a block contains a call to `pthread_mutex_lock` then the out effect of that block is that the mutex upon which that function was called is recorded as locked within the synchronisation token set of the successor state.

In our current model of POSIX threads, the only time in effects are recorded is for calls to `pthread_cond_wait`. This is due to the fact that this function has two effects on the synchronisation tokens that exist in the program, on arrival it will release the mutex

being held while it waits for its condition to become true. This effect is not one which is applied when the function returns, but rather when the function is initially called. When the condition which is being waited on becomes true, the function returns and the mutex which was released is re-taken, constituting the out effect of the function call and the block in which it resides.

Once we have recorded the values of the synchronisation tokens which have been encountered at a given state, we can use them to determine which transitions should be enabled within our automaton from a particular state. For example, if a block contains a call to `pthread_mutex_lock` on a mutex which we have recorded as having been locked by another thread previously in the trace, then any transition out of that block should not be enabled as we know that it will violate the semantics of the POSIX threading library and so will be infeasible.

By enforcing the semantics of POSIX thread synchronisation operations on the structure of our automaton as it is explored, we avoid having to explore traces within the program which will be obviously infeasible, reducing the size of our automaton and thus the number of solver calls necessary. This is mostly possible due to the simple and well defined behaviour of the tokens involved in thread synchronisation but can possibly be extended to other easily recognisable constructs within the program.

Fig. 3.8 shows an adapted version of our original example from Fig. 3.2 with some synchronisation added in order to make it a little less buggy. Each thread function has been modified so that the thread first must lock the mutex `m` before it is allowed to load, add and store back into one of `i` and `j`, effectively converting making the statements `i += j` and `j += i` perfectly atomic, something which we have previously observed was not the case with our original example. In `main` two calls to `pthread_join` have been added to ensure that both threads must complete their work before we check whether or not the Fibonacci sequence has been followed correctly. If we were to attempt to encode the behaviour of these synchronisation operations as terms and construct refinements for traces which violate the semantics of a mutex or a join, the structure of the automaton for this program would be effectively identical to the one given in Appendix B for our original example. In contrast, applying our approach of enforcing the semantics of all Pthread synchronisation operations, we construct the substantially reduced automaton shown in Fig. 3.9. The most obvious feature which demonstrates the effect of synchronisation is the existence of only one accepting state in the automaton, as we require that both of the child threads have terminated before we are able to reach the error checking condition on line 41.

3.5 Verification

In this section we describe some areas of interest in our adaptation of the single-threaded verification process, for a more complete description of trace abstraction refinement see Section 2.3 and the publications referenced within.

The primary challenges in adapting the existing trace abstraction refinement imple-

```
1  extern void __VERIFIER_error() __attribute__ ((__noreturn__));
2
3  #include <pthread.h>
4
5  int i = 1, j = 1;
6  pthread_mutex_t  m;
7
8  void *
9  f1(void* arg)
10 {
11     pthread_mutex_lock(&m);
12
13     j += i;
14
15     pthread_mutex_unlock(&m);
16     pthread_exit(NULL);
17 }
18
19 void *
20 f2(void* arg)
21 {
22     pthread_mutex_lock(&m);
23
24     i += j;
25
26     pthread_mutex_unlock(&m);
27     pthread_exit(NULL);
28 }
29
30 int
31 main(int argc, char **argv)
32 {
33   pthread_t t1, t2;
34
35   pthread_create(&t1, NULL, f1, NULL);
36   pthread_create(&t2, NULL, f2, NULL);
37
38   pthread_join(t1, NULL);
39   pthread_join(t2, NULL);
40
41   if (i + j != 5) {
42     ERROR: __VERIFIER_error();
43   }
44
45   return 0;
46 }
```

Figure 3.8: Example From Fig. 3.2 Adapted With Synchronisation

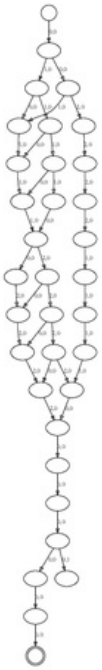


Figure 3.9: Synchronised Concurrent Automaton For Program in Fig. 3.8

mentation in Skink to accept our concurrent automaton were connected to the construction of SMT terms for a given trace, particularly in the conversion of the names in the trace into SSA form, and in generating appropriate interpolants for traces which are being refined.

As the existence of multiple threads within a program introduces the potential for multiple scopes within a trace, something which is not the case for a program which contains a single thread running a single function. In order to construct an SSA sequence of terms to be used to check feasbility of a trace it is necessary to associate a local name in each term not just with where it occurs in the sequence of stores but also with which thread it was executed on. In our approach, this is achieved by associating each thread in the source program with a namer which is responsible for keeping trace of the index of a variable's usage within a trace and also with annotating the name of that variable with the thread to which it belongs.

For the case of global names, our SSA form should be consistent across all threads to reflect the fact that modifying the value of a global variable on one thread will affect the result of that variable in other threads. This is achieved by having all threads share a single global namer, which is used any time a global name is encountered. From here, we

apply the standard scheme for computing the SSA form of a sequence of statements. We maintain a map of current indices for each variable and then increment the index when an operation which will modify the value of that variable is encountered.

When applying a refinement to the automaton a concurrent program, the process of building an interpolant for the rejected trace is complicated by the potential for loops within a thread to be interleaved with blocks from other threads. This complicates the detection of loops within a given trace and hence their removal in a refinement as the blocks that form the iterations of a loop will not necessarily be directly adjacent in a trace. For a single-threaded program, if we discover a trace of the form

$$\mathtt{B}_1,\ \mathtt{B}_2,\ \mathtt{B}_1,\ \mathtt{B}_2,\ \mathtt{B}_1,\ \mathtt{B}_2,\ \mathtt{E}$$

then we observe that $\mathtt{B}_1$ and $\mathtt{B}_2$ must form the body of a loop.

As we know that in a single threaded program that the blocks that form a repeating pattern in our trace must all be part of a loop, we can capture an arbitrary number of iterations through any loop by adding a backedge from the end of the first loop iteration to the beginning. In the case of our example this would constitute an edge from the first instance of $\mathtt{B}_2$ to the first instance of $\mathtt{B}_1$, which could capture any number of iterations through this loop. If adding this edge is successful, ie. if it is actually an interpolant for our original trace, then we do not need to consider adding any other edges in our interpolant, as we have already constructed an automaton which can produce this trace and all other traces which involve iterations through the loop consisting of $\mathtt{B}_1$ and $\mathtt{B}_2$.

In contrast, in the concurrenct case, a we may find a trace of the form

$$\mathtt{B}_{(0,\ 1)},\ \mathtt{B}_{(0,\ 2)},\ \mathtt{B}_{(0,\ 1)},\ \mathtt{B}_{(0,\ 2)},\ \mathtt{B}_{(1,\ 1)},\ \mathtt{B}_{(0,\ 1)},\ \mathtt{B}_{(0,\ 2)},\ \mathtt{B}_{(0,\ 1)},\ \mathtt{B}_{(0,\ 2)},\ \mathtt{E},$$

with each block now annotated with the identifier of the thread to which it belongs. In this trace, we observe the same repeating pattern of $\mathtt{B}_{(0,\ 1)}$ and $\mathtt{B}_{(0,\ 2)}$, which consitute a loop on thread 0. However, due to the presence of $\mathtt{B}_{(1,\ 1)}$ between two iterations of thread 0's loop, adding a single backedge will not capture all possible cases for this trace, as it will only capture iterations of the thread 0 loop before the interleaving of $\mathtt{B}_{(1,\ 1)}$ but not the iterations which happen afterwards.

The ability of blocks from other threads to interleave with loop iterations means that selecting backedges to apply when constructing an interpolant automaton is more complex. We cannot infer any information about the how future iterations of a loop may occur as we must be able to account for the pattern that makes up a loop in our trace to be interrupted arbitrarily. This means we must attempt to add backedges for every repeated pattern of blocks within a trace, allowing us to capture equivalent traces which have arbitrary numbers of loop iterations with blocks from other threads interleaved.

3.6 Reduction

In our approach we adapt the Source-DPOR algorithm as presented by Abdulla et al. in [1]. From the empirical results demonstrated in [1] and given the additional computational expense involved in computing wake-up trees, we have chosen not to adapt the

optimal partial order reduction technique. In order to clearly specify our adaptation of Source-DPOR we re-define some of the relations required by the algorithm in terms of our own approach, however in implementation these operations will likely remain generic, as specified by the original author.

In order to take advantage of the dynamic nature of SOURCE-DPOR and its symmetry with our approach to constructing the concurrent automaton, the exploration process carried out by the algorithm shares a role within the trace abstraction refinement process with trace generation. This is achieved, quite straight-forwardly, by adding a check at the start of the exploration of each state for the accepting status of the state we are in. If we find an accepting state at any point during the reduction exploration, we will return the trace that we have explored up until that point as a candidate error trace. This combination of trace generation with the reduction algorithm means that our reduction should not be viewed as a process which is taking a large automaton and converting it into a smaller one by removing equivalent traces, but rather one which traverses a large automaton with a strategy which allows it to only visit a small subset of the states.

Algorithm 1 Trace Generating Source-DPOR [1]

```
1:  function EXPLORE(T, sleep)
2:      if ISFINAL(T.state) then return TOTRACE(T)
3:      if ∃p ∈ ENABLED(T) \sleep then
4:          backtrack(T) ← {p}
5:          while ∃t ∈ (backtrack(T) \ sleep) do
6:              for all r ∈ T s.t. race(t, r) do
7:                  T' ← prefix(T, r)
8:                  intermediates ← NOTDEP(T, r)
9:                  candidates ← NOPREDECESSOR(T, intermediates) + t
10:                 if candidates ∩ backtrack(T') = ∅ then
11:                     backtrack(T') ← backtrack(T') ∪ {(some c ∈ candidates)}
12:             sleep' ← {s ∈ sleep | INDEPENDENT(t, s)}
13:             EXPLORE(succ(T, t), sleep')
14:             sleep ← sleep ∪ {t}
```

Algorithm 1 gives an overview of the adapted algorithm for programs with non-cyclic automaton. The form of SOURCE-DPOR follows that of a depth first search, but instead of maintaining a stack of all the "to visit" states which have been encountered, we maintain only a single linear path through the automaton which is being traversed given by a sequence of pairs of states and transitions, T. The job of managing which state should be visited next is managed by the source set, *backtrack*, which contains all the states which are yet to be visited when "backtracking" to a particular state. The source set is constructed lazily, with the source set of a given state able to be updated during the exploration of another state which further down in the program automaton. This mechanism of starting the source set of any particular state with only a single transition,

and then only adding additional states when a non-equivalent trace is found, is what allows the SOURCE-DPOR to avoid the exploration of equivalent states within the automaton. When all non-equivalent paths through a particular transition have been explored, it is added to the sleep set, *sleep*, excluding it from future exploration.

When deciding to explore a particular transition t from our current state, we enumerate all past transitions which are in a race with t. We define a transition r as being in a race with t for some trace T, if r occured before t in the trace, that the blocks which would be executed by the two transitions as part of the T are dependent and t was enabled at the time that r was taken. If this relation holds for two transitions, then we need to explore the case where the t was taken instead of r. This is achieved by updating the source set of the trace from which r was applied, given by T'.

From here, we compute all of the transitions that were taken between the application of r and our current trace T that are independent with r. This is given by $\text{NOTDEP}(T, r)$ which requires that the blocks that were executed by the transitions are not dependent and that they did not occur on the same thread. These transitions, plus our candidate transition t which was in a race with r are now our candidates for being added to the source set of T'. We then compute all of the available transitions from within this candidate set by taking all the transitions which have no dependent predecessor in the set $\text{NOPREDECESSOR}(T'')$. We note that this set will always include our original candidate t but potentially may also contain some other independent transitions that occured between T' and T.

