MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Handling Axiomatic Memory Models in Abstraction-Based Model Checking of Concurrent and Distributed Systems

Master's Thesis

Author
Levente Bajczi

Advisor
Dr. Vince Molnár

May 29, 2022

HALLGATÓI NYILATKOZAT

Alulírott *Bajczi Levente*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2022. május 29.

Bajczi Levente
hallgató

Contents

Kivonat

A kritikus hardver-szoftver rendszerek formális verifikációja egyre több területen elvárás. Emberéletek múlnak programok hibamentességén, és emiatt teljesen biztosnak kell lennünk abban, hogy soha nem fognak meghibásodni.

A gyengén rendezett kommunikációs formák jelentik az egyik legnagyobb problémát, amivel a state-of-the-art verifikációs eszközök is küzdenek. A látszólag tetszőleges átrendeződések kezelhetetlenül naggyá teszik az állapotteret a konvencionális eszközök számára.

Jelen dolgozatomban egy áttekintő elemzést nyújtok a párhuzamosság hatékony kezelésére képes verifikációs eszközökről, mind a szekvenciális, mind a gyengén rendezett esetben (1). Implementálom az egyik legjobban teljesítő eszköz algoritmusát a Theta keretrendszerben, melyet a Kritikus Rendszerek Kutatócsoport fejleszt és tart karban (2). Továbbá javasolok egy olyan megközelítést, mellyel a gyakorlatilag végtelen állapotterű bemeneteket is hatékonyan lehet kezelni, valamint implementálom az algoritmus fő lépéseit (3). Végezetül, alkalmazom a memóriamodellezési elveket üzenetalapú kommunikációs protokollok modellezésére, ezzel kiterjesztve a fent bemutatott algoritmusok és megközelítések használhatóságát (4).

Abstract

Formal verification of hardware-software critical systems is a necessity for an increasing number of applications. We trust the lives of people on the safety of computer software, and we have to be absolutely certain they will never fail.

One of the problems state-of-the-art verification tools struggle with is weakly ordered models of communication, both in the case of shared memory concurrency and distributed systems. The seemingly arbitrary reordering of accesses makes the state space unmanageably large for conventional approaches.

In this thesis, I survey the landscape of verification tools capable of performantly handling parallelism, both in the case of sequential consistency as well as weak memory ordering (1). I implement one of the best approaches in the verification framework Theta, a tool maintained by the Critical Systems Research Group (2). Furthermore, I propose a novel approach for handling practically infinite-state inputs for the weakly ordered memory models, and I create a proof-of-concept implementation of the main parts of the algorithm (3). Finally, I apply the memory modeling principles to message-based communication protocols, thereby widening the applicability of the algorithms and approaches shown before (4).

Chapter 1

Introduction

Safety critical hardware-software systems surround us in our daily lives. We rely on electronic brake systems to always slow down vehicles, automated power station management to never leave us without power and advanced collision avoidance systems to ensure no human error can lead to aviation catastrophes. However, developers of such systems can still be expected to make some mistakes, which could lead to faults causing many people to lose their lives or suffer outstanding monetary damages. The solution to this problem is tightly controlled quality assurance protocols, one part of which is to *verify* that the system will never find itself in an *unsafe* state under certain circumstances.

There are many methods of verifying safety. In the recent past, the most prominent technique has been testing, i.e., playing out scenarios with the system under test and comparing its responses to predetermined ones. While effective at catching common bugs, it is almost never exhaustive, meaning a proof of safety cannot be created via testing. In the scope of this paper, I deal with *formal verification*, which is a mathematically precise way of reasoning about a system's behavior. It is capable of both reporting bugs and capturing a proof a safety, but its performance overhead has been prohibitive of its use in the past. However, with the ever-growing pool of compute resources and the advancement of verification algorithms, its industrial use is starting to take hold and slowly replace (or at least complement) testing.

One of the techniques employed by formal verification is *model checking* [19], which uses a formal representation (a *model*), and by enumerating all its reachable states from some initial configuration, reason about its *unsafe* states. This method would solve all verification problems, as having access to the network of reachable states, any property over the state space would be trivial to check. However, in a general sense, this problem is *undecidable*: e.g., there cannot exist an algorithm that would determine whether any program terminates [42]. The discrepancy comes from the nature of the state space: it is not guaranteed to be finite, and in most real life cases, even finite-state systems have unmanageably large state spaces. Consider a simple program having two 32-bit integer variables. As each variable could have a possible 2^{32} different values, the size of the state space of the program would be 2^{32+32}, which would take up at least 18 exabytes of space given a very efficient 1 byte-per-state data structure. This phenomenon is called *state space explosion* [20], and most development effort towards model checking concentrates on the solution to this problem.

One possible solution to state space explosion is to employ *abstraction*, which partitions the concrete state space into groups that behave similarly. Using this method, the state space can become manageable and therefore verification may be feasible. However, the level of abstraction has to be fine-tuned so that it allows the efficient verification of the system while preserving enough information to not produce false alarms – which might be provided by a continuously refined abstraction level in a solution such as the Counterexample-Guided Abstraction Refinement (CEGAR) technique [18].

While abstraction solves many of the problems of conventional model checking, a new source of complexity arises when asynchronous components work together in a system, such as concurrent programs or distributed systems. Special techniques have to be employed to deal with this behavior, as the naive way of enumerating all interleavings of the parallel components cause the state space the explode. While partial order reduction (POR [26]) has been extensively used to combat this, recent work has shown that an *axiomatic* approach [7] works better at exploring states of concurrent programs. This is especially true for programs where memory access is not sequential by default, i.e., the order of instructions need not reflect the order of their apparent effects. In this case, the rules of the *memory consistency model* give the axioms, which guide the state exploration.

While many tools use the axiomatic approach to verify concurrent programs with weak semantics [7, 32, 21], it is usually complemented by another well-established verification technique such as stateless model checking [32], or bounded model checking [21]. However, no such solution exists that combines axiomatic verification with abstraction-based model checking – and furthermore, the aforementioned tools mainly concentrate on verifying software, rather than general parallelised architectures such as distributed systems.

In this thesis, my presented contributions are as follows:

I I survey and present the state of the art concerning software verification over axiomatic memory models

II I implement a customized version of the algorithm used by one of the best state-of-the-art verification tools

III I develop and implement an effective, abstraction-based state space exploration technique for handling axiomatic verification of concurrent software

IV I show an application of the *Cat* [8] memory modeling language to network-based communication protocols

The structure of the thesis is as follows: in Chapter 2 I introduce the necessary background the rest of this thesis builds upon. In Chapters 3-6, I elaborate on the four main contributions of my thesis (see above). Finally, in Chapter 7 I conclude the thesis by summarising its main points and observations.

Chapter 2

Background

This report builds upon the theories and findings of many fields of computer science, including embedded programming, formal software verification, memory modeling, concurrent software design and distributed systems. Some of these fields view the same topics slightly differently, e.g. software verification presumes a formal, mathematical model for the input program, while embedded programmers usually use the much lower abstraction level of source code to reason about properties of the software. This necessitates establishing the basis of the presented work to prevent misunderstanding among experts in these fields. This chapter introduces such concepts and defines their interpretation as used in the context of this work.

2.1 Safety-Critical Systems

If a hardware-software system was designed for a small selection of well-defined tasks, it is generally referred to as an *embedded system.* Such systems are not adept for general-purpose use, as their in- and outputs are often limited, and their software is seldom modifiable with the rare exception of program upgradability. *Embedded systems* can fulfil many kinds of tasks, ranging from operating the electrical windows on a car to performing complicated protocols for mid-air collision avoidance in airplanes (such as the Traffic Alert and Collision Avoidance System, *TCAS*), or providing a safety shutoff system for a nuclear plant.

Failure of an embedded system might be a minor nuisance or a serious safety problem, depending on the context of the application. If an electrical window fails on a car, the worst that can happen is some discomfort until the faulty unit is repaired or replaced – but failure of the TCAS might result in the collision of two airplanes where hundreds of lives are at risk. Any system that is designed to perform tasks where malfunction could lead to harm (physical or monetary) is classified as a *safety-critical system.*

Depending on the level of tolerable risk, a safety-critical system can be classified, e.g., according to *Safety Integrity Levels* (SIL) [30]. Several qualitative and quantitative measures are in place in such standards to mitigate dangerous failures, such as a controlled development workflow, thorough quality assurance and safety evaluation. For software components, the most widely used technique to assess safety is testing, i.e. running the program with defined sets of inputs and analyzing the outputs. This is not a definitive proof, as for untested inputs we cannot evaluate the behavior, but if the testing methodology is thorough enough, we can qualify the software as *probably safe* for the desired safety integrity level. To aid testing, formal methods such as model checking [19] (also see Section 2.2) and formal test generation [16] can be used, which are often too complex to be used on their own, but can support conventional testing methods.

With recent years' advancements, even safety-critical systems reached the point where scaling up in performance is next to impossible if only a single core is utilized [39]. The next logical step is to introduce multi-processor chips that can use smarter workload management to overcome the need for computing power. However, with multi-processor architectures and multi-threaded programs, the complexity of embedded systems surpasses the verification power of conventional testing, mainly due to the inherent nondeterminism of multi-threaded programs. When a program is strictly run on a single-core processor, it is relatively easy to guarantee that for a single set of inputs, the output of the program will always be the same. Therefore, it is enough to test each input set once, and assess the execution's results. With multi-threaded programs, an otherwise deterministic program can still produce different results based on timing differences among the processors, which cause different sections of the program to overlap in execution. Hence, testing is even less effective at proving safety, and a more formal method is often required.

A similar problem arises when instead of running on a multi-core processor, programs offload certain tasks to other participants in some network. This is the model of *distributed systems*, where in addition to the already high complexity of verifying separate subsystems on their own, their cooperative behavior needs to be verified as well. In some aspects, concurrent program execution and cooperative distributed systems operate under the same pretenses, but the way information is exchanged among them is inherently different.

2.2 Formal Software Verification

Formal software verification is a way to mathematically prove or disprove certain properties of an input program. Such properties might include *memory safety* (detecting use-after-free and other memory allocation problems), *reachability* (detecting if an unsafe state is reachable) or *termination* (detecting if the program will terminate in all its executions). In the context of this work, safety properties are always assumed to be *reachability* related, with a single unsafe state in the program.

Formal software verification often employs *model checking* [19], a technique that enumerates states of the input program and reasons about the properties of the states. For *reachability*-type queries, it is necessary to know whether the state marked as *unsafe* is reachable from the initial state(s) within a finite number of steps – if such a path exists, the program is deemed *unsafe* and safety cannot be guaranteed. In practice, generating all states of an input model is often infeasible, as even a single 32-bit variable will create 2^{32} different states according to its value. This phenomenon is called *state space explosion* [20], and counteracting it is required for any practically useful model checking algorithm.

A theoretical problem that verification tools have to face is the inherent *undecidability* of the model checking problem. Consider an arbitrary input program and an unsafe state at its exit point. To prove the (un)reachability of said state, the program's termination property has to be decided – which is proven to be undecidable [42]. This means, that any model checking algorithm will either be *incomplete*, i.e. some inputs will result in an *UNKNOWN* classification, or will produce *false* results in the form of *false alarms* and *missed bugs*.

A further problem of such algorithms is bridging the gap between the different abstraction levels in the verification workflow. Embedded programs are usually written in C or a similar high-level language, where concepts such as variable scopes, procedures and pointers make the lives of programmers easier, abstracting away the single instructions that will be generated by the compiler. However, the rich toolset of high-level languages greatly hinder the reasoning power of any formal method, as a formal model of the language semantics would be required. This is hard for some languages, and impossible to create for others: e.g. in the case of C++, the grammar is undecidable, i.e. there cannot exist any program that parses all C++-compliant code correctly[1]. To overcome these kinds of problems, the input programs are first transformed into a formal model, which can be done separately, as a pre-processing step, either by hand or in an automated way. However, it is important to keep in mind that the verification result of the model checking algorithm will only be valid for the formal model that served as its input, and not necessarily the source program – for that, verifying the result against the program's code might be necessary.

One such formalism is called a *Control Flow Automaton* (CFA) [13], which is mainly used to model programs.

Definition 1 (Control Flow Automata). *A control flow automaton is a tuple* $CFA = (V, L, l_0, E)$*, where:*

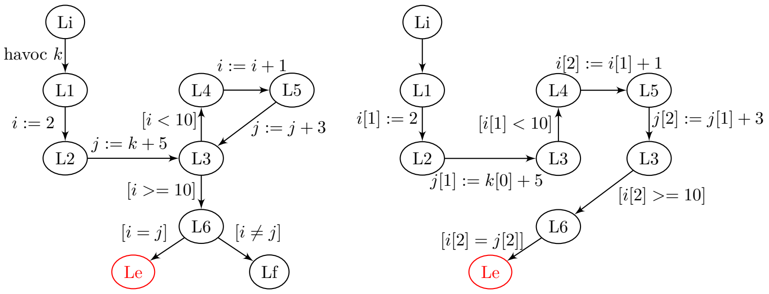
- V*: A set of variables*
- L*: A set of locations, representing the program counter (PC) in the program*
- $l_0 \in L$*: The initial location*
- $E \subseteq L \times Ops \times L$*: Directed edges in the CFA, describing the set of operations to be executed when the program advances to a new location*
  - $op \in Ops$*: An* assumption *of a predicate over* V *asserting its truth (i.e. an execution is only legal if the predicate is fulfilled), or an* assignment *of a new value to a* $v \in V$*. A special kind of assignment has the form* havoc v*, which assigns a non-deterministic value to* v*.* ▪

An *execution* of a CFA is a path over the directed edges E, starting from l_0, where at least one variable assignment exists that satisfies all assumptions of this path. Note that in the CFA, we use non-constant *variables*, which means multiple values can be assigned to a variable in a single execution. To keep track of variable values, *indexed constants* are used, where the index is increased with each assignment to the same variable. Assumptions and expressions always use the most recent *indexed constant*.

Consider the example in Figure 2.1. The program in Figure 2.1a reads a non-deterministic number k, then does several calculations over the variables i, j, k that includes a potential division-by-zero in line 9. This program is transformed into a CFA in Figure 2.1b, which includes an error location Le that represents the division-by-zero case, and a final location Lf which represents the successful termination of the program. An arbitrary path in this CFA leading to Le is presented in Figure 2.1c, which also shows the use of *indexed constants*. This format is called *Static single assignment*, as every indexed constant is assigned exactly once, meaning each value is invariant throughout the execution. Figure 2.1d shows the path feasibility query as a satisfiability formula, which can be given to an SMT-solver that can determine whether the path is a legal execution. In the presented case, there is a clear contradiction in the expressions over $i[2]$: if $i[1]$ is 2 then $i[2] := i[1] + 1$ means $i[2]$ must be 3 – which contradicts the $i[2] >= 10$ assertion. This means the path is not a feasible execution, and we have not yet determined the safety of the program.

[1] `https://blog.reverberate.org/2013/08/parsing-c-is-literally-undecidable.html`

```
int i, j, k;
k = ioread32();
i = 2;
j = k + 5;
while (i < 10) {
    i = i + 1;
    j = j + 3;
}
k = k / (i − j);
```



(a) C program with possible div.-by-zero **(b)** Control Flow Automaton for Figure 2.1a **(c)** Static single assignment form of a path in Figure 2.1b

$$i[1] = 2 \wedge j[1] = k[0] + 5 \wedge i[1] < 10 \wedge i[2] = i[1] + 1 \wedge j[2] = j[1] + 3 \wedge i[2] >= 10 \wedge i[2] = j[2]$$

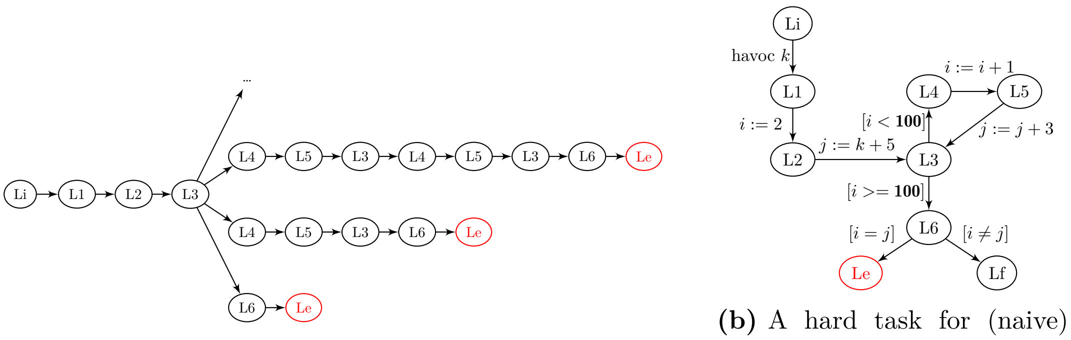
(d) SMT-expression of the path in Figure 2.1c

Figure 2.1: Mapping a program to a CFA

Note that while not explicitly shown, every *havoc* operation causes the index of the constant to increase, but nothing gets assigned to this constant. This means that the solver is free to choose any assignment, as long as it satisfies all other assertions that refer to this constant. Also, note that the example CFA in Figure 2.1b is not a correct mapping of the program in Figure 2.1a, as the values of the variables are entirely unbounded. This can be problematic if a path is found where the SMT solver reports the query as *satisfiable*, while in practice one of the variables would have wrapped around before reaching a value in the counterexample, making the path infeasible. This is a well-known limitation of SMT-based model checking, as fix-bit-width types cannot easily be mapped to mathematical integers. In the context of this work, best-effort practices are performed to counteract this phenomenon, namely, each *havoc* automatically implies an *assumption* over the variable's bounds (e.g. a C-like integer i will have an assumption that $-(2^{31}) \leq i \leq 2^{31} - 1$); and each *unsigned* integer type will wrap around when an out-of-bounds value is assigned to it, using modular arithmetic (e.g. a simple addition of the unsigned variable $u := u+1$ will be mapped to $u := (u + 1)\ mod\ 2^{32}$). As signed overflow is undefined in the C standard (and most other programming languages) [31], that case is unhandled and the values might fall out-of-bounds. A correct solution to this problem is to use bitvector arithmetic instead of integer arithmetic, but the implied performance overhead warrants the presented more performant approximation.

As we saw in Figure 2.1, the arbitrarily chosen path was not a feasible execution. However, that does not mean the program is safe – in fact, all paths would have to be checked first, to see if any produce the undesired division-by-zero problem. It is easy to see that due to the loop in the program, there are an infinite number of paths – to solve this model checking problem in a finite amount of time, many different approaches exist, but most of them fall into the following categories:

- Bounded Model Checking (BMC) – Starting from the initial state(s), check if an unsafe state is reachable within an expanding number of steps [15]
- k-Induction – Starting from the unsafe state(s), check if a safe state can be an expanding number of steps before any of the unsafe states [23]
- Explicit-State Model Checking – Mapping the states and transitions to an abstract state space using explicit-valued abstraction [29]



(a) Some paths to the error state in Figure 2.1b

(b) A hard task for (naive) BMC

Figure 2.2: Bounded Model Checking example and limitations

- Counterexample-Guided Abstraction Refinement (CEGAR) – Mapping the states and transitions to an abstract state, then refining the abstraction in subsequent iterations until a feasible counterexample is found, or safety is proven [18]

These techniques are generally used for different applications, e.g. BMC will usually find bugs the fastest but will not terminate if the state space is too big, while k-Induction is often capable of proving safety while not finding actually reachable unsafe states. Abstraction based techniques can be more complex and therefore slower, but they can both find bugs and prove safety relatively effectively. In the context of this work, I examined algorithms based on the BMC-approach, while the proposed new technique shows the abstraction-based state space exploration used by CEGAR. Therefore, I shall introduce these two approaches in detail below.

