VRIJE
UNIVERSITEIT
BRUSSEL

Master thesis submitted in partial fulfilment of the requirements for the degree of
Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

USING INTERVAL ANALYSIS TO IMPROVE THE VERIFICATION OF CONCURRENT PROGRAMS

Taoufik Cherroud

2024-2025

Promotor: Prof. Dr. Antonio Paolillo
Advisor: Dr. Hernán Ponce de León
Science and Bio-Engineering Sciences

VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van de graad van Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

GEBRUIK VAN INTERVAL ANALYSE OM DE VERIFICATIE VAN GELIJKTIJDIGE PROGRAMMA'S TE VERBETEREN

Taoufik Cherroud

2024-2025

Promotor: Prof. Dr. Antonio Paolillo
Advisor: Dr. Hernán Ponce de León

Wetenschappen en Bio-ingenieurswetenschappen

Abstract

Bounded model checking is a successful technique used to verify whether a program does not violate a specific property. Despite its success, it suffers from the state-space explosion problem. In the case of concurrent programs, this problem is further exacerbated due to an exponential number of thread interleavings. Additionally, the behaviour of concurrent programs depends on a memory model, which needs to be taken into account during verification. For sequential programs, interval analysis has seen prior success in alleviating the state-space explosion. However, to the best of our knowledge, it has not been used yet in the verification of concurrent programs. In this thesis, we introduce a memory-model-aware global interval analysis to Dartagnan, a bounded model checker, which takes into account thread interferences on shared memory addresses. This allows for more precise intervals to be derived, which can be used to optimise the encoding to improve verification time. We evaluated our work using different encoding and solving methods. This was done on 725 concurrent programs from the *ConcurrencySafety* benchmark set provided by the Software Verification Competition (SV-COMP). The evaluation shows that using our work verifies 1% more problems and shows a 1.7-7.5% decrease in terms of average time per task for all encoding and solving methods. Furthermore, we determined the computational overhead of the global analysis by benchmarking lock and lock-free data structures, where we found a linear relationship between the number of register-write events and overhead. Our results show that global interval analysis helps derive more bounds than a local analysis, which considers each thread as a sequential program. We expect this work to serve as a baseline for other tools to use a global interval analysis that considers memory models.

Abstract

Bounded model checking is een succesvolle techniek om te verifiëren dat een programma geen specifieke eigenschap schendt. Ondanks het succes, lijdt het aan het "state-space explosion" probleem. In het geval van gelijktijdige programma's wordt dit probleem verder verergerd door het exponentiële aantal "thread interleavings". Bovendien hangt het gedrag van gelijktijdige programma's af van geheugen modellen, waarmee rekening moet gehouden worden tijdens verificatie. Voor sequentiële programma's is interval analyse succesvol toegepast om te helpen met het "state-space explosion" probleem. Echter, voor zover onze kennis strekt, is het nog niet toegepast in de verificatie van gelijktijdige programma's. In deze thesis, stellen wij een geheugen-model-bewuste globale interval analyse voor in DARTAGNAN, een bounded model checker, dat thread-interferenties op gedeelde geheugen adressen in acht neemt. Dit maakt het mogelijk om preciezere intervallen af te leiden, die gebruikt worden om de encodering verder te optimaliseren en zo de verificatietijd te verkorten. We hebben ons werk geëvalueerd voor verschillende encodering- en oplossingsmethoden. Dit werd uitgevoerd op 725 gelijktijdigheidsproblemen uit de *ConcurrencySafety* benchmarkset van de software verificatie wedstrijd (SV-COMP). De evaluatie laat zien dat ons werk 1% meer problemen verifieert en toont een afname van 1.7-7.5% in termen van gemiddelde tijd per taak voor bepaalde encodering- en oplossingsmethoden. Verder hebben we de computationele overhead bepaald door lock- en lockvrije datastructuren te benchmarken, waarbij we een lineair verband vonden tussen het aantal register-write-events en de overhead. Onze resultaten tonen aan dat met een globale interval analyse meer preciezere intervallen afgeleid kunnen worden dan een lokale analyse, waarbij elke thread als een sequentieel programma wordt beschouwd. We verwachten dat dit werk als een uitgangspunt gebruikt kan worden voor andere tools om een globale aanpak te nemen tot interval analyse, dat geheugen modellen in acht neemt.

Acknowledgments

First, I would like to thank my promotor, Prof. Dr. Antonio Paolillo, for giving me the chance to work on this thesis and guiding me throughout the year with regular meetings. Second, I want to express my gratitude to my advisor, Dr. Hernán Ponce de León at Huawei Dresden Research Centre, for his endless knowledge in DARTAGNAN and interesting discussions to further improve this work. Finally, a massive thanks to my family and friends who supported me throughout this period.

Contents

Chapter 1

Introduction

Formal verification of concurrent programs is important due to hard-to-spot bugs such as potential data races, deadlocks, and safety violations. This problem is further exacerbated by memory models that define different multiprocessor behaviour depending on the architecture. Avoiding such bugs is usually done using synchronisation primitives such as locks. Two problems arise when using locks. First, the synchronisation primitives themselves may be buggy. Paolillo et al. [41] showed that the Linux `qspinlock` is incorrect under certain memory models using a verification tool called Dartagnan. Second, using locks can lead to bugs such as deadlocks, where two threads wait on each other to release some lock they need. This causes no progress to be made.

Bounded Model Checking (BMC) [8] is a technique for verifying that a program satisfies a property, such as deadlock-freedom. It takes three inputs: a program to verify, a property of interest, and an unrolling bound that specifies how many times the loops in the program must be unrolled. A BMC tool unrolls a program's loops up to the given bound and encodes it into a logical formula that represents all possible executions. Furthermore, for concurrent programs, all possible interleavings need to be considered and encoded. This logical formula is passed to an oracle, which reports if the program satisfies the property. Additionally, one must consider a memory model and filter out executions that are not valid under said model. One such BMC tool was mentioned earlier, namely Dartagnan.

Dartagnan [45] is an open-source BMC tool mainly maintained by Hernán Ponce de León. Compared to other tools that only support certain memory models [53, 30], Dartagnan takes a memory model as part of its input. This is then encoded as part of the logical formula, which makes Dartagnan highly modular. Despite the increase in encoding size, Dartagnan is scalable and competitive[1]. Part of this success is due to a so-called relation analysis, which helps reduce the overall encoding size and makes it more manageable for the oracle.

A limitation in BMC is the state-space-explosion problem. Where the number of possible states is exponential to the number of variables. This is a problem when exploring higher unrolling bounds. For concurrent programs, the number of thread interleavings is exponential in the number of statements present in a program [14]. Additionally, their behaviour depends on the memory models used by the underlying architecture. Despite this, many efficient tools exist to deal with these limitations, which we mentioned earlier. Another problem is that oracles are *black boxes*, we have little control over *how* they determine the satisfiability of a formula.

One way tools address these limitations is by using static analysis, similar to how a compiler uses static analysis to optimise a program's execution time. Static analysis determines informa-

[1] Dartagnan has won multiple gold medals from SV-COMP [5].

tion about a program without running it. This information can be used for optimisations such as reducing the program size, which results in fewer states. One such analysis is interval analysis [12]. This determines the upper and lower bounds for each program variable.

In previous attempts [37, 57] interval analysis has been successfully applied to alleviate the limitations of bounded model checking. To the extent of our knowledge, these attempts were concerned with the analysis and verification of sequential programs. One can apply the same technique if every thread is considered isolated and ignores communication with other threads via shared memory. However, ignoring shared memory may cause many trivial bounds. Prior work from Kusano and Wang [31] argues that considering memory models helps with precision. We make use of DARTAGNAN's relation analysis [22] to reduce the amount of thread interference possible when analysing events that depend on shared memory.

We propose a new global memory-model-aware interval analysis that considers shared memory to derive more bounds. It uses DARTAGNAN's relation analysis to increase the precision of the bounds that we derive. We build on existing research [37, 57] on interval analysis used in bounded model checking and extend it to concurrent programs. DARTAGNAN encodes its variables using bit vectors of a fixed width. Larger bit widths lead to larger encodings due to a procedure called *bit-blasting* [29]. With bounds, we can reduce the encoding size.

We aim to answer the following research question:

Can interval analysis be used to improve the verification time of concurrent programs?

We answer this question by evaluating our work using different encoding and solving methods. This was done on 725 concurrency problems from the *ConcurrencySafety* benchmark set provided by the software verification competition [51]. Our results show a 1% increase in the number of verification tasks solved as well as a 1.7-7.5% decrease in average time per task. Furthermore, we evaluated the computational overhead using programs that implement lock and lock-free data structures. In general, we observe a $< 3\%$ computational overhead and a 7% overhead in the worst case. Additionally, we note that the analysis time has a linear relationship with the number of register-write events. Our results show that we can derive more bounds with global interval analysis as opposed to using a thread-local analysis.

To summarise, we make the following contributions:

- We introduce interval analyses to DARTAGNAN that differ in precision. This includes a naive analysis, a local analysis, and a memory-model-aware global analysis.
- We use the bounds obtained from the interval analysis to add additional constraints to the encoding that improve verification time.
- We evaluated our work against known concurrency benchmark sets, including 725 SV-COMP verification tasks, lock and lock-free data structures.

The remainder of this thesis is structured as follows. Chapter 2 provides background information about DARTAGNAN, software verification, static analysis, and memory models. Subsequently, Chapter 3 details how we implement interval analysis in DARTAGNAN and use the results to optimise the encoding. Furthermore, Chapter 4 evaluates our work with respect to performance and computational overhead. After that, Chapter 5 discusses related work. Finally, Chapter 6 provides a conclusion and an outline for future work.

Chapter 2

Background

In this chapter, we give an overview of the relevant background needed to understand the ideas presented in this thesis. First, we cover how DARTAGNAN represents programs in Section 2.1. Second, we discuss satisfiability and bounded model checking in Section 2.2. Third, we give an overview of static analysis and the monotone framework in Section 2.3. Fourth, we discuss the relevance of memory models to this thesis in Section 2.4. Finally, we reflect on this chapter and discuss its relevance for Chapter 3 in Section 2.5.

2.1 Programs

In this section, we will discuss the programs we are analysing. First, we explain how DARTAGNAN processes an input program into the *core* intermediate representation. Then, we give an overview of the core intermediate representation, where we will discuss *events* and *expressions*. Finally, an example is explored for clarity.

2.1.1 Executions

The semantics of the input program are represented by a set of executions. An execution is a graph where the nodes represent executed events and edges represent relations between events. An event models different types of program instruction. First, memory events represent instructions that read and write to shared memory. Second, local events model assignments of local computations to local variables. Finally, branching events represent (un)conditional jumps. Each thread has a set of executed events. The *read-from* relation (`rf`) specifies from which stores a load can read, while the *coherence* (`co`) relation represents the order in which writes to a specific address hit shared memory. Furthermore, the `po` relation represents the *program order* within a thread [43, 22, 20]. These relations will be discussed again in Section 2.4.

2.1.2 Core Events

For this thesis, we are interested in analysing an Intermediate Representation (IR) of a program consisting of a specific subset of events. These events are called *core* events. They consist of memory, local, and branching events. Below, we give a grammar of the core events that we consider.

$\langle program \rangle \quad ::= \langle thread \rangle^+$

⟨*thread*⟩ ::= `thread` ⟨*event*⟩$^+$

⟨*event*⟩ ::= ⟨*local*⟩
| ⟨*memevent*⟩
| ⟨*label*⟩
| ⟨*condjump*⟩

⟨*local*⟩ ::= ⟨*reg*⟩ ← ⟨*expr*⟩

⟨*memevent*⟩ ::= ⟨*reg*⟩ = `load`(⟨*loc*⟩)
| `store`(⟨*loc*⟩,⟨*expr*⟩)

⟨*condjump*⟩ ::= `if` ⟨*pred*⟩ `then goto` ⟨*label*⟩
| `goto` ⟨*label*⟩

A program consists of one or more threads, which consist of one or more core events. Registers are thread-local variables, and memory locations are shared variables between threads.

Events

We can divide events into two general categories when considering their use of registers. An event is a register-writer if it writes the value of an expression into a register. Similarly, an event is a register-reader if it reads the value of a register.

Expressions

To determine integer bounds on registers, we must consider the various expressions that can occur in a program. DARTAGNAN supports expressions of different types such as the integer type, the Boolean type and aggregate type. We are interested in finding *integer* bounds for registers and will therefore only consider expressions of that type.

Below, we give a grammar of the expression that we consider.

⟨*expr*⟩ ::= ⟨*reg*⟩ | ⟨*loc*⟩ | ⟨*int*⟩ | ⟨*ite*⟩

⟨*int*⟩ ::= ⟨*lit*⟩
| ⟨*unary*⟩
| ⟨*binary*⟩
| ⟨*cast*⟩

⟨*unary*⟩ ::= ⟨*uop*⟩ ⟨*expr*⟩

⟨*binary*⟩ ::= ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩

⟨*cast*⟩ ::= ⟨*castop*⟩ expr ⟨*targetsize*⟩

⟨*uop*⟩ ::= - | ctlz | cttz

⟨*binop*⟩ ::= ⟨*arithmetic-binop*⟩
| ⟨*logical-binop*⟩
| ⟨*shift-operator*⟩

⟨*arithmetic-binop*⟩ ::= + | - | * | / | udiv | urem | srem

⟨*logical-binop*⟩ ::= | | & | ˆ

⟨*shift-binop*⟩ ::= >> | <<

⟨*castop*⟩ ::= trunc | sext

⟨*ite*⟩ ::= ⟨*pred*⟩ ? ⟨*expr*⟩ : ⟨*expr*⟩

Expressions return values that are stored in registers or memory locations. The simplest forms of integer expressions are integer literals and integer registers. Integer literals and registers are annotated with a type. For example, registers that are annotated with `bv8` can only hold values from the interval $[-128, 255]$ (registers could be signed or unsigned).

Unary operators are used to alter one expression. Binary operators combine two expressions to form compound expressions. DARTAGNAN supports arithmetic, logical and shift operators. The type of an integer expression can be cast to another type using casting operators. Registers can also contain addresses of memory locations, which are called pointers.

