Theory and Implementation of Software Bounded Model Checking

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurswissenschaften

der Fakultät für Informatik

des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Florian Merz

aus Herford

Tag der mündlichen Prüfung:	29.01.2016
Erster Gutachter:	Dr. Carsten Sinz
Zweiter Gutachter:	Prof. Dr. Bernhard Beckert
Externer Gutachter:	Prof. Dr. Armin Biere

Acknowledgments

This work would not have been possible without the invaluable scientific guidance and support of my supervisor Dr. Carsten Sinz.

I'm grateful to Prof. Dr. Armin Biere and Prof. Dr. Bernhard Beckert for agreeing to review my dissertation on such short notice. I'm also thankful to Armin Biere for inviting me to Linz for a research visit in 2010. The visit was tremendously helpful, as I learned a lot from him. Besides, the train ride to Linz sparked the idea for my first proper scientific contribution.

Furthermore, I wish to say thank you to Prof. Dr. Ralf Reussner and Prof. Dr. Klemens Böhm for agreeing to act as examiners for my thesis defense.

My colleague, Dr. Stephan Falke, greatly contributed to LLBMC's implementation as well as its participation in the International Software Verification Competition. My diploma thesis at Robert Bosch GmbH in 2008/2009 sparked my interest in this area of research. This would not have happened without my supervisors back then Dr. Hendrik Post and Thomas Gorges.

I'd also like to say thank my current and former colleagues at the KIT Tomáš Balyo, Lilian Beckert, Dr. Thorsten Bormer, Dr. Christian Engel, Dr. David Faragó, Dr. Aboubakr Achraf El Ghazi, Dr. Christoph Gladisch, Dr. Daniel Grahl, Sarah Grebing, Simon Greiner, Mihai Herda, Markus Iser, Dr. Vladimir Klebanov, Michael Kristen, Felix Kutzner, Dr. Tianhai Liu, Simone Meinhart, Dr. Mattias Ulbrich, Reimo Schaupp, Dr. Christoph Scheben, Alexander Weigl, Dr. Benjamin Weiß, and Dr. Frank Werner, for countless interesting discussions and conversations, for their support, and for numerous wonderfully distracting lunches.

I owe my gratitude to my parents for their love and support. Last but not least, I'd like thank my wife Christina, for proof reading this thesis, but far more importantly for being by my side throughout these years. Without you this would not have been possible. Thank you.

Florian Merz

Zusammenfassung (German Summary)

In den vergangenen Jahrzehnten hat die Bedeutung von Software stark zugenommen, so dass sie inzwischen beinahe allgegenwärtig geworden ist. Einen großen Anteil daran hat Software in sogenannten eingebetteten Systemen. Dabei handelt es sich um Computer, die lediglich einen Teil eines größeren technischen Systems darstellen. Gerade in Autos, Flugzeugen, Zügen und medizinischen Geräten übernehmen eingebettete Systeme oft auch eine steuernde Funktion. Dies wird am Beispiel der Fahrerassistenzsysteme deutlich, die als Einparkhilfen oder Abstands- und Spurhalteassistenten den Fahrer "mit-steuernd" unterstützen. Da Fehler in solchen Systemen meist großen finanziellen Schaden anrichten können und sogar teils eine Gefahr für Leib und Leben darstellen, gelten diese Systeme als sicherheitskritisch.

Die Entwicklung fehlerfreier sicherheitskritischer Systeme hat sich als enorme Herausforderung herausgestellt. Ein Beispiel hierfür ist der Erstflug der Ariane Rakete (Ariane Flug 501), der in der Explosion der 500 Millionen Euro teuren Rakete 37 Sekunden nach deren Abheben endete. Ursache hierfür war ein sogenannter Laufzeitfehler, der in der Qualitätssicherung nicht entdeckt wurde. Das Beispiel zeigt, dass klassische Methoden der Qualitätssicherung, insbesondere Software-Tests und leichtgewichtige statische Analyse, nicht die erforderliche Software-Qualität sicherstellen können. Dies ist wenig überraschend, können Tests in der Realität doch niemals eine hundertprozentige Abdeckung erzielen und daher lediglich die Anwesenheit von Fehlern anzeigen (*Falsifikation*), nicht jedoch deren Abwesenheit (*Verifikation*). Leichtgewichtige statische Analyseverfahren wiederum erkennen verdächtige Muster im Programmcode und können so auf Fehler hinweisen, bieten jedoch auch keine Verifikation. Die Software nachfolgender Generationen der Ariane Rakete wurden daher mit schwergewichtigen, verifizierenden *statischen Analyseverfahren* auf bestimmte Fehlerklassen geprüft.

Schwergewichtigere statische Analyseverfahren basieren meist auf formalen, mathematischen Methoden. Neben der *Abstrakten Interpretation*, welche bei der Entwicklung der Ariane Rakete zum Einsatz kam, gibt es hier zwei große Gruppen, die zu unterscheiden sind. Dies ist zum einen die *deduktive Verifikation*, bei der der mathematische Beweis der Korrektheit eines Programms im Mittelpunkt steht. Der Einsatz dieser Methoden stellt oft einen erheblichen Arbeitsaufwand dar, da die Beweise zu großen Teilen von Spezialisten von Hand erstellt werden müssen. Zum anderen gibt es die sogenannte *Modellprüfung*, bei der für ein gegebenes Modell (im modelltheoretischen Sinne) und eine gegebene Spezifikation (üblicherweise gegeben in temporaler Logik) vollautomatisch geprüft wird, ob das Modell die Spezifikation erfüllt.