If there is any transition in our set of candidate source set transitions that are not in the source set of T', we add that transition to the set and carry on searching for races. Once all the races for t have been explored, we construct the sleep set for our successor state, given by all the transitions in the sleep set of T which are independent with t and then explore the successor of T having applied t. Once we have fully explored all of the traces which include applying t from trace T, we add t to the sleep set for T and check to see if any new transitions have been added to the source set for T.

For programs containing loops, additional complexity is introduced to the algorithm due to two issues. First, because we only maintain a record of all of the traces which form part of our current trace, it is possible to get stuck in a cycle within the program automaton. We have no way of recognising if a state has been explored previously if it is not part of our current trace. Alongside this, as the pattern of exploration of the SOURCE-DPOR traversal is not deterministic there is the possibility of exploring traces which are equivalent to a member of the class of traces which have been removed by a previous refinement. A potential remedy for this is to "exhaust" the refinement automaton by attempting to explore all non-equivalent traces within the refinement automaton. This can be achieved by preferentially exploring transitions (necessarily backedges) which have been added by refinement automata before exploring transitions belonging to the original program automaton.

Chapter 4

Implementation

As mentioned in Section 1.1, the implementation of our approach is to take place as a component of the Skink verification tool[1], a currently closed source project developed within the Macquarie University Computing department. As such, many components of the trace abstraction refinement process have been produced by other members of the Skink project and so their implementation will not be described in detail in this report.

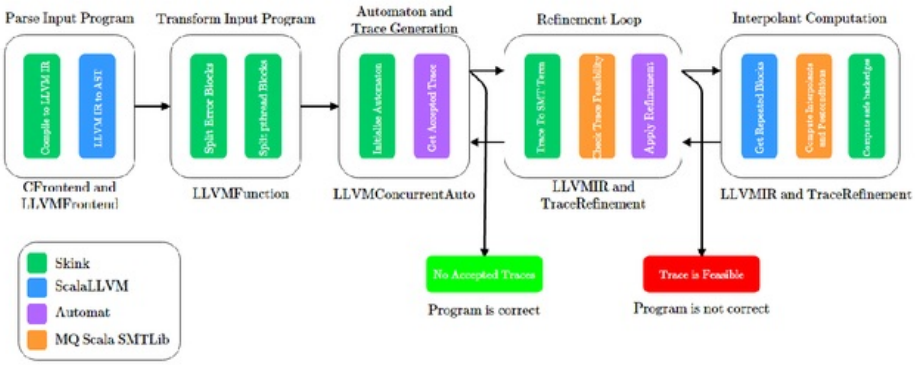


Figure 4.1: Block Diagram of Skink Multi-Threaded Verification Process

Fig. 4.1 provides a high level view of the structure of Skink with the main stages of the verification process listed above each block and the Skink classes which are responsible for each stage beneath each block. Within them, more fine grained tasks comprising each stage are noted, with each component coloured based on the library which provides the bulk of the functionality needed for that task. By inspecting the diagram we can observe the primary responsibilities of Skink, and most of the components which required implementation as part of the work described in this report. In this chapter, we describe the implementation of the program source transformation, concurrent automaton, SMT term

[1] https://bitbucket.org/inkytonik/skink

generation and adaptations made to the trace refinement and interpolant construction code.

4.1 Transformation of LLVM IR Source

Following the process introduced in Chapter 3, the first step in the handling of concurrent programs with our chosen input representation is the insertion of additional branches inside the LLVM IR `Block`s in order to allow interleavings and synchronisation between thread automata. All of the transformations on LLVM IR source code within Skink are done at a function level, with the `LLVMFunction` existing as our internal representation of an LLVM IR function within Skink. `LLVMFunction` is essentially a wrapper around an individual function's AST as provided by the ScalaLLVM library, and provides a number of methods either for preparing the function for verification and traversing the AST of that function.

`LLVMFunction` implements the `IRFunction` trait, which is used within Skink to provide a generic interface for interacting with the input program's source representation. This allows the central components of the trace abstraction refinement implementation to be de-coupled from a particular input language and also from the various tasks related to preparing the input program for verification. This design is particularly useful for our approach, as virtually all the material changes that are necessary to adapt trace abstraction refinement for concurrent programs exist outside the boundary of the algorithm itself.

The majority of the work for preparing an `LLVMFunction` for verification occurs inside `makeThreadVerifiable`, which is the function responsible for transforming the input program by inserting the new blocks that are required for our approach. The two main components of the `makeThreadVerifiable` function are given in Fig. 4.2. At the heart of this function is the nested function `splitOnPredicate` which takes a list of `MetaInstruction` instances from within a `Block` and splits them based on a predicate that is applied to each `MetaInstruction`. The predicate functions for both global memory access and Pthread API function calls are contained inside an `LLVMHelper` object and are constructed using a pattern match on the LLVM IR AST.

Once a separated list of `MetaInstruction` instances is acquired, they are re-constructed into a sequence of blocks, maintaining the same overall semantics as the original source program but with the addition of several new branches and labels to allow the necessary interleaving or synchronisation. This is achieved by iterating backwards over the blocks in the function, as we need to know the label of the next block in the function in order to create a branch to it from the previous block.

Similar work is done by the `makeErrorsVerifiable` function, which adds special labels before error locations are discovered within the program to allow easier recognition of final states during trace and SMT term generation. These two functions are composed in order to produce a new `FunctionBody` instance, which serves as the new canonical AST for the `LLVMFunction` when verification begins.

```
def splitOnPredicate(
    insns : List[MetaInstruction],
    pred : MetaInstruction => Boolean
) : List[List[MetaInstruction]] =
    insns.span(pred) match {
        case (Nil, Nil) => Nil
        case (Nil, access :: remains) => splitOnPredicate(remains, pred) match {
            case start :: end => List(access) :: start :: end
            case Nil          => List(List(access))
        }
        case (remains, Nil)                => List(remains)
        case (previous, access :: remains) => (previous :+ access) :: splitOnPredicate(
            remains, pred)
    }

def insertBranchOnGlobalAccess(block : Block) : Block = {
    val splitBlocks = splitOnPredicate(
        block.optMetaInstructions.toList,
        i => !isThreadPrimitive(i.instruction) && !isGlobalAccess(i.instruction)
    )

    if (splitBlocks.length <= 1) {
        block
    } else {
        val first = splitBlocks.head
        val rest = splitBlocks.drop(1).dropRight(1)
        val last = splitBlocks.last
        var label = makeLabelFromPrefix(block.optBlockLabel, "__threading")
        insertedBlocks += Block(BlockLabel(label), Vector(), None, last.toVector,
            block.metaTerminatorInstruction)
        var blockCount = 0
        for (b <- rest.reverse) {
            val newLabel = makeLabelFromPrefix(block.optBlockLabel, s"__threading.
                blockCount")
            insertedBlocks += Block(BlockLabel(newLabel), Vector(), None, b.toVector,
                MetaTerminatorInstruction(
                    Branch(Label(Local(label))),
                    Metadata(Vector())
                ))
            label = newLabel
            blockCount += 1
        }
        val startBlock = Block(block.optBlockLabel, Vector(), None, first.toVector,
            MetaTerminatorInstruction(
                Branch(Label(Local(label))),
                Metadata(Vector())
            ))
        startBlock
    }
}
```

Figure 4.2: `makeThreadVerifiable` Core Functions

4.2 Construction of Program Automaton

The concurrent program automaton is the most substantial addition to Skink that was necessary for the support of multi-threaded programs and was implemented in the class `LLVMConcurrentAuto`. In order to take best advantage of the work already done implementing automata and regular language operations, as well as trace generation, the implementation of our approach for program automaton construction follows the interface for a deterministic and complete automaton (DCA) as defined by Franck Cassez's automat library[2]. Adhering to an interface defined within automat also allows our automaton to plug into the existing trace refinement implementation with little modification required.

The DCA interface is relatively lightweight, with only four of the methods, `enabledIn`, `succ`, `isFinal` and `getInit` actually necessary to implement in order to produce a regular language of accepted traces. The state of our automaton is represented by a case class, `LLVMState`, which has two components, `threadLocs`, a `Map[Int -> String]` which records the name of the current block in each thread, and `syncTokens`, a `Map[String -> Bool]` used to record the state of the mutexes and conditions that have been encountered within the program. The labels of our automaton are another case class, `Choice`, with two fields, `threadId` an integer which identifies the thread that a label is being applied on and `branchId` for the index of the branch which is being taken. `Choice` is part of the generic interface shared by all source programs and is also used to construct the `Trace` objects which the trace abstraction refinement loop computes on. The implementation of traversal methods `succ` and `enabledIn` is shown in Fig. 4.3.

As described in Section 3.3, the fundamental process behind the construction of the program automaton is a traversal of the AST of each function which is active in the program at any given time, with an initial starting location at the first block in `main`. The primary method used for detecting enabled transitions and computing the successor for a state and transition pair is `nextBlocks`, which computes a set of `(Int, String)` tuples corresponding to the thread identifier and block name of all the blocks which can be reached from the current state. `nextBlocks` iterates over each location in the state's `threadLocs` map and performs a pattern match on the terminator instruction of the block corresponding with the state's locations. It is also responsible for applying the synchronisation semantics based on the synchronisation tokens in the current state, which is achieved by filtering the initial set of locations for blocks which are blocked using the method `isBlocked`. As nextBlocks is often re-computed for the same state during a traversal, it has been implemented as a Kiama[3] attribute to take advantage of the caching functionality provided by the `Attribution` class.

`isBlocked` is implemented by filtering the block for instructions which involve calls to Pthread API functions, before applying an extractor which takes the useful information from calls which may block the thread which that block occupies. The most complicated case for `isBlocked` comes in the form of `pthread_join`, due to the first argument of the function being a value rather than a pointer as is the case for most other Pthread

[2] https://bitbucket.org/franck44/automat/
[3] https://bitbucket.org/inkytonik/kiama

```
1  def succ(state : LLVMState, label : Choice) : LLVMState = {
2      val threadId = label.threadId
3      val branchId = label.branchId
4
5      // Find the next block for the selected branch
6      val (_, newBlock) = nextBlocks(state).filter(_._1 == threadId)(branchId)
7      var newThreadLocs = state.threadLocs + (threadId -> newBlock)
8
9      // Get thread creation info for this block
10     val block = ir.getBlockByName(threadId, state.threadLocs.get(threadId).get)
11     threadCreationEffects(block) match {
12         case Some((newThreadId, newBlockName)) => newThreadLocs = newThreadLocs + (
               newThreadId -> newBlockName)
13         case _                                 =>
14     }
15
16     var newSyncTokens = outSyncEffects(block, state.syncTokens)
17     newSyncTokens = inSyncEffects(ir.getBlockByName(threadId, newBlock), newSyncTokens)
18
19     LLVMState(newThreadLocs, newSyncTokens)
20 }
21
22 def enabledIn(state : LLVMState) : Set[Choice] = {
23     val buf = new ListBuffer[Choice]
24
25     for ((threadId, branches) <- nextBlocks(state).groupBy(_._1)) {
26         for (branchId <- 0 to branches.length - 1) {
27             buf += Choice(threadId, branchId)
28         }
29     }
30     buf.toSet
31 }
```

Figure 4.3: `succ` and `enabledIn` Method Implementations

functions. This means that the `pthread_t` variable which the join is targeting will not be part of the call to `pthread_join`, but rather will be loaded into a register and then the register will be passed to the join function. Currently, this is handled by assuming that the load from the `pthread_t` identifier will occur immediately before the call to `pthread_join`. This assumption has held for all of the synchronised programs that we have encountered so far, but it is not difficult to construct a counter-example for which this approach won't work. In the long term, applying a scheme similar to that which is used for indexing names within function ASTs (see Section 4.3) would allow for a more robust lookup of thread identifier names.