2.1.3 Preprocessing

One of DARTAGNAN's inputs is a program written in a supported language[1]. At this point, threads consist of many source language-specific events and intrinsic function calls. DARTAGNAN performs a preprocessing phase, where a program undergoes various passes. The goal of this phase is to transform and reduce the set of events such that they consist of only core events. We discuss some of the performed passes. First, function inlining replaces function call events with the events in the body of the called function. This is helpful since any analysis performed is now interprocedural, considering all called functions. Second, loop unrolling is a pass that duplicates loop bodies up to a certain bound. It is an essential step in bounded model checking and is further discussed in Section 2.2.3 and Section 3.1. Third, certain passes optimise the program. Constant propagation is a transformation that tries to discover values that are constant on all possible executions of a program. These constant values then get propagated to expressions that may be simplified to constants if all the operands are constant as well. One use for constant propagation is to simplify the code by eliminating assignments and removing unreachable conditional branches. DARTAGNAN uses a variant of constant propagation called Sparse Conditional Constant Propagation [54], which has a faster runtime and finds more constants. Finally, DARTAGNAN performs a pass to verify if all events present are the *core* events discussed above.

Example 1. Listing 1 shows a simple C program consisting of a for loop and a property that we want to verify. The loop updates a global variable n up to a bound N and then asserts that $n \geq 0$. The C program is compiled to LLVM IR [34] and then parsed and processed until it is in the core IR form shown in Listing 2. Lines 2 to 8 are the unrolled loop body, which uses `store` events to update the global variable n. The rest of the IR handles the property that we want to verify using conditional jumps. Note that the `assert` event on line 20 is also a core event, but is omitted from the previously introduced grammar as it is not relevant to the analysis.

2.2 Software Verification

In this section, we discuss bounded model checking. In Section 2.2.1 and Section 2.2.2, we review the backend of bounded model checking and common performance problems associated with it. Section 2.2.3 gives an overview of the whole process and discusses problems in the technique that this thesis addresses.

[1] This includes but is not limited to C, LLVM, SPIRV, and Litmus

```c
#include <assert.h>
#define N 2
int n = 0;
int main()
{
 for(int i = 0; i < N; i++) {
   n = i;
 }
 assert(n>=0);
}
```

Listing 1: Example of a C program. It is a loop that assigns a value to a global variable. It is annotated with an assertion that $n \geq 0$. The assertion is the property that we want to verify.

2.2.1 Satisfiability

The Boolean satisfiability problem (SAT) is the problem of finding a satisfying assignment given a propositional formula. This is an NP-complete problem and an active area of research [49]. The formula is typically in Conjunctive Normal Form (CNF), which is a conjunction of clauses. Each clause is a disjunction of literals, where a literal is a variable or a negated variable. An assignment is a map from variables to truth values. A clause is *satisfied* if there is an assignment that makes the clause true. Otherwise, it is *unsatisfied* if it is false or *unresolved* if it is neither. Clauses where all other literals are false except for one are considered *unit*. Unit clauses are used for an optimisation technique. A formula is *satisfiable* if there is an assignment that satisfies every clause, called a *model*. If a formula has no model, then it is unsatisfiable.

Example 2. Consider the following formula: $x \wedge (\neg x \vee y)$. It consists of two clauses: the unit clause x and $(\neg x \vee y)$. The formula has one model where $x = 1$ and $y = 1$.

Solvers are sophisticated programs that determine the satisfiability of a formula. One of the earliest successful algorithms is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [16, 15]. The algorithm searches for a satisfying assignment by assigning truth values to variables. If the partial assignment makes the formula unsatisfiable, the algorithm backtracks *chronologically* and tries to flip the value of the variable. When we encounter an already flipped variable, we undo the assignment and try to flip the assignment before it. Otherwise, the solver keeps making assignments until it finds a satisfying assignment. DPLL introduced several optimisations, such as unit propagation and pure literal elimination. Conflict-Driven Clause Learning (CDCL) is an extension of DPLL, which was first introduced with GRASP [35]. Currently, CDCL is the main method of solving satisfiability problems [7, 19, 50] and has been enhanced with many techniques [6]. These solvers compete with each other in the SAT competition [49]. The competition evaluates solvers on a set of benchmarks to see which solver gets the highest score. Its main objective is to foster progress in the performance of SAT solvers and motivate implementers to share their work. Participants are also required to provide new benchmarks, which contribute to a growing data set of SAT instances. Later in this chapter, we will cover SMT and software verification, which also have their respective competitions. One application of SAT solvers is as *oracles* embedded in solving different problems, which include SMT and bounded model checking.

```
thread  main()
  loop/itr_1:
     store(n, 0)
  loop/itr_2:
     store(n, 1)
  loop/itr_3:
     goto l1
  loop/bound:
  l1:
     r12 = load(n)
     if(0 <= r12) then goto l2
     goto l3
  l2:
     goto l4
  l3:
     goto l5
  l4:
     goto l6
  l5:
     assert(False)
  l6:
  END_OF_T0:
```

Listing 2: An example of the aforementioned C program that is unrolled up to N. Assignment instructions to global variables are compiled to `store` events. The `assert` statement is compiled to a $\langle condjump \rangle$ event leading to an `assert` event.

2.2.2 Satisfiability Modulo Theories

Another technique for modelling problems is Satisfiability Modulo Theories (SMT). This technique is based on determining the satisfiability of a First-Order Logic (FOL) formula. They do this by determining satisfiability with fixed interpretations for predicates and functions. We introduce the notation and background that will be used in this thesis, which is based on the book by Biere et al. [6].

Syntax and Theories

We briefly overview the syntax and semantics related to SMT. Formulas are constructed over a *signature* Σ, which consists of a set of predicate symbols and function symbols. Figure 2.1 describes a grammar for FOL. First, we have terms t, which can be constants c or functions f. Second, we have formulas which are unary predicates A or n-ary predicates p. Formulas can be negated or combined via logical operators. Semantics are defined with *structures*. Structures consist of a *domain* D and an *interpretation*. An interpretation assigns elements of Σ to elements of D. A structure satisfies a formula if the formula under the interpretation is true. Otherwise, it falsifies the formula. A theory $\mathcal{T}$ constrains an interpretation to a fixed interpretation of functions and predicates.

$$
\begin{aligned}
\langle t \rangle \quad &::= c \\
&\mid f(t_1, ..., t_n) \\
\langle \varphi \rangle \quad &::= A \\
&\mid p(t_1, ..., t_n) \\
&\mid t_1 = t_2 \mid \top \mid \bot \mid \neg\varphi_1 \\
&\mid \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2 \\
&\mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2
\end{aligned}
$$

Figure 2.1: Syntax for First Order Logic based on the book by Biere et al. [6].

$$
\begin{aligned}
\langle t \rangle \quad &::= t\, op\, t \mid \sim t \mid extend(t) \mid extract(t) \\
\langle \varphi \rangle \quad &::= t\, rel\, t \\
\langle rel \rangle \quad &::= = \mid < \\
\langle op \rangle \quad &::= + \mid - \mid \cdot \mid / \mid \ll \\
&\mid \gg \mid \lll \mid \ggg \mid \& \mid \mid \mid \oplus \mid \circ
\end{aligned}
$$

Figure 2.2: Syntax for the theory of bit vectors. Includes traditional arithmetic, relational and bitwise operators. Some operators, such as $\leq$, differ if the bit vectors are considered signed or not. This difference is indicated by a subscript S or U for signed and unsigned operators, respectively. Figure based on the book by Kroening et al. [29].

Solving

There are two main approaches for solving instances of the SMT problem. First, there is the eager approach. In this approach, an input formula is reduced to a propositional formula. The formula is then passed to a SAT solver, which decides it. A problem is that the translation may cause a significant increase in size, which may take an unreasonable amount of time to solve. Second, we have the lazy approach. This approach consists of two parts: a SAT solver and a specialised theory solver. The main idea is to use the SAT solver to decide the Boolean structure of the FOL formula and to use the theory solver to decide theory-specific parts. This approach has the advantage of using specialised decision procedures for solving predicates and functions for a specific theory.

A decision procedure for the lazy approach is DPLL($\mathcal{T}$) [21] where $\mathcal{T}$ is an abstract theory. This suggests that there is a main decision procedure, where we can provide a theory of choice as an argument. Many state-of-the-art solvers use the lazy approach, such as Microsoft's `z3` [39] and Yices2 [18].

Bit Vectors

The theory of fixed-width bit vectors allows us to represent values as bit vectors, similarly to how computers represent values internally. Additionally, the semantics of bit vector operations closely resemble how a computer handles them. This means that we can naturally account for behaviours such as arithmetic *overflows*.

We extend the syntax of Figure 2.1 to Figure 2.2. First, the syntax now contains equalities and inequalities for bit vectors. Furthermore, operators for bit vectors are included, such as arithmetic operators, bitwise operators, arithmetic shift operators, logical shift operators, extension operators, and extract operators.

To decide bit vector formulas, it is first converted to an equisatisfiable propositional formula[2] and then passed to a SAT solver. This is called *flattening* [29] or *bit-blasting*. Bit-blasting happens in three major steps:

1. The Boolean structure is created by introducing a new propositional variable for each subexpression in the formula. These new variables are then used to create a new equisatisfiable formula. This encoding is known as the Tseitin Encoding [55].
2. Each bit vector of width l is represented by l propositional variables. These propositional variables are subscripted to correspond to a bit in the bit vector. For example, b_0 is a propositional variable that represents the least significant bit of the bit vector b.
3. Every bit vector constraint is converted into a propositional formula that encodes its semantics. We refer to Kroening et al. [29] for an overview of different encodings.

Example 3. We elaborate on this process with a simple example. Consider the following bit vector formula:

$$(x = 0)$$

We introduce fresh variables for each subformula. The Boolean structure looks as follows:

$$a \wedge a \leftrightarrow x = 0$$

For this example, we assume a bit width of one, so we require only one propositional variable per bit vector. For the bit vector variable x we introduce the propositional variable x_0. The constant bit vector 0 is represented by the propositional variable c_0.

Now, to encode the equality $x = 0$, we use the logical equivalence operator. In the end, we have the following formula

$$\neg c_0 \wedge a \wedge a \leftrightarrow (x_0 \leftrightarrow c_0)$$

The formula would grow to 64 equivalence checks for 64-bit bit vectors.

Bit-blasting does not scale well for bit vectors with large bit widths. Kovasznai et al. [28] discuss the complexity of bit-blasting and how it can lead to an exponential explosion in formula size when bit widths increase.

2.2.3 Bounded Model Checking

In software verification, we want to check if a program adheres to a specification. This specification is called a *property*, which is expressed as a temporal logic formula[3]. There are two types of properties that are considered. Formal definitions of these property types are given by Alpern et al. [3], however, in much literature [14, 9], intuitive definitions are provided. First, a safety property states that "something bad will not happen". An example would be that we never reach a state where two threads are in a deadlock. Second, a liveness property specifies that "something good must happen eventually", such as program termination. If a property is violated, a *counterexample* is returned by the verifier. A counterexample is a trace of a program's execution that leads to a state that violates the property.

[2] A formula that preserves (un)satisfiability.

[3] Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) are examples of temporal logic that we found in the literature.

Bounded Model Checking (BMC) is a verification technique introduced by Biere et al. [8]. Previous techniques, such as BDD-based symbolic model checking, suffered from problems like state-space explosion [10]. BMC alleviates this problem by only considering programs up to a bound k and encoding a verification problem into a SAT or SMT instance. This makes BMC competent in detecting bugs and providing short counterexamples.

Deciding a Verification Task

Figure 2.3 shows an overview of the steps involved in DARTAGNAN to reduce a verification problem to an SMT problem. We will be using DARTAGNAN as an example. DARTAGNAN takes as input a program P, a bound k, a safety property S, and a memory model specification MM. After the program is unrolled up to k, the unrolled program P_U, S and MM are encoded into an SMT formula $\phi_{P_U} \wedge \phi_S \wedge \phi_{MM}$. A program is *fully unrolled* if all the loops in a program can no longer be unrolled further by higher k values. In other words, we unrolled each loop up to its maximum number of possible iterations. First, ϕ_{P_U} encodes all possible executions of P_U. Second, ϕ_{MM} makes sure that only executions that are consistent with MM are considered. Finally, ϕ_S is satisfied only if an execution violates S. Overall, this formula is satisfiable if there is an execution that violates the given property and the verifier returns a FAIL. Otherwise, if the formula is unsatisfiable and the program is fully unrolled, then the verifier returns a PASS. When the program is not fully unrolled and is unsatisfiable, then there could be a bug with unrolling bounds, and the verifier returns an UNKNOWN.

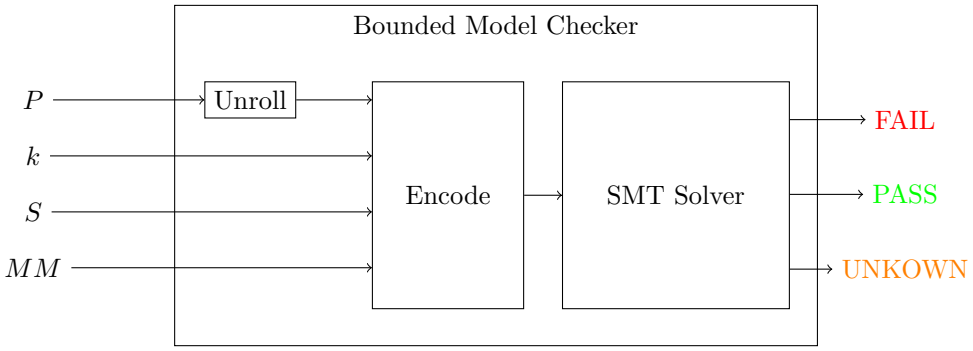


Figure 2.3: Displays the process a bounded model checker goes through to handle a verification task. Boxes indicate processes and arrows indicate input/output. P is the program under verification. k specifies how many times loops are unrolled. S is the property that we want to verify. We include the memory model MM as input to more closely resemble DARTAGNAN. We emphasise that this is a simplified representation of a BMC architecture.

Benefits and Drawbacks

We mentioned earlier that the main benefit of BMC is its ability to quickly find bugs at low unrolling bounds. Another benefit is that it does not require exponential space [9]. Moreover, satisfiability solvers also do not require much human interaction as opposed to previous techniques.

A drawback of BMC is that it can return UNKNOWN for a given verification instance. This means that it is generally incomplete, as we can increase the unrolling bound indefinitely. Another is that the encoding size increases as the bound increases, so a solver may take more

time with larger unrolling bounds. The increase in encoding size is exacerbated by the fact that we are verifying concurrent programs, as we need to account for all the possible interleavings. Lastly, a solver is a black box, which means that we do not have much control over *how* a solver decides a given formula. However, it is possible to optimise the encoding via techniques such as static analysis.