Der klassische Modellprüfungsansatz basiert auf der iterativen expliziten Auflistung der erreichbaren oder fehlerhaften Programmzustände (*Explizite Modellprüfung*). Klassische Modellprüfung wurde erfolgreich zur Prüfung von Protokollen und von abstrakten Modellen komplexerer Systeme eingesetzt. Die Methode leidet jedoch stark unter dem Problem der sogenannten *Zustandsraumexplosion*. Hierbei wächst die Menge der zu untersuchenden Zustände mit der Anzahl der Zustandsvariablen so stark, dass die Zustände nicht mehr effizient verwaltet werden kann. Eine Weiterentwicklung der Modellprüfung, die *symbolische Modellprüfung*, basiert auf der Darstellung der Zustandsmengen und -übergangsrelation des Programms als logische Formel mittels sogenannter *binärer Entscheidungsdiagramme*. Die Zustandsmengen

müssen dadurch nicht mehr explizit aufgezählt werden, wodurch es möglich wird, Systeme mit mehr als 10^{20} Zuständen zu analysieren.

Eine weitere Variante der Modellprüfung, die *beschränkte Modellprüfung* (*Bounded Model Checking*), basiert schließlich auf der Umformung eines beschränkten Teils des Modellprüfungsproblems in ein Problem der *Erfüllbarkeit der Aussagenlogik* (SAT). Das System wird dabei bis zu einer festgelegten Grenze k abgerollt und der Systemzustand und die Zustandsübergangsrelation jeweils k mal in einer aussagenlogischen Formel kodiert. Anschließend wird die Formel mithilfe eines leistungsfähigen SAT-Solvers gelöst. Die Grenze k kann dabei iterativ vergrößert werden, bis ein Fehler gefunden wurde. Die beschränkte Modellprüfung findet so kürzestmögliche Gegenbeispiele und ist verifizierend, falls eine ausreichend große Schranke erreicht wurde. Die Methode hat sich in der Praxis insbesondere bei der Qualitätssicherung von Mikroprozessorentwürfen bewährt.

Die vorliegende Arbeit beschäftigt sich mit Theorie und Implementierung von *Software Bounded Model Checking* (SBMC), also der Anwendung von Bounded Model Checking auf Software Systeme. Die Übertragung der Methode auf Software ist dabei keineswegs trivial, da sich Software-Systeme strukturell stark von Hardware-System oder abstrakten Modellen unterscheiden. Ein Grund hierfür ist die mannigfaltigere Struktur von Software, in der Schleifen und rekursive Funktionsaufrufe praktisch beliebig ineinander geschachtelt sein können. Dies erschwert das iterative Abrollen von Zuständen erheblich, und die beschränkte Modellprüfung kann daher auch nicht ohne weiteres auf Software-Systeme übertragen werden.

SBMC konzentriert sich primär auf sogenannte *Sicherheitseigenschaften*, also auf die Frage "ob etwas schlechtes passieren kann". Für die Sprachen C und C++, die in eingebetteten Systemen vorwiegend eingesetzt werden, stellen die sogenannten Laufzeitfehler eine wichtige Gruppe von Sicherheitseigenschaften dar. Dabei handelt es sich beispielsweise um die Division durch Null, Speicherzugriffsfehler und arithmetische Überläufe.

Die Arbeit bietet einen detaillierten Einblick in Theorie und Implementierung des Low-level Software Bounded Model Checking Werkzeugs *LLBMC*. Dieses wurde maßgeblich im Rahmen dieser Arbeit konzipiert und realisiert. LLBMC hat mehrmals an internationalen Software-Verifikationswettbewerben (SVCOMP-2011, SVCOMP-2012 und SVCOMP-2013) teilgenommen und dabei mehrere Gold-, Silber- und Bronzemedaillen in verschiedenen Kategorien gewonnen. Beim Vienna Summer Of Logic wurde das LLBMC-Team für diese Leistung mit der Kurt-Gödel-Medaille ausgezeichnet. Der Autor dieser Arbeit wurde darüber hinaus für seinen wissenschaftlichen Beitrag mit dem Intel Doctoral Student Honor Award geehrt.

Anstatt C-Programme direkt zu analysieren setzt LLBMC auf dem Compiler Framework LLVM auf und nutzt dessen Zwischensprache LLVM-IR als Eingabesprache. LLBMC nutzt die Compiler-Optimierungen von LLVM um die Analyse erheblich zu beschleunigen. Für die Übersetzung von C-Programmen in LLVM-IR wird dabei auf den Code Generator des Compiler Frameworks zurückgegriffen In der Dissertation wird beschrieben, wie dieser Generator so angepasst werden kann, dass bei der Übersetzung keine Information verloren geht die für die Prüfung der gewünschten Eigenschaften relevant ist.

In dieser Dissertation wird auch die sogenannte *Intermediate Logic Representation* (ILR) vorgestellt. ILR bildet den Kern von LLBMC, ist eng an das Design von LLVM-

IR angelehnt und bildet so das logische Gegenstück dazu. ILR ist dabei gleichzeitig ein Schema für Theorien der Prädikatenlogik erster Stufe, als auch Grundlage für *Termersetzungssysteme*, welche eine zentrale Rolle in LLBMC spielen.

Ein zentraler Bestandteil dieser Arbeit ist die detaillierte Darstellung der Kodierung von beschränkten Fragmente von LLVM-IR Programmabläufen in ILR. Eine wichtige Rolle bei diesem Prozess spielen spezialisierte Varianten von *Aufruf-* und *Kontrollflussgraphen*, die ebenfalls in dieser Promotionsschrift dargestellt werden. Diese Graphen kodieren dabei unter anderem die für den beschränkten Modellprüfungsprozess benötigten Schranken bezüglich der Anzahl der Schleifeniterationen und der Rekursionstiefe. Die Kodierung ist als ILR-basiertes Termersetzungssystem formalisiert. Die bei der Kodierung entstehende ILR-Formel ist dabei genau dann erfüllbar, wenn das untersuchte beschränkte Programmfragment eine Sicherheitseigenschaft verletzt. Ist dies der Fall, so kann das Modell der Formel auf ein vollständiges Gegenbeispiel auf Programmebene zurück abgebildet werden.