2.2.1 Bounded Model Checking (BMC)

As stated above, Bounded Model Checking (BMC) starts off with one or more initial states as the root of a tree graph, then repeatedly adds new states to the existing ones along transitions of the state space in a breadth-first-search (BFS) manner, i.e. all N-far states are added to the graph before any of the $(N + 1)$-far states. After reaching the upper bound of the analysis k, the graph is transformed into a satisfiability-modulo-theory (SMT) expression using the following recursive rule: *the expression at a node is the conjunction of its state expression and the disjunction of expressions in subsequent nodes.* This means that any junction will create an *Or* expression, and any paths will create an *And* expression. If this expression and the expression of the unsafe state are satisfiable together, the program is faulty as the unsafe state is reachable within this k-bound. Otherwise, the analysis continues building the graph with a new bound $k' > k$.

Take the example in Figure 2.1. The BMC algorithm will try to enumerate increasing-depth paths that end in the error location. The first few such paths can be seen in Figure 2.2a, but looking at the program in Figure 2.1a, these paths will be infeasible, as the value of i has not yet reached 10, i.e. the exit condition of the loop has not yet been fulfilled. After 8 iterations, and at a depth of 30, a feasible path will be found – if we assign -19 to k, the division will fail to due to the divisor being 0. Therefore, the program can be reported as *unsafe*.

Now consider the program in Figure 2.2b. The only difference to Figure 2.1b is the exit condition of the loop – instead of 8 iterations, the program must take 98 iterations before exiting the loop. For a naive BMC implementation, this means that it has to evaluate

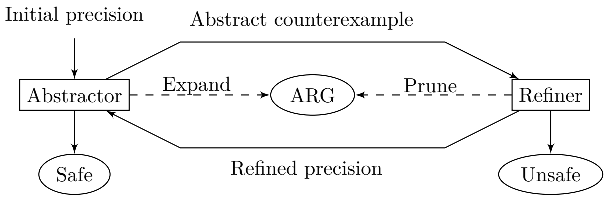


Figure 2.3: The CEGAR loop

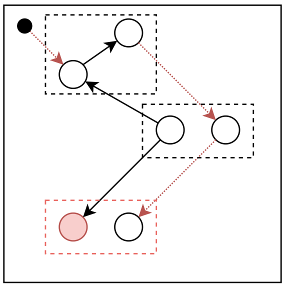


Figure 2.4: An ARG

97 infeasible paths before the counterexample is found. While possible, it might take a long time – and in the meantime, no proof of safety is possible as only specific traces are evaluated, rather than the whole program.

This trait of BMC (i.e. evaluating concrete traces rather than an abstract model) is both the source of its limitations, and its main advantage – if there is a bug, BMC is capable of finding it quickly, given it is not buried too deep in the program. This makes it appealing to use when a guarantee of safety is not required, but there is an incentive for finding bugs.

2.2.2 Counterexample-Guided Abstraction Refinement (CEGAR)

As opposed to BMC, CEGAR is a tool both for proving safety and finding bugs. While implementations of CEGAR are generally slower than BMC-based tools due to the algorithmic overhead; CEGAR can handle a superset of the tasks that BMC could solve (see Section 2.2.2.3). This means that the verification power of CEGAR is at least as big as that of BMC.

2.2.2.1 A Generic CEGAR Loop

CEGAR is a highly configurable verification algorithm [18], where the main product of the workflow is the *Abstract Reachability Graph* (ARG) [12]. The ARG is an overapproximation of the reachable state space, meaning a reachability in the concrete model implies reachability in the ARG, but not necessarily vice versa. This property can be seen in Figure 2.4: even though the filled-in error state is seemingly reachable if only the abstract states (denoted by rectangles) are taken into account, there is no actual path among the concrete states that could lead to the error state. Therefore, this level of abstraction is too high, and a more refined version is necessary.

A notable feature of ARGs is state coverability: if a state is found to be *covered* by another state, i.e. there is another state whose truth value is implied by the truth value of the current state (e.g. $S_1(a = 2, b = 3)$ implies $S_2(a = 2)$, therefore S_2 *covers* S_1). In this case, there is no need to expand the current state any further, as any observed behavior will also be observed by the *covering* state. This is a vital tool for combating state space explosion, as this means that covered states are handled only once. For example, if a program contains a loop that does not influence the reachability of the error state (e.g. waiting for an input), the only state we are interested in is the exit point of the loop – any in-between states are covered-by the loop header state, and therefore not expanded further.

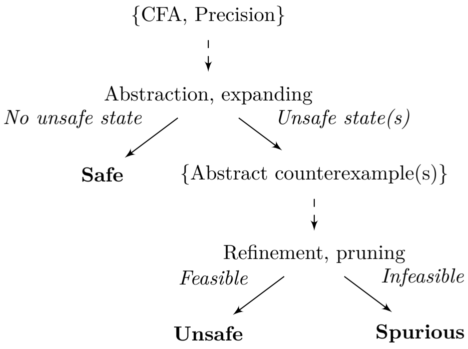


Figure 2.5: The CEGAR workflow

To achieve this continuous abstraction-then-refinement workflow, CEGAR works in a loop, as seen in Figure 2.3. This loop consists of two main algorithms working together: the abstractor and the refiner. The abstractor takes a precision describing the level of abstraction and the input model, and either creates a new ARG or expands the previously created, and pruned back ARG. This method is useful if a path is only found to be infeasible after a few feasible steps, and therefore the beginning of the ARG needs not to be re-created. When an ARG is ready, the abstractor checks whether an unsafe state is reachable, in which case the program is reported as *safe*. Note that safety is provable over an abstract state space, as it is always an over-approximation of the concrete state space and therefore the lack of an unsafe, abstract, reachable state implies the lack of an unsafe, concrete, reachable state as well.

If the created ARG does contain an unsafe state, the abstractor creates one or more *abstract counterexamples*, which are paths in the ARG leading to an unsafe state. These abstract counterexamples are then passed onto the refiner, which has multiple tasks: firstly, trace feasibility is evaluated (i.e. is at least one abstract counterexample concretizable, in which case the program is reported as *unsafe*). Then, if the counterexamples are infeasible, a new precision is created that is less abstract than the last one. Furthermore, the ARG is pruned back to the point where it became infeasible, then control is given back to the abstractor, where this abstraction-refinement cycle starts again.

The state of the algorithm after a single iteration of the CEGAR loop can have three values: *safe*, *unsafe* and *spurious*, as seen in Figure 2.5. The algorithm also produces proofs by default: for safety, a completely expanded ARG without an unsafe state suffices; and a feasible (concretizable) trace serves as the counterexample that shows the path to the bug in the program.

2.2.2.2 CEGAR Configuration Options

The CEGAR loop, as seen so far, is a declarative specification of the verification algorithm, i.e. only outcomes are specified and not the actual way to achieve said outcomes. This is due to the inherent configurability of CEGAR: as long as the parts are compatible with each other, many aspects of the algorithm can freely be swapped to other techniques fulfilling the same purpose.

As the possibilities are (almost) endless in terms of configurability, I only present the options provided by THETA[2], an open-source, generic and modular model checking frame-

[2] https://github.com/ftsrg/theta

work developed at the Critical Systems Research Group of Budapest University of Technology and Economics [40]. In the context of this report, I developed the proof-of-concept implementations of the presented algorithms in Theta. This choice is in part justified by the maturity of the framework (the implementation has been validated on thousands of input models, e.g. in the SV-COMP 2022 software verification competition[3]), and also based on my previous contributions to the framework, which are prerequisites for the work presented in this report. All implementation-specific details are published in [28] – I will only introduce those relevant to my work.

Theta implements the CEGAR loop with complete modularity in mind. It provides several built-in options for each swappable component, as well as an easy way to define custom ones. The two (arguably) most important ones are the *abstract domain* and the *refinement algorithm.*

2.2.2.2.1 Abstract Domain

The *abstract domain* specifies the basis of the abstraction, and by default, there are two *pure* domains implemented in Theta: the *explicit value* domain and the *predicate* domain. The latter is further divided, based on the way multiple predicates are handled inside a single state – there are *cartesian predicate abstraction*, *boolean predicate abstraction* and *split boolean predicate abstraction* domains. As previous results showed that software verification does not usually benefit from boolean predicate abstraction [28], I only focused on the *explicit* (EXPL) and *cartesian predicate* (PRED_CART) abstraction domains.

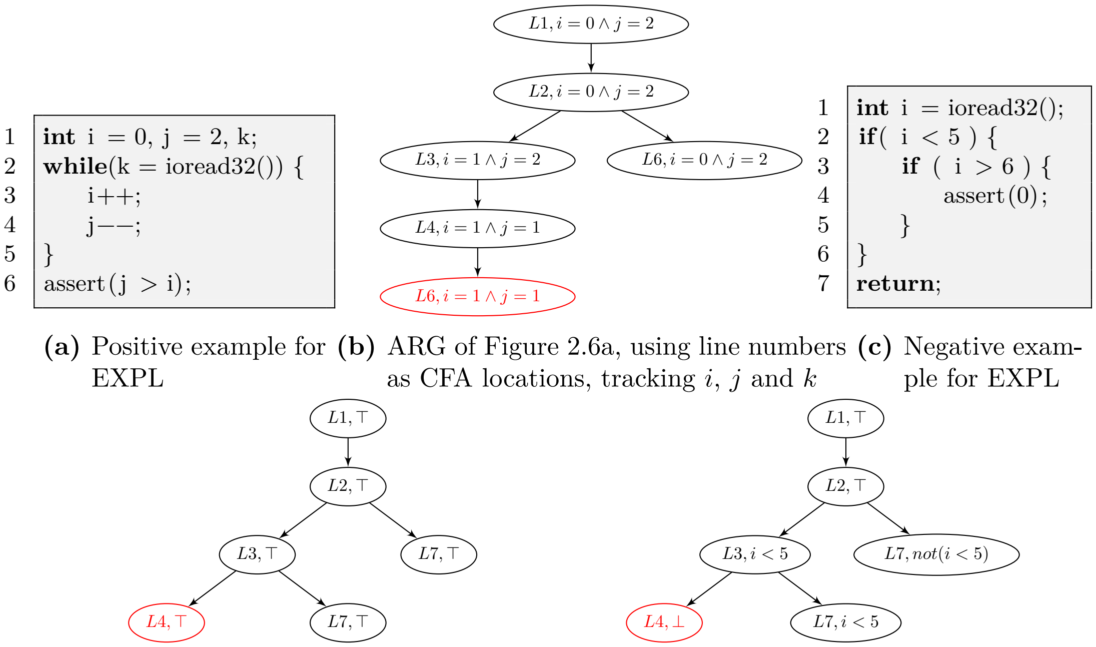
Definition 2 (Abstract Domain). *Formally, an* abstract domain *is a tuple* $D = (S, \top, \bot, \sqsubseteq, expr)$ *[28], where:*

- *S: Lattice of abstract states (possibly infinite)*
- *$\top \in S$: Top element*
- *$\bot \in S$: Bottom element*
- *$\sqsubseteq \subseteq S \times S$: Partial order over the lattice S*
- *$expr$: A mapping from an abstract state to an actual data state (i.e. an expression)*

To define an abstract domain, one has to give a mapping for each member of the tuple D.■

Explicit Domain The *explicit* abstraction domain defines the current abstraction precision as a set of *tracked* variables, i.e. variables whose values are of interest to us. Formally, the explicit domain can be defined as follows:

- S: A variable assignment of each *tracked* variable to a value of its domain, extended with top (arbitrary value) and bottom (no assignment possible) elements.
- $\top \in S$: No specific value is assigned to any of the tracked variables.
- $\bot \in S$: No assignment is possible to the tracked variables.
- $\sqsubseteq \subseteq S \times S$: $(s_1 \in S) \sqsubseteq (s_2 \in S) \iff (s_1 = s_2) \vee (s_1 = \bot) \vee (s_2 = \top)$
- *expr*: The conjunction of the equality expressions for each tracked variable and their value



```
int i = 0, j = 2, k;
while(k = ioread32()) {
    i++;
    j--;
}
assert(j > i);
```

```
int i = ioread32();
if( i < 5 ) {
    if ( i > 6 ) {
        assert(0);
    }
}
return;
```

(a) Positive example for EXPL

(b) ARG of Figure 2.6a, using line numbers as CFA locations, tracking i, j and k

(c) Negative example for EXPL

(d) ARG of Figure 2.6c, using line numbers as CFA locations, tracking i

(e) ARG of Figure 2.6c, using line numbers as CFA locations, tracking $i < 5$

Figure 2.6: Advantages and disadvantages of the EXPL domain w.r.t. PRED_CART

Note that when applying CEGAR on a CFA, the locations of the CFA are always explicitly tracked, as to always have a $1 : N$ relation between locations and states in the ARG.

The *explicit* abstraction domain also specifies a *maxenum* value, which is an upper bound on the enumeration of values to a variable in a single step – which can be useful if the domain of a variable is infinite or very large. Consider the program in Figure 2.6a: tracking the value of k is next to impossible, as it is always assigned a 32-bit non-deterministic number. Enumerating all possible states leads to the state space explosion we are trying to avoid. Therefore, the algorithm does not try to assign a value to k in any of the abstract states in the ARG in Figure 2.6b, even though the precision would allow it – instead, k is kept at its top element. However, even without the value of k, the algorithm is capable of deciding the safety of the ARG: after just one iteration, the assertion fails. In this example, this abstract counterexample also corresponds to a concrete trace, and therefore the refiner will most likely report the program as *unsafe*.

Predicate Domain The *cartesian predicate* abstraction domain defines the current abstraction precision as a set of tracked (and ponated)[4] predicates. Formally, the cartesian predicate domain can be defined as follows:

- S: A conjunction of predicates
- $\top \in S$: *True*
- $\bot \in S$: *False*
- $\sqsubseteq \subseteq S \times S$: $(s_1 \in S) \sqsubseteq (s_2 \in S) \iff (s_1 \implies s_2)$
- *expr*: The conjunction of the predicates applicable to the current state

[3] https://sv-comp.sosy-lab.org/2022/

[4] A ponated predicate means the outermost expression cannot be a *Not* operator.

Consider the program in Figure 2.6c. If we tried to solve this reachability problem with the explicit domain, we would get the ARG in Figure 2.6d even at the maximal precision of tracking all (one) variables. The unsafe state is reachable in the ARG, and therefore the abstractor cannot classify the input as *safe* – even though it is evident from the program's source that the assertion would never be reached, due to the contradicting $i < 5, i > 6$ assumptions. However, we cannot assign concrete values to i throughout building the ARG, as i can take up almost 2^{31} different values that would fulfil either criteria, and we can only evaluate one *if* statement at a time if the CFA contains different edges for them. (As a sidenote, this problem could also be solved by using large-block encoding (LBE) [14], but currently, Theta only supports a simple version of that).

In comparison, the cartesian predicate abstraction only needs the predicate $i < 5$ in the precision to deduce the safety of the program, as seen in the ARG in Figure 2.6e. Note the unsafe state in red: the ARG building algorithm correctly assigned the bottom element $\bot$ to its abstract state, as there was a contradiction in the path – meaning the ARG is complete and lacks unsafe states, and therefore the program is *safe*.

An aspect of the abstraction that was previously left out is the *transfer function*. The transfer function T maps sets of abstract states to the tuples consisting of an abstract state, a list of operations and a precision ($T : S \times Ops \times Prec \mapsto 2^S$). In practice, this determines the successor states of an abstract state in the ARG – which is a clear contradiction to the previous description of how an ARG is created (i.e. grouping concrete states together). While that was also a correct way of creating an ARG, it is not practical to create all concrete states just for being able to create abstract states out of it: instead, the transfer function is used to explore the abstract state space, and expand previously discovered abstract states. For example, given the EXPL domain, a precision tracking i, and an edge in the CFA assuming $i > 0 \wedge i < 4$ between locations l_1 and l_2; the state $s_0(l_1, \top)$ would have the following successor states: $\{s_1(l_2, i = 0),\ s_1(l_2, i = 1),\ s_1(l_2, i = 2)\}$.

Note that even though the transfer function assigns successor states to abstract states deterministically, the way these successor states are handled deeply influences the verification workflow: it is possible to visit and expand the first successor state in every instance, and therefore explore that state space in a depth-first manner (DFS), and it is also possible to visit all successors first before successors to those are visited and expanded (BFS). Any further combination of these techniques can also exist, such as an error-location-guided search (ERR), which will favor DFS more if the state is closer to the error location, but defaults to BFS otherwise [28].

2.2.2.2.2 Refinement Algorithm

As we have seen in Figure 2.6e, the predicate $i < 5$ in the precision was enough to guide the abstraction algorithm towards discovering that the program is *safe*. However, the discovery of this predicate is not trivial, and it can come from two sources: either from the initial precision (e.g. all assumes in the model), or the refinement algorithm will have to discover it while refuting the abstract counterexamples.

There are many different refinement algorithms implemented in Theta, but the relevant distinction among them in the context of this report is the following:

- Single-counterexample refinement: a single counterexample is generated from the unsafe ARG, and refuting it provides the new precision for the abstractor
- Multi-counterexample refinement: every counterexample is generated from the unsafe ARG, and a combined refutation provides the new precision for the abstractor

2.2.2.3 BMC Inside CEGAR

As mentioned above, CEGAR is at least as powerful of a verification tool as BMC, due to BMC being part of CEGAR. In order to justify this claim, consider the following:

- The CFA is extended with a counter c, which is increased after every statement in the model, starting with 0
- domain: EXPL
- initial precision: $\{c\}$
- search algorithm: BFS

This configuration will mimic the BMC algorithm, as it enumerates all paths in the program following the value of c. If BMC can find a counterexample, this method will be able to find it as well – and if BMC runs out of enumerable paths and classifies the program as *safe*, this technique will arrive at the same conclusion as well.

2.3 Multi-Processor Architectures

Modern hardware architectures almost universally offer concurrent memory access in a relaxed way. This means that read and write operations do not have to execute sequentially, the memory controller is free to reorder them (respecting certain constraints) to increase performance. The rules for such relaxed accesses is described by a *memory consistency model* (MCM). The specification of MCMs evolved from textual documentation through small "Litmus-tests" describing forbidden outcomes to well-defined axiomatic formal specifications of the execution semantics [10, 7, 38].