Once a successor block has been found by applying an enabled label, the effects of the newly selected block are applied. First, the block which is being transitioned out of, ie. the one which has been "executed" in our model, is examined for thread creation. This check is performed by the `threadCreationEffects` method, which uses pattern matching to extract the name of the `pthread_t` identifier and the function which will be executed by the thread. If a new thread is discovered, the `seenThreads` map is checked to ensure we are not creating a new threadId for a previously encountered thread (due to transitioning out of the same block from a different state). If this thread hasn't been encountered before, it is added to the `LLVMIR functionIds` map, which keeps track of

which `LLVMFunction` applies to a given thread identifier. Finally, the `threadCount` of `LLVMConcurrentAuto` is bumped and used as the identifier for the new thread.

If a new thread is discovered, it is applied to the `threadLocs` map of the updated state. Following this, the synchronisation effects of the transition are computed, beginning with `outSyncEffects` which computs the synchronisation effects of the block being transitioned out of. `outSyncEffects` relies on an extractor defined in `LLVMHelper` which is used to match extract the useful information from calls to Pthread API functions. This method handles initialisation of mutexes and conditions, the locking and unlocking of mutexes and the signalling of conditions. It is also used to handle the more complicated behaviour of `pthread_cond_wait` which has an out effect of locking the mutex that it had previously released. Following `outSyncEffects`, `inSyncEffects` applies the in effect of `pthread_cond_wait` prior to the thread being blocked, unlocking the mutex argument if the condition is currently false.

4.3 Verification Supporting Functionality

Adapting the existing implementation of trace abstraction refinement and its supporting operations was involved lifting the interface through which the program source and verification algorithm interacted from a function level to a program level. This is due to the previous sequential model used by Skink being centred around verifying an individual `IRFunction`. As a result, the `IRFunction` trait has been deleted from the `ir` package, with the only interface between the verification package and `ir` being via the trait `IR`, representing the source of a complete program. The source for the `IR` trait is given in Fig. 4.4.

The two main avenues of interaction between the source program and the trace abstraction refinement implementation are via the automaton representing the program, which is provided via the `dca` field of the `IR` trait and the `traceToTerms` method, which constructs an SMT term capturing the effects of a trace when it is projected back onto the source program. Term generation for an accepted trace begins with the construction of a `BlockTrace`, consisting of the sequence of blocks which are encountered when applying the trace to our source program. The `BlockTrace` is used to allow the effects of each statement in a trace to be converted to SSA form and then be encapsulated in an SMT term. The concurrent Skink method for constructiong a `BlockTrace` was adapted from the original synchonous version written by Tony Sloane. It iterates over each choice in the trace and computes a successor based on the `Choice` found at that location in the trace. The successor for each block given a particular choice is computed using the pre-existing `nextBlock` method of the `LLVMFunction` class. As it is possible for a trace to span over multiple functions, a map is used to store the current position of each thread after each transition as been applied. As with `nextBlocks`, `blockTrace` is implemented as an attribute to take advantage of the Kiama's caching functionality as it is a fundamental operation at multiple stages of the trace refinement process.

Having constructed a `BlockTrace`, term generation is managed by the `LLVMTermBuilder`

```
1  trait IR {
2
3      def execute() : (String, Int)
4
5      def functions : Vector[IRFunction]
6
7      def name : String
8
9      def show : String
10
11     def dca : DetAuto[_, Choice]
12
13     def traceToTerms(trace : Trace) : Seq[TypedTerm[BoolTerm, Term]]
14
15     def traceToSteps(failTrace : FailureTrace) : Seq[Step]
16
17     def traceToRepetitions(trace : Trace) : Seq[Seq[Int]]
18
19     def traceBlockEffect(trace : Trace, index : Int, choice : Int) : (TypedTerm[BoolTerm,
           Term], Map[String, Int])
20
21     def checkPost(pre : TypedTerm[BoolTerm, Term], trace : Trace, index : Int, choice :
           Choice, post : TypedTerm[BoolTerm, Term]) (implicit solver :
           ExtendedSMTLIB2Interpreter) : Try[Boolean]
22 }
```

Figure 4.4: `IR` Trait Implementation

```
1  lazy val blockTrace : Trace => BlockTrace =
2      attr {
3          case trace =>
4              import scala.collection.mutable.ListBuffer
5
6              var threadBlocks = Map[Int, Block]()
7              val blocks = new ListBuffer[Block]()
8              for (c <- trace.choices) {
9                  val threadFn = dca.getFunctionById(c.threadId).get
10                 val currBlock = threadBlocks.get(c.threadId) match {
11                     case Some(block) => block
12                     case None        => threadFn.function.functionBody.blocks(0)
13                 }
14                 threadBlocks = threadBlocks - c.threadId
15                 threadFn.nextBlock(currBlock, c.branchId) match {
16                     case Some(block) => threadBlocks = threadBlocks + (c.threadId ->
                           block)
17                     case None        => // There's no next block, panic
18                 }
19                 assert(threadBlocks.get(c.threadId).get != currBlock)
20                 blocks += currBlock
21             }
22             BlockTrace(blocks.toList, trace)
23     }
```

Figure 4.5: `blockTrace` Attribute Implementation

class, also originally written by Tony Sloane and mostly unchanged from its original implementation. The crucial difference between the sequential and concurrent Skink implementations for this phase of verification is in how the SSA conversion is managed. Each `LLVMTermBuilder` instance composes with a class implementing the `LLVMNamer` trait, which provides an interface allowing for the term builder to query the index and unique name of a variable use somewhere within a `Block`. As detailed in Section 3.5, in a concurrent program SSA conversion must be unique within a single thread, as a store to a local variable named `k` in `t1` should not affect the index of `k` in `t2`, but the indexes of global variables should be consistent across all threads. This means that a single `LLVMNamer` cannot manage the indexing of variables for all threads, and rather each function has its own namer, an instance of `LLVMFunctionNamer`, and a `BlockTrace` which contains only the blocks which are executed on that thread. In order to allow the global variable stores and loads to propagate across all threads, each `LLVMFunctionNamer` composes with an `LLVMGlobalNamer`, a singleton that is used any time a thread interacts with a global variable.

In order to keep track of when the index for a variable should be incremented, the `BlockTrace` for each function is converted into a tree with the Kiama tree relations functionality used to create a chain which allows an appropriately updated version of the map to be accessed for any node in the tree. When the use of a variable is found within the `BlockTrace` undergoing term generation, that use can be looked up in the Kiama decorated tree and the appropriate version of the index map for that point in the tree can be found. The namer also leverages the ScalaLLVM analysis module in order to resolve array lookups. Similar to the situation with `pthread_join` arguments, array lookups are performed via a special function with LLVM IR, `getelementptr`, which is used to find a pointer based on a memory address, type and index to an array. However when the item from within an array is accessed, the address of that item will first be loaded into a register before the store or load instruction is applied. This means that in order to detect array accesses, we need to be able to look back through the AST to determine if a regiter was set as the result of a `getelementptr` instruction. As this is an operation applied to an AST, each `LLVMFunctionNamer` is responsible for its own array lookup resolution.

The `traceToSteps` method is used as part of the witness generation module, using the `BlockTrace` for a trace to construct a sequence of steps which contain information about how a trace projects back onto the original source (C, not LLVM IR) program. This element relies on ScalaLLVM's analyser module again to allow us to extract information about where in the original C program a particular instruction came from, which facilitates the construction of a witness trace. Some work has been done to adapt witness generation for concurrent programs, but the expected behaviour of concurrent program witnesses in SV-COMP is not currently very well defined and so the main target for multi-threaded Skink is to maintain parity with sequential Skink's witness generation for single-threaded programs.

The methods `traceToBlockEffect`, `traceToRepetitions` and `checkPost` are used in the construction of interpolant automata with `checkPost` and `traceToBlockEffect` mostly unchanged from the original sequential versions. `traceToRepetitions` is the

```
1  case class DPOR[ S, L ](
2          autoToExplore : DetAuto[ S, L ],
3          processOf : L => Int,
4          independent : Seq[ L ] => ( Int, Int ) => Boolean
5  )
```

Figure 4.6: `DPOR` Class Interface

method used to detect patterns of repeated blocks within a trace which represent a loop in our program automaton that we may be able to capture with an interpolant automaton. This method was adapted to ensure that the patterns that were detect were occuring on a single thread by prepending the block names that were identified as part of the converted trace with the identifier of the thread upon which they occurred.

4.4 Reduction of Program Automaton

Our approach involves aligning the reduction traversal defined in SOURCE-DPOR with the trace generation phase of our verification process. As such, dynamic partial order reduction will effectively replace the search for an accepted trace within the automat implementation of formal languages, previously used within our refinement loop. As with the previous trace discovery traversal however, dynamic partial order reduction is a generic concept, and so our adaptation, TRACE GENERATING SOURCE-DPOR, is impemented as a component of automat, with an interface which allows the automaton which is undergoing reduction to provide a means for the dependence of labels within it to be determined.

The interface for DPOR (as implemented by Franck Cassez in automat) is shown in Fig 4.6 and shows the signature of the two methods which need to be provided by the automaton which is being traversed. The function `independent` provides a sequence of labels which can be used to re-construct the state of the automaton over the course of the trace and hence the equivalent source program statements. In the case of Skink, this corresponds to the standard process of projecting a trace onto the source program to produce a `BlockTrace`. The pair `(Int, Int)` then provides a pair of indices of labels within the trace, which can be mapped back to the two blocks which have been or would be executed as a result of the labels being applied. With two blocks available, determining their dependence can be achieved via a call to the function `areDependent`, which inspects the contents of each instruction call on each block and determines if the two blocks have stored to the same global variable or loaded and stored from the same global variable.

With this function defined, as well as the ability to map a label within the automaton to a particular process (or thread, in our parlance), the class `DPOR` is able to compute the NOTDEP and NOPREDECESSOR sets described in Section 3.6, and also to decide if two labels in the automaton are in a race. The EXPLORE function which forms the main body of Algorithm 1 is implemented as a recursive function, with the source set for each trace stored as a mutable map, allowing the call at the top of the stack to modify the source

```
1  def areDependent(a : Block, b : Block) : Boolean = {
2      def globalAccessNames(block : Block) : (Set[String], Set[String]) =
3          block.optMetaInstructions.map(_.instruction).foldLeft((Set[String](), Set[String
               ]())) {
4              case ((l, s), Load(_, _, _, _, Named(Global(n)), _))  => (l + n, s)
5              case ((l, s), Store(_, _, _, _, Named(Global(n)), _)) => (l, s + n)
6              case ((l, s), _)                                      => (l, s)
7          }
8
9      val (aStores, aLoads) = globalAccessNames(a)
10     val (bStores, bLoads) = globalAccessNames(b)
11     !((aStores & bStores).isEmpty && (aStores & bLoads).isEmpty && (aLoads & bStores).
           isEmpty)
12 }
```

Figure 4.7: `arDependent` Implementation For Resolving Dependency Of Two Blocks

set for previous calls. This enables the process by which non-equivalent interleavings are added to the traversal of previous states.