Die Erfüllbarkeit von ILR-Formeln wird in LLBMC mit Hilfe sogenannter SMT-Solver festgestellt. Es handelt sich dabei um hochperformante Implementierungen effizienter Entscheidungsverfahren ausgewählter Theorien der quantorenfreien Prädikatenlogik erster Stufe. Die vorliegende Arbeit beschreibt, wie hierfür ILR-Formeln mit Hilfe von Termersetzung auf Formeln der Theorien der Bitvektoren und Arrays reduziert werden können.

Ein weitere bedeutender Beitrag dieser Arbeit ist eine Erweiterung von ILR zur Unterstützung dynamischer Speicherallokation und zur Prüfung der Korrektheit von Speicherzugriffen. Darüber hinaus wird ein partielles Entscheidungsverfahren für diese Theorie vorgestellt, das die für das Software Bounded Model Checking relevante Teilmenge von Probleme durch Termersetzung auf ein reines Bitvector-Problem reduziert.

Abschließend beschreibt die Arbeit wie der Kern-Algorithmus von LLBMC weiterführend eingesetzt werden kann, beispielsweise um eine große Zahl von Sicherheitseigenschaften gleichzeitig zu prüfen.

Zusammenfassend bietet die Promotionsschrift einen detaillierten Einblick in die Architektur und Funktionsweise des Software Bounded Model Checkers LLBMC. Zu den Beiträgen der Dissertation gehören der Einsatz einer Compiler-Zwischensprache und Lösungsvorschläge für die daraus resultierenden Herausforderung, die Beschreibung einer Kodierung von LLVM-Programmen in einer speziell hierfür entwickelten Logik, sowie Entscheidungsverfahren die es ermöglichen Eigenschaften von Speicherzugriffs- und verwaltungsoperation zu prüfen. Der Fokus der Arbeit liegt dabei auf der Verbesserung von Präzision, Skalierbarkeit und Vertrauenswürdigkeit von Software Bounded Model Checking im Einsatz zur Verifikation von Laufzeitfehlern in eingebetteten Systemen.

Contents

Chapter 1

Introduction

Software has become ubiquitous in the last few decades. This can readily be seen on mobile phones and tablet computers, on personal computers and workstations at home and in our offices, and certainly not the least on the Internet.

Equally important these days, but far less visible, is embedded software, software controlling devices that are generally not perceived as computers at all. This includes devices such as cars, trains, airplanes, rockets, cash machines, pacemakers, radio therapy machines, and many more. A fair number of these systems are safety or security critical, meaning failures can cause injury, death or other harm to persons or their assets.

At the same time, it is a matter of common knowledge that every non-trivial software system contains a considerable number of undetected errors, as shown in [McC04]. In this book, McConnell estimates an industry standard of 15-50 errors per 1000 line of code. In the past, system failures caused by software bugs in safety or security critical systems have caused monetary loss, injury, and have last but not least cost lives.

Therac-25. The *Therac-25* was a radiation therapy machine produced by Atomic Energy of Canada Limited in the mid-80s and was subsequently in use in several hospitals across the United States and Canada. The machine was particularly versatile because it was designed to operate in two different modes: one for low-power, direct electron beam therapy, and another for x-ray therapy. The later mode operated by having a high-power electron beam collide into a target made of tungsten to convert the beam into x-rays.

Between June 1985 and January 1987 the Therac-25 was involved in multiple incidents, in which patients were exposed to about a hundred times the intended dose of radiation. This resulted in death of three patients and caused serious injuries to three more.

While the Therac-25 incidents were never officially investigated, the cause was eventually identified to be a software error by independent researchers in [LT93]. The radiation therapy machine's predecessors, the Therac-6 and Therac-20, used

hardware safety interlocks to ensure that only the low-power electron beam and x-rays could hit the patient but never the high-power electron beam directly.

In contrast, the Therac-25 relied solely on software safety interlocks for this, which turned out to be an unfortunate design decision. If the machine's operator pressed a specific sequence of keys in less than seven seconds, a race condition occurred and caused the software interlocks to fail. Ironically, the more experienced and therefore the faster the operator was in handling the machine, the more likely this race condition was to occur.

Finally, even though the Therac-25 did issue a warning in this case, this warning was routinely ignored by the operator due to the large number of spurious warnings the machine regularly generated. With this, the last opportunity to prevent the disaster was missed.

The exposure to high energy radiation caused serious symptoms of radiation poisoning with all of the affected patients and has even led to the deaths of three of them, as reported in Leveson and Turner [LT93].

Ariane flight 501. In 1987, the European Space Agency (ESA) initiated development of a new rocket, Ariane 5, a "major evolution for the the Ariane family" [ESA15]. The ESA considers the rocket "the cornerstone of Europe's independent access to space" [ESA15] and calls it "one of the most reliable launchers in the world" [ESA15] though this certainly was not true from the beginning.

On the fourth of June 1996 the first ever launch of an *Ariane 5* rocket, flight Ariane 501, was scheduled to take place at ESA's Kourou based launch site. The rocket was intended to bring a set of four satellites to orbit. After briefly being put on hold due to bad visibility, lift-off finally took place at 9:33:59 local time. Approximately 37 seconds later, the rocket went off course and shortly after self-destructed.

On the 19th of July 1996 the Ariane 501 Inquiry Board which was tasked with investigating the catastrophe published the report *Ariane 5 – Flight 501 Failure* [AIB96]. In this report they concluded that a software error was the primary cause of the event.

Software from the Ariane 4 system was reused in the Ariane-5 rocket. This software was not suited for the higher horizontal acceleration of this new, more powerful rocket. This led to an overflow during conversion of a 64-bit floating point number to a 16-bit signed integer. Range checks, though enabled for most of the system, were omitted for this particular operation for efficiency reasons.