2.3 Static Analysis

In this section, we discuss static analysis. The rest of this section is structured as follows. First, Section 2.3.1 gives an overview of static analysis and reviews its applications. Then, Section 2.3.2 introduces the *monotone framework* [27] for doing static data flow analysis. Finally, Section 2.3.3 briefly discusses alias analysis, which is used to support other analyses.

2.3.1 Motivation and Applications

Static analysis refers to the process of automatically computing information about a program. In contrast to dynamic analysis, we do not execute the program to obtain information. Compilers [23] use this information to optimise programs to produce optimised output. Choosing which analyses are performed depends on the optimisations that are wanted. Another use for static analysis is verification with techniques such as Abstract Interpretation [11]. One can verify properties such as no out-of-bounds accesses when indexing an array.

Similar to compilers, verification tools use static information to enhance the process and speed up verification times. For example, we mentioned earlier that DARTAGNAN preprocesses the program before encoding it into an SMT formula. This includes optimisations such as constant propagation, which requires an analysis to perform. The optimisations in the preprocessing step result in an enhanced encoding, which has faster solving times. These analyses are common in other verifiers [37, 30, 53]. Verifiers not only use analyses for performance enhancement but also for completeness. One can attempt to approximate the maximal times a loop can be executed, which is used to set the unwinding bound in BMC.

We mentioned approximation because interesting properties about a program are undecidable according to Rice's theorem [47]. Despite undecidability, static analysis has seen much success because we do not require the exact answer for every single program. Approximations can be enough for optimisations. For example, although we cannot always know the exact value of a variable at every program point, an interval analysis can be used to determine the range of values of a variable. This range can be used for a number of optimisations. Another example from compilers is live variable analysis. This analysis is important in improving register allocation, optimally using each register to store program variables, which is crucial to performance.

As analysis results are used for some subsequent goal, such as optimisation or verification, the soundness of the results is important. Indeed, unsound results may result in unwarranted optimisations. As a result, analyses are typically conservative as they give over-approximations or under-approximations about certain facts.

An important trade-off is between precision and performance. We want as precise information as possible, yet we also want reasonable performance. Examples include analyses used for assistance in software development. In the process of software development, analysis is used to warn about potential bugs or software anti-patterns [25]. An imprecise but faster analysis may yield many false positives[4], which resulted in less usage [14]. Whereas a precise but slower analysis may take too long to give useful answers.

[4] A reported bug that cannot happen due to the analysis being conservative.

2.3.2 Monotone Framework

In this section, we introduce the monotone framework for data flow analysis. This framework involves reasoning over information that is propagated through a program. Information is represented as abstract values defined by a lattice. With the abstract values, we establish equality constraints that define how the abstract values should *flow* through a program. These constraints are then solved using a fixpoint computation. The result is a solution where all constraints are satisfied. Furthermore, we introduce the role of transfer functions in data flow analysis.

This framework was introduced by Kam and Ullman [27], where they showed that monotonicity is a sufficient condition for data flow analyses in practice. For this thesis, we base our notation and definitions on the work by Møller and Schwartzbach [38].

First, we review control flow graphs. Second, we introduce lattices and abstract values. Third, a system of equations is defined, which is solved using a fixed point calculation. Finally, we introduce transfer functions and an algorithm to perform data flow analysis.

Control Flow Graphs

In this thesis, we are interested in *flow-sensitive* data flow analyses. In other words, the order in which program statements occur is relevant during design. This order is called the *control flow* of a program. The control flow of a program can be represented by a *control flow graph* (CFG). We use a formal definition from Kam and Ullman [27], which is adapted to our purposes.

Definition 1. A control flow graph is a four-tuple $G = (V, E, n_{entry}, n_{exit})$ where:

1. V is a finite set of *nodes*.
2. $E \subseteq V \times V$ called the *edges*. An edge (x, y) *enters* node y and *leaves* node x. x is the *predecessor* of y, and y is the *successor*. $pred(v)$ denotes the set of predecessors of v. Similarly, $succ(v)$ denotes the set of successors.
3. n_{entry} is the *entry* node.
4. n_{exit} is the *exit* node.

CFGs can be traversed *forwards* or *backwards*. Analyses that depend on future information in a program typically use a backwards approach, while programs that rely on older information use a forward approach. Both approaches can be applied to increase precision [52]. CFGs are commonly represented graphically, such as in Figure 2.4.

Lattices

The monotone framework uses *abstract values* to represent concrete values. Abstract values from a domain S are partially ordered by an ordering operator $\sqsubseteq$, which satisfies the following conditions:

- reflexivity: $\forall x \in S : x \sqsubseteq x$.
- transitivity: $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$.
- anti-symmetry: $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$.

A domain S and a partial ordering $\sqsubseteq$ form a *partially ordered set* or poset $(S, \sqsubseteq)$. Let $x, y \in S$, the *least upper bound* (or *join*) $x \sqcup y$ has the following properties:

```
main:
bv64 r0 <- 0;
bv64 r1 <- load(x);
if(x) then goto l1;
else goto l2;
l1:
r0 <- 5;
goto l3;
l2:
r0 <- 6;
goto l3:
l3:
assert(r0 >= 5);
```

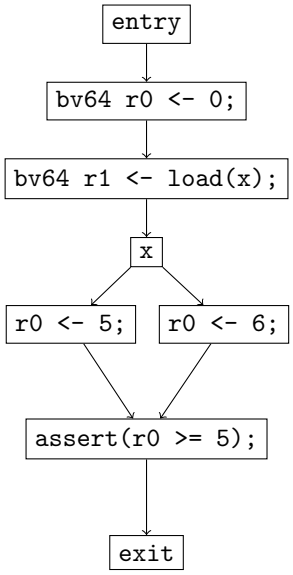


Figure 2.4: A side-to-side comparison with a program in DARTAGNAN IR (left) and its control flow graph (right).

- It is greater than both x and y: $\forall x, y \in S : x \sqsubseteq (x \sqcup y) \wedge y \sqsubseteq (x \sqcup y)$.
- It is smaller than all elements with previous property: $\forall x, y, z \in S : x \sqsubseteq z \wedge y \sqsubseteq z \implies (x \sqcup y) \sqsubseteq z$.

The *greatest lower bound* (or *meet*) operator $x \sqcap y$ is defined analogously. Intuitively, the join and meet operators allow us to combine information from different parts of a program at a control flow node. This combined information is a safe approximation for all paths that go through those specific nodes where paths converge.

A poset $(S, \sqsubseteq)$ extended with a join ($\sqcup$) and meet ($\sqcap$) operator $(S, \sqsubseteq, \sqcup, \sqcap)$ is called a *lattice* if $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in S$. If the join and meet exist for all arbitrary subsets of S, then the lattice is a *complete lattice*. A map lattice is a set of functions $f, g : A \to L$ that is ordered pointwise:

$$f \sqsubseteq g \iff \forall a_i \in A : f(a_i) \sqsubseteq g(a_i) \text{ where } f, g : A \to L$$

Fixed Points

The goal is to compute invariants at each control flow node of a program. We can set up a system of equations:

$$\begin{aligned} x_1 &= f(x_1, x_2, \ldots, x_n) \\ x_2 &= f(x_1, x_2, \ldots, x_n) \\ &\vdots \\ x_n &= f(x_1, x_2, \ldots, x_n) \end{aligned}$$

Where $x_1, x_2, \ldots, x_n$ are constraint variables and $f_1, f_2, \ldots, f_n : L^n \to L$ are constraint functions[5]. A constraint variable corresponds to a CFG node. A solution to the system of equations is an assignment to each constraint variable such that all equations are satisfied. Usually, we are not interested in any solution but the most precise one, which is given by the *least fixed point*.

Definition 2. A *fixed point* for f is a point $x \in L$ if $f(x) = x$. If $x \sqsubseteq y$ for all y that are fixed points for f, then x is the *least fixed point*.

A least fixed point always exists if f is monotone [38]. In practice, we use a transfer function. Programs can be represented by the map lattice $Node \to State$, with $Node$ the set of CFG nodes and $State$ an *abstract state lattice*. Let Var be the set of variables occurring in some program and L a lattice. An example of an abstract state lattice would be the map lattice $Var \to L$, which associates a variable from Var with some abstract value from L. A value from $State$ is called an *abstract state*. The *transfer function* of a node t_v specifies how an abstract state should be transformed when encountering a specific node v.

An algorithm to use the transfer function to propagate information is PropagationWorkList. Initially, it adds every control flow node to a work list W. Then, it iteratively removes a node v_i from W and computes a new value y using t_{v_i} for the abstract state x_i that corresponds with v_i. Afterwards, information is propagated to the abstract states of the nodes v_j that *depend* on v_i. If this information is *new*, then we add v_j to the work list and continue. Finally, the algorithm terminates when W is empty, and the algorithm returns the abstract states. In essence, this algorithm keeps adding new nodes to W while new information is still being computed. The algorithm terminates when there is no new information after applying the transfer functions, in other words, when we have reached a fixed point. Algorithm 1 shows pseudo-code of PropagationWorkList. Informally, the $dep(v)$ function computes the set of all nodes dependent on v.

Algorithm 1 PropagationWorkList algorithm [38]

Input: Nodes $V = \{v_1, \ldots, v_n\}$, Transfer functions $t_1, \ldots, t_n$
Output: An assignment for abstract states $(x_1, \ldots, x_n)$.

```
1:  procedure PropagationWorkList(t_1, ..., t_n)
2:      (x_1, ..., x_n) := (⊥, ..., ⊥)
3:      W := V
4:      while W ≠ ∅ do
5:          v_i := W.removeNext()
6:          y := t_{v_i}(x_i)
7:          for each v_j ∈ dep(v_i) do
8:              z := x_j ⊔ y
9:              if x_j ≠ z then
10:                 x_j := z
11:                 W.add(v_j)
12:             end if
13:         end for
14:     end while
15: return (x_1, ..., x_n)
16: end procedure
```

[5] L^n is called a product lattice where its elements are in $L \times L \times \cdots \times L$

2.3.3 Alias Analysis

Two variables *alias* each other if they both point to the same memory location. Aliases cause non-trivial issues for the data flow analyses, as we can no longer assume that we are reading (writing) from (to) distinct memory locations. To not lose much precision, we must make a coarse approximation of the possible memory location that we can access. This is called Alias Analysis[6], which computes the set of memory locations a variable *may* alias. Alias analysis is used to improve the precision of other analyses [22].

2.4 Memory Models

In this section, we give an overview of memory models. Memory models are essential in reasoning about the behaviour of concurrent programs, as different memory models induce different behaviours. First, in Section 2.4.1, we explain the difference between distinct memory models. Then, we discuss results from the literature relating to the soundness of static analyses under weak memory models in Section 2.4.2. Finally, a technique to optimise DARTAGNAN's encoding, called relation analysis, is covered in Section 2.4.3.

2.4.1 Different Shared-Memory behaviour

The behaviour of concurrent programs depends on a *memory model*. Memory models tell us what the legal set of behaviours is for a specific architecture [26, 46].

Read and write operations to shared memory may be reordered at runtime to increase performance. These reorderings are valid in sequential programs as long as the program's original behaviour remains unaltered. For concurrent programs, the behaviour of other threads that read from shared memory could be affected. We illustrate this with a common example in Figure 2.5.

Init: $x = 0, y = 0$

$T1$	$T2$
(1) `store`$(x, 1)$	(3) `store`$(y, 1)$
(2) $r1$ = `load(y)`	(4) $r2$ = `load(x)`

Observed? $r1 = 0, r2 = 0$

Figure 2.5: Litmus test to showcase potential memory reorderings.

In the example, the execution of two threads $T1$ and $T2$. $T1$ writes the value 1 in x and reads y into $r1$. $T2$ does the same but writes to y and reads x into $r2$. x and y are shared-memory locations while $r1$ and $r2$ are thread-local registers. We assume that x and y are initialised with 0. The question is now whether it is possible for $r1 = 0$ and $r2 = 0$? This answer depends on the memory model of the underlying architecture. If we assume *Sequential Consistency* (SC) [32], then this should not be possible.

Sequential consistency assumes that an execution of a program is an interleaving of program instructions of different processors, where the instructions of the same processor appear in the same order as the program they are executing. This order is called the *program order*. On other architectures, such as x86 [26], instructions from the same processor do not necessarily need to appear in program order in an execution. This memory model is *weaker* compared to sequential consistency. In general, memory models that allow more types of memory reorderings

[6]Some literature calls it "Pointer Analysis" or a "Points-to analysis"

are considered weaker than those that allow less. Table 2.1 shows a small table that compares the type of reorderings an architecture might allow. We consider four kinds of reorderings:

- Store/Store: a store with an older store.
- Load/Load: a load with an older load.
- Load/Store a store with an older load.
- Store/Load a load with an older store.

As shown in Table 2.1, SC does not allow any reordering, as it is the strongest. Total Store Ordering (TSO), the memory model used by x86, is weaker than SC as it allows at least one type of reordering. It is stronger than ARM, which allows any type of reordering [33].

Table 2.1: Table showing the different type of reorderings supported on different architectures.

	Store/Store	Load/Load	Load/Store	Store/Load
SC				
x86				X
ARM	X	X	X	X

2.4.2 Soundness of Static Analysis

We must tread carefully when designing a data flow analysis under weak memory models. Since we are working with weak memory models, a flow-sensitive forward data flow analysis may give unsound results. If we assume sequential consistency, then we need to consider the possible interleavings and not worry about potential reorderings. Earlier work from Rinard [48] suggests that data flow analyses may inadvertently be sound for weak memory models.

Work from Alglave et al. [2], identifies that the class data flow analyses are sound with respect to weak memory models. More specifically, if a non-relational data-flow analysis is sound for SC, then it is sound for weak memory models[7]. Interval analysis is a non-relational analysis. Additionally, they show that interval analysis is sound for weak memory models if it is sound for sequentially consistent executions. This is a valuable result as we only need to consider sequentially consistent executions when designing the analysis.