2.3.1 Memory Consistency Models

Generally, a memory model of an architecture can either be *operational* or *axiomatic* [7]. The former uses elements of the hardware platform such as queues and buffers to explain certain behaviors on the target architecture, which makes it easier to implement the architecture directly in hardware, but hinders reasoning on the software side [38]. In contrast, an *axiomatic* memory model uses a declarative approach to forbid certain sets of relations over memory accesses [10]. This approach proved to be better for reasoning about possible executions of concurrent programs, and therefore most software verification tools employ an *axiomatic* model to provide information on the guarantees of the hardware architecture [5, 3, 2, 25, 21] or the programming language [32, 35].

One *axiomatic* memory modeling language is CAT [8], which has been created to specify memory models for Herd [7] but has since seen widespread adoption due to its succinctness and expressivity.

The CAT language uses the notion of *candidate executions* to model possible executions of a program. A candidate execution is a directed, labelled graph, where each node corresponds to an *event* and is labelled by the specific *event sets* it belongs to; and each edge corresponds to a binary *relation* defined over the events. Event sets and relations come from both a predefined list of built-in elements, as well as the memory model.

2.3.1.1 Event Sets, Relations and Constraints

The predefined event sets consist of the following [8]:

- W: write events
- R: read events
- M: memory events ($M = W \cup R$)
- IW: initial writes stemming from the initial memory state
- FW: final writes observed at the end of test execution
- B: branching events
- RMW: read-modify-write events
- F: fence events
- XYZ: specific fence events from the architecture (such as `MFENCE`, `FENCE` and `SFENCE` for x86)

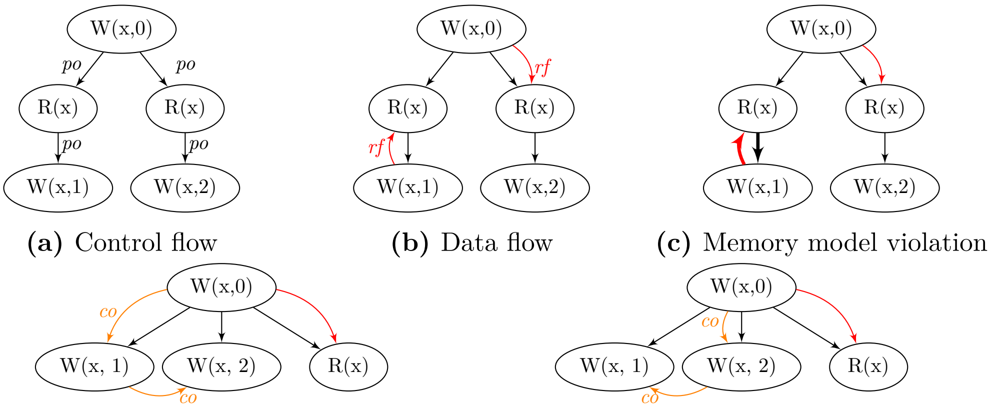
Derived event sets can be created by applying set operations over existing ones, such as union, intersection and difference. Furthermore, the (possibly filtered) Descartes-product of event sets can lead to *relations*. Some relations are predefined as follows [8]:

- po: program order
- addr: address dependency (the target event's memory location depends on the source read's value)
- data: data dependency (the target write's value depends on the source read's value)
- ctrl: control dependency (the target event is found in a branch controlled by an assumption depending on the source read's value)
- rmw: read-exclusive write-exclusive pair: the target exclusive write is a successful operation matched with the source exclusive read; or atomic RMW instructions
- amo: relate the read and write events of atomic RMW instructions
- id: identity (relate each event to itself)
- loc: same-location (relate all event-pairs that touch the same memory location)
- ext: external (relate all event-pairs that reside in *different* threads)
- int: internal (relate all event-pairs that reside in *the same* thread)
- rf: read-from (the target read's value is taken from the source write)
- co: coherence order (a total order of same-location write events) [5]

Derived relations can be created by applying one or more of the following operators over existing ones:

- Complement
- Domain
- Identity closure

[5] While CAT in itself defines co as a derived relation, it is a vital element of every execution and therefore I elevate it into this list



(a) Control flow **(b)** Data flow **(c)** Memory model violation

(d) Last value is 2, *Read* receives 0 **(e)** Last value is 1, *Read* receives 0

Figure 2.7: Candidate executions

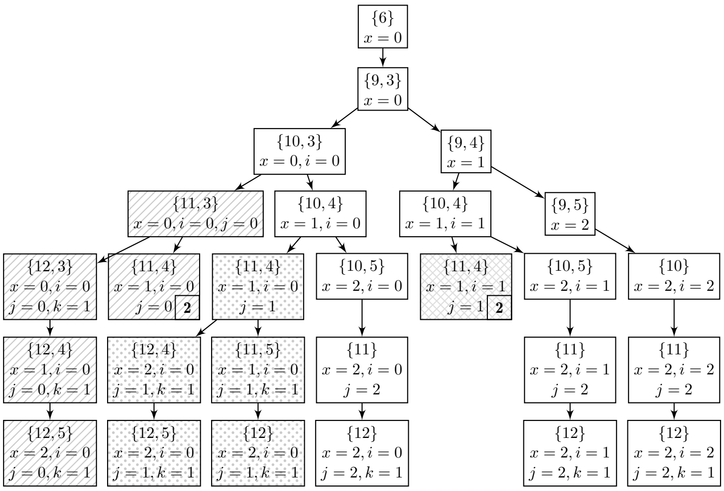
- Inverse
- Reflexive transitive closure
- Transitive closure
- Range
- To ID
- Sequence
- Union
- Difference
- Intersection

Finally, relations can be constrained to be *(ir)reflexive*, *(non-)empty* or *(a)cyclic*. *Candidate executions* can be checked against the memory model specification to decide if they are *consistent* or *in violation* with the candidate. For example, against the specification that no *Read* on a given thread shall receive a value from a later *Write* on the same thread (any path $(\mathit{po} \mid \mathit{rf})^{+}$ is acyclic), the example *candidate execution* in Figure 2.7c is in violation of the memory model. However, if a *candidate execution* is indeed *consistent*, it becomes an *execution graph*, describing an observable outcome on the given architecture.

Note the lack of other primitives on the execution graph. By populating the *po* and *rf* relations the necessary control and data flow is fully defined, and no further relation is necessary – most notably, the *coherence order co* [7, 21] (or *modification order* [32]), i.e. the total ordering of same-location *Write* events, is entirely superfluous. Consider the two *candidate executions* in Figures 2.7d and 2.7e, where this relation is not omitted. In both cases, the *Read* will read 0 from memory, but the order of the *Write* events differs. The outcome and overall execution of the program is not influenced by these differences other than the final observable value in memory. However, as the two executions are not the same graph, both variants will be produced when enumerating the possible *candidate executions*. Unless the state of the global memory is ever needed to be queried, it is enough to determine for each *Read* event the set of *Write* events it might receive data from. This does not mean that any derived relation that uses *co* as one of its operands will have to be completely re-written, but rather that we can simply leave *co* as unspecified with a few constraints: as long as there is a *co*-order that relates every same-location pair of writes in either in exactly one way (i.e. if elements a and b are same-location writes, either $co(a, b)$ or $co(b, a)$ is true, but not both), the execution is feasible.

```
1  int x;
2  void* thr(void* _) {
3    x = 1;
4    x = 2;
5  }
6  int main() {
7    pthread_t t;
8    pthread_create(&t, 0, thr, 0);
9    int i = x;
10   int j = x;
11   _Bool k = (j >= i);
12   pthread_join(handle, 0);
13   assert(k);
14 }
```

(a) Concurrent program testing sequentiality


{6} x = 0
{9,3} x = 0
{10,3} x = 0, i = 0
{9,4} x = 1
{11,3} x = 0, i = 0, j = 0
{10,4} x = 1, i = 0
{10,4} x = 1, i = 1
{9,5} x = 2
{12,3} x = 0, i = 0 j = 0, k = 1
{11,4} x = 1, i = 0 j = 0 2
{11,4} x = 1, i = 0 j = 1
{10,5} x = 2, i = 0
{11,4} x = 1, i = 1 j = 1 2
{10,5} x = 2, i = 1
{10} x = 2, i = 2
{12,4} x = 1, i = 0 j = 0, k = 1
{12,4} x = 2, i = 0 j = 1, k = 1
{11,5} x = 1, i = 0 j = 1, k = 1
{11} x = 2, i = 0 j = 2
{11} x = 2, i = 1 j = 2
{11} x = 2, i = 2 j = 2
{12,5} x = 2, i = 0 j = 0, k = 1
{12,5} x = 2, i = 0 j = 1, k = 1
{12} x = 2, i = 0 j = 1, k = 1
{12} x = 2, i = 0 j = 2, k = 1
{12} x = 2, i = 1 j = 2, k = 1
{12} x = 2, i = 2 j = 2, k = 1

(b) State space of Figure 2.8a, using tuples of line numbers as locations (States with bold numbers are not fully expanded)

Figure 2.8: State space exploration based on naive interleaving semantics

2.4 Analysis of Multi-Threaded Programs

Concurrent software verification algorithms also fall into two categories, depending on the execution semantics they employ. The *interleaving* semantics uses overlapping traces of the threads in the concurrent program to explain how it executes. This approach, when used naively, does not scale well due to the large number of possible unique executions. This is partially solved by utilizing e.g. *partial order reduction* (POR) [26], which significantly reduces the number of necessarily explored executions. In contrast, the *declarative* semantics of concurrent program executions uses partial orders to explain a specific execution. The *candidate executions* introduced above are examples to such a semantics, as the *po* and *rf* relations partially order the statements and yield a well-defined single execution of the program [7]. This *declarative* semantics has been shown to perform better on weak memory than the *interleaving* semantics over sequential memory, when implemented in model checking algorithms [6].

2.4.1 Interleaving Semantics

The naive way of dealing with concurrency is to strictly follow the definition of asynchronous systems, i.e. any of the threads may execute at any point in time, meaning every possible total order of the instruction has to be explored. This technique employs *naive interleaving semantics.*

Consider the example in Figure 2.8a (note that based on the C standard, a global variable will be initialized to 0, if no explicit value is assigned [31]). The main thread starts a worker thread, which writes two values to x, while the value of x is read twice. As the value of x increases monotonically, we assert that the latter read's value shall be at least big as the former's. We store this in a boolean k. If we tried to enumerate all executions based on the naive interleaving semantics, we would get the state space in Figure 2.8b – there are 10 different executions that can take place, as the $3+2$ operations between the start and end of the worker thread give rise to 10 total orders. However, if we examine the outcomes of the different executions, the set of possible end values is way smaller: x is always 2, j is always bigger than i and k is therefore always 1. Even though i and j can

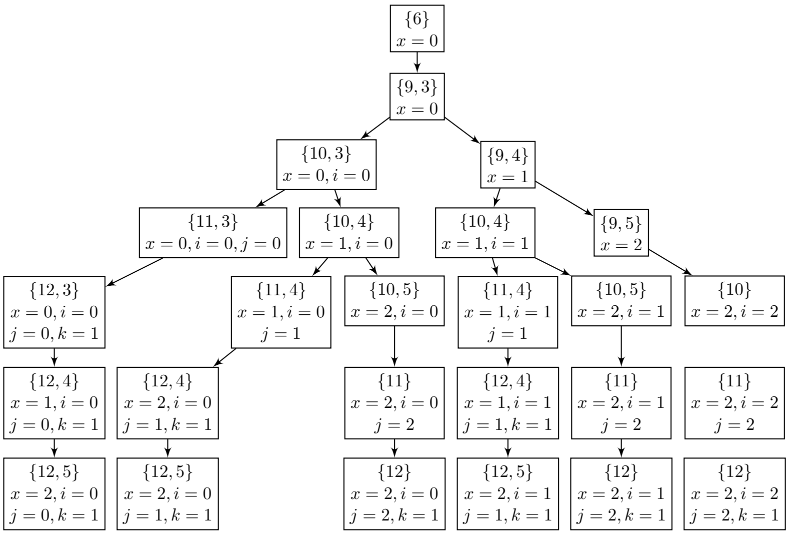


Figure 2.9: POR-based state space of Figure 2.8a, using tuples of line numbers as locations (main thread executes first)

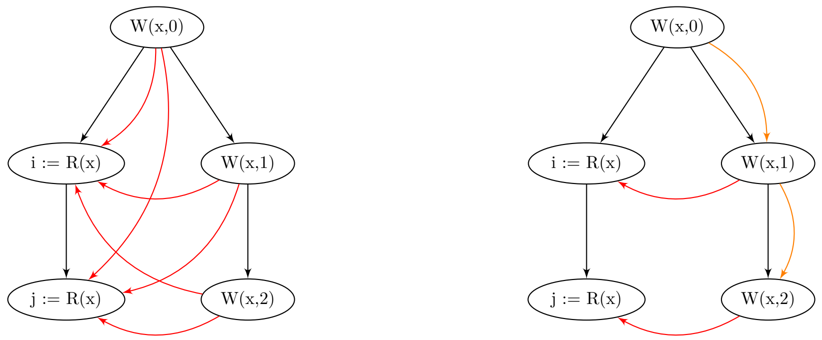
take up any one of the values from the set $\{(0,0),(0,1),(0,2),(1,1),(1,2),(2,2)\}$, this is still only 6 possible outcomes instead of the 10.

To explain this behavior, let us examine the three branches of the state space tree marked with patterns. In these executions, the values to i and j were decided early on, as the main thread progressed more than the worker thread. In theory, this would eliminate the need for further analysis, as any further action on either thread is independent of the other – x will be increased further, but no operation will use its value; and k is never used in any of the global memory accesses. However, the naive interleaving approach had to explore these subexecutions as well because it had no way of determining which operations would influence the final outcome and which ones would not.

An intuitive step to take is to discover independent pairs of transitions in the model, and forbid the exploration of both total orders. This technique is called *partial order reduction* (POR) [26], and it is widely used in the verification of concurrent systems (Even though there is a specialized version of POR called *dynamic partial order reduction* (DPOR) [24], which is shown to be more optimal, introducing and implementing that algorithm falls outside the scope of this work).

Consider the same input program in Figure 2.8a. If we apply POR based on a global-local partitioning of the transitions, where every transition touching a global memory object is considered dependent on each other, we get the state space in Figure 2.9. In this case, state space exploration was *optimal*, as each explored total order yielded a different outcome, and no possible outcome was left out.

Even though the presented example showed the POR algorithm to be *optimal*, this is not the case in every input program. For example, there could be another, totally independent y global variable, and two threads performing the same operations over y as over x – in this case, all total orders would have to be explored among accesses to the global variables as well, which would yield a suboptimal exploration. There are techniques mitigating this behavior (e.g. in [1], the authors have shown that there is an optimal DPOR, and also gave an example for such an algorithm), but the presented naive POR cannot deal with this problem.



(a) Abstract execution graph of Figure 2.8a **(b)** A concrete execution from 2.10a

Figure 2.10: Program verification based on declarative semantics

2.4.2 Declarative Semantics

To showcase the differences between the interleaving and declarative semantics, let us look at the same problem in Figure 2.8a. To generate the declarative state space of the program, an *abstract execution graph* is necessary – which is similar to a candidate execution, but Reads are not limited to a single *rf*-edge, and only *po*- and *rf*-edges are present. The semantics of such a construct is the following: all executions are *observable* which stem from a consistent candidate execution that is a subgraph of the abstract execution graph, and for which a satisfying total *co* order exists over same-location writes. Such an abstract execution graph can be seen in Figure 2.10a, and an example concrete execution is shown in Figure 2.10b.

Note that in this case, state space exploration is optimal by default: after the abstract execution graph is built (which is interleaving-free, and therefore can be built in a single-pass over the operations in the program), only different, and consistent execution graphs are enumerated.

2.4.3 Multi-Threaded CFA

In order to verify multi-threaded programs, a formalism supporting multi-threading is also necessary. As with single-threaded programs, a formalism encoding control-flow in the form of a program counter-like construct is advantageous – therefore, the basis of the chosen formalism is still a control flow automaton (CFA) [13]. However, this formalism has been extended in the following ways, giving rise to the eXtended Control Flow Automata (XCFA):

Definition 3. *eXtended Control Flow Automata (XCFA)*
An XCFA is a tuple $XCFA = (V_g, P)$, where:

- *V_g: Global variables*
- *P: Processes, which are tuples $P = (V_p, F, f_0)$, where:*
  - *V_p: Thread-local variables*
  - *F: Procedures, which are tuples $F = (V_l, CFA, P_{in}, P_{out})$, where:*
    - *V_l: Local variables*
    - *CFA: A conventional CFA (which can use $V_g \cup V_p \cup V_l$ as variables), extended with the following operations:*
      - Function call*s*
      - Start thread *and* join thread
      - Atomic begin *and* atomic end
      - Store*,* Load *and* Fence
    - *$P_{in} \subseteq V_l$: Input parameter variables, which are assigned when the function is called*
    - *$P_{out} \subseteq V_l$: Output parameter variables, which are returned when the function returns*
  - *$f_0 \in F$: The main function of the process (execution starts here)*

Semantically, an XCFA can either be static *or* dynamic*. In the former case, only the starting set of processes can execute. In the latter case, the* start thread *and* join thread *operations manipulate the set of enabled processes. In both cases, the processes fire* asynchronously*.* ▪

Note that variables can either be assigned via normal assignments (as in a conventional CFA), or through *store* and *load* operations. In the context of this work, I assume total sequentality for assignments, and only apply the memory model for the analysis of the designated memory access instructions.

Chapter 3

State of the Art

My contributions presented in this thesis mainly concentrate on the efficient handling of parallelism, handling both the sequential and weakly ordered case. In this chapter, I introduce the state-of-the-art tools for handling concurrency in these cases. For the sequential case, I concentrate on the algorithms employing a form of CEGAR, as that falls the closest to the scope of my work; while for the weakly ordered case I introduce the most advanced *bounded* algorithms, due to the lack of a general solution covering infinite-state programs[1].

3.1 Sequentially Ordered Concurrency

Most model checkers capable of verifying concurrent programs that employ a form of abstraction-refinement techniques use a pre-processing step to determine atomically executable (i.e. thread-local) transitions and global operations. Then, every interleaving is explored among these transitions when calculating abstract successor states. Note that these solutions employ a crude version of POR, with no dynamic element.

VVT [27] uses an LLVM-based front-end to verify C programs, which determines blocks of instructions that can execute atomically without interfering with the allowed set of outcomes. Then, these blocks serve as the individual transitions in a large-block encoded CEGAR loop.

In comparison, CPAchecker [11] uses a pre-processing step on the edges of the CFAs to determine thread-local and global operations, then uses a similar large-block encoded CEGAR loop. In addition, it uses several further optimization steps to be more performant; such as *waitlist ordering* and *partitioning of abstract states*.