The self destruction of Ariane 501 is estimated to have caused a total loss of 370 million US dollars [Lan97].

Toyota Camry. In 2005 on a highway off-ramp in Oklahoma, a *Toyota Camry* unexpectedly and allegedly unintended by the driver accelerated, veered off the road, and caused an accident in which one of the occupants of the car was killed.

After a ten month investigation by NASA in 2010 and 2011 no evidence for the car's electronics having caused the crash was identified, but such an error could not be

ruled out entirely either. There simply was not sufficient time to come to a clear conclusion.

On October 24th 2013 in the course of an Oklahoma County lawsuit, a second, more thorough investigation by embedded systems experts was conducted. After 20 month of thoroughly reviewing the Toyota source code, this investigation came to the conclusion, that the unintended acceleration "[...] was more likely than not caused by the death of a redacted-name task, [...] " [SRS13]. Furthermore, "Toyota's electronic throttle control system (ECTS) source code is of unreasonable quality" [Dun13], "Toyota's source code is defective and contains bugs, including bugs that can cause unintended acceleration (UA)" [Dun13], and "Toyota's fail safes are defective and inadequate [...] " [Dun13].

The software bug cost Toyota at least $2.2 billion dollars [Gan14] and ruined the company's image, which was previously known to produce very reliable cars, for years to come.

Boeing 787 Dreamliner. The *Boeing 787 Dreamliner* was and still is Boeing's prestige project, with the company pitching it as a "game-changer" [Gat11]. The total cost of the program was estimated by the Seattle Times to be $32 billion in September 2011, half of which are development costs [Gat11].

On the first of May 2015 news spread around the world about a defect in the Dreamliner's software having been found. The Federal Aviation Administration issued an airworthiness directive dictating that the plane's generators require a reboot after 120 days of uptime [FR14].

The Boeing Dreamliner has four generators that provide AC power to the plane, two of them attached to each of the plane's two turbines. A counter in these generator's software was programmed to increase by one every tenth of a second, starting at zero when the system was booted. The counter was implemented using a 32-bit signed integer and as a consequence a little over 284 days of continuous uptime the counter would overflow. In this event, the counter's value would switch from a large positive number $(2^{31} - 1)$ to a large negative number (-2^{31}). The system was programmed to react to a value less than zero by transitioning to a fail-safe state. In this state, the affected generator would not provide power to the plane any more.

If all four generators were booted at the same time, which they often are, they would also transition to this faile-safe state at the same time. This in turn would shut down the plane's AC power system, rendering the plane uncontrollable. Had this happened during take-off or landing, the consequences would have been catastrophic. Fortunately, the bug was found during laboratory testing at Boeing, so this bug never lead to an incident and no one was harmed.

Even despite the massive damage for Boeing as a brand, the company can still consider itself lucky in this case. But luck should not be relied upon when it comes to safety critical software. Methods for the development of reliable software are of major importance in the development of this type of software.

1.1 Motivation

The radiation therapy machine Therac 25, the Ariane 5 rocket, the Toyota Camry, and the Boeing 787 Dreamliner all have one thing in common: they rely heavily on so called *embedded systems*. Embedded systems are computer systems that are part of a larger electric or mechanical system.

Unfortunately, the terms failure, error, fault, and defect are defined and used differently and may cause considerable confusion. We will use the terms as defined in *ISO 26262-1 - Road Vehicles – Functional Safety* [ISO26262]. We will use the term *software failure* (or simply *failure*) to refer to to "termination of the ability of an element to perform a function as required" [ISO26262]. We will use the term *software fault* (or simply *fault*) to refer to an "abnormal condition that can cause an element or an item to fail" [ISO26262]. A *software error* (or simply *error*) is the "discrepancy between a computed, observed or measured value or condition, and the true, specified or theoretically correct value or condition" [ISO26262]. We will use the term *software defect* (or simply *defect*) synonymously with the term software fault. In general, faults in the program's code lead to errors which in turn lead to the system's failures. The terms are not used consistently in literature, e.g. Parhami [Par97] uses a finer distinction between fault and defect, however this refinement is not required in the scope of this thesis.

The Therac 25 incidents, the explosion of Ariane 501, the Toyota Camry accident, and and the issuing of the airworthiness directive for the Boeing Dreamliner all share a common cause: their embedded software systems contained defects and these defects caused the failure of the system as a whole. All these systems share another property: they are *safety critical*. Failure of these systems can cause injury or death to human beings.

The failure of Ariane 501 and the Dreamliner incidents show that even in an industry with extremely strict safety standards, software defects and the resulting system failures are hard, if not impossible, to rule out completely. The Camry's failure on the other hand shows that these kinds of error can equally well occur in everyday systems, such as cars.

Software failures such as the ones listed above can have a wide range of different causes. In the case of Therac 25 this was a timing and concurrency error. Another cause is misconfiguration, as was the case for the crash of the Airbus A400M in Sevilla in May 2015 [Gal15]. In the case of the Toyota Camry not a single specific cause was made public, certainly to some degree because of the sheer number of different quality issues in the code. Last, but certainly not least, The Ariane 501 and the Dreamliner incidents were caused by runtime errors, the error type which this dissertation is primarily motivated by and focused on.

Runtime errors are errors which occur while the software system is running. This is in contrast to *compile time errors*, errors which occur when the software is compiled. Some of the most well known examples for runtime errors are the division by zero, arithmetic overflows, and array index out of bounds accesses. Runtime errors can result in the immediate termination of the program. Whether this happens or not depends mostly on which programming language the program is written in. For example the Java programming language's form of a runtime error, an uncaught `RuntimeException`, will always terminate the program. In contrast, for C it depends

to some degree on the particular kind of runtime error and to some degree on the implementation, meaning primarily the compiler and the target architecture. The premature termination of the program differentiates runtime errors from *functional errors*. Functional errors result in incorrect output, but never terminate the program prematurely. Note that runtime errors that do not cause termination of the program frequently result in subsequent functional errors.