Fig. 4.7 shows the implementation of the function `areDependent` which is used to determine the dependency of two blocks based on their interactions with the global variables of a program. This function, contained in the `LLVMHelper` object, is used within a method defined on the `LLVMConcurrentAuto` class that can be passed to the `DPOR` class in automat for use in generating traces from a reduced traversal of the program automaton. When its implementation is finalised the `DPOR` object that is used to construct the reduced set of representative traces will take the place of the automat defined `Lang` class within Skink's trace refinement implementation, and can be thought of as a reduced formal language which only accepts a single representative trace (or due to our implementation of the non-optimal algorithm, a small number of traces) for each equivalence class within the program automaton.

When an initial working version of the `DPOR` class for acyclic automaton with only linear refinement has been completed, we can begin to consider how our implementation can be further developed to track and attempt to exhaust the interpolant automata defined labels within the refined program automaton during each trace generation round, as well as a means for detecting and avoiding previously unfolded traces.

Chapter 5

Results

At the time of writing, an implementation of the process of constructing a formal automaton representation of a multi-threaded C program has been implemented within Skink. Further work has begun on implementation of the `Source-DPOR` algorithm as a generic reduced language generator within automat. Using our implementation of the approach described in Chapter 3, we are able to demonstrate equivalence with the previous, synchronous version of Skink, as well as to verify a multi-threaded C programs with some constraints applied to the complexity and operations used within the program.

With the SV-COMP submission deadline being the 7th of December, one month after the submission date of this report, our work to complete an implementation of `Source-DPOR` and to introduce support for additional functionality needed for successful participation in the SV-COMP concurrency category is on track for completion in time to meet our stated aim of competing in SV-COMP 2017. In order to prepare our implementation for SV-COMP and to demonstrate its ability to function within the competition environment, the benchmarking tool BenchExec[1] has been used to run and measure all of the benchmark programs which are described in this chapter.

All benchmarks listed in this chapter were run on a Ubuntu 16.04 LTS virtual machine with a 3.3Ghz quad core i5 CPU. We note the likely differency in execution time of benchmarks on our host to those which are achieved as part of SV-COMP due to the substantially smaller amount of memory and single core assigned to each verification task. All benchmarks were limited to 500MB of ram (as compared to the 15GB available in SV-COMP) with the timeout for all benchmarks except our own concurrency set fixed at 500 seconds to allow benchmarks to be repeated quickly during development and to allow us to observe the variation in run-time of each benchmark program across multiple runs.

[1] https://github.com/sosy-lab/benchexec

5.1 Single-Threaded Programs

One of the primary advantages that we sought to demonstrate with the implementation of our approach in Skink was the ability of our concurrent automaton and surrounding verfication supporting functionality to replicate the results of the existing sequential Skink implementation, allowing a single version of the tool to used for all types of programs within SV-COMP and beyond. As the both the concurrent and sequential implementations are under ongoing development prior to the 2017 SV-COMP submission date, rather than benchmark against a set of large SV-COMP categories, we use a set of simple benchmarks which demonstrate all the basic program constructs which are supported by Skink at present. The benchmark set, Skink Simple, is used by the devlopers as represenatitives of SV-COMP programs and consists of eleven benchmark programs which have been handwritten or adapted from existing SV-COMP benchmarks to serve as a quickly applicable system test suite for use in ongoing work on the development of Skink's sequential implementation.

The naming scheme of the programs follows that which is used in SV-COMP, with the name of the program, for example `array-hard`, suffixed with the correctness of the program (true for correct, false for incorrect) and the property which is being checked (which in all cases for our benchmarks is the reachability of an error assertion call). The benchmark programs used are:

- **array-hard_true-unreach-call.c** This program tests two of the more advanced features that are currently implemented in Skink, consisting of a `main` method which declares an array of 10 integers and then a for loop which iterates over the first five indexes of the array and assigns them to the value of their index. An assertion then checks that the final location in the array that was assigned at index 4 is equal to 4. Proving the correctness of this program requires us to construct an interpolant automaton for the loop which precludes us from exploring infeasible error traces which iterate through the loop more than the fixed number which is given in the program. It also requires that we compute postconditions on the effect of array look-ups, something which is non-trivial in Skink for reasons described in Section 4.3.

- **array-sequence_true-unreach-call.c** Similar to the previous program but without a loop or the need for interpolants, this is a simple benchmark for Skink's ability to correctly handle loads and stores with a local array of integers.

- **eca-like_false-unreach-call.c** In this benchmark an infinite while loop is run which checks the value of two variables initialised with `__VERIFIER_nondet_int` and if a condition is met branches to an error location. Although this test involves a loop, the existence of a feasible error trace within it means that interpolants are not strictly necessary and passing this benchmark requires only correct modeling of `__VERIFIER_nondet_int` and the simple control flow statements in the program.

- **multiple-error-calls_false-unreach-call.c** As with the previous benchmark, this is a test for simple control flow constructs with a program with two error locations (as the program contains both a call to `__VERIFIER_error` and `__VERIRIFER_assert`).
- **simple-fuction_false-unreach-call.c** As no version of Skink currently supports function calls, this benchmark checks that a simple function is correctly inlined and that the control flow is correctly explored, leading to the discovery of a feasible error trace.
- **simple-fuction_true-unreach-call.c** An alternate version of the previous program, a change to the `__VERIFIER_assert` call inside the simple function changes the benchmark from being incorrect to correct.
- **simple-if_false-unreach-call.c** Similar to the simple function benchmark, this program contains a conditional statement which if true results in a call to `__VERIFIER_error`.
- **simple-if_true-unreach-call.c** An alternate version of the previous program, a change to the condition inside the if statement changes the benchmark from being incorrect to correct.
- **simple-loop-array_true-unreach-call.c** Another arrays and loops benchmark, this program creates an array of 10 integers and runs a while loopawhich assigns each position in the array to the value of its index. Again, this requires the correct computation of an interpolant for the while loop to avoid exploration of an infinite number of traces through this loop and also the correct resolution of SMT terms for array lookups.
- **simple-loop_false-unreach-call.c** This program contains a simple loop that runs while a nondeterministic integer is greater than zero and decrements the integer at each iteration of the loop. As it is an incorrect program, there is no need for an interpolant as we will terminate as soon as a fesible error trace is discovered.
- **test-interpolant-franck_true-unreach-call.c** The most complicated interpolant computations of any benchmark in the set, this program introduces two variables `x` and `y` and runs a loop which depends on the value of `x` but modifies the value of both variables. After the loop terminates, a condition is asserted which depends on `y`. Computing an interpolant on this program is slightly more involved than the array benchmarks, as the number of iterations through the loop can affect the truth of the final assertion, although the loop bound ensures that it is not violated.

The results of benchmarking both sequential and concurrent Skink on the Skink-Simple benchmark set are given in Table 5.1. First, we note that both sequential and concurrent Skink are able to find the correct result for all 11 benchmarks in the suite, and that the overall time taken and memory usage for each benchmark is comparable between the two implementations. There is a small apparent overhead in the concurrent implementation, with an overall additional runtime of 3.8 seconds for the full suite, or

Table 5.1: Single-Threaded Benchmark results

	Sequential Skink			Concurrent Skink				
Benchmark Name	**Result**	**CPU Time (s)**	**Mem. Usage (MB)**	**Result**	**CPU Time (s)**	**Mem. Usage (MB)**	**Δ CPU Time (s)**	**Δ Mem. Usage (MB)**
array-hard_true-unreach-call.c	True	4.64	155	True	4.04	139	-0.6	-16
array-sequence_true-unreach-call.c	True	3.69	142	True	5.04	118	1.35	-24
eca-like_false-unreach-call.c	False	3.88	140	False	4.85	122	1.16	-18
multiple-error-calls_false-unreach-call.c	False	3.63	118	False	4.42	142	0.54	24
simple-function_false-unreach-call.c	False	5.51	148	False	5.40	146	-0.11	-2
simple-function_true-unreach-call.c	True	5.21	147	True	6.35	148	1.14	1
simple-if_false-unreach-call.c	False	3.83	116	False	4.30	142	0.47	25
simple-if_true-unreach-call.c	True	4.04	142	True	4.46	123	0.42	-19
simple-loop-array_true-unreach-call.c	True	5.77	149	True	6.12	150	0.35	1
simple-loop_false-unreach-call.c	False	5.46	149	False	5.94	146	0.48	-2
test-interpolant-franck_true-unreach-call.c	True	5.14	148	True	5.44	116	0.3	-32
Total	11/11	50.8	1555	11/11	54.6	1492	3.8	-63

a 7.5% increase over the time taken by the sequential implementation. Possible reasons for this time difference include the additional transformation steps which are applied (although without any effect in the case of these single threaded programs) to the source program's LLVM IR and the additional overhead created by the construction of the local and global `LLVMNamer` instances which are used for term generation. Observations from repeated runs of the benchmarks on both versions of Skink imply that this time difference is consistent across most of the benchmarks in the suite, and does not appear to be connected to a particular feature of any of the benchmark programs. It should also be noted that small differences in execution time, in the region of $\pm 10\%$ are likely due to fluctuations resources availability on the host machine, as the benchmarks were not run in a closed system and were sharing a CPU core with the operating system and other programs on the host. The full BenchExec generated reports for both the sequential and concurrent Skink runs of the Simple Skink set are provided in Appendix C.

On larger sequential benchmarks, like those in the loops category of SV-COMP, the trade-offs between our program automaton and the representation used in the sequential version of Skink become less obvious, with no consistent winner in cpu time or memory usage. Fig 5.2 provides a comparison of the results between the sequential and concurrent implementations of Skink for a small subset of the SV-COMP loops category, taking all of the programs that involve computing a sum with a loop. As we make no claims about improvements to the correctness of our concurrent implementation over the synchronous version, the cases where one or both of the tools returns an incorrect result are not considered in our comparison, and is potentially due to a difference in the interpolant automaton construction strategy used by the version of sequential Skink that was used.

One of the clear features of the successful results is that the difference in performance between the two implementations does not appear to be a function of the size of the program, but is likely more sensitive to the structure of the program and the interpolant automaton construction strategy applied by the two versions. We note however that in all the cases that sequential Skink is able to attain the correct result for, our concurrent implementation can achieve the same result with similar CPU time and memory consumption – in keeping with our expectation that the equivalence of the two approaches over sequential programs generalises for more complicated examples. The BenchExec output for the sum benchmarks for both the sequential and concurrent Skink implementations are available in Appendix C.

Table 5.2: SV-COMP Loops sum Benchmark Results

	Sequential Skink			Concurrent Skink				
Benchmark Name	**Result**	**CPU Time (s)**	**Mem. Usage (MB)**	**Result**	**CPU Time (s)**	**Mem. Usage (MB)**	**Δ CPU Time (s)**	**Δ Mem. Usage (MB)**
sum01_bug02_false-unreach-call_true-termination.c	True	268	513	False	88.5	499	-179.5	-14
sum01_bug02_sum01...true-termination.c	True	28	415	True	24.2	484	0	0
sum01_false-unreach-call_true-termination.c	True	140	515	False	125	520	-15	5
sum03_false-unreach-call_true-termination.c	Timeout	616	582	Timeout	617	675	0	0
sum04_false-unreach-call_true-termination.c	True	29.6	423	True	23.4	474	0	0
sum_array_false-unreach-call.c	False	24.8	476	False	11.8	363	-13	-113
sum01_true-unreach-call_true-termination.c	True	9.92	267	False	10.5	318	0.58	51
sum03_true-unreach-call_true-termination.c	True	24.5	417	True	26.6	482	2.1	65
sum04_true-unreach-call_true-termination.c	True	6.31	228	True	8.3	280	1.99	52
sum_array_true-unreach-call.c	Timeout	560	562	False	176	508	-384	54
Total	11/11	1710	4380	7/10	1110	4604	-587	100

5.2 Concurrent Programs

The benchmarks which are used as part of SV-COMP typically test a number of pieces of concurrent functionality in concert with one another. During development, we defined a series of our own simple and more targeted benchmarks which allow us to both test our implemenetation and observe how our approach handles the various building blocks of a full multi-threaded program. The results we present have been produced without the application of a reduction algorithm to the traces explored by the refinement process and so in some cases required a large number of iterations to terminate as all of equivalent interleavings of concurrent blocks were explored.