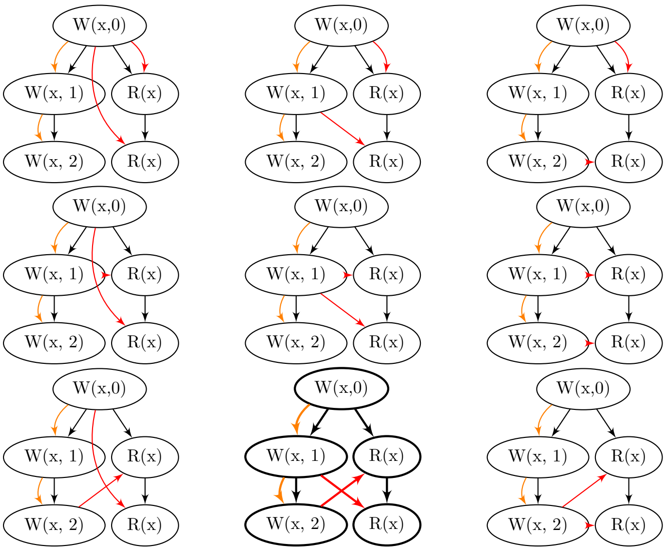
3.2 Weakly Ordered Concurrency

The algorithm presented in this report has been heavily influenced by three existing tools, namely, Herd [7], Dartagnan [21] and rcmc [32]. In this section, I will present the approaches employed by these tools.

[1] At the time of writing this report, I have no knowledge of any approach that utilizes any form of abstract reasoning over declarative semantics.

```
ExampleArch same−loc
{ x=0; }
P0              | P1
MOV [x],$1   | MOV R0,[x]
MOV [x],$2   | MOV R1,[x]
exists  (1:R0=2 /\ 1:R1=1)
```

(a) Litmus test *same-loc*



(b) Candidate executions with fixed *co*- and *po*-orders

Figure 3.1: Herd's input litmus test and the generated candidate executions

3.2.1 Herd

Herd is a memory model simulator [7]. It expects a memory model specification written in the CAT language [8] and a litmus test. Litmus tests are small, assembly-level concurrent programs that include accesses to global memory, as well as constraints on local variables. Litmus tests are widely used to specify guarantees of memory models, e.g. Intel most notably only uses such programs as the specification of the X86 memory model [17]. For example, a memory model rule might forbid the reordering of same-location accesses. The corresponding litmus test in Figure 3.1a has two threads: a producer with two consecutive *Write* events, and a consumer with two consecutive *Read* events, all to the same location. Any execution is forbidden where the consumer observes the two written values in reverse order, i.e. the value of $R1$ is 1 from the earlier *Write*, while the value of $R0$ is 2 from the second *Write*. This outcome is only possible when either the *Reads* or the *Writes* have been reordered.

For a given memory model and litmus test, the question is whether the forbidden behavior is *observable* on the target architecture. To answer this question, Herd will first generate all candidate executions of the litmus test. This is done in an enumerative way: for each primitive relation every semantically correct combination will be explored [7], as seen in Figure 3.1b. After enumeration, the candidate executions are filtered based on whether they are consistent with the specified memory model. If any consistent execution graph of the litmus test produces the forbidden outcome, the specified behavior is *observable* and the litmus test fails. For the example in Figure 3.1a, there is one such candidate execution (given fixed *co*- and *po*-orders), highlighted in bold in Figure 3.1b.

The example in Figure 3.1 also shows that the number of *candidate executions* is generally much higher than the number of *consistent execution graphs*. Given a memory model rule that forbids the reordering of same-location accesses, only 6 *candidate executions* are *consistent* out of the 9 in Figure 3.1b given the fixed *co*-order. However, the total number of *candidate executions* are much higher. The *Write* events can be ordered by any of their permutations, as the algorithm cannot assume that any of those partial orders is inconsistent without taking the memory model into account. However, given the forbidden

```
int x = 0;
void thr1(void* _) {
    x = 1;
    x = 2;
}
void thr2(void* _) {
    int r0 = x;
}
```

(a) Example input

```
int x = 0;
int y = 0;
void thr1(void* _) {
    y = 1;
    x = 1;
}
void thr2(void* _) {
    int r0 = x;
    int r1 = y;
    assert (!( r0 == 1  &&
               r1 == 0 ));
}
```

(b) Input causing false positives over SC

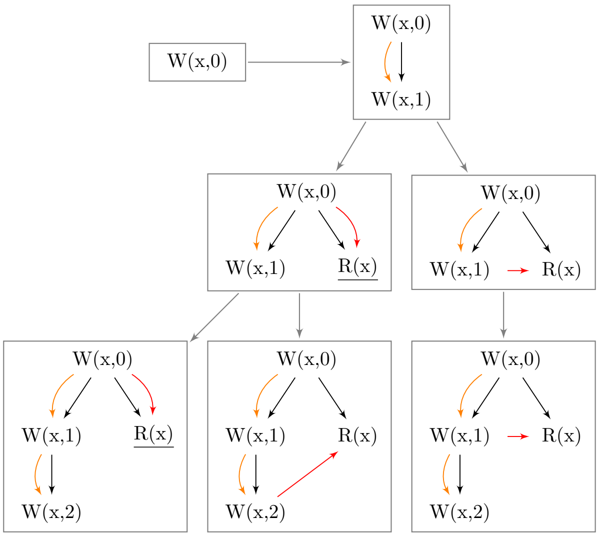
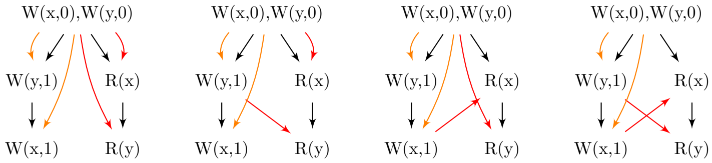


Figure 3.3: Exploring the program in Figure 3.2a



(a) $r0 = 0$ $r1 = 0$ **(b)** $r0 = 0$ $r1 = 1$ **(c)** $\underline{r0 = 1}$ $\underline{r1 = 0}$ **(d)** $r0 = 1$ $r1 = 1$

Figure 3.4: Execution graphs generated by RCMC

same-location reordering, only the one in Figure 3.1b is consistent with the memory model. This puts the number of all candidate executions at $3! * 9 = 54$, and the percentage of consistent execution graphs at 11.1%. For larger programs, this ratio is even smaller, as the number of unnecessary partial orders becomes higher. This observation is also established by the practical evaluation of the RCMC tool, which only generates consistent execution graphs [32].

The goal of HERD is not general program verification, but rather architectural verification. Litmus tests are by definition small programs, and therefore it is unnecessary to optimize the algorithm in HERD for input size. For anything larger than an ordinary litmus test, HERD will most likely time out while enumerating the candidate executions. This prompted the development of smarter *candidate execution* generation, such as RCMC [32].

3.2.2 Rcmc

The novelty of RCMC is its smart exploration algorithm. In each step of its algorithm, RCMC will only generate consistent execution graphs, and no execution graph is ever explored twice. The implemented *stateless* model checking algorithm receives a concurrent C/C++ program with optional assertions, and enumerates all consistent executions as its output. If in any of the execution graphs the assertion is violated, or a non-atomic concurrent access occurs, the tool reports the program as *unsafe* immediately. Note that the memory model is *not* an input, as the C/C++ concurrency model (as formalized in the repaired RC11 memory model [34]) is hard-coded into the algorithm. This significantly reduces the applicability of the tool for custom architectures and potentially yields false positive results.

Consider the input program in Figure 3.2a. Two threads are executing concurrently, one writing to memory and another reading from it. Note that atomic accesses have been replaced with regular assignments for the sake of brevity. For the sake of this example, *relaxed* accesses can be assumed.

The execution of the algorithm can be seen in Figure 3.3. RCMC will start by recording the initial values in a node, then one-by-one adding the statements of the program. Any time a *Read* event is added, each subexecution is explored where *Read* receives a value from any existing *Write* event. In exactly one of the subexecutions, *Read* remains *revisitable*, i.e. a later *Write* can provide it a value. *Revisitable* nodes are underlined in the example. Each time a *Write* event is added, each subexecution is explored where the newly added *Write* provides a value for any combination of currently *revisitable Reads.* Furthermore, each consistent *co*-order is also explored, but in Figure 3.3, this is deterministic due to *po.* In the example in Figure 3.3, the order of recorded nodes alternates between the two threads, starting with a *Write* event to x.

The novelty behind the algorithm is to use *revisitable* nodes to mark a single subexecution where a given *Read* event's value is not final. If more than one such subexecution existed, adding a subsequent *Write* event could generate redundant subexplorations [32].

Consider the input program in Figure 3.2b and the generated execution graphs in Figure 3.4. Depending on the received values in the second thread, an assertion failure can occur. The condition of the assertion means that the second *Write* event executed *before* the previous one. This is observable in Figure 3.4c. Considering C/C++ can generally run on any architecture, one cannot assume that the hardware is not e.g. sequentially consistent (SC). SC guarantees that no statements will be reordered, and therefore the assertion is never violated. RCMC, however, reports it as unsafe because C/C++ does not guarantee this assumption, and therefore this can be categorized as a false positive result. This is not a shortcoming of the algorithm itself, but rather of the approach: one cannot assume that the memory model of a programming language is independent of the target architecture [41]. Such a false result might shadow actual problems in the input program, and is therefore inherently unsafe.

Another problem of RCMC is the suboptimal exploration of executions when multiple threads write the same variable. As noted above, exploring artificially generated *co*-orders is detrimental to the number of execution graphs. In the worst case, each new *Write* event will effectively multiply the number of subexecutions by the factor of existing *Write* events to the same variable, even if only one thread observes the value. In this case, enumerating all execution graphs where this *Read* reads from a different *Write* would suffice, yet this is multiplied by the factorial of the number of *Write* events, as seen in Figures 2.7d and 2.7e.

3.2.2.1 GenMC

As an imprevement to RCMC, GENMC [33] promises to deliver a memory model-aware, stateless model checking algorithm. This enables the verification of software running on custom memory models with an approach very close to that of RCMC. However, the boundedness of the algorithm is still a considerable drawback (as it is with RCMC as well).

	Software verification	Parametric memory model	Scalable	Optimal execution enumeration	Handle unbounded state spaces
Herd	✗	✓	✗	✗	✗
rcmc	✓	✗	✓	✓*	✗
Dartagnan	✓	✓	✓	N/A	✗

Figure 3.5: Comparison of related verification tools

3.2.3 Dartagnan

Most of the concerns above are addressed by Dartagnan, a bounded model checker that uses memory models as modules [21, 25]. Dartagnan expects a concurrent program and a memory model as inputs, and using the conjunction of SMT-encoded expressions determines whether an unsafe state is reachable within a given bound. To achieve this, Dartagnan unrolls and encodes the concurrent program as an SMT-expression; encodes the unsafe state as another SMT-expression; and encodes the input memory model as an SMT-expression. If the conjunction of the expressions above is *satisfiable*, the unsafe state is *reachable* and therefore the concurrent program is *unsafe*.

Dartagnan is a software verification tool, complete with an integration to Smack [37], an LLVM-based program transformation tool that allows Dartagnan to work on formal models rather than source-level programs. The gap between the higher-level LLVM-IR and the ISA of the target architecture is bridged by using *compiler mappings* for translating e.g. memory ordering primitives [21]. This is a conventional procedure [41], but special attention has to be paid to ensure the compiler mappings represent an actual compiler's behavior that might be used to compile the examined program later on.

In comparison with Herd and rcmc, Dartagnan (and its companion tool, Porthos [21]) is not capable of enumerating consistent executions. Even though as a reachability checker, Dartagnan is not expected to provide this feature, it could be useful to provide a way to use the tools embedded into other verification algorithms for handling concurrent parts of an otherwise independent set of threads. In this case, an unsafe state might not only be dependent on the concurrent parts of the program, and therefore Dartagnan could not handle it on its own.

Evaluating the five criteria in Figure 3.5 reveals that none of the tools fulfil every aspect. Herd is not capable of software verification due to scaling issues caused by its suboptimal execution enumeration approach. rcmc is not parametric and therefore only C/C++ guarantees are assumed, and it uses artificially generated *co*-orders which increase the number of explored execution graphs. Dartagnan cannot enumerate consistent execution graphs. Furthermore, neither solution can handle unbounded programs.

Chapter 4

SMT-Based Verification over Declarative Semantics

In most memory model-aware verification tools (such as Herd [7] and Rcmc [32]), the consistent candidate executions are create *generatively*, i.e., by starting from an empty execution and adding events and locations until a complete candidate execution is created. If reachability of some error state is required, their approach is to enumerate all consistent executions, and see whether any include the given state. In contrast, SMT-based verification tools such as Dartagnan use an SMT-solver instead, and encode the unsafe state explicitly in the query. While this introduces the complexity of dealing with an SMT-solver, the acquired benefit of answering reachability queries directly make it worthwile.

In this chapter, I introduce my implementation of an SMT-based algorithm for verifying reachability over declarative semantics, influenced greatly by Dartagnan. Throughout this chapter, all my work can be assumed to have taken place in Theta [40], a configurable and modular verification framework I am a developer of[1]. However, the implementation is subject to change in the future, and therefore I do not consider it an artifact of this thesis.

4.1 Representing the Input Program

Theta includes support for a diverse set of input formalisms, such as Extended Timed Automata (XTA), Symbolic Transition Systems (STS) and Control Flow Automata (CFA) [28]. While all of these have roughly the same expressive power, they lack certain features that make it hard to represent parallelism. Most notably, none of these provide any information on the structural composition of the source model. Therefore, I opted to implement and use a version of the previously introduced eXtended Control Flow Automata (XCFA) formalism. In this section, I introduce this implementation an elaborate on its features.

Theta achieves its modularity by clearly abstracting formalism-specific parts of the verification algorithms away from the core analysis modules [22, 28]. Furthermore, for each formalism, the code is divided into three main packages:

- the core formalism, containing the classes used for the in-memory representation of the model;

[1] `https://github.com/ftsrg/theta`

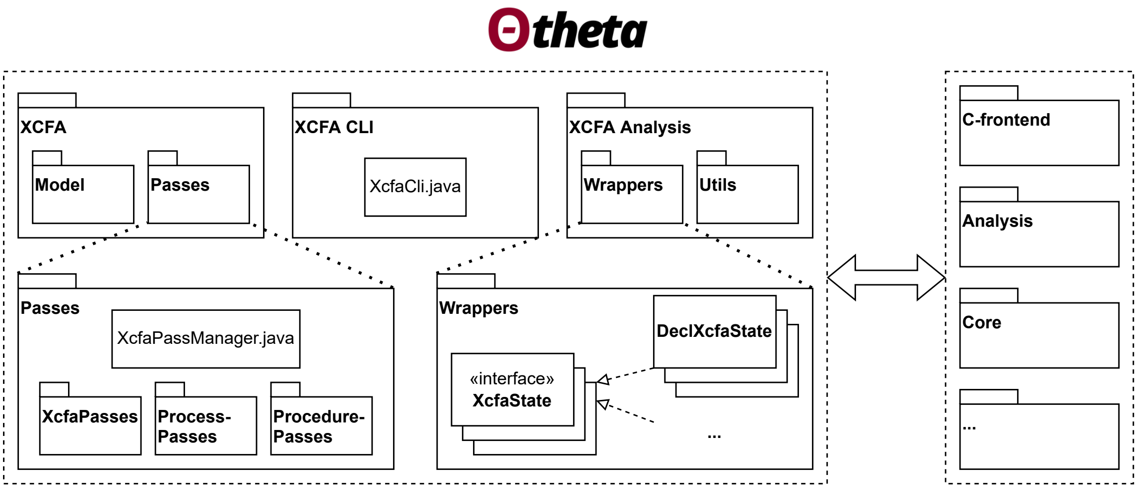


Figure 4.1: High-Level Architecture of THETA, focusing on the XCFA subproject

- the analysis package, containing the wrappers that provide access to the formalism for the higher-level analyses; and
- the CLI package, which provides an executable frontend to running THETA's algorithms on the formalism.

To follow this pattern, the XCFA subproject is also divided into the three corresponding packages: `xcfa`, `xcfa-analysis` and `xcfa-cli`, as seen in Figure 4.1. The main entry point is the `XcfaCli` class from the `xcfa-cli` package, which provides the configuration options that govern the verification process. This will instantiate the necessary wrappers from the `xcfa-analysis` package, and parse the input file to create an in-memory XCFA from the `xcfa` package using the THETA's C-Frontend implementation (more on this in Section 4.1.1). Currently, the XCFA formalism does not define a DSL, as input is expected to only come as C programs when using this tool. (To use another source, such as litmus tests, one can use the XCFA infrastructure as a library – see Section 4.1.2).

An XCFA consists of processes, which in turn consist of procedures. Reflecting this, the XCFA in-memory representation found under the `xcfa/model` package is also multi-layered, consisting of the elements seen in Figure 4.2. Note that some relations are left out from this diagram, as to keep the figure easy to interpret.

One of the features of an XFCA is that it can be *static* or *dynamic*, i.e., it can start all processes at startup or spawn new ones during execution. In the context of this work, both configuration options can be observed, as most programs follow the *dynamic* option, while the models of distributed systems rarely need to spin up new devices while working, and therefore it can be assumed that all processes are known before launch.

The `xcfa-analysis` package is unique among the other analysis packages of THETA, as it includes more than one set of wrappers. This was necessary to provide different mappings for single- and multi-threaded algorithms, as well as interleaving- and declarative semantics-based implementations for the latter.

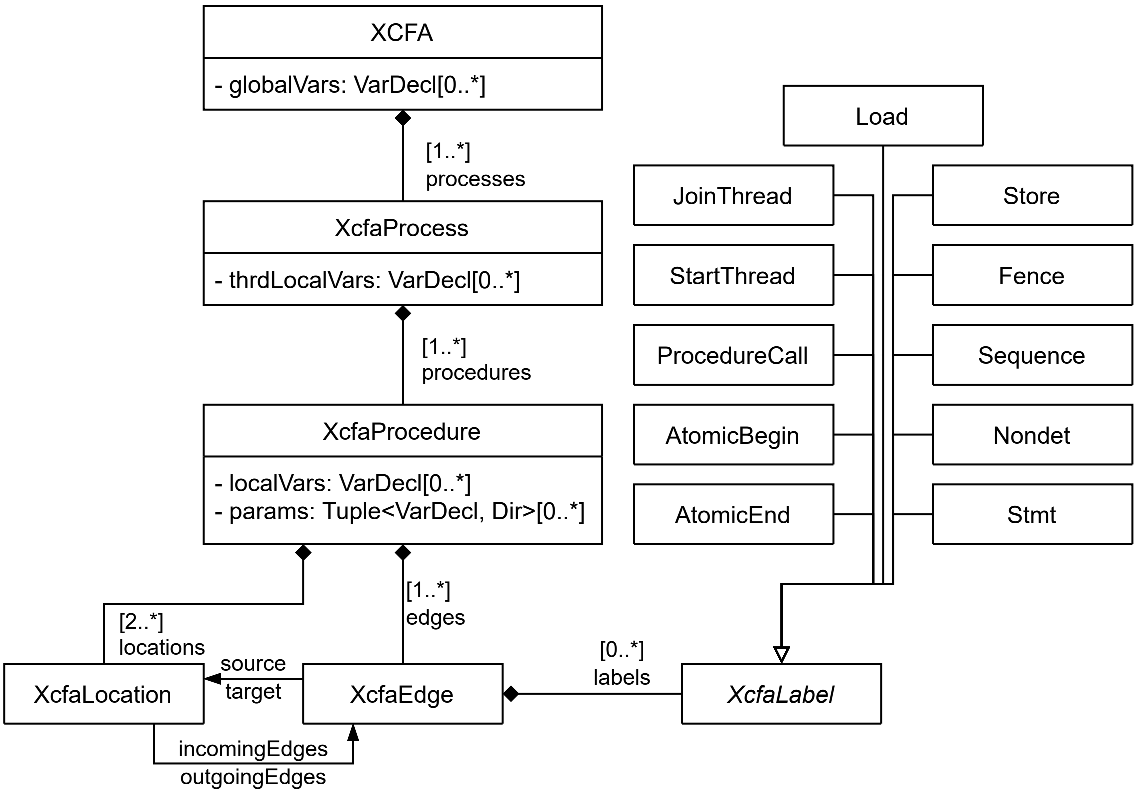


Figure 4.2: Class diagram of a subset of relations inside the XCFA representation

4.1.1 Parsing C Programs

Currently, the only fully supported frontend to the XCFA subproject is the C frontend. This is an ANTLR[2]-based parser that supports C programs in the format of SV-COMP[3], a verification competition for C verifiers. This C frontend creates an intermediary representation (*not* an XCFA, as to be formalism-independent), which can be further transformed into any formalism, such as the XCFA. To be as efficient as possible, this process follows the workflow outlined in our paper on efficient frontends for C verification [9], i.e., it consists of two steps: first the raw C program is transformed into a verbose XCFA, after which several optimization- and elimination passes can transform it into its final form (governed by `XcfaPassManager`). This approach has been validated by both our results on this year's SV-COMP [4] and the tests in our FormaliSE paper [9].