The examples mentioned above show that current methods of software quality assurance are not yet sufficient to prevent such incidents. This is to a large degree because most methods in use today, including software testing, code reviews, and light-weight static code analysis, can only provide falsification but not verification. *Falsification* methods are able to detect faults in the code, but they are not able to prove their absence. *Verification* methods, on the other hand, provide guarantees, as strong as mathematical proofs, of the program's correctness.

Verification methods can be roughly split into two kinds: interactive tools and automatic tools. Interactive tools are often labor intensive, requiring a specialist to construct a proof of correctness by hand. In contrast, fully automatic tools do not require human interaction but often do not scale sufficiently well for the systems with millions of lines of code, a common size for embedded systems these days. Some error types are easier to verify automatically than others. For example, functional errors are often hard to verify, as are concurrency errors. Verifying the absence of runtime errors however is more feasible, though certainly not trivial, to do.

The explosion of Ariane 501 shows the dramatic consequences of runtime errors, while the issuing of the airworthiness directive concerning Boeing's Dreamliner illustrates nicely that runtime errors, while often underestimated, still are an unsolved problem. Consequently, static code analysis for the verification of runtime errors is an important challenge in the development and quality assurance of safety critical embedded software.

1.2 Challenges

These days, most industrially used methods for verification of runtime errors in embedded systems are based on abstract interpretation by Cousot and Cousot [CC77], e.g. the tool Astrée [Mau04]. However, as reported by Post et al. [Pos+08], the large number of spurious errors generated by these tools still poses a major drawback in their industrial application. Post et al. propose using software bounded model checking to reduce the number of false positives generated by abstract interpretation tools.

Bounded model checking (see section 2.2.5) is a variant of model checking which "[...] is widely accepted as an effective technique [...] " [Bie+03]. Software bounded model checking is the application of bounded model checking to software systems. Software bounded model checking operates in multiple steps: First a source code transformation is performed during which all loops in the program are unrolled a given number of times and all function calls are inlined up to a given depth. The resulting single, large function is then encoded in a bitvector formula. Finally, this formula is translated to a propositional formula in conjunctive normal form and then solved by an off-the-shelf SAT solver.

CBMC was the first publicly available software bounded model checker and was introduced by Clarke et al. [CKL04] in 2004. CBMC has largely defined the concept of software bounded model checking. In the following, when we refer to CBMC we also include its derivatives SMT-CBMC [AMP06] and ESBMC [CFM09]. Both tools are based on SMT solvers (see section 2.1.2) instead of SAT solver but keep to CBMC architecture apart from that.

Despite of CBMC's success there is still a number of challenges to be solved before software bounded model checking can be successfully applied in an industrial setting, namely

- increasing the precision,
- reliability and trustworthiness of the tool, as required in the context of tool qualification according to the relevant standard documents, e.g. [ISO26262],
- improving scalability up to the sizes of contemporary embedded systems,
- and extensive language support, in particular including the features and language variants used in the embedded software industry,

1.2.1 Precision

The first challenge approached in this thesis is precision. Precision is the capability to faithfully model a system's behavior for static analysis. Imprecise tools might generate false positives or false negatives. A false positive occurs when a tool reports an error which cannot happen in reality. A large number of false positives severely decreases developers' acceptance of a tool. In contrast, a false negative occurs when the tool is supposed to detect an error that can happen in reality but fails to do so. However, avoiding false negatives is particularly important if a tool is used not for falsification but for verification of safety critical software. If the tool is the only measure in the software development process to detect or prevent a specific kind of error, the software system's safety can be compromised.

C and C++ are notoriously hard to verify. Even though both languages are statically typed, they are also weakly typed. This means the type system can easily be subverted and therefore cannot be relied upon by a static analysis tool. Unfortunately, the type system is regularly subverted in embedded software development for reasons of efficiency. Furthermore, embedded software often makes use of low-level operations and optimizations, such as bit-shifts, bit-packing, and bit-stuffing. These require precise handling down to the bit-level which disqualifies many analysis methods, in particular those based on mathematical integers.

CBMC is based on bitvectors and therefore already supports bit-precision in many regards. However, CBMC's modeling of memory is not as precise as its modeling of bitvectors, due to its use of type based aliasing analysis. This can cause false positives as well as false negatives. Most static analysis tools are not perfectly precise, which can be acceptable if the limitations of the tool and the assumptions it makes during verification are communicated clearly. However, this is often not the case for software bounded model checkers.

1.2.2 Trustworthiness

When examining the results of the International Software Verification Competition [Bey12; BW13; Bey14] carefully, one cannot fail to notice that nearly every one of the participating tools produces incorrect results for some of the benchmarks. Based on our own experience and observations during participation, we assume that a significant number of these incorrect results are not caused by programmer's errors but can be considered evidence for limitations and imprecisions of the method implemented in the tool.[1]. as well as different interpretations of a programs semantics[2].

However, even though nearly all tools have such limitations and imprecisions, most tools do not communicate these clearly. Instead, most publications about static analysis tools (and this seems to be especially true for software bounded model checking) are limited to pseudo-code descriptions of the higher-level algorithms. Finer details, e.g. how the tool encodes a program's semantics in logic, are often not elaborated upon.

While CBMC's most prominent assumption, the bound, is clearly communicated, others are not. For example, CBMC changed its handling of strict-aliasing sometime between version 3.8 and 4.2. Where CBMC previously assumed a may-alias relation in certain situations, at some point the tool started assuming a must-alias relation. This change to the tool's type-based alias analysis is a considerable change to the tool's assumptions about the program's semantics and can have clearly observable effects on the tool's results. Nonetheless, neither the semantics before nor the semantics after the change are specified.