2.4.3 Relation Analysis

Dartagnan encodes the memory model as part of the SMT formula to ensure that executions are consistent with the possible behaviours that it allows. We mentioned that Dartagnan supports memory models as inputs. This is part of the memory model specification language `cat` [1]. In `cat`, memory models define relations between pairs of memory events that must hold in a valid execution of a program. Furthermore, constraints can be defined over relations to which they adhere. Dartagnan encodes these relations as part of the SMT formula, which means that the encoding size increases. To remedy this, work from Gavrilenko et al. [22] introduces *relation analysis*, which helps Dartagnan scale well. It does this by computing so-called *may pairs* and *active pairs*, which are a static over-approximation of the domains of the relations. Only active

[7]The memory model must exhibit uniprocessor behaviour, which equivalent to SC between events that access the same memory address.

pairs, which are pairs that adhere to the constraints defined in the `cat` model, are added to the encoding. This, in turn, decreases the encoding size.

A relevant relation is the *read-from* (`rf`). A pair (w, r) where w is a write event and r is a read event, is in `rf` if r reads the value that w wrote. Relation analysis and `rf` help in increasing the precision of our interval analysis.

2.5 Conclusion

To summarise this chapter, we gave an overview of DARTAGNAN's IR, where events represent program instructions. After this, we introduced satisfiability, bit vectors, and software verification with BMC. We also discussed problems such as the state-space explosion problem and the increase in formula size with bit-blasting. We try to alleviate these problems in the next chapter. Then, we gave an overview of static analysis and the monotone framework. We will be using the framework to implement our interval analysis in the next chapter. Finally, we discussed memory models and reported on results from the literature that will help when implementing interval analysis.

Chapter 3

Interval Analysis

In this chapter, we discuss the different interval analyses that we introduced to DARTAGNAN. An *interval* for a program variable x, denoted $[l, u]$, where $l \leq x \leq u$, is the range of values between l and u that x can have. An *interval analysis* [12], approximates intervals for every variable in the program. Knowing the intervals of variables assists in optimisations regarding bit widths. Of course, knowing the bounds of variables allows for the reduction of their representation. In other words, we want to approximate the minimum number of bits needed to represent a variable. The technique has been used in the hardware field [52] and optimising compilers [54]. This analysis is a generalisation of constant propagation as singleton intervals. That is to say, $[l, u]$ is considered a constant if $l = u$. The generalisation allows for more optimisations regarding program simplifications. The rest of the sections describe the design of the interval analysis. Section 3.1 describes the analysis on a local level, only considering `local` events on registers. Section 3.2 expands on the local analysis by adding non-trivial evaluations for `store` and `load` events. In Section 3.3, we specify how we use the results from the interval analysis to add bounds to the SMT encoding of DARTAGNAN. Finally, we conclude on this chapter in Section 3.4.

3.1 Local Analysis

In the local analysis, we consider each thread to be a sequential program and analyse them with disregard for other threads. For this approach, we use already established interval analysis techniques. We use the monotone framework to describe the analysis. The lattice of intervals is $(I, \sqsubseteq, \sqcup, \sqcap)$[1]. Where:

$$
\begin{aligned}
&I = [l, u], l, h \in \mathbb{Z} \\
&\sqsubseteq : [l_1, u_1] \sqsubseteq [l_2, u_2] \iff l_2 \leq l_1 \wedge h_1 \leq h_2 \\
&\sqcup : [l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(h_1, h_2)] \\
&\sqcap : [l_1, u_1] \sqcap [l_2, u_2] = [\max(l_1, l_2), \min(h_1, h_2)]
\end{aligned}
$$

First, the abstract values I range over a domain of intervals consisting of integers. Second, one interval is more precise than another if its bounds are tighter. Finally, joining two intervals is computed by determining the interval that contains the most values. We associate an interval with each register via the abstract state $\sigma \in \mathit{Reg} \to I$, where Reg is the set of all registers. Furthermore, each event is associated with an abstract state $\mathit{Event} \to (\mathit{Reg} \to I)$ with Event

[1] We do not use $\sqcap$ but still define it

the set containing all events in a program. The *join* function combines information from different events:

$$join(e) = \bigsqcup_{s \in pred(e)} [\![s]\!]^2$$

With $e, s \in Event$ and $pred(e)$ being the predecessor events in the corresponding CFG. Note that information is propagated forward through the program. Additionally, let $Event \to (Reg \to I)$ be the $EventState$ lattice. We associate a $ProgramState$ lattice with each thread $t \in Thread$. The result of this analysis will be an element of $Thread \to EventState$.

Transfer Function

This analysis targets register-writer events, more specifically, `local` and `load` events. For `local` events, the transfer function is:

$$t_{\langle reg \rangle = \langle expr \rangle}(\sigma) = \sigma[\langle reg \rangle \to \widehat{eval}(s, \langle expr \rangle)]$$

We will refer to this as $t_{\texttt{local}}$ for ease in the future. We update the abstract state of the predecessor nodes with a mapping between $\langle reg \rangle$ and its abstract evaluation. The transfer function for the load event is trivial:

$$t_{\langle reg \rangle = \texttt{load}(\langle expr \rangle)}(\sigma) = \sigma[\langle reg \rangle \to \top]$$

Likewise, we will refer to the transfer function as $t_{\texttt{load}}$. Since we consider each thread as an isolated sequential program with no knowledge of other threads, we assume that a `load` can read any value. Other events simply propagate information and do not compute new information. The abstract evaluation function takes in an abstract state and computes a new interval from I. Its semantics for each expression are detailed in Table A.1.

Boundedness

Note that this lattice is infinite, which can lead to an infinite ascending chain through the lattice:

$$i_1 \sqsubseteq i_2 \sqsubseteq \dots$$

where $i_j \in I$. This infinite chain means that calculating a fixed point may take a long time or diverge [38]. We mentioned that the programs we analyse are *bounded*, which means that we do not have loops that may execute indefinitely. This simplifies the analysis further by not having to consider infinite chains.

Algorithm

We will now describe Algorithm 2, which is used to carry out the analysis. The algorithm expects a program P with threads T. First, we initialise θ, which will contain the final result of the analysis. Second, we initialise a work list W with all events from T and associate each event e_i with an abstract state σ_i, this association is called ϵ. It is assumed that all events in W are in program order. Third, we analyse an event e_i from the front of the work list and update ϵ with the new abstract state. The AnalyseEvent function applies the correct transfer function to event e_i and abstract state σ_i. This new abstract state is propagated through all the successors of e_i. Fourth, when a thread is analysed, we update θ, and finish the loop once all threads are analysed. Finally, θ is returned, and the analysis terminates.

[2] The double bracket notation means the abstract state of event s.

The algorithm takes only one pass to reach a fixed point. Indeed, at each event, new information is propagated forward only once. Considering the absence of loops and therefore back-edges in the CFG, we never compute new information in nodes that occur before it. Thus, a re-computation is never triggered, and a fixed point is reached after one pass through a thread.

The goal of this algorithm is to derive tight bounds with minimal overhead. Techniques that could help increase the precision of the local algorithm are out of scope of this thesis. One such technique is *control sensitivity* [38]. This technique takes into account information from conditional statements and uses them for increased precision. Methods using linear programming (LP) [57] are also possible but were deemed too expensive for our purposes. One would need to represent constraints explicitly and invoke an external LP solver.

Algorithm 2 Local interval analysis

Input: A program P
Output: $\theta \in ThreadState$

```
1:  procedure LocalAnalysis(P)
2:      for each thread T ∈ P do
3:          ϵ := [e_1 → σ_i, ..., e_n → σ_n]
4:          W := ∅
5:          W.enqueueAll(T)
6:          while W ≠ ∅ do
7:              e_i := W.remove()
8:              σ_i := ϵ(e_i)
9:              σ_i := AnalyseEvent(e_i,σ_i)
10:             ϵ[e_i → σ_i]
11:             for each e_j ∈ succ(e_i) do
12:                 σ_j = join(e_j)
13:             end for
14:         end while
15:         θ[T → ϵ]
16:     end for
17: return θ
18: end procedure
```

3.2 Global Analysis

For the global analysis, we focus on improving the precision to derive more bounds. We ignored shared memory to make a thread-local analysis. Another approach is to incorporate information from other threads. This, in turn, will help derive more bounds for registers that read data from shared memory. The main challenge here is to model interactions between threads. Of course, for flow-sensitive forward analysis, information may flow to more events than just its successors. This means that we have to consider different possible interleavings when propagating information. This fine-grained approach yields unreasonable analysis times. Rinard [48] mentions that this fine-grained approach is not necessary, and a coarse-grained approach does not affect the final result. The coarse-grained approach only considers parts of the program where threads can communicate. One way threads communicate is via shared memory. Thus, the goal is to find shared memory accesses and model the effect of different threads modifying the same address. This is also known as *thread-interference*. In our case, consider the example in Figure 3.1. Here,

we see three threads that access shared memory. Thread P_0 loads information from shared memory, and the arrows indicate from which `store` events it can read. So the register `r0` at that specific event contains the combined information of all the `store` events it can read from.

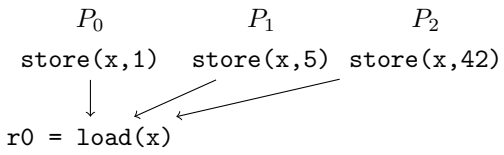


Figure 3.1: Example showcasing thread interference when loading a value from address `x`. We must consider all possible stores that could happen.

The main problem for the global analysis is finding all `store` events that may influence the value read by a `load`. This is not trivial due to the presence of aliases. We can make use of DARTAGNAN's alias analysis to find them. However, using alias analysis assumes the weakest memory model. If we assume a strong memory model, such as sequential consistency, the number of `store` events can potentially decrease.

Remember, DARTAGNAN takes a memory model specification as input. We can exploit this information to improve precision using results from relation analysis. Recall from Section 2.4.3 that relation analysis approximates relations defined by a memory model which contains pairs of events. The approximation of the `rf` relation can reduce the number of possible `store` events we must consider when joining information. This makes the analysis *memory-model-aware* as it incorporates information from a given memory model. Figure 3.2 and Figure 3.3 show the difference between using alias analysis and relation analysis under sequential consistency.

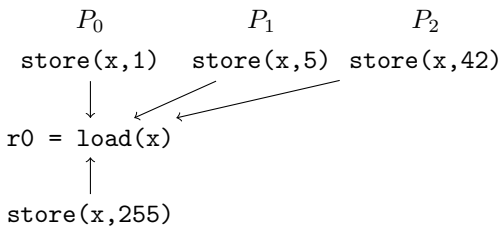


Figure 3.2: Example using alias analysis to discover `store` events.

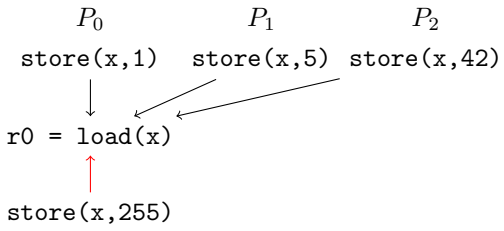


Figure 3.3: Example using relation analysis to discover `store` events under sequential consistency. The arrow in red indicates that it cannot read from the `store`.

Updated Transfer Function

We alter the transfer function for `load` events to consider all stores from which it can load data. First, we define $\mathcal{S}$ as the set of all stores and $\mathcal{L}$ as the set of all loads in the program. The function $ps : \mathcal{L} \to \mathcal{P}(\mathcal{S})$ where:

$$ps(l) = \{s \mid \forall s \in \mathcal{S} : (s, l) \in \texttt{rf}\}$$

Collects all the stores using the `rf` relation. The expression that a store $s \in \mathcal{S}$ writes to an address is denoted $expr(s)$. Abstractly evaluating this set of stores is done with the function $evalStore : \mathcal{S} \to I$ where:

$$evalStore(s) = \widehat{eval}(\sigma_s, expr(s))$$

With σ_s being the abstract state of store s. Finally, we define the transfer function $t_{\texttt{load}}$ as:

$$t_{\texttt{load}}(\sigma) = \sigma[\, \langle reg \rangle \to \bigsqcup_{s \in ps(\texttt{load})} evalStore(s)\,]$$

Updated Algorithm

The analysis consists of two parts: an enhanced version of the local analysis and a fixed point computation. The local analysis is enhanced with the new transfer function defined earlier. Furthermore, because of the sequential nature of the analysis, it could be possible that some information is not available whenever we join `store` events. Consider Figure 3.4, if we analyse T_0 before T_1, then we do not have any information about r_0 yet and derive $r_0 \to \top$. To not miss such bounds, we re-analyse each thread with local analysis until a fixed point is reached. This process is shown in Algorithm 3.

Init: $x = 0$

T_0	T_1
$r0 = \texttt{load}(x)$	$r1 = 1$
	$\texttt{store}(x, r1)$

Figure 3.4: Example illustrating a potential scenario during analysis.

We developed another version inspired by the value range propagation algorithm from Patterson [42], where the control flow was analysed first, followed by the data flow using SSA [13] edges. However, initial explorations of this approach revealed that it took too long to converge for our purposes.

3.3 Optimisations

In this section, we discuss how to use the analysis results to optimise the encoding in DARTAGNAN. The optimisations are an extension of the encoding phase and guide the solver to find solutions more quickly.

3.3.1 Constraints

Currently, variables are encoded with their size in the intermediate representation. For example, a variable with size `bv64` is encoded as a bit vector with length 64. During the encoding phase, we add extra constraints that bound the range of a variable. Let r be a bit vector variable in the

Algorithm 3 Global Analysis

Input: A program P
Output: $\theta \in ThreadState$
  procedure GLOBALANALYSIS(P)
    $\theta' := \bot$
    repeat
      $\theta' := \theta$
      $\theta :=$ LOCALANALYSIS(P)
    until $\theta' == \theta$
  return θ
  end procedure

SMT encoding and $[l, u]$ its interval from the interval analysis. Taking into account the possible sign, the lower bound constraints are as follows:

$$\begin{cases} (l \leq_S r) & l < 0 \vee u < 0 \\ (l \leq_U r) & l \geq 0 \wedge u \geq 0 \end{cases}$$

And the upper bound constraints:

$$\begin{cases} (r \leq_S u) & l < 0 \vee u < 0 \\ (r \leq_U u) & r \geq 0 \wedge u \geq 0 \end{cases}$$

These constraints, in turn, help the SMT solver optimise the encoding further. For example, SMT solvers such as `z3` try to reduce the bit width of bit vectors [17]. The reduced width in the bit vectors makes it such that the result of the bit-blasting procedure yields a smaller encoding. Besides a smaller encoding, the added constraints and, in turn clauses, could assist in unit propagation for a SAT solver.