4.1.2 Parsing Litmus Tests

A new and experimental feature of `Theta` is to use programs in the litmus format of `Herd` [7] as input, through the XCFA formalism. The implementation can be found in the `litmus2xcfa` package, and currently it only supports the `AArch64`-flavor of the litmus format, as this architecture provides the most interesting memory model out of the common ones (x86, arm32 and power are too simple, the linux kernel memory model is out of the scope of this implementation and RC11 does not have enough associated litmus tests used by other verification tools). The current implementation is based on the ANTLR-

[2] `https://www.antlr.org/`

[3] `https://sv-comp.sosy-lab.org/`

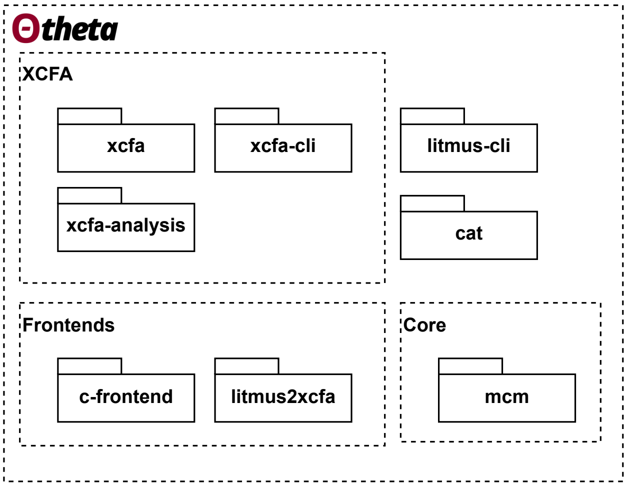


Figure 4.3: Summary of new features in `Theta`

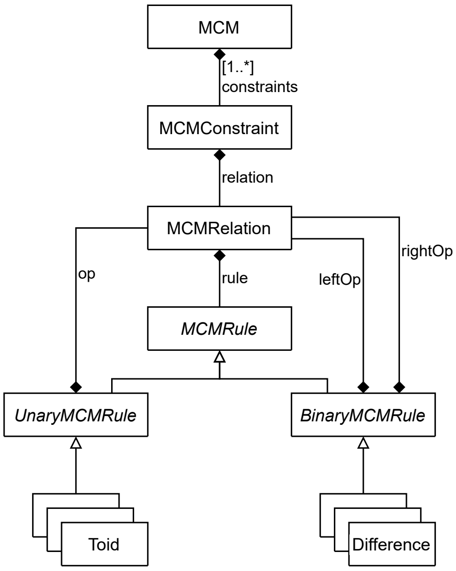


Figure 4.4: MCM representation

grammars provided by `Dartagnan`'s public repository[4], and a set of custom visitors are used to transform it into an XCFA.

To go along with this frontend, a new CLI tool has also been added to THETA, called `litmus-cli`, to be used as the entry point to litmus test verification.

4.2 Representing the Memory Model

The second input to the verification workflow is the memory model. It is assumed to be provided in the Cat format, and I used the ANTLR grammar used by `Dartagnan` as the basis to this process. First, I extended this grammar by further concepts used by Cat (such as functions, procedures and file inclusion) and I created the in-memory representation for the memory model's elements. To not constrain myself to just using Cat in the future, I provide an abstraction over its syntax to only include mathematical elements such as sets and relations – this is to follow the patterns seen in other parts of THETA. Therefore, I created a new analysis package called `mcm` inside THETA's core subproject, and I placed the necessary classes there (see Figure 4.4). This is also the place where I created the analysis package to be introduced in Section 4.3.

For a summary of the new features in `Theta`, see Figure 4.3: the `XCFA` project contains the three subprojects used for program verification via the XCFA formalism, which use the `c-frontend` package to parse C programs into XCFA models; the `core` subproject has been extended with a new package called `mcm` which is responsible for storing memory models and verifying concurrent- and distributed systems over declarative semantics; and the two standalone packages `litmus-cli` and `cat` can be used to parse (with the help of `litmus2xcfa`) and run (via the `mcm` package) analyses over litmus tests.

[4] `https://github.com/hernanponcedeleon/Dat3M/`

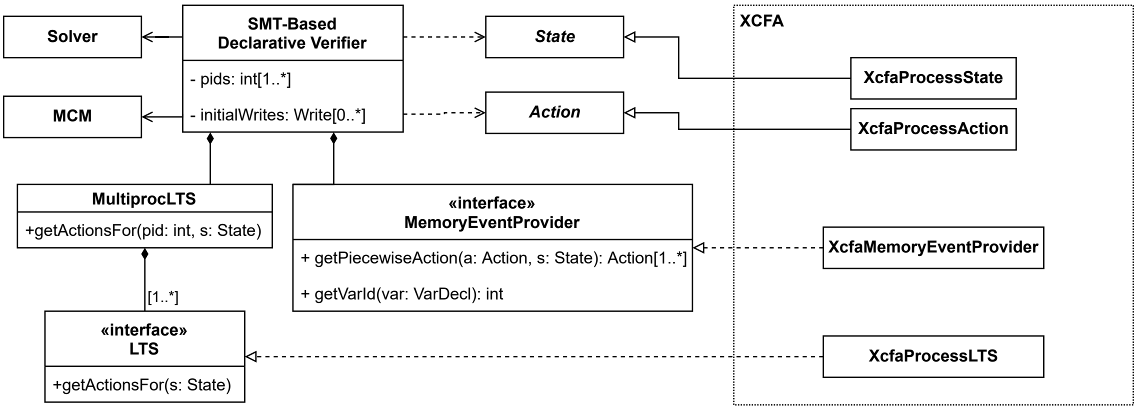

Solver
SMT-Based Declarative Verifier
- pids: int[1..*]
- initialWrites: Write[0..*]
State
Action
MCM
XCFA
XcfaProcessState
XcfaProcessAction
MultiprocLTS
+getActionsFor(pid: int, s: State)
«interface» MemoryEventProvider
+ getPiecewiseAction(a: Action, s: State): Action[1..*]
+ getVarId(var: VarDecl): int
XcfaMemoryEventProvider
[1..*]
«interface» LTS
+getActionsFor(s: State)
XcfaProcessLTS

Figure 4.5: Class diagram of the necessary elements used in the algorithm

4.3 Implementing the Core Algorithm

Having parsed and stored a memory model and a program, the goals regarding their verification are the following:

1. If an error property is given, find a *consistent* execution of the program that satisfies the error property. If found, flag that as a *counterexample*, and if not, provide a proof of safety.
2. If no error property is given, enumerate all *consistent* execution graphs.

Note that in this chapter, the input program has to be constrained *not* to include loops. This requirement is set to avoid infinite loops over paths in the program, as the algorithm does not keep track of the current state and therefore all transitions are always enabled. This is solved by the algorithm proposed in Chapter 6.

4.3.1 Overview of the Approach

The goal is to create a single SMT-formula Φ, which is only satisfiable by consistent executions (if an unsafe property is also given, then it should satisfy that as well). This will come from the following sources:

1. Control and data flow ϕ_{CDF};
2. Memory model ϕ_{MCM}; and
3. The unsafe property ϕ_{ERR}.

Then, using $\Phi := \Phi_{CDF} \wedge \Phi_{MCM} \wedge \Phi_{ERR}$, an SMT-solver can either find a satisfying model for Φ and therefore produce a counterexample, or report the program as *safe.*

To get the above mentioned encodings, an overview of the necessary elements can be seen in Figure 4.5.

4.3.2 Encoding Control and Data Flow

To encode the control and data flow of the input program, the workflow consists of three main steps:

1. Explore the program's memory events and branches
2. Encode the branch conditions as SMT-formulae
3. Constrain the events to forbid taking multiple branches in a single thread, and enforce causality in the program order

Exploring the program's memory events take use of the two interfaces connected to the main verification class, namely `MultiprocLTS` and `MemoryEventProvider`. Their respective tasks are:

- Provide a list of outgoing transitions from a state (which is analogous with a location in this context) on a given process,
- Provide a slicing of an action (i.e., a transition) into a sequence of sub-actions that are either *normal* actions or *memory events*, and map each variable to an index.

Using these elements, the basic control flow can be established in the input program – as each memory event is discovered via querying the `MemoryEventProvider` for the transitions supplied by the `MultiprocLTS`. This results in a tree of events which will provide the *po*-relation.

Next, the control flow tree has to be encoded into the Φ_{CDF} SMT-formula. To do this, for each event i a new constant t_i is introduced, which must be *true* to include the corresponding event in the final execution (the *in-trace* unary relation). To achieve well-formedness, the following constraints must be placed over the values of t:

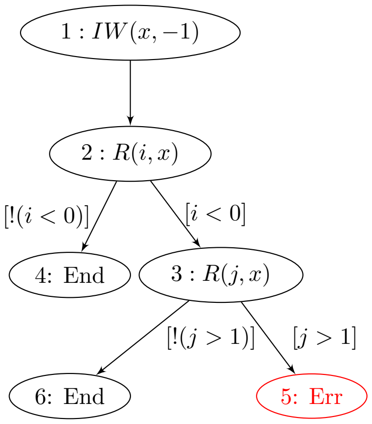
1. For each initial write, t_i should be true.
2. For each t_i, all previous events must be *true* for t_i to possibly be true.
3. For each t_i, exactly one successor must be *true*
   (a) At most one because a choice must be made about the successor branch; and
   (b) At least one because we would like to maximize the executions

Furthermore, we must deal with the labels on the edges of the program. As we use an SMT-encoding, first we have to create a static single assignment (SSA) form, which will have indexed constants instead of the variables on the edges; then constrain the resulting formulae to be *true* when their transition's target state is chosen to be in-trace.

The resulting workflow is demonstrated in the example in Figure 4.6. First, the memory events and branches of the program in Figure 4.6a are explored and thus Figure 4.6b is created (given an arbitrary initial write, here -1), which has 6 numbered events with a single error state. This can be directly encoded into SMT as seen in Figure 4.6, along the four constraint types introduced above. Note that reads and writes are *not* encoded via Φ_{CDF}, as that will be taken care of by the memory model. Also note that a general solution could not use the `xor` function to encode the successor constraint, as more than two successors are possible, and should use a combination of assertions ensuring that exactly one successor is selected.

```
if(x < 0) {
    if(x > 1) {
        reach_error();
    }
}
return;
```

(a) Example program



(b) Memory events and *po*

```
(assert t1)
```

(c) Initial writes

```
(assert (=> t1 t2))
(assert (=> t2
            (xor t3 t4)))
(assert (=> t3
            (xor t6 t5)))
```

(d) Successors

```
(assert (=> t2 t1))
(assert (=> t3 t2))
(assert (=> t4 t2))
(assert (=> t5 t3))
(assert (=> t6 t3))
```

(e) Ancestors

```
(assert (=> t3 (< i 0)))
(assert (=> t4
            (not (< i 0))))
(assert (=> t5 (> j 1)))
(assert (=> t6
            (not (> j 1))))
```

(f) Assumptions

(g) Φ_{CDF} encoding (SMT-LIB format)

Figure 4.6: Exploring the control and data flow of a single process

4.3.3 Encoding the Memory Model

Having encoded the control and data flow of the program, the accompanying memory model must be encoded as well. Mathematically, the memory model serves as the meta-model for the candidate execution graphs – therefore it can be encoded as constraints over a labelled graph. To achieve this, a boolean constant can be introduced for each pair of events in the control flow graph for every defined relation. Then, by imposing constraints stemming from the memory model over these constants, the memory model can be encoded over the specific program.

Consider the memory model in Figure 4.7a. It defines a simple rule, stating that a read event cannot read from a po-previous source. This introduces a relation `rf-same`, and a constraint `no-int-read`. Therefore, $6 * 6 = 36$ new constants are added to the SMT-query, each corresponding to a tuple $\{source, target, relation\}$. Then, their values can be constrained by the rules in the memory model such as `no-int-read`: in Figure 4.7b and 4.7c a few assertions can be seen that demonstrate this process. The conjunction of such rules will give the memory model encoding Φ_{MCM}.

Note that based on the architecture of the memory model, a single rule can be broken up into multiple sub-rules. For example, the rule

$$let\ rf\text{-}sibling = (rf\hat{}\text{-}1; rf)\backslash id \tag{4.1}$$

can be broken up into

$$\begin{gathered} let\ rule1 = (rf\hat{}\text{-}1; rf) \\ let\ rf\text{-}sibling = rule1\backslash id. \end{gathered} \tag{4.2}$$

```
let  rf—same = rf & po
empty rf—same as no—int—read
```

(a) Simple memory model

```
(assert  ( iff  rf—same_1_1
            (and rf_1_1 po_1_1)))
(assert  ( iff  rf—same_1_2
            (and rf_1_2 po_1_2)))
...
```

(b) `rf-same` encoding

```
(assert  (not (or
            rf—same_1_1
            rf—same_1_2
            ...
            rf—same_6_6)))
```

(c) `no-int-read` rule

Figure 4.7: Encoding the memory model

While this effectively multiplies the number of relations (and hence, constants) in the SMT-formula, the rules and hence the implementation can be much simpler.

Note that the built-in relations *po*, *int*, *ext*, etc. need to be explicitly encoded. Beside *rf* and *co*, all built-in relations are clearly defined after exploration, and therefore their values can directly be encoded into Φ_{MCM}. It is important to define all relation instances as either *true* or *false*, because their values are grounded in the semantics of the program, rather than that of the memory model – and the SMT-solver should not receive an ambiguous input in this context.

The last element of the memory model encoding is the equality constraint among reads and writes of the same *rf* edge. This will enforce that if a read-from edge is chosen to be included in the execution graph, the read actually receives its value from the corresponding write.

4.3.4 Enumerating Solutions

To enumerate the solutions to the verification problem, first the error property Φ_{Err} has to be encoded. This is, however, easy in the case of reachability: if the property is some data state then encode it directly, and if it is location reachability, assert that the violating location is included in the trace (t_i is *true*). With this, Φ can be constructed, and the SMT-solver can be queried. Then, while there is a satisfying model to Φ, new solutions can be found by adding the negated conjunction of all *rf*-relations – which will surely produce new consistent executions, until such constructs exist.

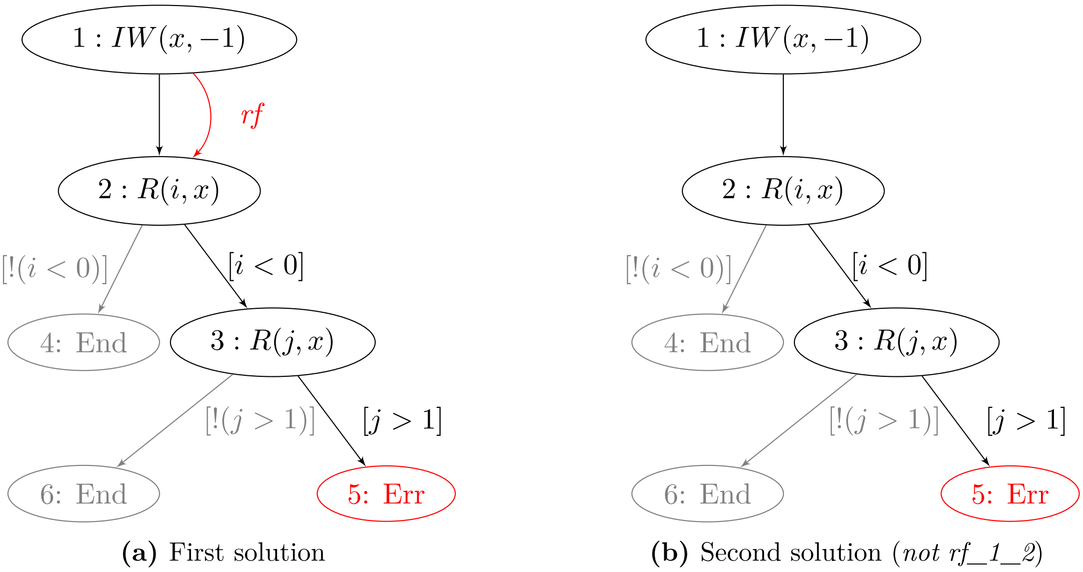
In the case of the program in Figure 4.6a and the empty memory model, some satisfying solutions can be found in Figure 4.8.

4.4 Encoding Well-Formedness Constraints

Notice the problem with Figure 4.8. There are reads which read from *nothing*, i.e., are equivalent to *havoc*s. This is due to the lack of constraints over the well-formedness of the generated relations *rf* and *co*. This is problematic, because their semantics are lost during the transformation from memory model to execution graph – and the solution is to introduce a standard library of rules and constraints that always govern their behavior.