As with the Skink-Simple benchmark set, our own multi-threaded benchmarks, Skink-Concurrency, follow the SV-COMP naming scheme to allow BenchExec to detect whether the result returned by Skink is the expected one or not. Our concurrent benchmarks consist of:

- **concurrent-loop_true-unreach-call.c** This program is a simple benchmark for the effect of a loop which runs concurrently with the main thread. The body of the child thread assigns 5 to global variable `a` and runs a loop which iterates 10 times but which does not effect the value of `a`. In the body of main, a check is made on `a > 5` and, if it succeeds, `__VERIFIER_error` is called. While it is therefore possible to compute an interpolant for the loop inside the body of the child thread which captures the effect of any number of iterations, the potential for the loop to be interrupted by interleaved blocks from main means that without a reduction there are a large number of interleavings which we need to discover and construct the interpolants for, using the scheme described in Section 3.5.

- **concurrent-loop_false-unreach-call.c** A modified version of the previous program but with a change to the condition in main to introduce a feasible error trace into the program. While this is a program which contains an error location and so not all paths of execution need to be explored, there are still a large number of possible interleavings of the blocks from main and the blocks that constitute the loop inside the child thread which mean that discovery of a feasible error trace without reduction is a substantial computation.

- **sync-concurrent-loop_true-unreach-call.c** A modification of the original concurrent-loop program which uses a join in the main thread, converting the loop in the child

thread into an effectively synchronous trace, as it cannot be interleaved with any instruction from main.

- **fib-threads_false-unreach-call.c** This is the example program shown in Fig. 3.2, which constitutes two threads intended to cooperate to construct Fibonacci numbers. The two global variables `i` and `j` with `i` added to `j` in one thread and `j` added to `i` in the other. If the two threads interleave in the ideal way, the final value of `i + j` should be 5, the sixth Fibonacci number. However, as there is no synchronisation constraint on when `main` checks the value of `i + j`, there are many feasible error traces in this program, the most obvious being the one where neither thread begins to execute before main checks the error condition.

- **fib-threads_true-unreach-call.c** A modification of the previous program to relax the condition on `i + j` in order to account for any of the possible orders of execution for the program. Although the condition has been relaxed, proving this program correct still necessitates exploring every possible interleaving of the load and store instructions on each thread with one another and the condition in main and so without reduction represents a substantial verification task.

- **sync-fib-threads_true-unreach-call.c** A further modification of the original Fibonacci threads program, show in Fig. 3.8, which introduces the use of a mutex and joins to enforce the atomicity of the `+=` operator in each thread and to ensure that main does not check the condition on the value of `i + j` until both threads have terminated. The use of synchronisation operations substantially reduces the size of automaton which needs to be explored, even without the use of partial order reduction, and makes this program substantially faster than the unsynchronised versions to verify.

- **simple-threads_false-unreach-call.c** A simple test which performs an assignment from one global variable into another in two concurrent threads and contains a check in `main` on the value of their sum. Similar to the Fibonacci programs but the use of a direct assignment rather than addition reduces the size of the AST and the number of potential interleavings within the program. As with the incorrect Fibonacci program, although the existence of a feasible error trace in the program means we do not need to explore all error paths, the number of error traces that exist in the unreduced program means that finding a counter-example is non-trivial.

- **simple-threads_true-unreach-call.c** The previous program modified to accept all possible results for the sum `i + j`, requiring all error traces within the program to be explored.

- **sync-threads_true-unreach-call.c** A benchmark for synchronisation operations, this program uses all of the Pthread API synchronisation types and functions that are supported within Skink. A global variable `num` is initialised as 0, along with a `pthread_mutex_t`, `m`, and a `pthread_cond_t`, `five`, which are used to synchronise

Table 5.3: Skink Concurrency Set Benchmarks

Benchmark Name (s)	Result	CPU Time (s)	Memory Usage (MB)
concurrent-loop_true-unreach-call.c	True	5583	507
concurrent-loop_false-unreach-call.c	False	5.34	500
sync-concurrent-loop_true-unreach-call.c	True	6.25	250
fib-threads_false-unreach-call.c	False	4.73	203
fib-threads_true-unreach-call.c	True	3842	507
sync-fib-threads_true-unreach-call.c	True	27.2	488
simple-threads_false-unreach-call.c	False	4.67	177
simple-threads_true-unreach-call.c	True	2249	157
sync-threads_true-unreach-call.c	True	18.9	500
sync-threads_false-unreach-call.c	False	4.72	210
Total	10/10	11746	3474

the two threads. The mutex `m` is used to ensure the two threads cannot interleave one another, with the condition `five` used to enforce an ordering on the two threads. In main, the two threads are joined by the main thread to ensure that both have terminated before the error condition is checked. In the first thread, the value of `num` is set to 5 and then the condition `five` is signalled, allowing the second thread to unblock and set `num` to 10 once the mutex `m` is released by the first thread. After the two threas have made their assignment and terminated, main checks that the value of `num` is 10. While this program has more lines of code than any other benchmark in our set, the enforcement of the synchronisation semantics on the program automaton mean that the number of error traces that need to be explored is quite small.

Table 5.3 shows the results of verification the Skink Concurrency set. As expected, the use of of partial order reduction exploring all possible interleavings of the thread interactions in the `fib-threads` and `simple-threads` benchmarks in order to prove correctness for the true cases is an expensive process, requiring thousands of iterations of the refinement loop. Also demonstrated by these results is the effect of applying the semantics of synchronisation to our program automaton, with the synchronised version of the `fib-threads` and `concurrent-loop` programs able to be proven correct in substantially less time than the equivalent unsynchronised programs. Overall the results from our local benchmark are indicative of the value of partial order reduction. The three programs which took the most time to verify all represent good case studies for our implementation of Source-DPOR.

Table 5.4 shows the results of running our current implementation of Skink against the pthread sub-category of SV-COMP's concurrency category, which contains the simplest concurrent programs which are available for verification within SV-COMP. The results of achieved for these benchmarks are demonstrative of the two significant missing features in the current implementation of Skink. Somewhat surprisingly, our lack of an implementation of partial order reduction is not at fault for any of the failures from this set of benchmarks, but rather a lack of support for the creation of threads within loops using

Table 5.4: SV-COMP Concurrency pthread Set Benchmarks

Benchmark Name (s)	Result	CPU Time (s)	Memory Usage (MB)
bigshot_*.c	Unknown		
fib_bench_*.c	Timeout	500	500
indexer_true-unreach-call.c	Timeout	500	416
queue_ok_*.c	Unknown		
sigma_false-unreach-call.c	Unknown		
singleton_false-unreach-call.c	Unknown		
stack_*.c	Unknown		
stateful01_false-unreach-call.c	False	5.92	157
stateful01_true-unreach-call.c	True	131	500
twostage_3_false-unreach-call.c	Unknown		
sync01_true-unreach-call.c	True	403	500

an array of `pthread_t` identifiers (such as in the `stack`, `queue` and `sigma` benchmarks) and dynamic memory allocation (used in `bigshot`). The remaining unsolved group of benchmarks, `fib-bench`, represent a set which would likely benefit somewhat from partial order reduction but which still require the exploration of a huge number of non-equivalent traces as the two threads operate exclusively on global data and each interleaving has the potential to produce a different final result.

Rather than run these benchmarks against other SV-COMP concurreny category competitors locally, we direct the reader to the results from SV-COMP 2016[2] to gain an understanding of how our results compare to other similar tools. We note that all of the tools which were able to successfully verify the `fib_bench_true` cases are based on bounded model checking (such as Lazy-CSEQ and MU-CSEQ [18, 22]) or, as in the case of CIVL [24], verify specific safety properties, which in neither case are able to provide a guarantee of correctness. In contrast, Impara, a tool which impelements predicate abstraction refinement, an approach which shares the soundness of trace abstraction refinement and thus which must explore all non-equivalent traces in a program, timed out on all of the `fib_bench` programs.

One encouraging aspect of our results so far is that we have been able to successfuly verify programs which were only handled in SV-COMP 2016 by bounded model checking tools which do not produce a proof of correctness. The gap between what bounded approaches to verification and those which seek to prove provide a guarantee of partial correctness is clear though, with the programs which Skink is able to verify correctness of in a number minutes taking the most successful sequentialisation based tools a matter of seconds. The full results for both the Skink-Concurrency and SV-COMP concurrency benchmark sets, as produced by BenchExec, are given in Appendix D.

[2] https://sv-comp.sosy-lab.org/2016/results/results-verified/Concurrency.table.html

5.3 Scala As a Platform For Verification

As well as demonstrating the ability of our approach to successfuly verify concurrent programs, our work on the implementation of the necessary functionality to support our approach in Scala is demonstrative of the value of a number of language features in this type of work. Skink forms an interesting case study in the application of work by Tony Sloane and his collaborators on the use of Scala as a platform for working with embedded domain specific languages (DSLs) [20, 21], as it relies heavily on the embedded DSLs implemented within its surrounding libraries. At the centre of most of the implementation work described in this report are interactions with the AST of the source LLVM IR program as constructed by ScalaLLVM[3], supported by the sbt-rats![4] parser generator and the Kiama language processing framework[5]. The ability to arbitrarily traverse and transform our source program as a tree defined by our own LLVM IR DSL and compute relations on nodes within our AST using Kiama allows for code surrounding the construction of our automaton to be expressive and easy to write. The pretty printing functionality generated by sbt-rats! is also invaluable for debugging.

We also observe the power of Scala pattern matching and extractors as they apply to our representation of LLVM IR, with all of our interactions with the source program during the analysis reducing to a pattern match on a function's AST. The best example of this is found when computing the effects of a block on the synchronisation tokens of a successor state, which involves inspecting the context of the block for calls to a large number of different Pthread API functions. Each `Call` instruction in LLVM IR contains a number of arguments, most of which aren't useful when attempting to identify the name and arguments of a particular function call, so in order to hide some of this unnecessary information, we introduce an extractor which allows us to pattern match on an LLVM IR instruction if it is a call to a function with a specific name, or alternatively to allow us to collect the arguments of a function call with a name that matches the one we supplied to the pattern match.

The source for the `PthreadOperation` extractor used to collect information about Pthread synchronisation function calls is given in Fig. 5.1. The number of arguments that are to be expected and whether or not we are interested in returning their value depends on which synchronisation function is called, with the name and arguments of the function call collected by the `GlobalFunctionCallWithArgs` extractor and the name of each argument extracted by `ValueArgName`. The number of potential cases involved makes this quite an involved pattern match but the number of ignored fields in each pattern match is massive improvement over matching directly on the LLVM IR AST. Also as expected, the expressiveness and readability of this quite complicated pattern match is a substantial improvement over the type of code would be required to achieve the same collection of information using an if/else construct.