Besides bit vectors, DARTAGNAN supports an integer encoding where variables are encoded using integers. Arithmetic is used to represent binary operations between variables. Using the bounds obtained from interval analysis, we can also introduce constraints for this encoding:

$$(l \leq r) \wedge (r \geq u)$$

Where r is an integer variable in the SMT encoding. The bounds help tighten the domain of the integer variable. Tighter bounds mean a smaller search space to explore for the solver, which can assist in achieving faster solving times.

3.4 Conclusion

In this chapter, we discussed a local and a global interval analysis. The local interval analysis from Section 3.1 considered only `local` events and derived trivial bounds for `load` events. Afterwards, in Section 3.2, we introduced a global interval analysis that derives more precise bounds for `load` events by considering thread-interferences. Additionally, we made the global analysis memory-model-aware by using relation analysis to search for thread-interferences. Finally, we specified the constraints that we will add using the bounds obtained from the interval analysis in Section 3.3. In the next chapter, we will evaluate our work.

Chapter 4

Evaluation

In this chapter, the focus is on evaluating our work. We collect results to help answer our research question:

Can interval analysis be used to improve the verification time of concurrent programs?

There are two main categories that we want to evaluate. First, we evaluate the verification time of Dartagnan with the interval analysis under different encoding parameters in Section 4.1. Second, interval analysis adds additional computational overhead, which increases verification time. In Section 4.2, we aim to measure the effect of the overhead compared with the number of register-writes in a program. Table 4.1 shows the specifications of the machine on which we ran all the experiments. We use two different tools for benchmarking:

- BenchExec [4] is the official benchmarking tool used to run a set of verification tasks for SV-COMP.
- Benchkit [40] is a Python-based performance evaluation framework which provides fine-grained control over the benchmarking process.

For all experiments, if no unrolling bound is specified, it is increased incrementally until a PASS or FAIL result is achieved. Regarding memory models, Dartagnan offers a set of specifications. For SV-COMP benchmarks, we use the `svcomp.cat` memory model specification. For other benchmarks, we use the specification for sequential consistency (`sc.cat`). Additionally, we implemented our analyses on Dartagnan version `4.3.0`.

Table 4.1: Table showing the parameters of the hardware and software of the system that ran the benchmarks.

Component	**Specification**
CPU	AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx (4 cores @ 2.1 GHz, 3.7 GHz boost, 8 threads)
RAM	8GB Samsung M471A1K43CB1-CTD SODIMM DDR4 Synchronous Unbuffered (Unregistered) 2667 MHz (0.4 ns)
OS	Fedora Linux 42 (Workstation Edition) (kernel version: 6.14.6-300.fc42.x86_64)

The software verification competition (SV-COMP) [5] is an annual competition where verifiers are evaluated. It provides a comparison of current state-of-the-art verifiers. More importantly, it

has established a set of benchmarks against which verifiers can be evaluated. For our purposes, we use the *ConcurrencySafety* set, which contains 725 concurrency problems. The competition evaluates a verifier on the verification track using a scoring schema shown in Table 4.2.

Table 4.2: Table showing the scoring schema of SV-COMP for verification tasks. Incorrect answers are punished heavily than correct answers. A FALSE is the same as a FAIL, and TRUE is the same as a PASS. Based on SV-COMP [5].

Points	Reported result	Description
0	UKNOWN	Failure to compute verification result, out of resources, program crash.
+1	FALSE (correct)	The error in the program was found and a violation witness was confirmed.
-16	FALSE (incorrect)	An error is reported for a program that fulfils the specification (false alarm, incomplete analysis).
+2	TRUE (correct)	The program was analysed to be free of errors and a correctness witness was confirmed.
-32	TRUE (incorrect)	The program had an error, but the competition candidate did not find it (missed bug, unsound analysis).

4.1 Different Encoding and Solving Methods

This experiment involves comparing the verification time of DARTAGNAN with different interval analyses. First, a naive analysis that derives $\top$ values for each register, which serves as a baseline. Second, a local analysis that ignores shared memory. Finally, our global analysis that derives tighter bounds. We hypothesise that global analysis should help provide better verification times and verify more tasks than both the naive and local analysis. The parameter space we explore is given in Table 4.3. We try triples of parameters for each run. For instance, a triple for one run would be (`encoding.integers=true`,`method=lazy`,`program.analysis.interval=naive`). One run consists of a BENCHEXEC execution on the *ConcurrencySafety* benchmark set with a two-minute timeout for each verification task.

Table 4.3: Table detailing the encoding parameters of DARTAGNAN that were varied during the experiment.

Parameter	Description
`encoding.integers`	Specifies whether variables are encoded using integers or bit vectors. Can be `true` or `false`.
`method`	Instructs DARTAGNAN to use the `eager` or `lazy` solving method. The `lazy` method uses a specialised theory solver [24].
`program.analysis.interval`	Specifies which interval analysis to use. The options are: `naive`, `local`, and `global`.

4.1.1 Overall Results

We observe that global interval analysis was able to verify 0.1-1% more tasks with a timeout of two minutes for each task (see Table 4.4) than both the naive and local analysis. This is especially true for the eager encoding, where we see a 0.5-1% increase in the number of correct bugs reported. Note that with these benchmarks, global interval analysis did not degrade DARTAGNAN in terms of correct tasks solved for most configurations. We notice a 0.28% decrease in the number of tasks solved when using integers and the `lazy` method. The local analysis verifies 0.27-1% fewer tasks than the naive analysis for most configurations. Regarding the overall CPU time of a run, we see that using global analysis results in a reduction of 1.73-7.53% in average time per task. Local analysis shows a 2.74-4.75% increase in average time per task for some configurations. However, it also shows 3.46% decrease in time per task for Integer-Lazy.

4.1.2 Trends in Verification Time

The trend in verification time is shown in Figure 4.1. The y-axis indicates CPU time in seconds, and the x-axis indicates the *n-th fastest result*. We see in Figures 4.1a, 4.1c, and 4.1d that using either of the three analyses resulted in similar verification times. Observe that, for verification tasks longer than 20-30 seconds, using global interval analysis resulted in lower CPU times compared to using local or naive. This suggests that global interval analysis only starts to give a meaningful speed-up at higher CPU times. Furthermore, it makes certain configurations scale better in comparison to using naive or local analysis.

Looking at Figure 4.1b, both runs did not diverge much in terms of CPU time. This experiment gives a general overview of the *ConcurrencySafety* set. However, further investigation is needed to see for which class of verification tasks we are getting the most gain, which is not covered in this thesis.

Our results suggest that global analysis outperforms both the local and naive analysis. Additionally, local analysis does not provide an improvement over naive analysis and performs worse for some configurations (see Figure 4.1c and Table 4.4b). We suspect that this is because the potential performance gain does not compensate enough for the additional computation overhead.

Table 4.4: Tables showing the number of verification tasks solved under two minutes, the overall time one run takes to finish, and the associated SV-COMP score according to the scoring schema outlined in Table 4.2. For each entry representing tasks solved, we add the percentage compared to the total number of tasks in parentheses. Percentages for the overall time show the percentage difference between the average time per task of the naive analysis and the average time per task of the specific sub-table. Furthermore, these percentages are computed by normalising the time with respect to the number of correct tasks verified. Colours for tasks solved represent the comparison with the naive analysis. Green cells mean a higher percentage of correct tasks solved, while red means a lower percentage. The cells remain uncoloured if they stay the same. Colours for overall time represent a comparison with the average time per task compared to the naive analysis. Green colours represent a lower average time per task, while red represents a higher average time per task. One should consider the number of tasks solved before considering overall time as a lower number of tasks solved leads to a lower overall time.

(a) Naive analysis (baseline)

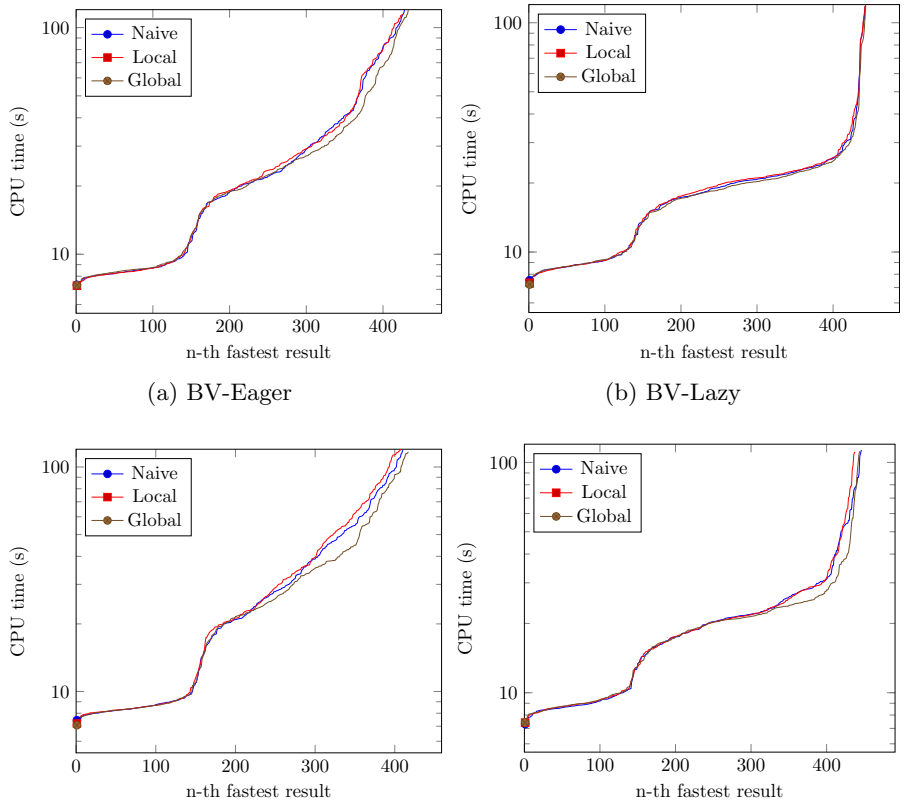
Parameters	**Total Corr. (%)**	**Corr. True (%)**	**Corr. False (%)**	**Overall time (%)**	**Score**
Integer-Eager	410 (56.55)	149 (20.55)	261 (36)	3h25m29s (0)	495
Integer-Lazy	446 (61.52)	153 (21.10)	293 (40.41)	2h37m46s (0)	535
BV-Eager	428 (59.03)	149 (20.55)	279 (38.48)	3h18m45s (0)	481
BV-Lazy	442 (60.97)	151 (20.83)	291 (40.13)	2h19m38s (0)	497

(b) Local analysis

Parameters	**Total Corr. (%)**	**Corr. True (%)**	**Corr. False (%)**	**Overall time (%)**	**Score**
Integer-Eager	408 (56.28)	148 (20.41)	260 (35.86)	3h34m12s (4.75)	492
Integer-Lazy	438 (60.41)	147 (20.28)	291 (40.14)	2h29m34s (-3.46)	521
BV-Eager	426 (58.76)	149 (20.55)	277 (38.21)	3h18m17s (0.23)	479
BV-Lazy	443 (61.10)	152 (20.97)	291 (40.14)	2h23m47s (2.74)	499

(c) Global analysis

Parameters	**Total Corr. (%)**	**Corr. True (%)**	**Corr. False (%)**	**Overall time (%)**	**Score**
Integer-Eager	417 (57.52)	150 (20.69)	267 (36.83)	3h20m12s (-4.21)	503
Integer-Lazy	444 (61.24)	152 (20.97)	292 (40.28)	2h25m14s (-7.53)	532
BV-Eager	433 (59.72)	149 (20.55)	284 (39.17)	3h12m51s (-4.09)	486
BV-Lazy	443 (61.10)	152 (20.97)	291 (40.14)	2h17m31s (-1.73)	499



(a) BV-Eager (b) BV-Lazy

(c) Integer-Eager (d) Integer-Lazy

Figure 4.1: These plots compare the CPU times (s) between the naive (baseline), local, and global analysis for matching parameter pairs. The y-axis is the CPU time in seconds, and the x-axis is the *n-th fastest result*. Using global analysis appears to begins to take effect after 20-30 seconds for all configurations except BV-Lazy.

4.2 Overhead Cost

Interval analysis comes with a computational overhead. This experiment aims to measure the overhead against programs with a varying number of registers. We use two sets of programs:

- a set of lock-free data structures `lfds`.
- a set of lock implementation `locks`.

In particular, the number of registers in lock-free data structures grows considerably with higher unrolling bounds and has higher verification times. We use a BENCHKIT campaign, verifying each program with an unrolling bound of six to get a PASS on each benchmark except for one. Additionally, we used the integer encoding together with the `eager` method and used no timeout. Tables 4.5 and 4.6 show the data gathered after benchmarking `lfds` and `locks`, respectively. Average size indicates the length of the interval. Total register reads indicate the total number of register-reads in a program. These are the registers that eventually get encoded as variables in the encoding. We add the percentage of registers that we bound in between parentheses. Total register writes roughly[1] correspond with the number of events we need to analyse. Looking at Figure 4.3, the overhead of global interval analysis is low compared to the total runtime ($< 3\%$). However, there are programs where the overhead can take almost a minute (see `safe_stack.c` in Table 4.5). This program is characterised by a large number of register-write events compared to the other programs. This corresponds with the number of events that need to be analysed. In general, we notice a linear relationship between register-write events and interval analysis time (see Figure 4.2).

If we observe Figure 4.4, we see that global interval analysis had a varied performance in terms of bounded register percentage. Some programs had a high bound percentage ($< 80\%$) relative to the amount of register-reads, some exhibited a low bounded percentage ($\sim 11\%$). For the lock-free data structures (Figure 4.4b) we see five programs with $< 26\%$ bounded register percentage. We suspect this may be due to the implementation not supporting many of the possible operators. Furthermore, the interval analysis found small bounds (2-3 average size) for many registers. Keep in mind, a small average size does not indicate reduced bit width, as the interval can have a high lower bound in absolute value. It does indicate a smaller integer domain for a register that reduces the search space.

For the local analysis, the results are given in Tables 4.7 and 4.8. Figure 4.5 shows that many programs have a low computational overhead ($\leq 1\%$). In Figure 4.4, we see that there are many programs where the local analysis did not derive many interesting bounds. Additionally, in Figure 4.6a, there are programs with 0% bounded registers. Our results suggest that global interval analysis was able to derive more bounds for these benchmarks. For example, for `hash_table.c` we derive 62.82% more registers.