A commonly used relation will be the `both-in-trace` relation, defined by the following rule:

$$let\ both\text{-}in\text{-}trace = T * T \tag{4.3}$$



(a) First solution

(b) Second solution (*not rf_1_2*)

Figure 4.8: Consistent executions of Figure 4.6a over Figure 4.7a (grayed out parts are not in-trace)

where T denotes the in-trace relation from earlier. Using this relation, it can be ensured that *rf* and *co* relations must only span among in-trace events:

$$\begin{gathered} \textit{empty rf} \setminus \textit{both-in-trace as rfMustBeChosen} \\ \textit{empty co} \setminus \textit{both-in-trace as coMustBeChosen} \end{gathered} \tag{4.4}$$

Next, we need to constrain the domain of the two relations. We know that *co* edges span between writes only, while *rf* edges will only go from writes towards reads:

$$\begin{gathered} \textit{empty rf} \setminus (W * R) \textit{ as rfOnlyBetweenWriteRead} \\ \textit{empty co} \setminus (W * W) \textit{ as coOnlyBetweenWrites} \end{gathered} \tag{4.5}$$

Furthermore, we know that both relations only target same-location variables:

$$\begin{gathered} \textit{empty rf} \setminus \textit{loc as rfReadsSameVar} \\ \textit{empty co} \setminus \textit{loc as coForSameVar} \end{gathered} \tag{4.6}$$

We also know that each read event has at most one *rf* edge:

$$\textit{empty (rf; rf\^{}-1)} \setminus \textit{id as onlyOneRfPerRead} \tag{4.7}$$

Furthermore, each read event has at least one *rf* edge:

$$\textit{empty } (R\&T) \setminus \textit{range(rf) as everyReadReads} \tag{4.8}$$

And finally, we know that *co* relations form a total order over each location and initial writes are always ordered-before regular writes:

$$\begin{gathered} \textit{empty } ((IW * W) \mathbin{\&} \textit{loc} \mathbin{\&} \textit{both-in-trace}) \setminus \textit{id} \setminus \textit{co as coInitialFirst} \\ \textit{empty } ((W * W) \mathbin{\&} \textit{loc} \mathbin{\&} \textit{both-in-trace}) \setminus \textit{id} \setminus \textit{co} \setminus \textit{co\^{}-1 as coTotalOrder0} \\ \textit{empty co as coTotalOrder1} \end{gathered} \tag{4.9}$$

```
let  both−in−trace = (T ∗ T)

empty rf \ both−in−trace as rfMustBeChosen
empty co \ both−in−trace as coMustBeChosen

empty rf \ (W ∗ R) as rfOnlyBetweenWriteRead
empty rf \ loc as rfReadsSameVar
empty (rf; rf^−1)\id as onlyOneRfPerRead
empty (R & T) \ range(rf) as everyReadReads


empty co \ (W ∗ W) as coOnlyBetweenWrites
empty co \ loc as coForSameVar
empty ((IW ∗ W) & loc & both−in−trace) \ id \ co as
      coInitialFirst
empty ((W ∗ W) & loc & both−in−trace) \ id \ co \ co^−1
      as coTotalOrder0
acyclic co as coTotalOrder1
```

Figure 4.9: `stdlib.cat` well-formedness constraints

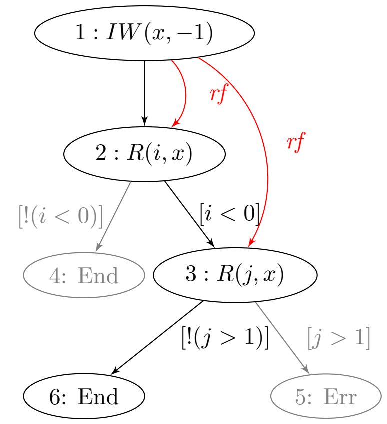


Figure 4.10: Only solution

A summary of these well-formedness constraints can be seen in Figure 4.9.

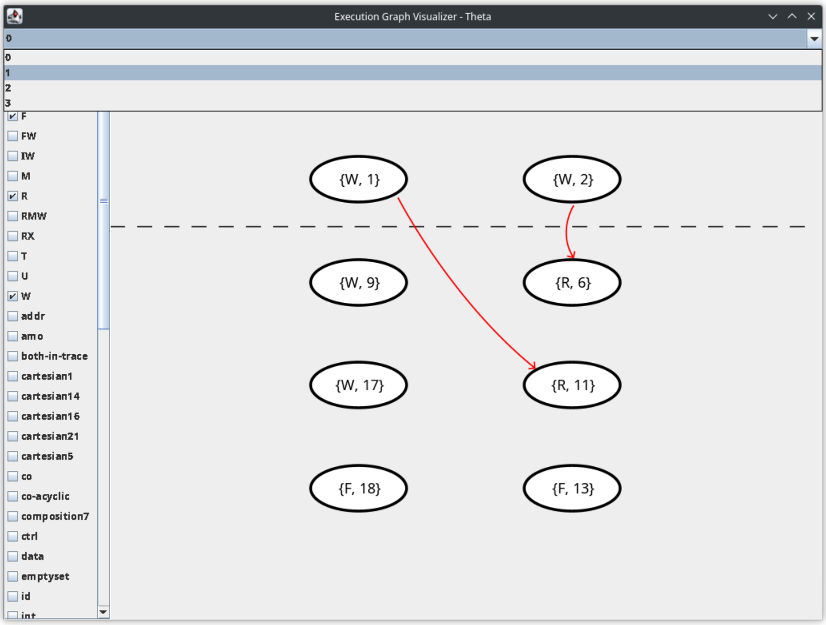
Using this, the program in Figure 4.6a can be classified as *SAFE* over the empty memory model, as no execution exists that leads to the error state. The only consistent execution (minus the error property) is given in Figure 4.10.

4.5 Providing a User Interface

Currently, Theta mainly supports litmus tests as the source for concurrent programs. It also has support for parsing C programs via the above mentioned C frontend, but the higher abstraction level of C hinders exact reasoning over its memory accesses, and therefore the proof-of-concept implementation focuses on litmus tests instead, which can be translated almost verbatim. Note that this is *not* a limitation of the proposed approach, but purely an implementation-specific one.

To verify a litmus test with respect to a certain memory model, Theta provides the already mentioned tool `litmus-cli`. It is a command line tool that provides the following configuration switches:

- `--cat`: the path to the input memory model
- `--litmus`: the path to the input litmus test
- `--loglevel`: the detailedness of the standard output
- `--print-xcfa`: dump the parsed XCFA as a .dot file for Graphviz visualization
- `--smt-home`: the path to the SMT home directory where solvers are installed (dee [22])
- `--solver`: the SMT solver's name and version to be used
- `--visualize`: visualize the consistent execution graphs



0

0
1
2
3

☑ F
□ FW
□ IW
□ M
☑ R
□ RMW
□ RX
□ T
□ U
☑ W
□ addr
□ amo
□ both-in-trace
□ cartesian1
□ cartesian14
□ cartesian16
□ cartesian21
□ cartesian5
□ co
□ co-acyclic
□ composition7
□ ctrl
□ data
□ emptyset
□ id
□ int

Figure 4.11: Graphical interface of the `litmus-cli` frontend

The last option, `--visualize`, will open a graphical window that shows an interactive execution visualization (see Figure 4.11). On the top, there is a drop-down that lets the user choose a solution to visualize. On the left, each checkbox corresponds to a relation or event set in the memory model, and by selecting some of them, they will be drawn on the graph. By default, only the 3 main event sets (Writes, Reads, Fences) and the *rf*-relation is visible.

4.6 Verifying the Implementation

To verify the implementation, I ran a selection from the litmus tests provided by DARTAGNAN[5] to see if both HERD and THETA produce the same results. In particular, I was interested in the number of consistent execution graphs the two tools produce.

I ran tests on 50 arbitrarily chosen litmus tests that had a diverse set of features among them (varying numbers of processes, labels, arithmetic- and branching instructions, etc.). HERD sometimes reported a higher number of executions than THETA, as some *co*-orders influenced the final memory state and therefore HERD enumerated all total orders; while THETA concentrated only on the *rf* relation (as it was expected). Besides this discrepancy, no further differences were found between the two tools, meaning the proof-of-concept implementation is most likely free of any serious bugs – although, of course, exact correctness could not be ensured using only testing.

[5] `https://github.com/hernanponcedeleon/Dat3M`

Chapter 5

Abstraction-Based State Space Exploration over Declarative Semantics

As mentioned before, the implemented algorithm in Chapter 4 is not capable of verifying programs containing loops. The main reason is the naive exploration of transitions from any given state – only the location is taken into account using the labelled transition system (LTS) implementation of the formalism. To fix this, more established tools (such as Dartagnan [21]) will use a loop unrolling method to get the necessary representation, but the technical obstacle remains the same.

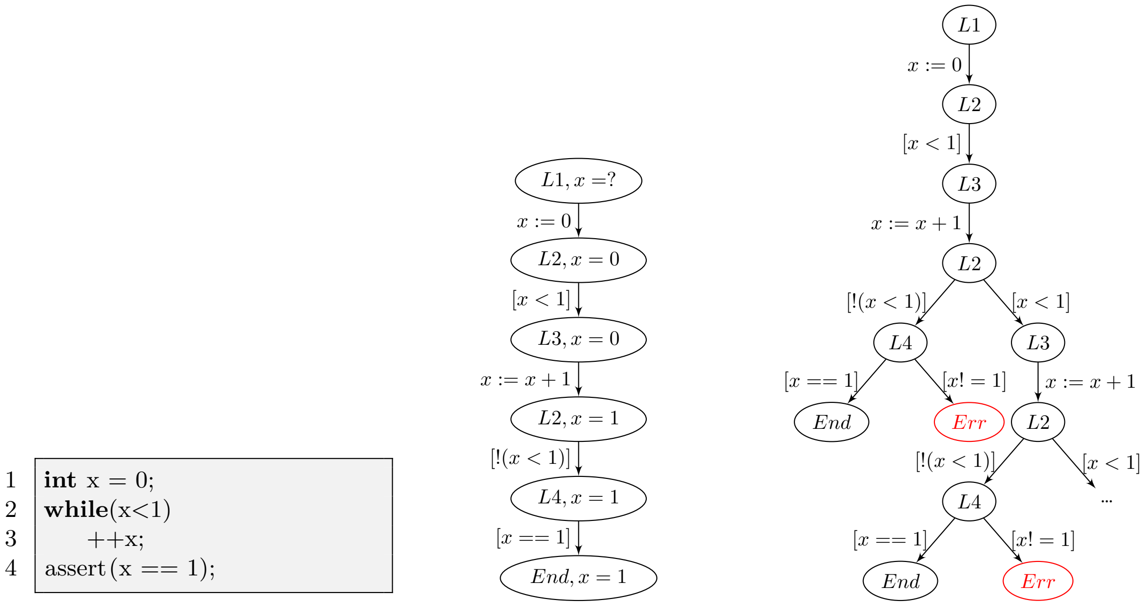
In this chapter, I propose a novel, CEGAR-like extension to the demonstrated approach, which should solve the problem of unbounded loops *as well as* handle any input more efficiently.

5.1 High-Level Overview

The main shortcoming of the SMT-based naive algorithm is the not taking data states into account when choosing possible transitions. For example, consider the program in Figure 5.1a: it contains a useless loop that will only execute once, and therefore the only enabled transition from the body of the loop should be its exit transition (see Figure 5.1b). However, we only know this if we pay attention to the value of x – and without that information, we must take all outgoing transitions into account, thereby creating an iteratively expanding state space (see Figure 5.1c).

However, if we take all variables and all their values into account, the state space will explode. A conventional solution to that is abstraction, and one its most widespread application to software verification is Counterexample-Guided Abstraction Refinement (CEGAR) [18]. However, applying abstraction over declarative semantics is tricky, because the standard way of state space exploration must be adapted.

The key idea behind applying CEGAR to declarative semantics is the use of multiple, co-operating *abstract reachability graphs* (ARG) [13], one per thread (or participant, process, etc.). Every thread builds its own ARG as a conventional, single-threaded program would, with the exception to memory accesses. With those, it also builds a global *abstract execution graph*, which will guide the *rf*-edges constraining the dataflow, as well as add them as *havoc* (in the case of reads) or *no-op* (in the case of fences and writes) statements



```
int x = 0;
while(x<1)
    ++x;
assert(x == 1);
```

(a) Bounded loop **(b)** Concrete state space **(c)** BMC over a loop

Figure 5.1: Demonstrating the problem with handling loops in a bounded setting

```
f0 = 1;          | f1 = 1;
t = 1;           | t = 0;
while(t && f1)   | while(!t && f0)
  { }            |   { }
assert(!crit);   | assert(!crit);
crit=1;          | crit=1;
crit=0;          | crit=0;
f0 = 0;          | f1 = 0;
```

Figure 5.2: Peterson's algorithm

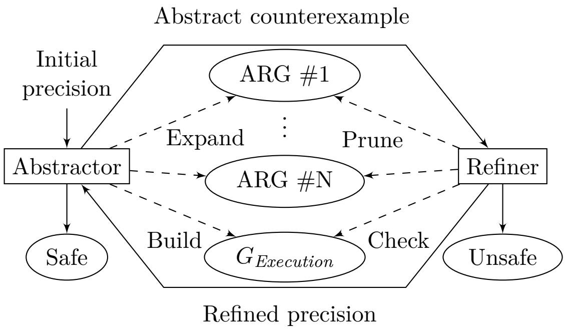


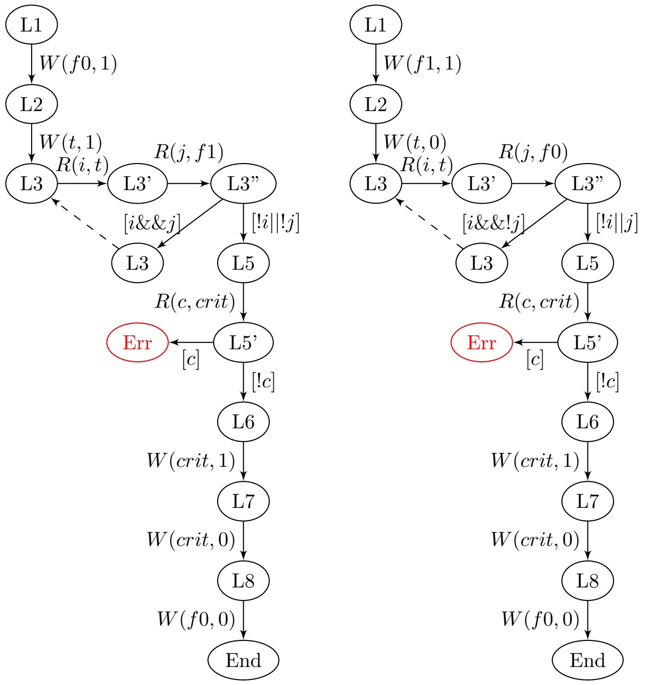
Figure 5.3: The declarative CEGAR loop

to the ARG of their executing thread. When an error state is discovered, all of the ARGs must be encoded into a single SMT-formula, and together with the abstract execution graph, an SMT-solver can be used to check for satisfiability and thereby reachability. If the error state is unreachable, a refutation can be used to refine the abstraction precision of each thread, and the state space exploration can restart. If no error state is reachable in the ARG, we can conclude safety. This workflow is summarized in Figure 5.3.

5.2 Representative Example

To demonstrate how the proposed approach works, I shall verify the program in Figure 5.2 (Peterson's algorithm [36]) over `SC` using its methodology. Note that some details might be left out to preserve conciseness, but a formalization of the steps is available in Section 5.3.

Let the initial precision for this verification run be and empty explicit-valued precision. With this, we construct the abstract reachability graph for each thread, as seen in Figure 5.4. Note the *covering* edges that link partially ordered-before states to their successors



(a) Thread 1

(b) Thread 2

Figure 5.4: ARGs of Figure 5.2

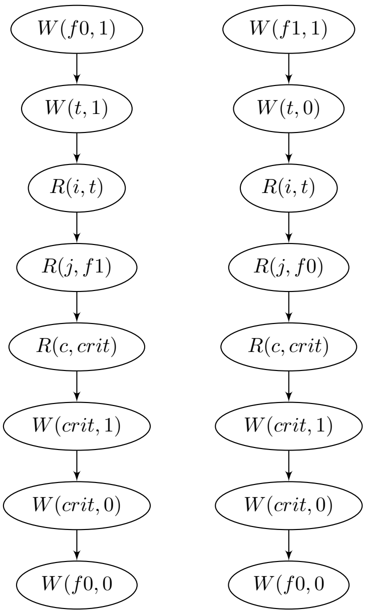


Figure 5.5: Events of Figure 5.4

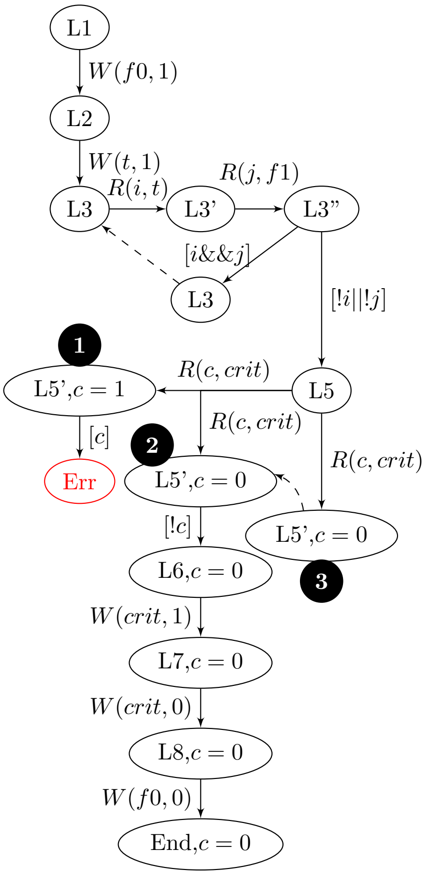
(which, in this case, is only location-dependent). Also note that some edges contain a numbered memory event – this is how the algorithm encodes the *abstract execution graph* of the multi-threaded program (the corresponding *po*-graph can be seen in Figure 5.5). As these ARGs allow the error state to be reached, an abstract counterexample can be constructed from them – for this, let us choose the first thread's error location.

In a conventional, single-threaded CEGAR loop, an abstract counterexample is a *trace*, i.e., a list of alternating states and transitions. As in a multi-threaded setting the states might be influenced by the global state, we must take the entire ARG of every other thread into account, and provide the counterexample as a *reached* node in the form of the *in-trace* (T) relation. This is added to the execution graph's specification via a specially tagged event.