Also used by Skink and taking advantage of the appropriateness of Scala for hosting

[3] https://bitbucket.org/inkytonik/scalallvm
[4] https://bitbucket.org/inkytonik/sbt-rats
[5] https://bitbucket.org/inkytonik/kiama

```
object PthreadOperation {
    def unapplySeq(insn : MetaInstruction) : Option[Seq[String]] =
        insn match {
            case GlobalFunctionCallWithArgs(callName,
                Vector(ValueArgName(Global(syncToken)))
                ) if List(
                "pthread_mutex_lock",
                "pthread_mutex_unlock",
                "pthread_cond_signal"
            ).contains(callName) =>
                Some(List(callName, syncToken))
            case GlobalFunctionCallWithArgs(callName,
                Vector(ValueArgName(Global(syncToken)))
                ) if callName == "pthread_cond_condition" =>
                Some(List(callName, syncToken))
            case GlobalFunctionCallWithArgs(callName,
                Vector(ValueArgName(Global(syncToken)),
                    _)
                ) if List("pthread_mutex_init", "pthread_cond_init").contains(callName)
                    =>
                Some(List(callName, syncToken))
            case GlobalFunctionCallWithArgs(callName,
                  Vector(ValueArgName(Global(syncToken)),
                  ValueArgName(Named(Global(returnMutex))))
                ) if callName == "pthread_cond_wait" =>
                Some(List(callName, syncToken, returnMutex))
            case GlobalFunctionCallWithArgs(callName,
                    Vector(ValueArgName(Global(syncToken)),
                    _)
                ) if callName == "pthread_join" =>
                Some(List(callName, threadNameRegister))
            case _ =>
                None
        }
}
```

Figure 5.1: `PthreadOperation` Extractor Object

embedded DSLs is MQ-Scala-SMTLib[6], used by Skink to construct a logical representation of a trace and then for interactions with an SMTLib compliant solver to compute the feasibility of the trace. As with ScalaLLVM, MQ-Scala-SMTLib introduces a family of types and operators which allows the construction of each SMT term to resemble the real logical syntax of an SMT propositional logic statement and leverages the generated pretty-printing of sbt-rats! to improve the debugging environment for term generation within Skink.

[6] https://bitbucket.org/franck44/mq-scala-smtlib

Chapter 6

Conclusions

The goal of our work was to re-explore the method introduced by Cassez and Ziegler in [8] to adapt trace abstraction refinement based verification for concurrent programs and to produce an approach which is able to apply this verification technique to multi-threaded C programs in the SV-COMP concurrency category. An approach for applying partial order reduction to our program representation, as introduced in [8] was also explored, with the possibility of improving on the prior work by applying a more effective reduction algorithm, defined in [1]. It was on this basis that our approach was derived and described in Chapter 3.

We described in Section 3.1 the process undertaken to transform a source C program via Clang and LLVM into an intermediate representation which, once parsed by a language processing library ScalaLLVM, can be transformed to allow interleavings of the blocks comprising the CFG of each function in the program to be explored. It was observed that our approach of using LLVM IR as a representation of our source program during verification provides a number of advantages over statement level representations, as it allows instruction level resolution of global memory access. This is necessary for correct modeling of atomicity within our analysis, but through its grouping of sequences of instructions in blocks, can still provide a succinct means for representing a particular path of execution within a program.

In order to represent the synchronised product of the thread automata in a concurrent program, a scheme for dynamically generating the structure of an automaton which represents the control flow for the complete concurrent program was introduced in Section 3.3. This is acheived by traversing the AST of the source program in order to discover available branches from a given thread and block which become the labels for our representation. The location of the most recently encountered block for each thread comprises the state of the automaton. We detailed the means by which new thread creation can be discovered by inspecting the syntax of blocks being "executed" within a particular path of execution in our program, and also the method for detecting and applying the synchronisation operations and tokens provided by the Pthread API.

To achieve reduction of our program automaton without losing the benefits of its dynamic construction, we adapted Source-DPOR algorithm introduced by Abdulla et al.

in [1]. This is described in Section 3.6 and further an explanation of its fundamental operations in the context of our approach, with the ongoing implementation of this adaptation described in Section 4.4.

In Chapter 4 we recounted details of the implementation of our approach as a fork of the existing trace abstraction refinement tool, Skink. This includes an explanation of how the previous structure within the tool was adapted as well as new functionality which was added as part of Skink's LLVM package and also within the external library automat to support the verification of concurrent programs.

In our results we provided an overview of how our implementation performs in comparison to the previous sequential implementation of trace abstraction refinement which existed in Skink over a set of simple sequential benchmarks. This was supplemented by a small subset of more difficult benchmarks from the SV-COMP loops category. Having demonstrated the abilites of our program automaton in sequential cases, we next showed its performance on a set of our own concurrent benchmarks and in particular its ability to perform well on programs which employ Pthread synchronisation functionality, due to the enforcement of the semantics of Pthread synchronisation on the struction of the automaton. As a preview to our ongoing work in preparing our implementation for the upcoming SV-COMP 2017, we provided the results from running our implementation against the SV-COMP concurrency pthread category, and commented on the current shortcomings of our implementation and how they effect our results on the pthread benchmarks.

Finally, we discussed the role of Skink as a case study for the application of software language engineering techniques within Scala. This included the ways in which our implementation took advantage of the language features and surrounding tools in order to allow succinct and expressive interactions with the source program representation from ScalaLLVM as well as with the SMT term construction provided by MQ-Scala-SMTLib.

Chapter 7

Future Work

7.1 Exploration of Partial Order Reduction

Although we have discovered that a large number of the benchmarks in the SV-COMP concurrency category would not benefit significantly from partial order reduction, the ability to explore a minimal or near-minimal set of representative traces in every program encountered remains a valuable feature for Skink as a verification tool. Our immediate plans for partial order reduction in Skink revolve around completing our generic implementation of the Trace-Generating Source-DPOR algorithm described in Section 3.6 and the supporting operations within Skink, before benchmarking the resulting reduction against our previous, un-reduced representation.

Following this supplementing our simplified approach with the wake-up tree data structure described in [1] will allow us to explore the trade-off between producing a provably minimal set of representative traces and the increased computation time and memory consumption required to manage the additional data structure.

Another avenue of exploration for the partial order reduction method used in Skink comes in the form of a new technique described by Chatterjee et al. in [9], which defines a new means of defining equivalence on traces within a program. Termed *observational equivalence* it is distinct from *Mazurkiewicz equivalence* which is used by all of the partial order reduction algorithms described in Section 2.4. Using this new means of considering equivalence, the authors introduce a new dynamic partial order reduction algorithm, DC-DPOR, which is shown experimentally to produce a substantial improvement in reduction over the original dynamic partial order reduction algorithm introduced by Flanagan and Godefroid in [11].

7.2 Recognising and Handling Thread Creation In Loops

As recognised in Section 5.2, limited by of our current implementation of concurrent verification in Skink is our ability to recognise and handle calls to Pthread library functions, particularly `pthread_create`. In the short term, the most straightforward approach to overcoming this limitation appears to be to apply loop unrolling, a technique which is very common amongst bounded model checking tools. It can be quite easily applied as an initial step in the compilation of our source program from C into LLVM IR, alongside the inlining of functions. By un-rolling the loops containing calls to Pthread function calls, it will be possible to distinguish between a call which is being re-visited due to existing within a loop and a call that is being re-visited at a different location within a trace. It also makes it possible to statically determine the identifier used as an argument to each Pthread API call, which is necessary for the application of our approach for enforcing synchronisation semantics on the program automaton.

Going forward, we can apply a similar technique as we have done for synchronisation. We will attempt to detect the structure of the loop which contains calls to Pthread API functions and use our knowledge of the variable's counter or bounds in order to keep track of which identifier is being used as an argument to the function being called within the loop. A prior means of identifying important values within a source program's syntax and tracking them as part of the trace exploration, rather than via refinement, is described by Cassez et al. in [6] where it was used to handle programs with loops for which an interpolant could not be discovered.

7.3 Adapting Program Automata For Functions

One of the potential extensions for our AST traversal approach for the construction of our program automaton is to support non-inlined function calls. The general scheme would be to recognise function calls in a similar fashion as we do for calls to Pthread API functions, but instead of applying our model of the entire function call's effect on the structure of our automaton, we apply the semantics of the call instruction itself and jump from the current location to the first block of the function which was called. There are some non-trivial issues surrounding this adaptation of our existing machine, including the problem of mapping between the actual and formal arguments of the function during term generation, the propagation of return values and the recording of where in the source program each function should return to.

This approach lacks some of the power of one previously described technique for modeling function calls in trace refinement in [7]. It requires the effect of a function call to be re-computed every time a new trace containing that function call is explored, rather than re-using a previously computed summary of the pre and post-conditions of the function. It is however, a fairly flexible approach, and requires almost no changes to the refinement algorithm itself as the ultimate output of our automaton is just a trace to an error

location.

Chapter 8

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
ATP	Automated Theorem Prover
CEGAR	Counter-Example Guided Abstraction Refinement
CFG	Control Flow Graph
CPU	Central Processing Unit
DCA	Deterministic and Complete Automata
IR	Intermediate Representation
NFA	Non-deterministic Finite Automata
POR	Partial Order Reduction
SMT	Satisfiability Modulo Theories
SSA	Static Single Assignment
TAR	Trace Abstraction Refinement

Bibliography

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 373–384. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535845

[2] D. Beyer, "Reliable and reproducible competition results with benchexec and witnesses report on sv-comp 2016," in *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 887–904. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-49674-9_55

[3] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," 2003.

[4] N. Bjorner and L. de Moura, "Applications of smt solvers to program verification," in *Notes for the Summer School on Formal Techniques*. Springer, 2014.

[5] F. Cassez, "Automated software verification." [Online]. Available: http://science.mq.edu.au/~fcassez/software-verif.html

[6] F. Cassez, T. Matsuoka, E. Pierzchalski, and N. Smyth, "Perentie: Modular trace refinement and selective value tracking - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, 2015, pp. 439–442. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46681-0_39

[7] F. Cassez, C. Müller, and K. Burnett, "Summary-based inter-procedural analysis via modular trace refinement," in *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, 2014, pp. 545–556. [Online]. Available: http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2014.545

[8] F. Cassez and F. Ziegler, "Verification of concurrent programs using trace abstraction refinement," in *Logic for Programming, Artificial Intelligence, and*

Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings, ser. Lecture Notes in Computer Science, vol. 9450. Springer, 2015, B - International Conferences, pp. 233–248. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48899-7_17

[9] K. Chatterjee, A. Pavlogiannis, N. Sinha, and K. Vaidya, "Data-centric dynamic partial order reduction," 2016.

[10] S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle, "OpenJDK's java.utils.collection.sort() is broken: The good, the bad and the worst case," in *Computer Aided Verification*. Springer Science + Business Media, 2015, pp. 273–289. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21690-4_16

[11] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: ACM, 2005, pp. 110–121. [Online]. Available: http://doi.acm.org/10.1145/1040305.1040315

[12] P. Godefroid, Ed., *Partial-Order Methods for the Verification of Concurrent Systems*. Springer Berlin Heidelberg, 1996. [Online]. Available: http://dx.doi.org/10.1007/3-540-60761-7

[13] P. Godefroid, "Software model checking: The Verisoft approach," *Form. Methods Syst. Des.*, vol. 26, no. 2, pp. 77–101, Mar. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10703-005-1489-x

[14] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski, "Ultimate automizer with smtinterpol - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, 2013, pp. 641–643. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36742-7_53

[15] M. Heizmann, J. Hoenicke, and A. Podelski, "Refinement of trace abstraction," in *Static Analysis*. Springer Science + Business Media, 2009, pp. 69–85. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03237-0_7

[16] M. Heizmann, "Software model checking for people who love automata," in *Computer Aided Verification*. Springer Science + Business Media, 2013, pp. 36–52. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39799-8_2

[17] G. J. Holzmann and D. Bosnacki, "Multi-core model checking with SPIN," in *2007 IEEE International Parallel and Distributed Processing Symposium*. Institute of Electrical & Electronics Engineers (IEEE), 2007. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2007.370410

[18] O. Inverso, T. Nguyen, E. Tomasco, B. Fischer, S. L. Torre, and G. Parlato, "Lazy-cseq 1.0:(competition contribution)," October 2015. [Online]. Available: http://eprints.soton.ac.uk/387010/

[19] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006. [Online]. Available: http://dx.doi.org/10.1109/MC.2006.180

[20] A. M. Sloane, "Lightweight language processing in kiama." in *GTTSE*, 2009, pp. 408–425.