If the tool's assumptions about the program's semantics are not clear to the user, the user loses trust in the tool with every unexpected result. In order to increase the tool's trustworthiness it is necessary to document these assumptions.

A complete and clear specification is also important if the tool is to be qualified to act as an integral part of the development process for safety critical code. *ISO 26262-1 - Road Vehicles – Functional Safety* [ISO26262], which is the relevant standards document for the automotive industry, provides four methods for tool qualification. These methods include 1) proven-by-use reasoning, 2) an evaluation of the software development process, 3) a stringent validation of the tool, and 4) application of a software development process according to a safety standard, e.g. [ISO26262] itself. For a static analysis tool originating from a research project, 1), 2), and 4) are barely an option, leaving 3) as the only viable option. A stringent evaluation of the tool requires a clear and complete specification of the tool's limitations and imprecisions.

1.2.3 Scalability

The third challenge approached in this thesis is scalability. Scalability is the capability of a static analysis tool to successfully analyze increasingly large software systems. Embedded systems have grown steadily in recent years. For example, embedded

[1] Based on the lively discussions that occur on the competitions' mailing list during the preparation phase, we conclude that most participants try to fix any programmer's errors which are triggered by the competitions' benchmark set before submitting their tool. We certainly did.

[2] The C standard is notorious for leaving much undefined or unspecified.

systems in the automotive industry have reached sizes of several million lines of code. Any static analysis aiming to analyze such systems needs to scale accordingly.

Scalability is related to runtime and memory footprint. The unrolling of loops and the inlining of functions, which are at the core of software bounded model checking, are primarily responsible for the method's large memory requirements. For CBMC and its derivatives this is amplified by their implementations, which realize these operations as source to source transformations. This often causes these tools to fail even before the program is even converted to logic. Werner and Faragó [WF10] describe how this limitation became apparent when CBMC was used to prove correctness of sensor network applications.

1.2.4 Extensive Language Support

The fourth challenge approached in this thesis is full support for an industrially-used programming language. Embedded systems are often developed under constraints concerning processing power and memory available on the target architecture. This often leads to the use of the more obscure language features and optimizations which are often not well supported by static analysis tools. In addition, the languages evolve and change whenever new versions of the standard documents are released and modern compilers adopt new features early on, often even before the standard is finalized.

It is notable that CBMC supports a real programming language instead of simply a toy language or a simplified subset of a real language, which both are only useful in the academic context. Nonetheless, for several years, CBMC often failed at parsing programs and, as noted by Werner and Faragó [WF10], this requires workarounds to be able to use CBMC at all. This has improved notably, but considering the time needed for this, it can be assumed that this was a laborious task. Workarounds frequently involve code transformations, which are often fragile, in particular for C++. Furthermore, these code transformations are often done by the user with the help of scripts, which considerably reduces the trustworthiness of the process as a whole.

1.3 Contributions

This section briefly highlights the major contributions of this dissertation. The thesis and its contributions are closely linked to the software bounded model checking tool LLBMC, which is presented in depth in section 3.1.

LLBMC was one of the first tools to use LLVM's intermediate representation as an input language for a static analysis tool (see section 3.2 and [MFS12; FMS13c]). Using LLVM-IR greatly reduces the complexity of writing an academic static analysis tool targeting C and C++ and thereby enabled LLBMC surpass CBMC's level of *language support* quickly. As we describe in [FMS13b], the use of LLVM enabled LLBMC to use compiler optimizations for improving LLBMC's *performance*. Using a compiler IR is not without its drawbacks, as extra care needs to be taken to maintain the tools *precision*. Suitable measures have been addressed in section 3.3.

As discussed in Merz et al. [MFS12] and Falke et al. [FMS13c], LLBMC uses a flat memory model with a single symbolic array to model the process's entire address space. This greatly increases the tool's *precision* concerning memory related operations in particular when the type system is subverted by the programmer. Disadvantages of using a flat memory model concerning the tool's *performance* and *scalability* are counteracted with suitable formula simplifications and by using fixed addresses for most memory objects. LLBMC's performance in the Software Verification Competition, in particular in comparison with the competing software bounded model checker ESBMC, has shown that the approach is *precise* and *performs* well (see [FMS13b]).[3]

LLBMC's C *language support* extends itself also to the language's standard library. Most notable are two different approaches to C-style dynamic memory management which were presented in [SFM10; FMS11] and which are implemented in LLBMC. In chapter 5, a third approach is introduced which is a minor enhancement of the approach we presented in [FMS11] but, more importantly, comes with a proof of correctness. The approach is a *precise* formalization of the semantics of `malloc` and `free` and `realloc`.

Finally, the thesis provides a formalization of its encoding in chapter 4. This acts as a specification and documentation of how LLBMC models the C language, including its limitations and imprecisions. This aims to increase the tool's *trustworthiness*.

1.3.1 Additional Contributions

In addition to the research discussed and elaborated upon in this thesis, the author contributed to a number of scientific papers during the creation of this dissertation, which are also concerned with different aspects of software bounded model checking.

One aspect concerns the C standard library's functions `memset`, `memmove` and `memcpy`, which are frequently used in embedded projects. In software bounded model checking the loops contained in these functions must be unrolled, which can take up considerable amounts of memory. In [FSM12] we introduce a compact representation for these particular functions via an extension to the theory of arrays, as well as multiple decision procedures for the extended theory. The compact representation aims at increasing LLBMC's scalability for programs which use these functions. In [FMS13a] we expand upon this by introducing λ-functions for loop summarization, which also results in a compact representation for `memset`, `memcpy` and `memmove` and in addition for a range of similarly structured loops, too. Because of the compact representation of these core functions both approaches increase scalability if these functions are used in the source code or inserted by the compiler. Furthermore, the approach improves language support because `memset`, `memcpy`, and `memmove` function calls of any size are supported without the need for a loop bound.