The refiner needs to first check whether the counterexample is concretizable, i.e. if a consistent concrete execution allows the same state to be reached. To do this, it performs the same process as the algorithm in Chapter 4 for checking if a consistent execution can be produced – in which case, it reports this as a *counterexample*. If no such execution can be produced however (as is the case for Figure 5.4), it needs to produce a refutation why it is not possible. One option is to use *interpolation*, a feature provided by SMT-solvers, to find where the execution goes wrong. With this, a new precision can be created that holds all the variables we need to track in the next iteration of CEGAR. In this case, the new precision can be an explicit-valued precision containing the following variables:

1. The local variable `c` from the first thread
2. The global variable `crit`

With this precision, new ARGs must be built. The prefix to the $R(c, crit)$ instruction remains the same on both threads, but the further process differs. Because the `crit` variable



(a) Thread 1

Figure 5.6: ARGs of Figure 5.2

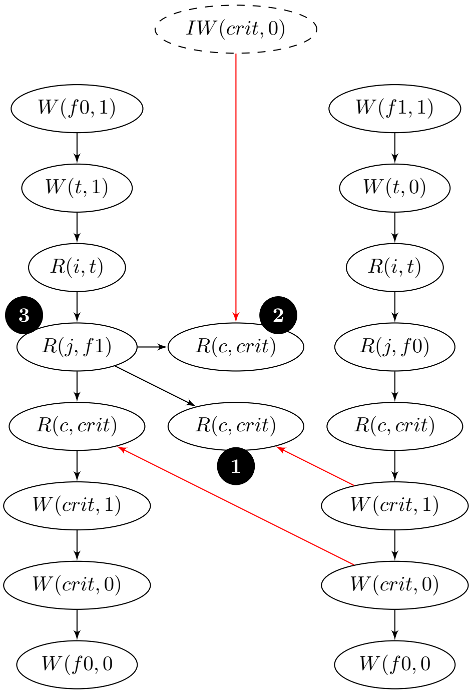


Figure 5.7: Events of Figure 5.6

is in the precision, we cannot handle a read from this variable as a *havoc* (because that was the erroneous over-approximation last time). Instead, we must put the read event into a collection of so-called *revisitable reads*, (inspired by Rcmc [32]), and explore every possible combination of existing write-read combination as a potential *rf*-edge. Furthermore, all successive additions of write events must try to supply a value to all revisitable reads.

The transfer function must also be instructed to take the source writes' states into account when calculating the resulting state. This process can be seen in Figure 5.6. Note how the resulting state of $L5$ changes based on where it received a value from (the different *rf*-edges are depicted in Figure 5.7).

With this new ARG, we can produce an abstract counterexample once again, which will not be concretizable. If in the next iteration we take all variables into account (where the verification is headed), we will find that it is impossible to reach the error state, and therefore we can conclude that Peterson's algorithm works for mutual exclusion.

5.3 Formalization

In this section, I formalize algorithms and data structures of the proposed approach.

5.3.1 Data Structures

The algorithms mainly work on a single data structure, the one holding the ARGs and the memory events. This can be formalized as a tuple $\mathit{MultiARG} = (IW, ARGs, \mathit{Relations})$, holding a collection of initial write events, an associative array of ARGs (keyed with unique process IDs), as well as elements to the unary- and binary relations defined by the memory model. Each ARG is a node- and edge-labelled, directed graph $G(V, E, l_v, l_e)$, where:

- V: vertices
- $E \subseteq V \times V$: edges
- $l_v : V \to S$: vertex labeling function, evaluating to a *state*
- $l_e : E \to (A \cup \{R, W, F\})$: edge labeling function, evaluating to either an *action* or a *memory event* (read, write or fence)

States and actions are formalism-dependent, but both should be evaluable to SMT-expressions. Memory events need to hold the following information:

- Read: unique ID, local variable, global variable, ordering primitive (*tag*)
- Write: unique ID, local variable, global variable, ordering primitive (*tag*)
- Fence: unique ID, ordering primitive (*tag*)

Relations are stored as collections of elements for *positive* and *negative* facts. With no stored fact, a given element is considered *unknown*, i.e., it may be true. The mapping of memory events to atoms in relations are given by the events' unique ID.

5.3.2 Algorithms

The algorithm follows a regular CEGAR loop, with a few modifications. This section mainly focuses on these modifications, but I also present some of the details from the conventional algorithm as to provide a clearer picture for how the proposed approach works.

5.3.2.1 Overview

On the highest level, the algorithm expects an initial precision, an XCFA and a memory model. Then, in a loop, it will create a `MultiARG` from the program over the memory model with the given precision, and report *safety* if no error state is reachable. Otherwise, it creates a counterexample and tries to concretize it. If it succeeds, the program is *unsafe*; and if it fails, a new precision is created that forbids the same spurious counterexample to be generated in the future, and the loop can start over again.

Algorithm 1: CEGAR loop

```
input  : InitPrec
input  : XCFA
input  : MCM
output: SafetyResult
RunningPrec ← InitPrec;
while true do
    MultiARG ← Abstract (RunningPrec, XCFA, MCM);
    if Safe(MultiARG) then
        return Safe
    else
        CEx ← ExtractCex(MultiARG);
        if Conretizable(CEx, MultiARG, MCM) then
            return Unsafe
        else
            RunningPrec ← Refine(CEx, MultiARG, MCM);
        end
    end
end
```

5.3.2.2 `Abstract`

The `Abstract` function takes as inputs a precision, an XCFA and a memory model, and creates a `MultiARG`.

Algorithm 2: `Abstract`: Abstraction of an XCFA with a given precision

```
input  : Prec
input  : XCFA
input  : MCM
output: MultiARG
MultiARG ← (InitialWrites, {Initialize (ARG0), ...}, InitialRelations);
WaitSet ← {InitialStates};
Reads ← {};
Writes ← {};
for s ∈ InitialStates do
    Transitions ← Successors (s);
    for t ∈ Transitions do
        if Var (t) ∈ Prec then
            (WaitSet, Reads, Writes) ← HandleMemoryEvent (t, WaitSet, Reads,
              Writes);
        else
            NewStates ← TransFunc (s, t);
            MultiArg ← MultiArg + NewStates;
            WaitSet ← WaitSet∪NewStates;
        end
    end
end
return MultiARG
```

Algorithm 3: `HandleMemoryEvent`: Handling an in-precision memory event

```
input  : Transition
input  : WaitSet
input  : Reads
input  : Writes
output: (NewWaitSet, NewReads, NewWrites)
if Type (Transition) == Read (as r) then
    NewReads ← Reads∪{$r$};
    NewWaitSet ← WaitSet;
    for $w \in Writes[var == var(r)]$ do
        NewWaitSet ← NewWaitSet∪ReadsFrom($w$,$r$);
    end
    return (NewWaitSet, NewReads, Writes);
else if Type (Transition) == Write (as w) then
    NewWrites ← Writes∪{$w$};
    NewWaitSet ← WaitSet;
    for $r \in Reads[var == var(w)]$ do
        NewWaitSet ← NewWaitSet∪ReadsFrom($w$,$r$);
    end
    return (NewWaitSet, Reads, NewWrites);
else
    return (WaitSet, Reads, Writes);
end
```

5.3.2.3 `ExtractCex`

As mentioned above, the counterexample to the verification problem is an in-trace event in an error state. Given the MultiARG, this is trivial to compute (traverse all ARGs, find one state that violates the safety property, and report the counterexample).

5.3.2.4 `Concretizable`

This sub-algorithm is explained in-detail in Chapter 4 and therefore I will not reiterate it here. Note that it does not matter that we use an abstract view in the ARGs, as the concretizer will have to work with the actual statements and relations.

5.3.2.5 `Refine`

Refining the precision is hard to generalize, because of the many different approaches. For some of the possibilities, see the technical paper of Theta [28]. The main difference from conventional refinement demonstrated in that paper stems from the use of memory models, which cause encoded relations to show up in the interpolant – the refinement algorithm needs to be aware of this caveat, and add the impacted variable to the precision.

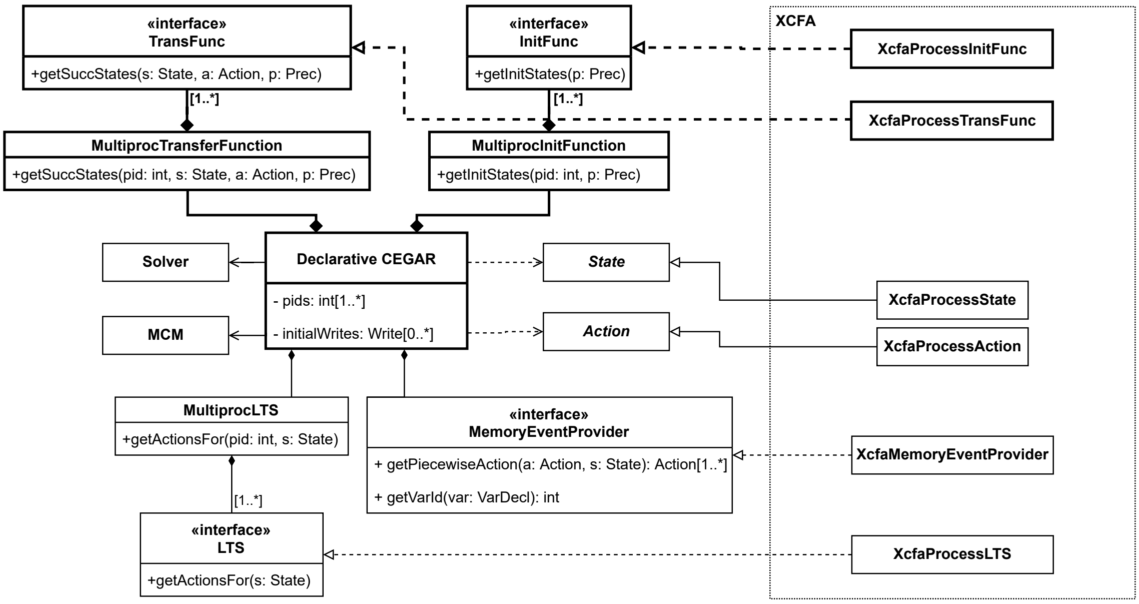


Figure 5.8: Summary of the implementation

5.4 Proof of Concept Implementation

As demonstrated by the example in Section 5.2, the most complicated parts of the proposed approach are:

1. the abstract state space exploration with a given precision; and
2. the concretization process.

Therefore, I implemented these parts in Theta, as a preparation to realizing a fully functional CEGAR loop using Theta's toolset and the presented new algorithm (which is out of the scope of this thesis).

The implementation uses a number of wrapper objects that give access to formalism-specific elements of the input program. This is summarized in Figure 5.8, with elements differing from Figure 4.5 highlighted. These elements are:

1. Transfer function: given a state, an action and a precision, enumerate all successor states
2. Init function: given a precision, enumerate all initial states

As a fully automatic abstraction refinement process has not yet been implemented, extensive evaluation of the process is not yet feasible. This implementation has mainly been a guide to understand and optimize the ARG building process, for which I used the same testing suite as in Chapter 4 with differing levels of precision, both from the cartesian predicate and the explicit domain. So far, all preliminary tests have shown that the implementation is sound, as no *false negatives* have been found (which would be the telltale sign of a badly-designed abstraction, due to the lack of over-approximation).

Chapter 6

Applying Memory Models on Communication Protocols

So far, the main focus has been put on variable *read* and *write* accesses. This is the standard model for different processor cores communicating with each other through a shared memory area, and the memory models have mainly concentrated on this process. However, if we want to model communication among participants of a distributed system, we must redefine these terms to better suit our needs. In this chapter, I explore the expansion of memory modeling concepts to commonly used communication protocols such as UDP and TCP, which could serve as the basis of many, more complex protocols.

6.1 Defining the Scope

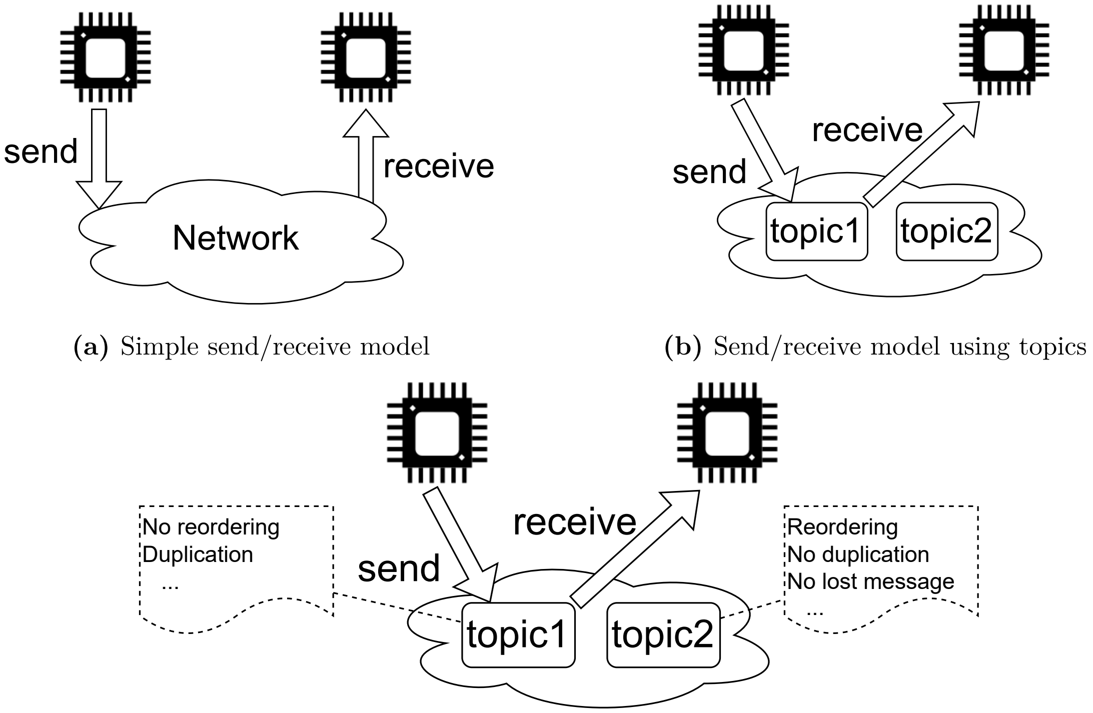
Throughout this chapter, the standard model for communication can be seen in Figure 6.1a. Several devices are communicating via some sort of network, where participants can either *send* or *receive* data. This standard model can be extended with further elements, such as topics (as seen in Figure 6.1b), which restrict the pairings of sent and received data; or Quality of Sevice (QoS) settings that govern how each topic (or segment) behaves in terms of message handling (as seen in Figure 6.1c).

Note that while different protocols will define different send and receive semantics (such as publish/subscribe, broadcast and direct communication, etc.), those details will be encoded in their *memory model*, rather than defining separate standard models for every scenario. In Section 6.4, I will be exploring the applicability of this approach to show that it does not constrain the communication protocols in this way.

6.1.1 Communication Behavior Patterns

When it came to shared memory communication, the reordering of memory accesses was the main source of complexity. For network communication, however, multiple times of problems can occur:

- Two messages were sent in one order, but observed in another by at least one receiver
- A single message was sent, but multiple messages were received with identical contents



(a) Simple send/receive model

(b) Send/receive model using topics

(c) Send/receive model using topics with different QoS settings

Figure 6.1: Models of distributed communication

- A message was sent, but no-one received it

Note that the last two problematic events are normal when we talk about memory accesses, but might be unusual and/or unwanted when network communication is used. Based on this and similar observations, the list of behavior patterns to model in the context of this thesis is the following:

1. Reordering: does the reception of messages follow some total order by all participants?
2. Duplication: can multiple receptions by the same participant occur from a single sent message?
3. Reliability: is the reception of a sent message guaranteed?
4. Broadcasting: can multiple participants receive a single sent message?
5. Sending synchronicity: can a participant issue new events before a sent message is received?
6. Blocking reception: can a participant receive from nowhere, or will all receive instructions block?

6.2 Modeling Behaviour Patterns

To model the previously established patterns, I used my proof-of-concept implementation of a memory model based verification tool introduced in Chapter 4. Note however, that

I did not use the well-formedness constraint concerning the *rf*-relation's mandatory existence for each read established in Equation 4.8, as that was mainly true for shared memory applications. The other ones still hold. Also, an added requirement is that no read shall be able to receive the value written by an initial write – that concept is generally not present in message-based system. For this, we need to include the following in `stdlib.cat`:

$$\textit{empty}\ \ \textit{rf}\,\&\,(\textit{IW} * \textit{R})\ \textit{as noIW} \tag{6.1}$$

In this section, I create easily pluggable memory modeling constructs in the form of a *standard library for message-based communication*, which is realized by two Cat models per pattern (one negative and one positive). These files help with the rapid prototyping and evaluation of communication protocols, by making it possible to include them and therefore enforce or forbid their contents.

6.2.1 Reordering Messages

The reordering of events is a problem both concurrent software and distributed systems observe. With no reordering, we expect all *rf* edges to conform to a strictly *sequential* execution – which can be achieved using the following memory model constraint:

$$\begin{gathered}\textit{let fr} \ = \ (\textit{rf\^{}-1}\ ;\ \textit{co})\ \backslash \textit{id} \\ \textit{acyclic}\ (\textit{po} \mid \textit{co} \mid \textit{rf} \mid \textit{fr})\ \textit{as sc}\end{gathered} \tag{no-reordering.cat}$$

If a protocol wants to allow reordering, we still need to pay attention to causality – meaning a previous reception should never read a value published by a later send. With no other constraint, this can be realized by disallowing cycles in the *po* and *rf* relations:

$$\textit{acyclic}\ (\textit{po} \mid \textit{rf})\ \textit{as causality} \tag{causality.cat}$$

Of course, finer control over reordering messages *can* be expressed using Cat – but that problem is well-covered by the conventional memory modeling of concurrent programs [7].

6.2.2 Duplicating Messages

With concurrent software, it is not unusual to have a write's value passed to multiple reads – if that value is in-memory, any number of read might receive it. However, with message-based communication, each message is restricted to be received once in most cases (either globally, or by participant). To model global uniqueness using Cat, the following construct can be used:

$$\textit{empty}\ (\textit{rf\^{}-1; rf})\ \backslash\ \textit{id as noDup} \tag{global-no-dup.cat}$$

This rule can be relaxed by allowing each *participant* to read the value once:

$$\textit{empty}\ (\textit{rf\^{}-1; rf})\ \backslash\ \textit{id}\ \backslash\ \textit{int as intNoDup} \tag{participant-no-dup.cat}$$

If duplication should be allowed, no constraint is necessary – this is the most relaxed model.