[21] A. M. Sloane, F. Cassez, and S. Buckley, "The sbt-rats parser generator plugin for scala (tool paper)," in *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, ser. SCALA 2016. New York, NY, USA: ACM, 2016, pp. 110–113. [Online]. Available: http://doi.acm.org/10.1145/2998392.3001580

[22] E. Tomasco, T. Nguyen, O. Inverso, B. Fischer, S. L. Torre, and G. Parlato, "Mu-cseq 0.4: individual memory location unwindings: (competition contribution)," April 2016. [Online]. Available: http://eprints.soton.ac.uk/386736/

[23] B. Wachter, D. Kroening, and J. Ouaknine, "Verifying multi-threaded software with impact," in *2013 Formal Methods in Computer-Aided Design*. Institute of Electrical & Electronics Engineers (IEEE), oct 2013. [Online]. Available: http://dx.doi.org/10.1109/FMCAD.2013.6679412

[24] M. Zheng, J. G. Edenhofner, Z. Luo, M. J. Gerrard, M. S. Rogers, M. B. Dwyer, and S. F. Siegel, "CIVL: applying a general concurrency verification framework to c/pthreads programs (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, 2016, pp. 908–911. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-49674-9_57

[25] F. Ziegler, "Verification of concurrent programs via partial-order reduction and trace refinement," Master's thesis, University of Augsburg, 2014.

Appendix A

Weekly Meetings Record

Consultation Meetings Attendance Form

Week	Date	Comments (if applicable)	Student's Signature	Supervisor's Signature
1	5/8	Skype meeting		
2	11/8	Skype meeting		
3	18/8	Skype meeting		
4	25/8	Skype meeting via email.		
5	1/9	Skype meeting		
6	8/9	Skype meeting		
7	15/9	Skype meeting		
8	6/10	Skype meeting		
9	13/10	Skype meeting		
10	20/10	Skype meeting		
11	27/10	Skype meeting		
12	3/11	Skype meeting		

Appendix B

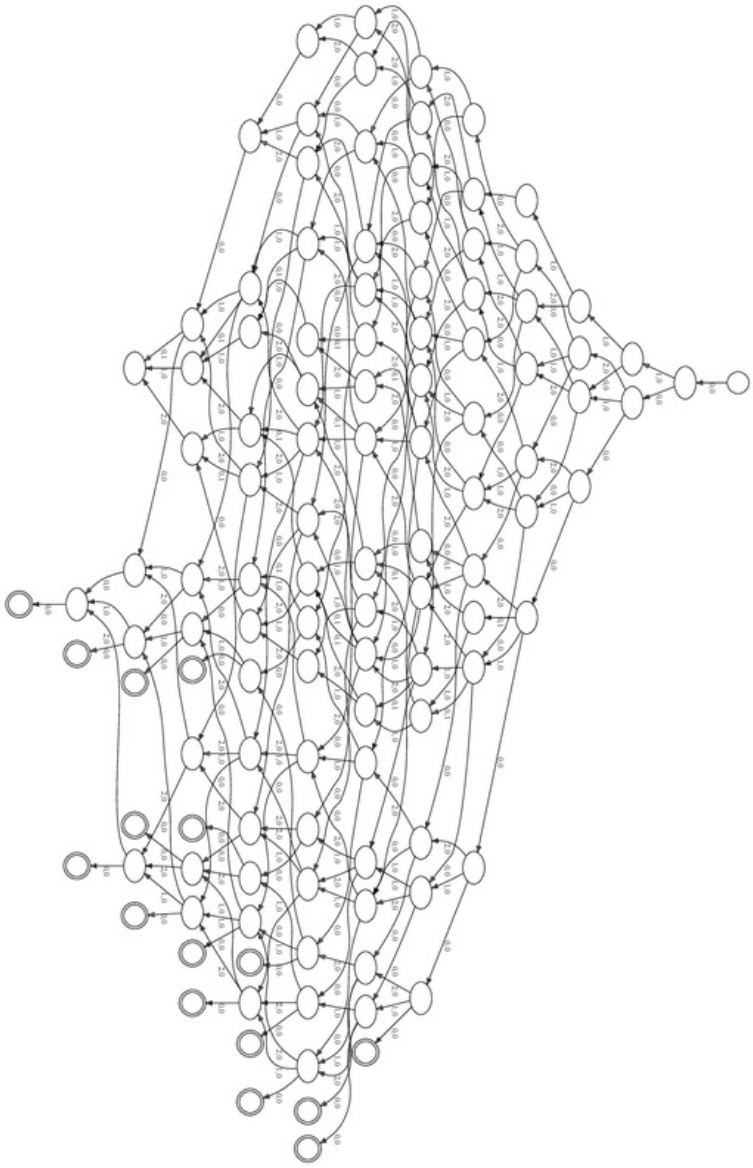
Transformed LLVM IR Program and Automaton For Example in Fig. 3.2

B.1 Transformed LLVM IR Source Code

```
%union.pthread_attr_t = type { i64, [48 x i8] }

@i = global i32 1, align 4
@j = global i32 1, align 4

define i8* @f1(i8* %arg) #0 {

    %1 = alloca i8*, align 8
    %2 = alloca i8*, align 8
    store i8* %arg, i8** %2, align 8
    call void @llvm.dbg.declare(metadata i8** %2, metadata !23, metadata !24) , !dbg !25
    %3 = load i32, i32* @i, align 4, !dbg !26
    br label %__threading.1.nolabel

    %7 = load i8*, i8** %1, !dbg !29
    ret i8* %7, !dbg !29

  __threading.nolabel:
    call void @pthread_exit(i8* null) #4 , !dbg !28
    unreachable, !dbg !28

  __threading.0.nolabel:
    %5 = add nsw i32 %4, %3, !dbg !27
    store i32 %5, i32* @j, align 4, !dbg !27
    br label %__threading.nolabel

  __threading.1.nolabel:
    %4 = load i32, i32* @j, align 4, !dbg !27
    br label %__threading.0.nolabel

}

define i8* @f2(i8* %arg) #0 {

    %1 = alloca i8*, align 8
    %2 = alloca i8*, align 8
    store i8* %arg, i8** %2, align 8
    call void @llvm.dbg.declare(metadata i8** %2, metadata !30, metadata !24) , !dbg !31
    %3 = load i32, i32* @j, align 4, !dbg !32
    br label %__threading.1.nolabel

    %7 = load i8*, i8** %1, !dbg !35
    ret i8* %7, !dbg !35

  __threading.nolabel:
    call void @pthread_exit(i8* null) #4 , !dbg !34
    unreachable, !dbg !34

  __threading.0.nolabel:
    %5 = add nsw i32 %4, %3, !dbg !33
    store i32 %5, i32* @i, align 4, !dbg !33
    br label %__threading.nolabel

```

```
54    __threading.1.nolabel:
55      %4 = load i32, i32* @i, align 4, !dbg !33
56      br label %__threading.0.nolabel
57
58  }
59
60  define i32 @main(i32 %argc, i8** %argv) #0 {
61
62      %1 = alloca i32, align 4
63      %2 = alloca i32, align 4
64      %3 = alloca i8**, align 8
65      %t1 = alloca i64, align 8
66      %t2 = alloca i64, align 8
67      store i32 0, i32* %1
68      store i32 %argc, i32* %2, align 4
69      call void @llvm.dbg.declare(metadata i32* %2, metadata !36, metadata !24) , !dbg !37
70      store i8** %argv, i8*** %3, align 8
71      call void @llvm.dbg.declare(metadata i8*** %3, metadata !38, metadata !24) , !dbg !39
72      call void @llvm.dbg.declare(metadata i64* %t1, metadata !40, metadata !24) , !dbg !44
73      call void @llvm.dbg.declare(metadata i64* %t2, metadata !45, metadata !24) , !dbg !46
74      %4 = call i32 @pthread_create(i64* %t1, %union.pthread_attr_t* null, i8* (i8*)* @f1, i8* null) #5 ,
            !dbg !47
75      br label %__threading.2.nolabel
76
77    ; <label>:10
78      br label %11, !dbg !55
79
80    ; <label>:11
81      br label %__error.11
82
83    ; <label>:12
84      ret i32 0, !dbg !59
85
86    __threading.nolabel:
87      %8 = add nsw i32 %6, %7, !dbg !52
88      %9 = icmp ne i32 %8, 5, !dbg !53
89      br i1 %9, label %10, label %12, !dbg !54
90
91    __threading.0.nolabel:
92      %7 = load i32, i32* @j, align 4, !dbg !51
93      br label %__threading.nolabel
94
95    __threading.1.nolabel:
96      %6 = load i32, i32* @i, align 4, !dbg !49
97      br label %__threading.0.nolabel
98
99    __threading.2.nolabel:
100     %5 = call i32 @pthread_create(i64* %t2, %union.pthread_attr_t* null, i8* (i8*)* @f2, i8* null) #5 ,
            !dbg !48
101     br label %__threading.1.nolabel
102
103   __error.11:
104     call void (...) @__VERIFIER_error() #4 , !dbg !57
105     unreachable, !dbg !57
106
107 }
```

B.2 Full Concurrent Automaton



Appendix C

BenchExec Results For Sequential Benchmarks

C.1 Sequential Skink Simple Set Benchmark Results

Tool	skink			
Limits	timelimit: 500 s, memlimit: 500 MB, CPU core limit: 1			
Host	matt-VirtualBox			
OS	Linux 4.4.0-45-generic x86_64			
System	CPU: Intel Core i5-2500K CPU @ 3.30GHz, cores: 1, frequency: 3300 MHz; RAM: 4144 MB			
Date of execution	2016-11-06 07:36:39 AEDT			
Run set	skink17			
programs/simple/	status	cputime (s)	walltime (s)	memUsage
eca-like_false-unreach-call.c	false(reach)	4.64	5.32	155451392
multiple-error-calls_false-unreach-call.c	false(reach)	3.69	3.71	142540800
simple-function_false-unreach-call.c	false(reach)	3.88	3.91	139976704
simple-if_false-unreach-call.c	false(reach)	3.63	3.65	117796864
simple-loop_false-unreach-call.c	false(reach)	5.51	5.54	147988480
array-hard_true-unreach-call.c	true	5.21	5.22	146649088
array-sequence_true-unreach-call.c	true	3.83	3.88	116453376
simple-function_true-unreach-call.c	true	4.04	4.06	142163968
simple-if_true-unreach-call.c	true	5.77	5.79	149065728
simple-loop-array_true-unreach-call.c	true	5.46	5.49	149057536
test-interpolant-franck_true-unreach-call.c	true	5.14	5.16	147910656
programs/simple/	status	cputime (s)	walltime (s)	memUsage
total tasks	11	50.8	51.7	1555054592
local summary	-	50.5	53.1	-
correct results	11	50.8	51.7	1555054592
correct true	6	29.4	29.6	851300352
correct false	5	21.3	22.1	703754240
incorrect results	0	-	-	-
incorrect true	0	-	-	-
incorrect false	0	-	-	-
score (11 tasks, max score: 17)	**17**	-	-	-
Run set	skink17			