Abstract testing is a novel concept first introduced in [Pos+09] which uses software bounded model checking for the symbolic execution of abstract test cases. The method is designed to bridge the gap between requirements engineering and software testing. This works because fewer of these abstract test cases are required and they

[3] LLBMC not only won multiple medals throughout the years but in 2013 it was also the only tool that did not generate a single incorrect result (section 6.3).

map more directly to the requirements. We advanced upon this idea in [Mer+15; Mer+10].

In [Bec+11] we use software bounded model checking to support the annotation-based verification process of tools such as VCC (see [Coh+09a]). When a proof does not close during verification it is often not immediately apparent if the proof did not close because the property does not hold or if the property holds but the tool was not able to proof it. In this context, the annotations are translated to executable code and passed to LLBMC with a small bound. If LLBMC provides a counterexample, the user knows the proof did not close because the property does not hold.

Finally, in [MSF12] we examine the challenges one faces when comparing software analysis tools. This reflects the authors' experiences during participation in the International Software Verification Competitions.

1.4 Overview of This Thesis

This section provides an overview of the following chapters of this dissertation.

Chapter 2 gives an introduction into the theoretical background required for understanding this thesis and provides an overview of related methods and the state of the art. We provide a minimal introduction into first-order logic, satisfiability modulo theories, the theory of bitvectors, the theory of arrays, and linear temporal logic in section 2.1. This is mainly to establish notational conventions used throughout this dissertation as we assume the reader is familiar with logic in computer science.

Chapter 3 introduces LLBMC, an award winning, state of the art software bounded model checking tool. The chapter starts off with an overview of the tool in section 3.1. This is followed by an informal introduction into LLVM's compiler intermediate representation LLVM-IR in section 3.2. Section 3.3 discusses the relationship between the source language and the compiler intermediate representation and how to translate the former into the latter with respect to C's concept of undefined behavior. Section 3.4 briefly shows how compiler optimizations interact with the use of a compiler framework as front end for a model checking tool and how optimizations can be used to improve model checker performance. LLBMC's approach to software bounded model checking makes control flow graphs and in particular call graphs explicit. These graphs are introduced in section 3.5 and section 3.6. Last but not least, section 3.7 presents ILR, LLBMC's intermediate logic representation and core data structure.

Chapter 4 presents the encoding of LLVM-IR programs in ILR formulæ via term rewriting. The chapter introduces a language extension to ILR which comprises of a number of sorts for different constructs of the LLVM-IR language, such as instructions and basic blocks, as well as a number of functions for reasoning about such language constructs, e.g. the memory state at certain points of time during execution. The sorts and functions are introduced in section 4.1. Sections 4.2 to 4.5 present a number of rewrite rules based on the newly introduced sorts and functions which, in combination, can be used for encoding bounded fragments of traces through LLVM-IR programs in ILR. Section 4.6 concludes the chapter by introducing the term rewriting system as a whole, providing variations thereof.

Chapter 5 discusses how dynamic memory allocation is handled in LLBMC. Section 5.1 provides an overview of dynamic memory related sections of the C standard. Section 5.2 introduces a theory of dynamic memory allocation for C. Section 5.3 defines a partial decision procedure for a subset of the theory presented in the previous section, which is based on the reduction of problems of this theory to pure bitvector (and arrays of bitvectors) problems. Section 5.4 shows how the theory can be used to model dynamic memory allocation in LLVM-IR programs. Finally, section 5.5 extends the concept of the safety of instructions to memory accessing instructions, thereby showing how the theory can be used to detect memory access errors for statically, dynamically, and automatically allocated memory in LLVM-IR programs.

Chapter 6 wraps up all matters concerning LLBMC by showing the formula simplifications in section 6.1, by presenting how ILR formulæ are converted to SMT-LIB formulæ, and by discussing how satisfiability of ILR formulæ is decided in section 6.2 and how this can be used to its best effects. Furthermore, section 6.3 provides a brief evaluation of LLBMC's performance in the International Software Verification Competitions.

Chapter 2

Theoretical Background and State of the Art

This chapter introduces terminology and concepts used throughout this thesis and provides an overview of the state of the art in software bounded model checking. Section 2.1.1 introduces terminology related to first-order logic, section 2.1.2 gives a short introduction into satisfiability modulo theories, section 2.1.5 introduces notational conventions for temporal logic, while section 2.1.6 does the same for term rewriting systems, and section 2.1.7 provides a few definitions concerning graphs which are used in this thesis.

Section 2.2.1 gives a short overview of the history of formal verification. Section 2.2.2 introduces the formal verification method of model checking. Section 2.2.3 introduces the therefrom derived method of symbolic model checking. Section 2.2.4 introduces bounded model checking, and finally, section 2.2.5 presents software bounded model checking and provides related work and the state of the art in this field of research.

2.1 Theoretical Background and Foundations

This section of the dissertation provides an introduction into the notational conventions used throughout this dissertation. Nonetheless, familiarity with many-sorted first-order logic and a basic understanding of temporal logic are required to understand this dissertation as whole.

2.1.1 Many-Sorted First-Order Logic

This section briefly introduces many-sorted first-order logic, primarily to introduce the notational conventions used through this thesis but also as a starting point for a brief discussion of the closely related Satisfiability Modulo Theories in section 2.1.2.

We will use $\top$ for true, $\bot$ for false, $\neg$ for negation, $\wedge$ for conjunction, $\vee$ for disjunction, $\rightarrow$ for implication and $\leftrightarrow$ for logical equality.

A *signature* Σ is a quadruple $(S_\Sigma, F_\Sigma, P_\Sigma, \alpha_\Sigma)$ where S_Σ is a set of sorts, F_Σ is a set of function symbols, P_Σ is a set of predicate symbols, and α_Σ assigns sorts to the function and predicate symbols. A function's or predicate's arity is given by the functional $\mathrm{arity}(f)$ with $f \in F_\Sigma \cup P_\Sigma$. Σ-*terms*, Σ-*formulæ*, and Σ-*sentences* are defined in the usual way.