4.3 Conclusion

In this chapter, we evaluated our work using two experiments. In the first experiment, we saw that using global interval analysis provided an increase in the number of tasks solved compared to using naive or local analysis. Additionally, it provided a decrease in average time per task. This suggests that our approach performs better than both the naive and local analysis. Furthermore, we noticed that using global interval analysis starts to become effective at longer verification tasks. In the second experiment, we measured the computational overhead of the local and

[1] We only analyse registers than can hold integers.

global analysis. We noticed that the overhead is low for both analyses. Additionally, our results confirm that we derive more bounds using global interval analysis for both benchmark sets.

Table 4.5: Table showing statistics from the global interval analysis on the Lock-free data structures benchmark set. All programs showed a low average interval size (≤ 5), except for one (`wsq.c`). Most programs showed a low bounded percentage ($\leq 17\%$). Most programs showed a low computational overhead for the global analysis($\leq 1\%$), except for `hash_table.c` and `safe_stack.c`.

Program	Average Interval Size	Total Register Reads (Bounded %)	Total Register Writes	Status Log	Interval Analysis Time (secs) (Overhead%)	Benchmark Time (secs)
chase-lev.c	5	87 (44.83)	78	PASS	0.037 (1.089)	3.396
hash_table.c	3	156 (85.26)	209	PASS	0.065 (2.143)	3.033
ms.c	2	750 (14.93)	581	PASS	0.461 (0.003)	13997.671
safe_stack.c	2	2898 (14.22)	2456	UNKNOWN	43.903 (7.263)	604.45
treiber.c	2	365 (25.21)	318	PASS	0.196 (0.053)	373.07
wsq.c	528857991	757 (17.7)	806	PASS	0.592 (0.008)	7614.063
dglm.c	2	684 (17.25)	549	PASS	0.255 (0.04)	637.567

Table 4.6: Table showing statistics from the global interval analysis on the Lock benchmark set. All lock programs exhibited a low average interval size (2-3). Some programs resulted in a high bounded percentage ($> 80\%$), while some showed a low bounded percentage ($\leq 11\%$) Computational overhead is low in general ($\leq 1\%$).

Program	Average Interval Size	Total Register Reads (Bounded %)	Total Register Writes	Status Log	Interval Analysis Time (secs) (Overhead%)	Benchmark Time (secs)
clh_mutex.c	2	138 (58.7)	77	PASS	0.028 (1.069)	2.62
cna.c	2	410 (3.66)	224	PASS	0.093 (0.558)	16.667
mutex.c	3	162 (68.52)	241	PASS	0.113 (0.266)	42.485
pthread_mutex.c	2	17 (82.35)	61	PASS	0.026 (1.177)	2.208
seqlock.c	2	138 (32.61)	144	PASS	0.056 (1.135)	4.934
spinlock.c	2	87 (86.21)	130	PASS	0.037 (0.997)	3.712
ticket_awnsb_mutex.c	3	130 (20)	89	PASS	0.044 (0.966)	4.555
ticketlock.c	3	27 (11.11)	31	PASS	0.017 (0.791)	2.149
ttas.c	2	51 (76.47)	58	PASS	0.027 (0.606)	4.455

Table 4.7: Table showing statistics from the local interval analysis on the Lock-free data structures benchmark set. All programs show a low analysis overhead (< 1%). None of the programs exceed 23% bounded percentage using the local interval analysis.

Program	Average Interval Size	Total Register Reads (Bounded %)	Total Register Writes	Status Log	Interval Analysis Time (secs) (Overhead%)	Benchmark Time (secs)
chase-lev.c	2	87 (3.45)	78	PASS	0.025 (0.595)	4.2
hash_table.c	2	156 (22.44)	209	PASS	0.039 (1.184)	3.294
ms.c	2	750 (11.2)	581	PASS	0.072 (0)	15661.285
safe_stack.c	2	2898 (14.08)	2456	UNKNOWN	0.221 (0.029)	768.247
treiber.c	2	365 (23.01)	318	PASS	0.05 (0.015)	334.331
wsq.c	2	757 (1.32)	806	PASS	0.213 (0.005)	4387.911
dglm.c	2	684 (12.28)	549	PASS	0.079 (0.012)	662.249

Table 4.8: Table showing statistics from the local interval analysis on the Lock benchmark set. There is a low overhead for each program (< 1%). Many programs derived no interesting bounds (0%).

Program	Average Interval Size	Total Register Reads (Bounded %)	Total Register Writes	Status Log	Interval Analysis Time (secs) (Overhead%)	Benchmark Time (secs)
clh_mutex.c	0	138 (0)	77	PASS	0.025 (0.825)	3.032
cna.c	2	410 (2.93)	224	PASS	0.049 (0.326)	15.032
mutex.c	2	162 (29.63)	241	PASS	0.045 (0.092)	48.845
pthread_mutex.c	0	17 (0)	61	PASS	0.016 (0.603)	2.652
seqlock.c	2	138 (26.09)	144	PASS	0.036 (0.574)	6.274
spinlock.c	2	87 (41.38)	130	PASS	0.031 (0.709)	4.37
ticket_awnsb_mutex.c	0	130 (0)	89	PASS	0.026 (0.504)	5.163
ticketlock.c	0	27 (0)	31	PASS	0.014 (0.555)	2.524
ttas.c	0	51 (0)	58	PASS	0.019 (0.391)	4.86

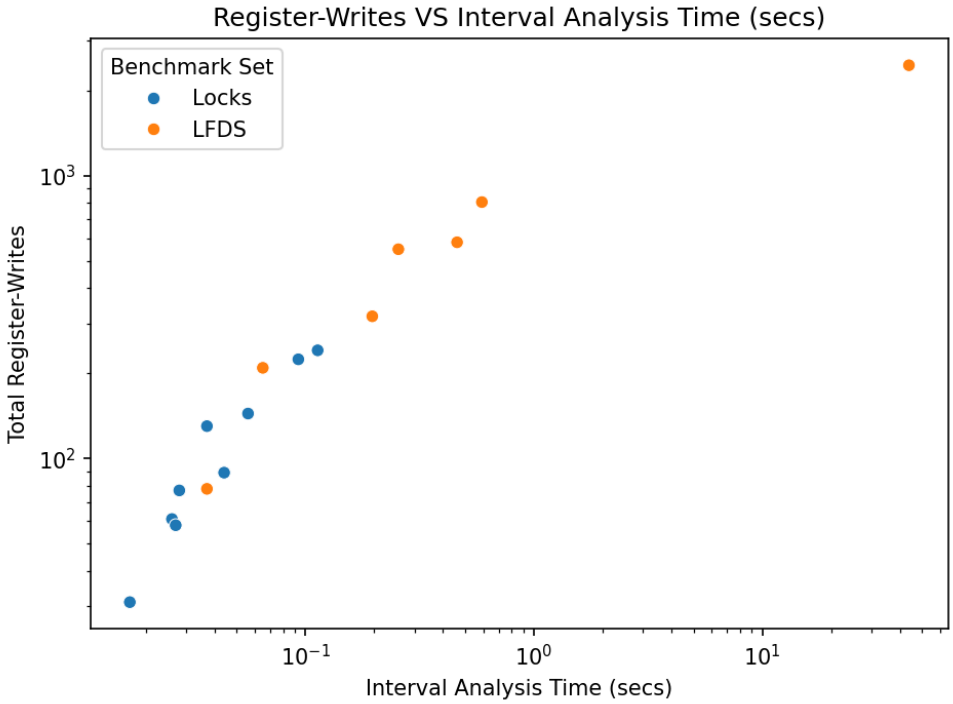
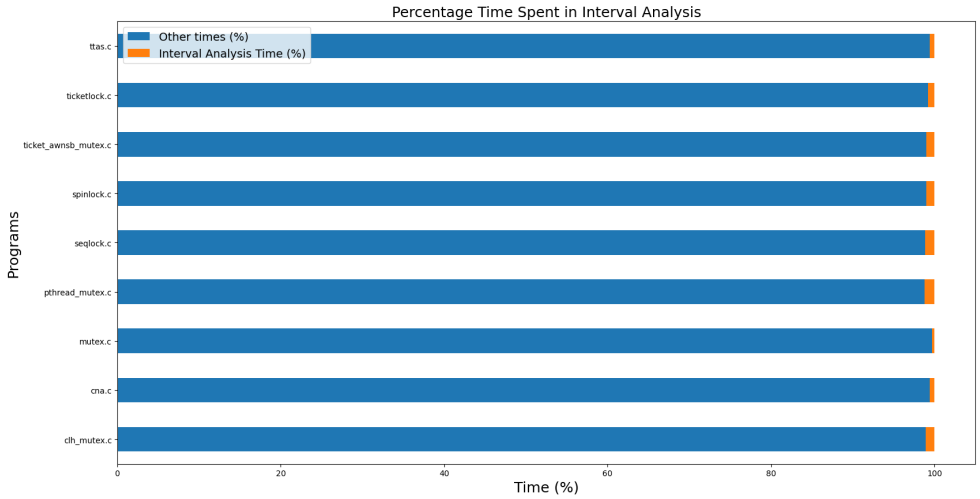
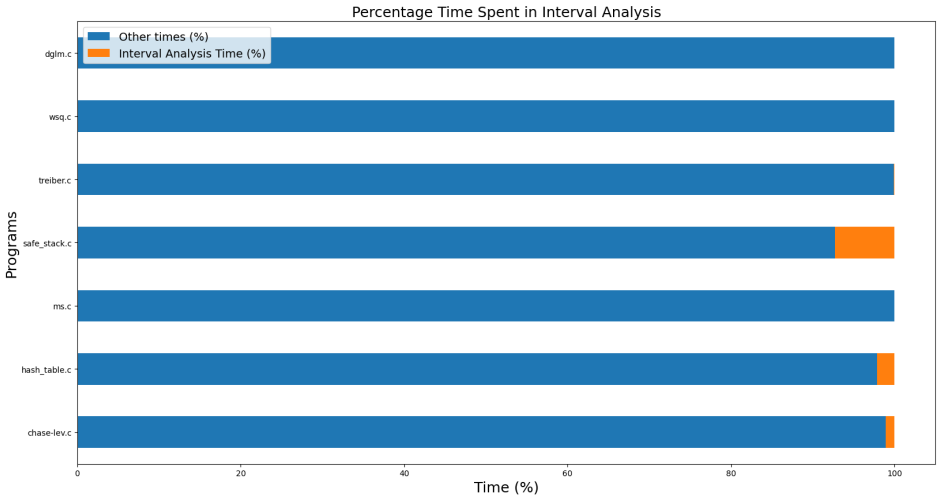


Figure 4.2: Scatter plot showing a linear relation between the number of register-write events and analysis time.

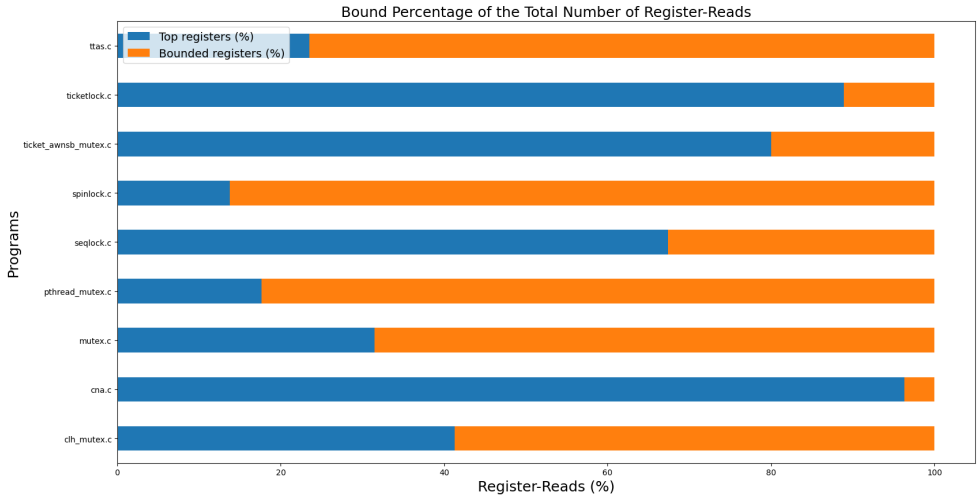


(a) Locks

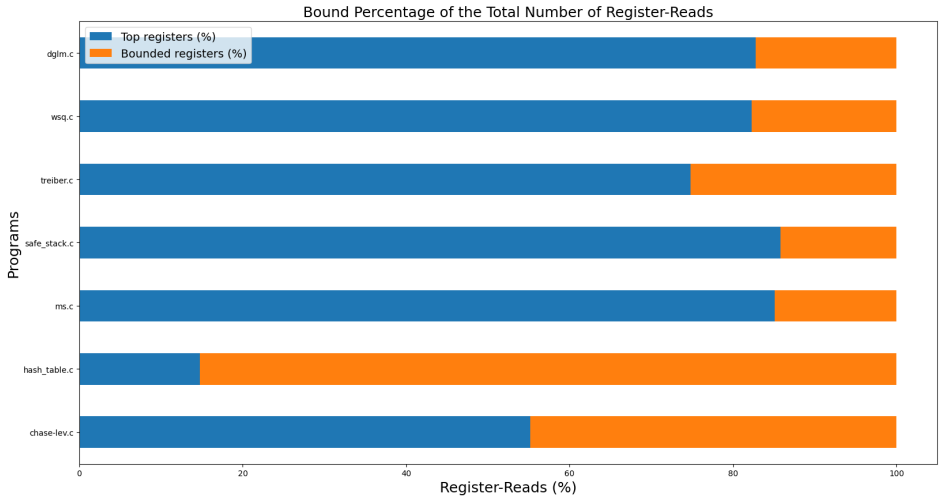


(b) Lock-free data structures

Figure 4.3: Stacked bar charts showing the overhead of the global interval analysis time relative to rest of the verification time for locks (a) and lock-free data structures (b). The overhead of the analysis is low compared to the benchmark time, except for `safe_stack.c`, which has 2456 register-write events that need to be analysed.