C.2 Concurrent Skink Simple Set Benchmark Results

Tool	skink			
Limits	timelimit: 500 s, memlimit: 500 MB, CPU core limit: 1			
Host	matt-VirtualBox			
OS	Linux 4.4.0-45-generic x86_64			
System	CPU: Intel Core i5-2500K CPU @ 3.30GHz, cores: 1, frequency: 3300 MHz; RAM: 4144 MB			
Date of execution	2016-11-06 07:34:50 AEDT			
Run set	skink17			
programs/simple/	status	cputime (s)	walltime (s)	memUsage
eca-like_false-unreach-call.c	false(reach)	4.04	4.06	139194368
multiple-error-calls_false-unreach-call.c	false(reach)	5.03	5.04	118235136
simple-function_false-unreach-call.c	false(reach)	4.16	4.85	121675776
simple-if_false-unreach-call.c	false(reach)	4.23	4.42	141963264
simple-loop_false-unreach-call.c	false(reach)	5.20	5.40	145956864
array-hard_true-unreach-call.c	true	6.07	6.35	147771392
array-sequence_true-unreach-call.c	true	4.21	4.30	142446592
simple-function_true-unreach-call.c	true	4.43	4.46	122613760
simple-if_true-unreach-call.c	true	6.10	6.12	150147072
simple-loop-array_true-unreach-call.c	true	5.74	5.94	146456576
test-interpolant-franck_true-unreach-call.c	true	5.35	5.44	116322304
programs/simple/	status	cputime (s)	walltime (s)	memUsage
total tasks	11	54.6	56.4	1492783104
local summary	–	54.2	57.9	–
correct results	11	54.6	56.4	1492783104
correct true	6	31.9	32.6	825757696
correct false	5	22.7	23.8	667025408
incorrect results	0	–	–	–
incorrect true	0	–	–	–
incorrect false	0	–	–	–
score (11 tasks, max score: 17)	**17**	–	–	–
Run set	skink17			

C.3 Sequential Skink SV-COMP Loops sum Benchmark Results

Tool	skink			
Limits	timelimit: 500 s, memlimit: 2000 MB, CPU core limit: 1			
Host	matt-VirtualBox			
OS	Linux 4.4.0-45-generic x86_64			
System	CPU: Intel Core i5-2500K CPU @ 3.30GHz, cores: 3, frequency: 3300 MHz; RAM: 4144 MB			
Date of execution	2016-11-07 14:25:13 AEDT			
Run set	skink17			
.../.../Dropbox/Thesis/sv-benchmarks/c/loops/	status	cputime (s)	walltime (s)	memUsage
sum01_bug02_false-unreach-call_true-termination.i	**true**	268	268	513441792
sum01_bug02_sum01_bug02_base.case_false-unreach-call_true-termination.i	**true**	28.0	28.0	415416320
sum01_false-unreach-call_true-termination.i	**true**	140	140	514797568
sum03_false-unreach-call_true-termination.i	**timeout**	616	501	582422528
sum04_false-unreach-call_true-termination.i	**true**	29.6	29.7	422580224
sum_array_false-unreach-call.i	false(reach)	24.8	24.9	476082176
sum01_true-unreach-call_true-termination.i	true	9.92	9.95	266649600
sum03_true-unreach-call_false-termination.i	true	24.5	24.6	417132544
sum04_true-unreach-call_true-termination.i	true	6.31	6.35	227794944
sum_array_true-unreach-call.i	**timeout**	560	501	561672192
.../.../Dropbox/Thesis/sv-benchmarks/c/loops/	status	cputime (s)	walltime (s)	memUsage
total tasks	10	1710	1530	4397989888
local summary	-	452	1710	-
correct results	4	65.6	65.7	1387659264
correct true	3	40.7	40.8	911577088
correct false	1	24.8	24.9	476082176
incorrect results	4	465	466	1866235904
incorrect true	4	465	466	1866235904
incorrect false	0	-	-	-
score (10 tasks, max score: 14)	**-121**	-	-	-
Run set	skink17			

C.4 Concurrent Skink SV-COMP Loops sum Benchmark Results

Tool	skink			
Limits	timelimit: 500 s, memlimit: 2000 MB, CPU core limit: 1			
Host	matt-VirtualBox			
OS	Linux 4.4.0-45-generic x86_64			
System	CPU: Intel Core i5-2500K CPU @ 3.30GHz, cores: 3, frequency: 3300 MHz; RAM: 4144 MB			
Date of execution	2016-11-07 13:48:19 AEDT			
Run set	skink17			
.../sv-benchmarks/c/loops/	status	cputime (s)	walltime (s)	memUsage
sum01_bug02_false-unreach-call_true-termination.i	false(reach)	88.5	88.7	498728960
sum01_bug02_sum01_bug02_base.case_false-unreach-call_true-termination.i	**true**	24.2	24.3	484085760
sum01_false-unreach-call_true-termination.i	false(reach)	125	125	520093696
sum03_false-unreach-call_true-termination.i	timeout	617	531	674978840
sum04_false-unreach-call_true-termination.i	**true**	23.4	23.5	474062848
sum_array_false-unreach-call.i	false(reach)	11.8	11.9	363180032
sum01_true-unreach-call_true-termination.i	true	10.5	10.5	317698048
sum03_true-unreach-call_false-termination.i	true	26.5	26.6	481972224
sum04_true-unreach-call_true-termination.i	true	8.30	8.32	280694784
sum_array_true-unreach-call.i	true	176	176	508100608
.../sv-benchmarks/c/loops/	status	cputime (s)	walltime (s)	memUsage
total tasks	10	1110	1030	4603596800
local summary	-	449	1180	-
correct results	7	447	447	2970468352
correct true	4	221	221	1588465664
correct false	3	226	226	1382002688
incorrect results	2	47.6	47.8	958148608
incorrect true	2	47.6	47.8	958148608
incorrect false	0	-	-	-
score (10 tasks, max score: 14)	**-53**	-	-	-
Run set	skink17			

Appendix D

BenchExec Results For Concurrent Benchmarks

D.1 Concurrent Skink Simple Concurrency Benchmark Results

Tool	skink			
Limits	timelimit: 10000 s, memlimit: 500 MB, CPU core limit: 1			
Host	matt-VirtualBox			
OS	Linux 4.4.0-45-generic x86_64			
System	CPU: Intel Core i5-2500K CPU @ 3.30GHz, cores: 3, frequency: 3300 MHz; RAM: 4144 MB			
Date of execution	2016-11-07 13:40:47 AEDT			
Run set	skink17			
programs/concurrency/	status	cputime (s)	walltime (s)	memUsage
concurrent-loop_false-unreach-call.c	false(reach)	5.34	5.37	224477184
fib-threads_false-unreach-call.c	false(reach)	4.73	4.75	203280384
simple-threads_false-unreach-call.c	false(reach)	4.67	4.70	176902144
sync-threads_false-unreach-call.c	false(reach)	4.72	4.74	209907712
concurrent-loop_true-unreach-call.c	true	5583	5584	507232256
fib-threads_true-unreach-call.c	true	3842	3842	495529984
simple-threads_true-unreach-call.c	true	2249	2251	505413632
sync-fib-threads_true-unreach-call.c	true	27.2	27.2	488062976
sync-threads_true-unreach-call.c	true	18.9	19.0	413597696
synch-concurrent-loop_true-unreach-call.c	true	6.25	6.28	250421248
programs/concurrency/	status	cputime (s)	walltime (s)	memUsage
total tasks	10	11746	11748	3474825216
local summary	–	11746	11748	–
correct results	10	11746	11748	3474825216
correct true	6	11726	11726	2660257792
correct false	4	19.5	19.6	814567424
incorrect results	0	–	–	–
incorrect true	0	–	–	–
incorrect false	0	–	–	–
score (10 tasks, max score: 16)	**16**	–	–	–
Run set	skink17			

D.2 Concurrent Skink SV-COMP Concurrency pthread Benchmark Results

Tool	skink			
Limits	timelimit: 500 s, memlimit: 500 MB, CPU core limit: 1			
Host	matt-VirtualBox			
OS	Linux 4.4.0-45-generic x86_64			
System	CPU: Intel Core i5-2500K CPU @ 3.30GHz, cores: 1, frequency: 3300 MHz; RAM: 4144 MB			
Date of execution	2016-11-06 15:54:57 AEDT			
Run set	skink17			
../sv-benchmarks/c/pthread/	status	cputime (s)	walltime (s)	memUsage
bigshot_p_false-unreach-call.i	unknown	4.51	4.54	147890176
fib_bench_false-unreach-call.i	timeout	500	500	493389440
fib_bench_longer_false-unreach-call.i	timeout	500	500	491480704
fib_bench_longest_false-unreach-call.i	timeout	500	500	499738240
lazy01_false-unreach-call.i	false(reach)	10.6	10.6	167981056
queue_false-unreach-call.i	unknown	10.7	10.7	234676224
queue_longer_false-unreach-call.i	unknown	10.8	10.8	253280256
queue_longest_false-unreach-call.i	unknown	10.8	10.8	250281984
reorder_2_false-unreach-call.i	unknown	8.57	8.60	199667712
reorder_5_false-unreach-call.i	unknown	10.0	10.1	201895936
sigma_false-unreach-call.i	unknown	5.34	5.37	154046464
singleton_false-unreach-call.i	unknown	5.35	5.37	152203264
stack_false-unreach-call.i	unknown	7.00	7.03	192663552
stack_longer_false-unreach-call.i	unknown	7.11	7.15	188411904
stack_longest_false-unreach-call.i	unknown	7.69	7.70	193081344
stateful01_false-unreach-call.i	false(reach)	5.92	5.97	157835264
twostage_3_false-unreach-call.i	unknown	20.4	20.5	454930432
bigshot_s2_true-unreach-call.i	unknown	4.17	4.19	146112512
bigshot_s_true-unreach-call.i	unknown	4.28	4.30	496923520
fib_bench_longer_true-unreach-call.i	timeout	500	500	489234432
fib_bench_longest_true-unreach-call.i	timeout	500	500	492912640
fib_bench_true-unreach-call.i	timeout	500	500	484839424
indexer_true-unreach-call.i	timeout	500	500	495891456
queue_ok_longer_true-unreach-call.i	unknown	8.78	8.80	224038912
queue_ok_longest_true-unreach-call.i	unknown	8.88	8.94	219242496
queue_ok_true-unreach-call.i	unknown	9.05	9.16	223760384
stack_longer_true-unreach-call.i	unknown	6.78	6.85	158654464
stack_longest_true-unreach-call.i	unknown	6.96	7.00	190472192
stack_true-unreach-call.i	unknown	7.18	7.22	189968384
stateful01_true-unreach-call.i	true	131	132	499998720
sync01_true-unreach-call.i	true	403	406	499998720
../sv-benchmarks/c/pthread/	status	cputime (s)	walltime (s)	memUsage
total tasks	31	4214	4194	6865502208
local summary	–	3914	4310	–
correct results	4	551	555	1325813760
correct true	2	535	538	999997440
correct false	2	16.5	16.6	325816320
incorrect results	0	–	–	–
incorrect true	0	–	–	–
incorrect false	0	–	–	–
score (31 tasks, max score: 45)	**6**	–	–	–
Run set	skink17			