6.2.3 Losing Messages

By default, it is not forbidden for writes to be *lost*, i.e., have no associated *rf*-edge. It might be required, however, that all messages must be received by at least one participant, for example in the case of TCP: a send operation cannot be considered successful, if no-one received (and acknowledged) it. This requirement can be formalized in the following constraint:

$$\textit{empty } (W\&T) \setminus \textit{domain(rf) as noSendLost} \tag{no-loss.cat}$$

6.2.4 Broadcasting Messages

To further constrain the execution seen in no-loss.cat, it can also be a requirement that *all* other participants must receive a value from a particular write:

$$\begin{gathered} \textit{let rf-int} \;=\; \textit{rf; int} \\ \textit{empty } ((W\&T) * (R\&T))\ \&\ \textit{ext} \setminus \textit{rf} \setminus \textit{rf-int as allMustReceive} \end{gathered} \tag{broadcast.cat}$$

6.2.5 Sending Synchronously

Some protocols require all *send* operations to find all its respective *receive* operation(s) before proceeding with execution. This is called *synchronous* messaging, while *asynchronous* refers to the case where the value is committed to some sort of distributed memory, and any reception can receive it later. Normally, concurrent programs are asynchronous and therefore no extra constraint has to be placed over such executions, but synchronicity requires the use of the following construct:

$$\begin{gathered} \textit{let fr} \;=\; (\textit{rf\^{}-1}\ ;\ \textit{co}) \setminus \textit{id} \\ \textit{let po-com} \;=\; \textit{po} \mid \textit{co} \mid \textit{rf} \mid \textit{fr} \\ \textit{let po-com-chain} \;=\; \textit{po-com}^{*} \setminus \textit{rf} \\ \textit{empty } (\textit{rf / rf\^{}-1})\ \&\ \textit{po-com-chain as allRWCoincide} \end{gathered} \tag{synchronous.cat}$$

6.2.6 Blocking Reception

Normally, a memory operation could never fail, as the value in the memory is always established. However, in the case of network communication, it is possible to receive *nothing* as the result of an attempted reception. In this case, the implementation might *block* and wait for a valid value, which is handled implicitly by not modeling any timing information (and therefore arbitrary time differences are allowed among any two events). They also might just fail and continue execution normally (with possibly a random value as their received payload). This is modelled by *not* including the *rf*-related constraints from Equation 4.8, and their counterpart (i.e., no read shall return without receiving a value) need the following rules:

$$\textit{empty } (R\&T) \setminus \textit{range(rf) as everyReadReads} \tag{blocking.cat}$$

However, even in the non-blocking case, it might be necessary to enforce that a reception can only be ignored if no suitable write exists. To do this, we first have to calculate potential *rf*-edges, then assert that if a read has one, than it also needs to have an actual *rf*-edge. However, this is impossible to do generally, as the *rf* edge can be constrained in any number of ways. Therefore, we expect the user to provide a function called `illegal_rf()`, which will provide the necessary constraints:

$$\begin{aligned} &\textit{let pot-rf} \ = \ (W * R) \ \& \ \textit{loc} \setminus \textit{illegal\_rf}() \\ &\quad \textit{empty range(pot-rf)} \setminus \textit{range(rf)} \end{aligned} \qquad \text{(no-ignored-write.cat)}$$

6.3 Extending the Standard Model

As mentioned above and show in Figure 6.1, the standard model can be extended with topics, QoS settings, etc. So far, to encode the *send* and *receive* events, the *write* and *read* constructs were used, all writing a single global variable. If topics are to be used, each topic can correspond to a different shared variable. Furthermore, each segment of the network under different QoS settings can be either tagged if the send/receive events are influenced, or be placed in a *sr* (scope) relation [7] that influences the applicable rules.

Note that having *topics* is not that exotic of a behavior. For many protocols, the communication is point-to-point, i.e., the sender sends its message to an addressee. In this case, each *topic*

Notice that the modeled patterns above use the *rf*, *po*, etc. relations directly, while due to scoping, some rules might be differently applied to different parts of the execution. To solve this, all rules above are placed inside procedures, which receive these relations indirectly as parameters. For example, the rule (no-loss.cat) will look like this:

$$\begin{aligned} &\textit{procedure no-loss(W, T, rf)} \ = \\ &\ \textit{empty} \ (W \& T) \setminus \textit{domain(rf) as noSendLost} \\ &\textit{end} \end{aligned} \qquad \text{(no-loss.cat)}$$

6.4 Applicability of the Approach

To see how well the above presented approach works, I model a few real-life protocols and show how they behave. Throughout this section, it is important to keep in mind that the proposed approach is *declarative*, i.e., the effects of the communication protocols are modeled, not their exact behaviour or implementation.

6.4.1 User Datagram Protocol (UDP)

UDP is a simple, ”no-guarantees”, point-to-point or point-to-multipoint communication protocol. It allows reordering, duplication and message loss as well. We assume that the received messages are not erroneous (or are detected and discarded, which case is handled under message loss). In this example, I opted to used the point-to-point version.

Based on its guarantees, UDP needs the following patterns:

- Reordering, but causality

```
send(x, 1)        | i = receive(x)
send(x, 2)        | send(x, i)
nop               | j = receive(x)
```

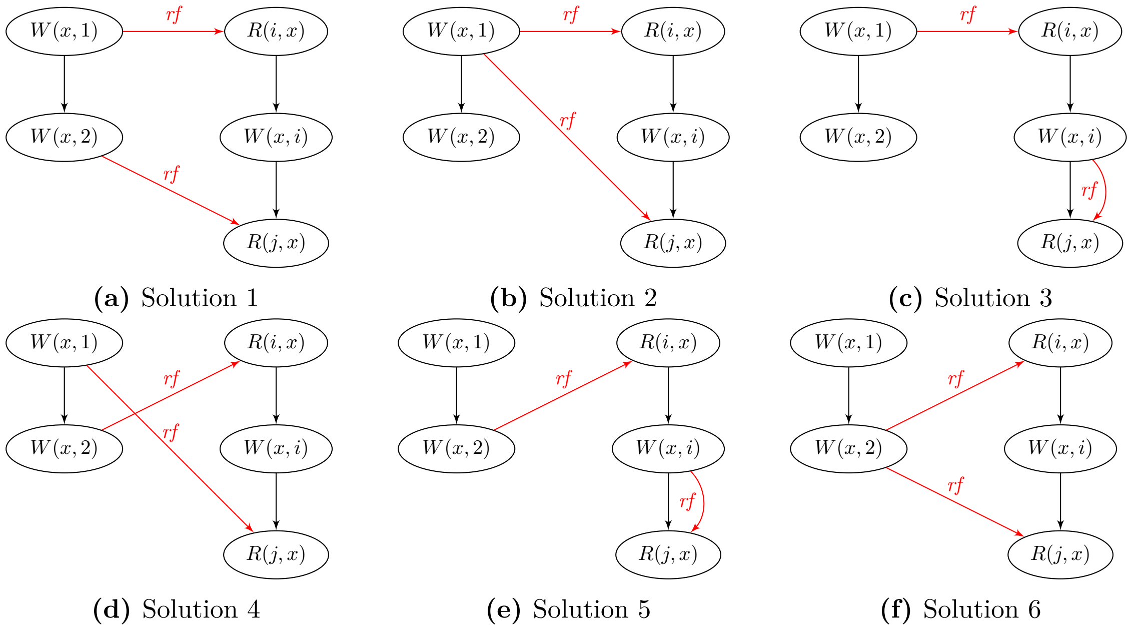
Figure 6.2: Litmus test

```
UDP

include "causality.cat"
 call causality(po, rf)

include "blocking.cat"
 call blocking(R, T, rf)
```

Figure 6.3: UDP memory model



(a) Solution 1

(b) Solution 2

(c) Solution 3

(d) Solution 4

(e) Solution 5

(f) Solution 6

Figure 6.4: Solutions to Figure 6.2 over 6.3

- Duplication
- Loss
- No broadcasting
- Asynchronous
- Reception is blocking

To showcase these properties, we would like to verify the safety of the litmus test in Figure 6.2. To do this, I have created the memory model for UDP, seen in Figure 6.3, which produced the solutions seen in Figure 6.4. It shows that the memory model works as intended:

- Solution 2 shows *duplication* and *loss*
- Solution 4 shows *reordering* and *asynchronous*-ness
- No solution has been generated that contradicts *causality*, even though the second participant could have read from its own future

```
send(x, 1)       | i = receive(x)
i = receive(y)   | send(y, i)
send(x, 2)       | j = receive(x)
```

Figure 6.5: Litmus test

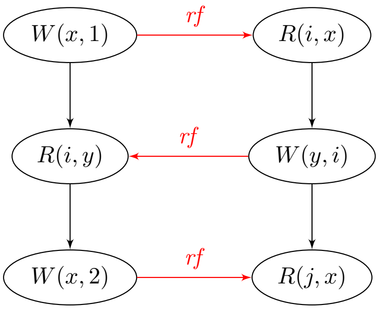


Figure 6.6: Solution to Figure 6.5 over 6.7

```
TCP

include "no−reordering.cat"
call sc(rf, co, id, po)

include "global−no−dup.cat"
call nodup(rf, id)

include "no−loss.cat"
call noSendLost(W, IW, T, rf)

include "synchronous.cat"
call sync(rf, co, id, po)

include "blocking.cat"
call blocking(R, T, rf)
```

Figure 6.7: TCP model

- No solution has been generated that contradicts the *blocking* effect, as no read was left without a corresponding *rf*-edge

Even though one test is not definitive proof that the memory model is perfect, the lack of a contradiction is very promising.

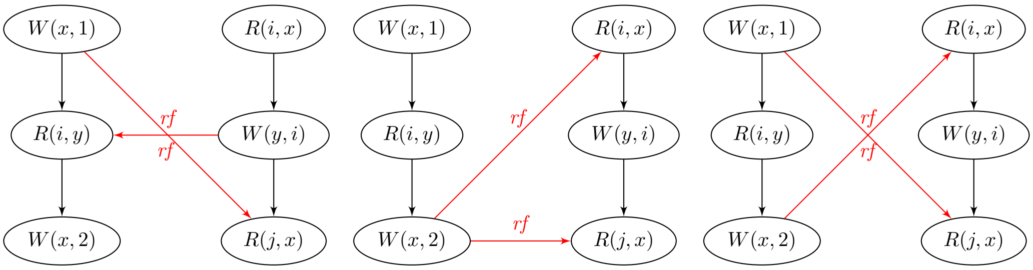
6.4.2 Transmission Control Protocol (TCP)

In contrast to UDP, TCP is a very reliable, mainly point-to-point communication protocol. It guarantees that all sent messages are delivered, without reordering or duplication. Therefore, it needs to follow the following patterns:

- No reordering
- No duplication
- No loss
- No broadcasting
- Synchronous
- Reception is blocking

The corresponding memory model can be seen in Figure 6.7. I could not use the same litmus test as for UDP (Figure 6.2), as the mismatched number of send and receive events would make it impossible to generate any solutions. Therefore, I used a similar litmus test (see Figure 6.5) to generate consistent executions, which can be seen in Figure 6.6. Notice that only a single execution was generated: this is because the strict rules of TCP disallow any other execution. See Figure 6.8 for all the solutions that were eliminated by its rules.

Consider the forbidden execution in Figure 6.8a. It would violate a number of patterns: it loses the value of the last write (*loss*), there is a read that received no value (*non-blocking*),



(a) Forbidden solution 1 **(b)** Forbidden solution 2 **(c)** Forbidden solution 3

Figure 6.8: Some solutions to Figure 6.5 forbidden by 6.7

and it is not possible that the two pairs of read-write events occurred simultaneously (*asynchronous*). In addition, Figure 6.8b shows *duplication*, and Figure 6.8c shows *reordering* of events. This shows that the litmus test *could* produce forbidden results, but the constructed memory model (rightly) eliminates them.

Chapter 7

Conclusion

Formal software verification is a very relevant topic nowadays. More and more domains require a formal proof of safety, as the price of failure is too high – from making sure a space probe will never fail as maintenance is impossible, to power plants and aviation endangering the lives of billions of people if not done properly, the need for a formal method for finding and eliminating bugs is a growing trend.

In this thesis, my main focus has been on the efficient handling of weakly-ordered programs and systems. In this case, the presumption that the order of events is governed only by the order of lines in the source code is violated, and therefore most conventional techniques fail to address it.

As my first contribution to this topic, I presented in this thesis a survey of the state of the art regarding verification approaches handling parallelism, with a focus on the weakly ordered case. I have shown that while some tools and algorithms exist, they fail to address the problem of verifying unbounded inputs over a specific memory model.

To gain a deeper understanding of the problem, as well as to provide a technical platform for further research, I implemented the most promising existing algorithm in Theta [40], from the bounded model checker Dartagnan [21]. This has been my second contribution of this thesis.

Based on the knowledge gained from the survey and the implementation, I proposde an approach that combines abstraction (a strong-suit of Theta) with axiomatic memory models to provide a way of handling infinite-state programs over declarative semantics. My third contribution consists of this theoretical proposal, as well as the proof-of-concept implementation of the main parts of the algorithm.

Finally, as my fourth contribution, I extended the reach of memory modeling to the field of distributed systems by providing a practical application of the memory modeling language Cat [8] to message-based protocols, with a complete implementation for the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).

I think my contributions will help further shape the direction where axiomatic memory modeling, and its formal verification are headed. I am planning on continuing this work, first by providing a deeper integration of the proposed approach into conventional CEGAR, and therefore enabling the rapid development of algorithms and configurations that work well for this case. Furthermore, I am planning on pursuing the extensions of memory modeling to other domains as well, where such constructs might help formal reasoning tools handle their problems more efficiently.

List of Figures

Bibliography

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, et al. Optimal Dynamic Partial Order Reduction. *SIGPLAN Not.*, 49(1):373–384, January 2014. ISSN 0362-1340. DOI: `10.1145/2578855.2535845`.

[2] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, et al. Stateless Model Checking for TSO and PSO. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015*, volume 9035 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2015. DOI: `10.1007/978-3-662-46681-0_28`.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, et al. Stateless Model Checking for POWER. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016*, volume 9780 of *Lecture Notes in Computer Science*, pages 134–156. Springer, 2016. DOI: `10.1007/978-3-319-41540-6_8`.

[4] Zsófia Ádám, Levente Bajczi, Mihály Dobos-Kovács, et al. Theta: portfolio of CEGAR-based analyses with dynamic algorithm selection (Competition Contribution). In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2022*, volume 13244 of *Lecture Notes in Computer Science*, pages 474–478. Springer, 2022. DOI: `10.1007/978-3-030-99527-0_34`.

[5] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013. DOI: `10.1007/978-3-642-39799-8_9`.

[6] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 141–157, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39799-8.

[7] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014. DOI: `10.1145/2627752`.

[8] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, abs/1608.07531, 2016.

[9] Levente Bajczi, Zsófia Ádám, and Vince Molnár. C for Yourself: Comparison of Front-End Techniques for Formal Verification. In *2022 IEEE/ACM 10th International*

Conference on Formal Methods in Software Engineering (FormaliSE), 2022. DOI: 10.1145/3524482.3527646.

[10] Mark Batty, Scott Owens, Susmit Sarkar, et al. Mathematizing C++ concurrency. In Thomas Ball and Mooly Sagiv, editors, *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, pages 55–66. ACM, 2011. DOI: 10.1145/1926385.1926394.

[11] Dirk Beyer and Karlheinz Friedberger. A light-weight approach for verifying multi-threaded programs with CPAchecker. *Electronic Proceedings in Theoretical Computer Science*, 233:61–71, 2016. DOI: 10.4204/eptcs.233.6.

[12] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, et al. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, September 2007. DOI: 10.1007/s10009-007-0044-z.

[13] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. *Computer Aided Verification*, page 504–518, 2007. DOI: 10.1007/978-3-540-73368-3_51.

[14] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, et al. Software model checking via large-block encoding. *2009 Formal Methods in Computer-Aided Design*, 2009. DOI: 10.1109/fmcad.2009.5351147.

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, et al. Bounded model checking. *Advances in Computers*, page 117–148, 2003. DOI: 10.1016/s0065-2458(03)58003-2.

[16] Paul E. Black, Paul Ammann, and Wei Ding. *Model checkers in software testing.* U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, 2002.

[17] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and Litmus tests. In Albert Cohen and Martin T. Vechev, editors, *38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, pages 467–481. ACM, 2017. DOI: 10.1145/3062341.3062353.

[18] Edmund Clarke, Orna Grumberg, Somesh Jha, et al. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, September 2003. DOI: 10.1145/876638.876643.

[19] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking.* MIT Press, 1999.

[20] Edmund M. Clarke, William Klieber, Miloš Nováček, et al. Model checking and the state explosion problem. *Lecture Notes in Computer Science*, page 1–30, 2012. DOI: 10.1007/978-3-642-35746-6_1.

[21] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, et al. BMC with Memory Models as Modules. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018*, pages 1–9. IEEE, 2018. DOI: 10.23919/FMCAD.2018.8603021.

[22] Mihály Dobos-Kovács. On the verification of safety-critical embedded software systems. Master's thesis, Budapest University of Technology and Economics, 2021.

[23] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, et al. Software verification using K-Induction. *Static Analysis*, page 351–368, 2011. DOI: 10.1007/978-3-642-23702-7_26.

[24] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 110–121, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 158113830X. DOI: 10.1145/1040305.1040315.

[25] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, et al. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019*, volume 11561 of *Lecture Notes in Computer Science*, pages 355–365. Springer, 2019. DOI: 10.1007/978-3-030-25540-4_19.

[26] Patrice Godefroid, Jan van Leeuwen, Juris Hartmanis, et al. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Citeseer, 1996.

[27] Henning Günther, Alfons Laarman, and Georg Weissenbacher. Vienna verification tool: IC3 for parallel software. *Tools and Algorithms for the Construction and Analysis of Systems*, page 954–957, 2016. DOI: 10.1007/978-3-662-49674-9_69.

[28] Ákos Hajdu and Zoltán Micskei. Efficient Strategies for CEGAR-based Model Checking. *Journal of Automated Reasoning*, 64(6):1051–1091, 2020. DOI: 10.1007/s10817-019-09535-x.

[29] Gerard J. Holzmann. Explicit-state model checking. *Handbook of Model Checking*, page 153–171, 2018. DOI: 10.1007/978-3-319-10575-8_5.

[30] IEC 61508:2010. Functional safety of electrical/electronic/programmable electronic safety-related systems. International standard, International Electrotechnical Commission, April 2010.

[31] ISO/IEC 9899:201x. Programming languages — C. International standard, International Organization for Standardization, International Electrotechnical Commission, December 2010.

[32] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, et al. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.*, 2(POPL): 17:1–17:32, 2018. DOI: 10.1145/3158105.

[33] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In Kathryn S. McKinley and Kathleen Fisher, editors, *Conference on Programming Language Design and Implementation, PLDI 2019*, pages 96–110. ACM, 2019. DOI: 10.1145/3314221.3314609.

[34] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, et al. Repairing Sequential Consistency in C/C++11. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 618–632, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. DOI: 10.1145/3062341.3062352.

[35] Brian Norris and Brian Demsky. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.*, 38(3), May 2016. ISSN 0164-0925. DOI: 10.1145/2806886.

[36] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12 (3):115–116, 1981. DOI: 10.1016/0020-0190(81)90106-X.

[37] Zvonimir Rakamaric and Michael Emmi. SMACK: Decoupling Source Language Details from Verifier Implementations. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014. DOI: 10.1007/978-3-319-08867-9_7.

[38] Susmit Sarkar, Peter Sewell, Jade Alglave, et al. Understanding POWER multiprocessors. In Mary W. Hall and David A. Padua, editors, *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 175–186. ACM, 2011. DOI: 10.1145/1993498.1993520.

[39] Thomas N. Theis and H.-S. Philip Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science; Engineering*, 19(2):41–50, 2017. DOI: 10.1109/mcse.2017.29.

[40] Tamás Tóth, Ákos Hajdu, András Vörös, et al. Theta: a Framework for Abstraction Refinement-Based Model Checking. In Daryl Stewart and Georg Weissenbacher, editors, *17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. ISBN 978-0-9835678-7-5. DOI: 10.23919/FMCAD.2017.8102257.

[41] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, et al. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In Yunji Chen, Olivier Temam, and John Carter, editors, *22nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017*, pages 119–133. ACM, 2017. DOI: 10.1145/3037697.3037719.

[42] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math*, 58:345–363, 1936.