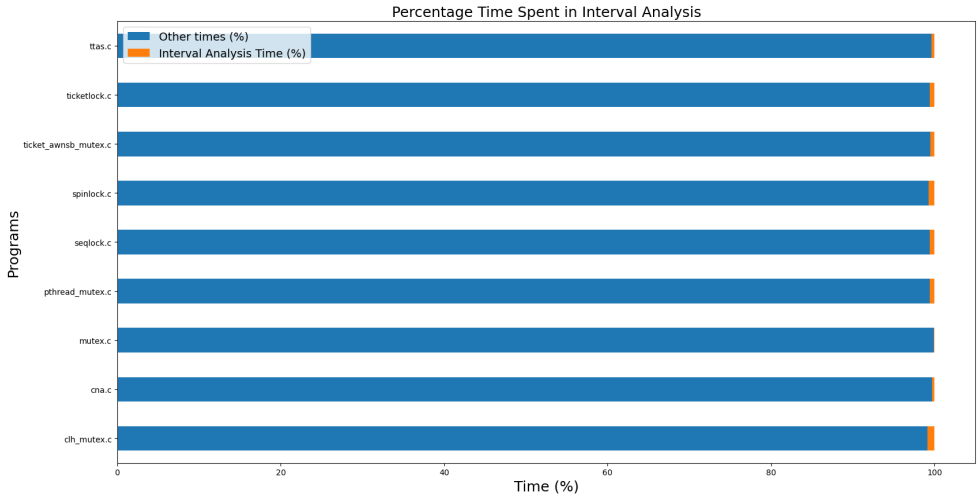


(a) Locks

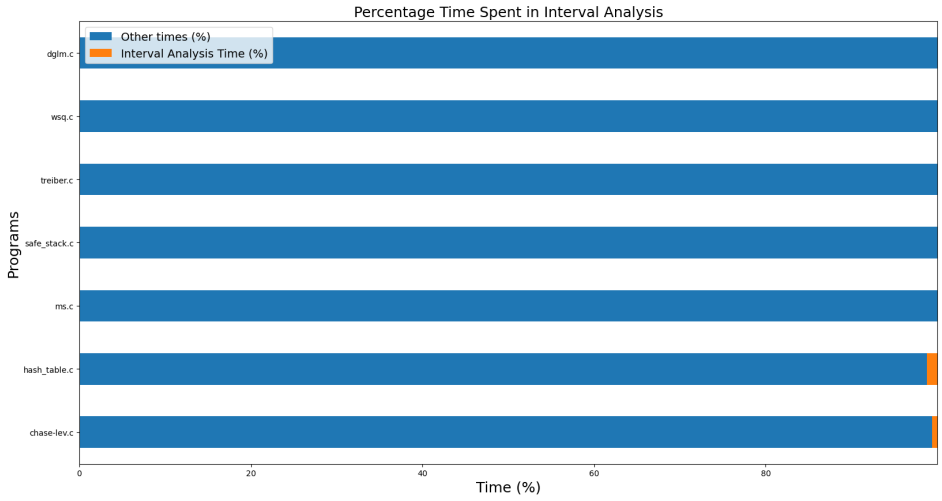


(b) Lock-free data structures

Figure 4.4: Stacked bar charts showing the percentage of bounded register-reads compared to $\top$ register-reads that the global analysis computed for locks (a) and lock-free data structures (b). The bounded percentages in (a) are varied, while (b) exhibits many low bounded percentages.

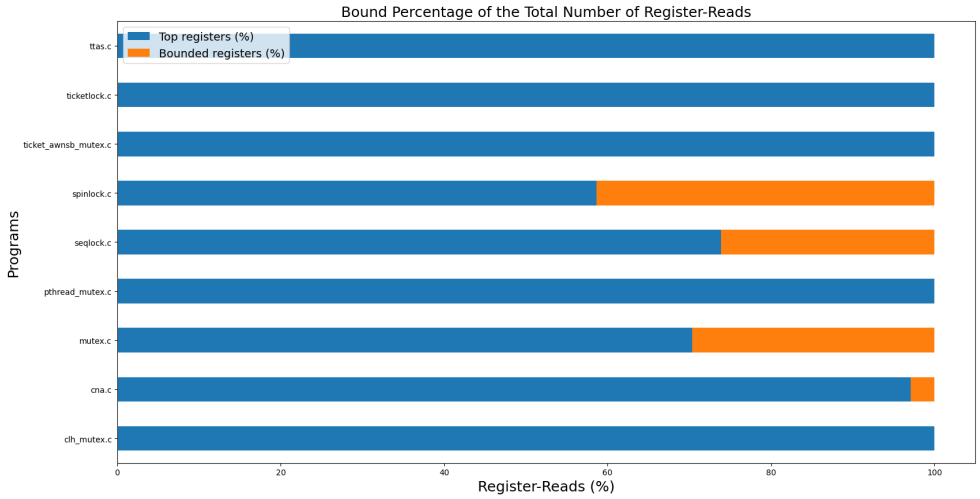


(a) Locks

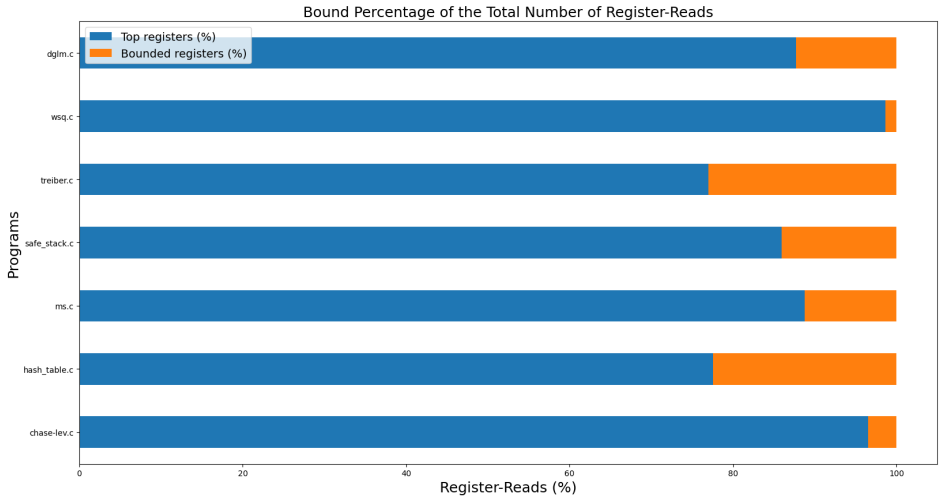


(b) Lock-free data structures

Figure 4.5: Stacked bar charts showing the overhead of the local interval analysis time relative to rest of the verification time for locks (a) and lock-free data structures (b). The overhead of the analysis is low compared to the benchmark time.



(a) Locks



(b) Lock-free data structures

Figure 4.6: Stacked bar charts showing the percentage of bounded register-reads compared to $\top$ register-reads that the local analysis computed for locks (a) and lock-free data structures (b). Most bounded percentages in (a) are low (0), while (b) exhibits many low bounded percentages.

Chapter 5

Related Work

In this chapter, we review additional related work with this thesis. We provide summaries of papers about interval analysis in verification and memory-model-aware static analysis.

5.1 Interval Analysis in Verification

The work of Menezes et al. [37] is the most recent and closely related to our work. They carry out a case study on the use of interval analysis in BMC. They extend a verifier called ESBMC [36] with interval analysis using abstract interpretation. Three perspectives about using interval analysis are researched:

- How it is best *applied*.
- The impact of *precision*.
- The impact of different interval *representations*.

In terms of application, they use interval analysis to optimise program size. Additionally, similar to this thesis, they introduce invariants for variables via instrumentation, which help the solver. They showed that optimisation and instrumentation showed very similar results. They found that more precise interval computation did not influence the SMT solving time greatly with $\leq 1\%$ impact. For representation, the integer domain and wrapped domain were used. Wrapped domain caused an additional overhead and resulted in fewer benchmarks solved. Overall, they reported a 5% improvement compared to not using interval analysis with a slight increase in resource consumption.

In Section 3.1, we mentioned work by Zaks et al. [57] that explores bit width analysis in model checking. They use LP techniques to derive bounds for variables to create smaller verification models. They introduce the notion of *Bounded Interval Analysis* where only perform interval analysis up to a certain number of steps. We differ from this idea by considering an analysis on a bounded program. They report a reduction in model size using a constraint-based approach on public domain benchmarks. The method was also used on proprietary software where more properties were able to be (dis)proven using interval analysis.

5.2 Static Analysis of Concurrent Programs

The global analysis takes into account memory models and is thus memory-model-aware. Work by Kusano and Wang [31] proposed a similar approach in the context of abstract interpreta-

tion. They introduce a memory-model-aware static analysis method based on thread-modular abstract interpretation. This is done by introducing a framework for deducing possible thread interferences. We differ from this approach as we use relation analysis to determine the possible stores a load may read from. Furthermore, they evaluated their method against other abstract analysers. Their results suggest that it increases accuracy as the approach provided significantly more correctness proofs at the cost of a runtime overhead versus other approaches.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we have introduced a global memory-model-aware interval analysis to the verification of concurrent programs that takes into account thread interferences on shared memory addresses. We extended a bounded model checker, DARTAGNAN, with multiple interval analysis. First, we introduced a thread-local analysis over bounded programs that did not consider shared memory. Building on this local analysis, we incorporated the non-trivial analysis of `load` events to derive more tight bounds. This analysis performed an enhanced local analysis until a fixed point is reached. To further increase the precision of global analysis, we utilised the information of relations such as `rf` from DARTAGNAN's relation analysis. The computed bounds were used to optimise the bit vector and integer encoding.

We evaluated our work on the *ConcurrencySafety* benchmark set of SV-COMP, which includes 725 concurrency problems. We observed a 0.1-1% improvement in the number of verification tasks solved for most configuration triples when using global interval analysis. One triple caused a small degradation in the number of tasks solved. Local analysis showed a 0.27-1% decrease in the number of tasks solved. However, in terms of average time per task, global interval analysis showed a consistent improvement for all triples (1.7-7.5%). We observed that using local analysis resulted in some configurations that saw an increase in average time per task. Our results suggest that using a global approach is more promising at higher verification times than a local one. Additionally, we measured the computational overhead for lock and lock-free data structures. We found that, for most benchmarks except for one, the overhead is low ($< 3\%$) in comparison with the overall run time. Furthermore, we noticed a linear relation between the number of register-write events and analysis time. In terms of deriving bounds, the analysis had a varied performance where we noticed 11-80% of total register-reads bounded. Our results show that global interval analysis helps derive more bounds than a local analysis that considers each thread as a sequential program.

We repeat our research question from the Introduction.

Can interval analysis be used to improve the verification time of concurrent programs?

Using our results, we draw the conclusion that a global interval analysis can improve the verification times of concurrent programs and that a local analysis can increase verification time.

6.2 Future Work

In this final section, we discuss future work, which outlines potential avenues of research. This includes using ideas from the literature to apply interval analysis in other areas. For example, we mentioned that Menezes et al. [37] use bounds to simplify a program. We can apply similar ideas to Dartagnan and measure its impact for concurrent programs.

Additionally, we propose exploring more variables for evaluation. First, weaker memory models increase verification time due to an increasing number of possible behaviours. We mentioned that we evaluated our work under sequential consistency, but we predict that results may differ when considering weaker models. Second, we evaluated our work on short to moderately long benchmarks. However, it remains to be seen what the effect is on benchmarks that run for days or weeks. Third, we gave a general overview of the results for the SV-COMP benchmarks. A more fine-grained view is needed to see on which class of concurrency tasks we receive the most performance gain, similar to Menezes et al. [37]. Finally, finding a relation between the number of bounded registers and the change in verification time may help in the explanation of the observed results. We mentioned how interval analysis attempts to help with the bit-blasting procedure for an optimised encoding. Exploring experiments that quantify the effect of this optimisation would be valuable in further evaluating the gain of introducing interval analysis.

We plan to open-source are works and merge it with Dartagnan [44]. As we mentioned in related work, interval analysis in the context of bounded model checking is not new. However, to the best of our knowledge, our work is unique in the context of concurrent programs and using relation analysis to increase precision. We expect this work to serve as a baseline for other tools to consider a global approach to interval analysis that considers weak memory models.

Bibliography

[1] Jade Alglave, Patrick Cousot, and Luc Maranget. *Syntax and semantics of the weak consistency model specification language cat*. Aug. 30, 2016. DOI: 10.48550/arXiv.1608.07531. arXiv: 1608.07531[cs]. URL: http://arxiv.org/abs/1608.07531 (visited on 06/30/2025).

[2] Jade Alglave et al. "Soundness of Data Flow Analyses for Weak Memory Models". In: *Programming Languages and Systems*. Ed. by Hongseok Yang. Berlin, Heidelberg: Springer, 2011, pp. 272–288. ISBN: 978-3-642-25318-8. DOI: 10.1007/978-3-642-25318-8_21.

[3] Bowen Alpern and Fred B. Schneider. "Defining liveness". In: *Information Processing Letters* 21.4 (Oct. 1985). Publisher: Elsevier BV, pp. 181–185. ISSN: 0020-0190. DOI: 10.1016/0020-0190(85)90056-0. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019085900560 (visited on 07/18/2025).

[4] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Reliable benchmarking: requirements and solutions". In: *International Journal on Software Tools for Technology Transfer* 21.1 (Feb. 1, 2019), pp. 1–29. ISSN: 1433-2787. DOI: 10.1007/s10009-017-0469-y. URL: https://doi.org/10.1007/s10009-017-0469-y (visited on 08/11/2025).

[5] Dirk Beyer and Jan Strejček. "Improvements in Software Verification and Witness Validation: SV-COMP 2025". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Arie Gurfinkel and Marijn Heule. Cham: Springer Nature Switzerland, 2025, pp. 151–186. ISBN: 978-3-031-90660-2. DOI: 10.1007/978-3-031-90660-2_9.

[6] A. Biere et al. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. NLD: IOS Press, Jan. 2009. 980 pp. ISBN: 978-1-58603-929-5.

[7] Armin Biere et al. "CaDiCaL 2.0". In: *Computer Aided Verification*. Ed. by Arie Gurfinkel and Vijay Ganesh. Cham: Springer Nature Switzerland, 2024, pp. 133–152. ISBN: 978-3-031-65627-9. DOI: 10.1007/978-3-031-65627-9_7.

[8] Armin Biere et al. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W. Rance Cleaveland. Berlin, Heidelberg: Springer, 1999, pp. 193–207. ISBN: 978-3-540-49059-3. DOI: 10.1007/3-540-49059-0_14.

[9] Edmund Clarke et al. "Bounded Model Checking Using Satisfiability Solving". In: *Formal Methods in System Design* 19.1 (July 1, 2001), pp. 7–34. ISSN: 1572-8102. DOI: 10.1023/A:1011276507260. URL: https://doi.org/10.1023/A:1011276507260 (visited on 12/01/2024).

[10] Edmund M. Clarke et al. “Model Checking and the State Explosion Problem”. In: *Tools for Practical Software Verification*. Ed. by Bertrand Meyer and Martin Nordio. Vol. 7682. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. ISBN: 978-3-642-35745-9 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1. URL: http://link.springer.com/10.1007/978-3-642-35746-6_1 (visited on 08/11/2025).

[11] Patrick Cousot. *Principles of Abstract Interpretation*. Cambridge, MA, USA: MIT Press, Sept. 21, 2021. 832 pp. ISBN: 978-0-262-04490-5.

[12] Patrick Cousot and Radhia Cousot. “Static determination of dynamic properties of programs”. In: *Proceedings of the 2nd International Symposium on Programming, Paris, France* (1976). Publisher: Dunod, pp. 106–130.

[13] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1, 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: https://dl.acm.org/doi/10.1145/115372.115320 (visited on 08/11/2025).

[14] Vijay D’Silva, Daniel Kroening, and Georg Weissenbacher. “A Survey of Automated Techniques for Formal Software Verification”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (July 2008). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 1165–1178. ISSN: 0278-0070. DOI: 10.1109/tcad.2008.923410. URL: http://ieeexplore.ieee.org/document/4544862/ (visited on 05/23/2025).

[15] Martin Davis, George Logemann, and Donald Loveland. “A machine program for theorem-proving”. In: *Commun. ACM* 5.7 (July 1, 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: https://dl.acm.org/doi/10.1145/368273.368557 (visited on 07/09/2025).

[16] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (July 1, 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: https://dl.acm.org/doi/10.1145/321033.321034 (visited on 07/09/2025).

[17] *Does Z3 use bounds on Bitvectors to reduce them to lower size/width? · Issue #6137 · Z3Prover/z3*. GitHub. URL: https://github.com/Z3Prover/z3/issues/6137 (visited on 08/11/2025).

[18] Bruno Dutertre. “Yices 2.2”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 737–744. ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_49.

[19] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer, 2004, pp. 502–518. ISBN: 978-3-540-24605-3. DOI: 10.1007/978-3-540-24605-3_37.

[20] Florian Furbach. “Verification with Memory Models as Input”. PhD thesis. Technische Universität Kaiserslautern, 2021. DOI: 10.26204/KLUEDO/6367. URL: https://kluedo.ub.rptu.de/frontdoor/index/index/docId/6367 (visited on 08/20/2025).

[21] Harald Ganzinger et al. “DPLL(T): Fast Decision Procedures”. In: *Computer Aided Verification*. Ed. by Rajeev Alur and Doron A. Peled. Berlin, Heidelberg: Springer, 2004, pp. 175–188. ISBN: 978-3-540-27813-9. DOI: 10.1007/978-3-540-27813-9_14.

[22] Natalia Gavrilenko et al. "BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings". In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Cham: Springer International Publishing, 2019, pp. 355–365. ISBN: 978-3-030-25540-4. DOI: 10.1007/978-3-030-25540-4_19.

[23] *GCC, the GNU Compiler Collection - GNU Project*. URL: https://gcc.gnu.org/ (visited on 08/11/2025).

[24] Thomas Haas, Roland Meyer, and Hernán Ponce De León. "CAAT: consistency as a theory". In: *Proceedings of the ACM on Programming Languages* 6 (OOPSLA2 Oct. 31, 2022). Publisher: Association for Computing Machinery (ACM), pp. 114–144. ISSN: 2475-1421. DOI: 10.1145/3563292. URL: https://dl.acm.org/doi/10.1145/3563292 (visited on 07/19/2025).

[25] David Hovemeyer and William Pugh. "Finding more null pointer bugs, but not too many". In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. PASTE '07. New York, NY, USA: Association for Computing Machinery, June 13, 2007, pp. 9–14. ISBN: 978-1-59593-595-3. DOI: 10.1145/1251535.1251537. URL: https://doi.org/10.1145/1251535.1251537 (visited on 08/11/2025).

[26] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*. Vol. 3A.

[27] John B. Kam and Jeffrey D. Ullman. "Monotone data flow analysis frameworks". In: *Acta Inf.* 7.3 (Sept. 1, 1977), pp. 305–317. ISSN: 0001-5903. DOI: 10.1007/BF00290339. URL: https://doi.org/10.1007/BF00290339 (visited on 07/23/2025).

[28] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. "Complexity of Fixed-Size Bit-Vector Logics". In: *Theory of Computing Systems* 59.2 (Aug. 1, 2016), pp. 323–376. ISSN: 1433-0490. DOI: 10.1007/s00224-015-9653-1. URL: https://doi.org/10.1007/s00224-015-9653-1 (visited on 07/15/2025).

[29] Daniel Kroening, R. E. Bryant, and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. 1 p. ISBN: 978-3-540-74104-6 978-3-540-74105-3. DOI: 10.1007/978-3-540-74105-3.

[30] Daniel Kroening and Michael Tautschnig. "CBMC – C Bounded Model Checker". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Berlin, Heidelberg: Springer, 2014, pp. 389–391. ISBN: 978-3-642-54862-8. DOI: 10.1007/978-3-642-54862-8_26.

[31] Markus Kusano and Chao Wang. "Thread-modular static analysis for relaxed memory models". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, Aug. 21, 2017, pp. 337–348. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106243. URL: https://doi.org/10.1145/3106237.3106243 (visited on 12/01/2024).

[32] Leslie Lamport. "How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor". In: *IEEE Trans. Comput.* 46.7 (July 1, 1997), pp. 779–782. ISSN: 0018-9340. DOI: 10.1109/12.599898. URL: https://doi.org/10.1109/12.599898 (visited on 08/11/2025).

[33] *Learn the architecture - ARMv8-A memory systems*. URL: https://developer.arm.com/documentation/100941/0101/The-memory-model (visited on 08/23/2025).

[34] *LLVM Language Reference Manual — LLVM 22.0.0git documentation*. URL: https://llvm.org/docs/LangRef.html (visited on 08/11/2025).

[35] João P. Marques-Silva and Karem A. Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability". In: *IEEE Trans. Comput.* 48.5 (1999), pp. 506–521. ISSN: 0018-9340. DOI: 10.1109/12.769433. URL: https://doi.org/10.1109/12.769433 (visited on 08/11/2025).

[36] Rafael Sá Menezes et al. "ESBMC v7.4: Harnessing the Power of Intervals: (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14572. Series Title: Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2024, pp. 376–380. ISBN: 978-3-031-57255-5 978-3-031-57256-2. DOI: 10.1007/978-3-031-57256-2_24. URL: https://link.springer.com/10.1007/978-3-031-57256-2_24 (visited on 12/10/2024).

[37] Rafael Sá Menezes et al. *Interval Analysis in Industrial-Scale BMC Software Verifiers: A Case Study*. June 21, 2024. DOI: 10.48550/arXiv.2406.15281. arXiv: 2406.15281. URL: http://arxiv.org/abs/2406.15281 (visited on 10/25/2024).

[38] Anders Møller and Michael I. Schwartzbach. *Static Program Analysis*. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/. Oct. 2018.

[39] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[40] Antonio Paolillo. *open-s4c/benchkit*. Dec. 9, 2024. URL: https://github.com/open-s4c/benchkit (visited on 12/10/2024).

[41] Antonio Paolillo et al. *Verifying and Optimizing Compact NUMA-Aware Locks on Weak Memory Models*. July 9, 2022. DOI: 10.48550/arXiv.2111.15240. arXiv: 2111.15240[cs]. URL: http://arxiv.org/abs/2111.15240 (visited on 04/08/2025).

[42] Jason R. C. Patterson. "Accurate static branch prediction by value range propagation". In: *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. PLDI '95. New York, NY, USA: Association for Computing Machinery, June 1, 1995, pp. 67–78. ISBN: 978-0-89791-697-4. DOI: 10.1145/207110.207117. URL: https://dl.acm.org/doi/10.1145/207110.207117 (visited on 08/11/2025).

[43] Hernan Ponce-de-Leon et al. "BMC with Memory Models as Modules". In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. 2018 Formal Methods in Computer Aided Design (FMCAD). Austin, TX: IEEE, Oct. 2018, pp. 1–9. ISBN: 978-0-9835678-8-2. DOI: 10.23919/FMCAD.2018.8603021. URL: https://ieeexplore.ieee.org/document/8603021/ (visited on 07/04/2025).

[44] Hernán Ponce-de-León. *hernanponcedeleon/Dat3M*. Aug. 18, 2025. URL: https://github.com/hernanponcedeleon/Dat3M (visited on 08/25/2025).

[45] Hernán Ponce-de-León et al. "Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Armin Biere and David Parker. Vol. 12079. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 378–382. ISBN: 978-3-030-45236-0 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_24. URL: http://link.springer.com/10.1007/978-3-030-45237-7_24 (visited on 12/01/2024).

[46] *POWER Memory Model*. URL: https://www.cl.cam.ac.uk/~pes20/ppcmem/ (visited on 07/03/2025).

[47] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Transactions of the American Mathematical Society* 74.2 (1953). Publisher: American Mathematical Society, pp. 358–366. ISSN: 0002-9947. DOI: 10.2307/1990888. URL: https://www.jstor.org/stable/1990888 (visited on 08/11/2025).

[48] Martin Rinard. "Analysis of Multithreaded Programs". In: *Static Analysis*. Ed. by Patrick Cousot. Berlin, Heidelberg: Springer, 2001, pp. 1–19. ISBN: 978-3-540-47764-8. DOI: 10.1007/3-540-47764-0_1.

[49] *SAT 2025*. SAT 2025. URL: https://satisfiability.org/SAT25/ (visited on 07/16/2025).

[50] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *Theory and Applications of Satisfiability Testing - SAT 2009*. Ed. by Oliver Kullmann. Berlin, Heidelberg: Springer, 2009, pp. 244–257. ISBN: 978-3-642-02777-2. DOI: 10.1007/978-3-642-02777-2_24.

[51] *SoSy-Lab / Benchmarking / SV-Benchmarks · GitLab*. GitLab. Aug. 7, 2025. URL: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks (visited on 08/11/2025).

[52] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. "Bidwidth analysis with application to silicon compilation". In: *SIGPLAN Not.* 35.5 (May 1, 2000), pp. 108–120. ISSN: 0362-1340. DOI: 10.1145/358438.349317. URL: https://dl.acm.org/doi/10.1145/358438.349317 (visited on 03/25/2025).

[53] *thufv/Deagle at 19e267151cca620cb1d24bc109a451b8a0e617f8*. URL: https://github.com/thufv/Deagle/tree/19e267151cca620cb1d24bc109a451b8a0e617f8 (visited on 10/25/2024).

[54] *Tree SSA passes (GNU Compiler Collection (GCC) Internals)*. URL: https://gcc.gnu.org/onlinedocs/gccint/Tree-SSA-passes.html (visited on 08/11/2025).

[55] G. S. Tseitin. "On the Complexity of Derivation in Propositional Calculus". In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28. URL: https://doi.org/10.1007/978-3-642-81955-1_28 (visited on 07/14/2025).

[56] Henry S. Warren. *Hacker's delight*. 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2013. 494 pp. ISBN: 978-0-321-84268-8.

[57] A. Zaks et al. "Bitwidth Reduction via Symbolic Interval Analysis for Software Model Checking". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.8 (Aug. 2008), pp. 1513–1517. ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2008.925777. URL: http://ieeexplore.ieee.org/document/4527113/ (visited on 01/09/2025).

Appendix

Appendix A

Tables

Table A.1: Table detailing the abstract evaluation function. The second column gives the result of an abstract evaluation. The implementation of the binary operators is given in Table A.2. Cast operators return $\top$ if $\langle targetsize \rangle$ is smaller than the original bit width, otherwise it returns the identity. Unary operators are not implemented and return $\top$.

Expression	Result
$\widehat{eval}(\sigma, \langle lit \rangle)$	$[\langle lit \rangle, \langle lit \rangle]$
$\widehat{eval}(\sigma, \langle reg \rangle)$	$\sigma(\langle reg \rangle)$
$\widehat{eval}(\sigma, \langle pred \rangle \, ? \, \langle expr_1 \rangle : \langle expr_2 \rangle)$	$\widehat{eval}(\sigma, \langle expr_1 \rangle) \sqcup \widehat{eval}(\sigma, \langle expr_2 \rangle)$
$\widehat{eval}(\sigma, \langle expr \rangle_1 \langle binop \rangle \langle expr \rangle)$	$\langle binop \rangle(\widehat{eval}(\sigma, \langle expr \rangle_1), \widehat{eval}(\sigma, \langle expr \rangle_2))$
$\widehat{eval}(\sigma, \langle castop \rangle \langle expr \rangle \langle targetsize \rangle)$	$\langle castop \rangle(\widehat{eval}(\sigma, \langle expr \rangle), \langle targetsize \rangle)$
$\widehat{eval}(\sigma, \langle uop \rangle \langle expr \rangle)$	$\langle uop \rangle(\widehat{eval}(\sigma, \langle expr \rangle))$

Table A.2: Table specifying the implemented operators in the current interval analysis based on Menezes et al. [37]. Implementations for the bitwise & and | are based on work from Warren [56].

Operator	Result
$[l_1, u_1] + [l_1, u_2]$	$[l_1 + l_2, u_1 + u_2]$
$[l_1, u_1] - [l_2, u_2]$	$[l_1 - u_2, u_1 - l_2]$
$[l_1, u_1] * [l_2, u_2]$	$[min(l_1 * u_1, l_1 * u_2, l_2 * u_1, l_2 * u_2), max(l_1 * u_1, l_1 * u_2, l_2 * u_1, l_2 * u_2)]$
$[l_1, u_1] \vert [l_2 + u_2]$	$[minOr(l_1, l_2, u_1, u_2), maxOr(l_1, l_2, u_1, u_2)]$
$[l_1, u_1] \& [l_2 + u_2]$	$[minAnd(l_1, l_2, u_1, u_2), maxAnd(l_1, l_2, u_1, u_2)]$