A Σ-*structure* $\mathcal{D}$ contains non-empty, pairwise disjoint sets D_σ for every sort $\sigma \in S_\Sigma$, specifying the *domain* of σ, and an interpretation function I mapping function symbols in F_Σ and the predicate symbols in P_Σ to functions and predicates while respecting sorts and arities. We use $I(f)$ to denote the interpretation of $f \in F_\Sigma$ in $\mathcal{D}$ and $I(p)$ to denote the interpretation of $p \in P_\Sigma$ in $\mathcal{D}$. The interpretation of an arbitrary term t of sort σ in $\mathcal{D}$ is denoted $[\![t]\!]^{\mathcal{D}} \in D_\sigma$ and defined in the standard way. Similarly, $[\![\phi]\!]^{\mathcal{D}} \in \{\top, \bot\}$ denotes the truth value of a formula ϕ in $\mathcal{D}$. Finally, a structure $\mathcal{D}$ is a *model* of a formula ϕ if $[\![\phi]\!]^{\mathcal{D}} = \top$. A formula ϕ is called *satisfiable*, iff a model of it exists and *unsatisfiable* otherwise. We will call an algorithm that decides satisfiability of a formula a *decision procedure*.

A Σ-*theory* $\mathcal{T}$ is a set of Σ-sentences, its axioms. A Σ-theory is *single-sorted* if $|S_\Sigma| = 1$. For a single-sorted theory $\mathcal{T}_i$, its only sort is usually denoted by σ_i. Two signatures $\Sigma_1 = (S_1, F_1, P_1, \alpha_1)$ and $\Sigma_2 = (S_i, F_i, P_i, \alpha_i)$ are *disjoint*, if $F_1 \cap F_2 = \emptyset \wedge P_1 \cap P_2 = \emptyset$. A Σ_1-theory $\mathcal{T}_1$ and a Σ_2-theory $\mathcal{T}_2$ are *disjoint* if Σ_1 and Σ_2 are disjoint. The *combined theory* $\mathcal{T}_1 \oplus \mathcal{T}_2$ of two disjoint theories $\mathcal{T}_1$ and $\mathcal{T}_2$ is the $(\Sigma_1 \cup \Sigma_2)$-theory containing the union of $\mathcal{T}_1$'s and $\mathcal{T}_2$'s axioms.

The *equality* symbol $=_\sigma$ is implicitly defined for most sorts σ, though an explicit definition will be provided whenever considered beneficial. The symbol is not part of any signature Σ and is always interpreted as the identity relation over σ. Its subscript will be omitted usually.

For brevity we will also omit information about the sorts of terms and variables, though only if it can be deduced from their use as arguments in functions and predicates.

2.1.2 Satisfiability Modulo Theories

First-order logic is popular in formal methods research because it is sufficiently expressive to formulate a large number of interesting problems from this domain. However, due to quantifiers first-order logic in general is undecidable. Nonetheless, for a wide range of interesting problems a decidable subset of first-order logic exists.

Over time, a number of particularly versatile first-order logic theories has emerged and this consequently lead to growing research interest in these theories. Among these theories are for example the theory of equality and uninterpreted functions, the theory of linear integer arithmetics, the theory of linear real arithmetics, the theory of bitvectors, and the theory of arrays. The research on these theories resulted in a number of highly-optimized decision procedures. For example, efficient decision procedures for the theory of equality of uninterpreted functions often make use of the union-find algorithm, while decision procedures for the theory of integers often leverage the well-known Simplex algorithm.

Satisfiability module theories (*SMT*) is the field of research concerned with the satisfiability of many-sorted first-order logic formulæ with respect to certain theo-

ries. Initiated in 2003 by Ranise and Tinelli [RT03] SMT underwent a process of standardization via the *SMT-LIB* initiative and language. The standard is actively developed further and is now at version 2.5, presented in [BFT15]. If the term SMT is used in this thesis it is always used with the SMT-LIB standard in mind.

SMT differs from first-order logic in few, but very notable aspects. Likely the most important difference between SMT and first-order logic is the fact that in SMT logics, all sorts and most function symbols are already interpreted. In addition, in first-order logic a theory is a set of sentences, its axioms. In contrast, an SMT theory can be seen as a restriction on the set of possible models for a formula. This restriction can be formalized as a set of axioms, but it does not have to be. As a consequence, not every SMT theory is also a theory in first-order logic. Satisfiability checking in SMT is done "modulo" these background theories, hence the name Satisfiability Modulo Theories.

Furthermore, SMT, as defined in the SMT-LIB standard, allows for parametric and derived sorts, while this is uncommon in first-order logic. For example, the theory of bitvectors does not define a single bitvector sort, but an infinite number of sorts distinguished by an integer parameter indicating the bitwidth. Similarly, the theory of arrays is derived from an index and an element sort. In this chapter, parametric and derived sorts will be indicated by a parentheses-enclosed, comma-separated list of the parameters and sorts used, e.g. the sort $\sigma_{BV}(n)$, with σ_{BV} being the sort symbol for bitvectors and n being an integer, gives the sort of bitvectors of size n, and the sort $\sigma_A(\sigma I, \sigma E)$, with σ_A being the sort symbol for arrays and σ_E and σ_I being other sorts, indicates the array sort derived from the index sort σ_I and element sort σ_E. Note that we will use a different notation in chapters 3 to 5, which follows LLVM-IR's notation for types more closely.

When referring to functions defined in any of the theories defined by SMT-LIB, we will follow SMT-LIB's naming of the those functions, though we will use first-order logic syntax instead of SMT-LIBs Lisp-inspired syntax. We use a subscript to specify additional information, wherever required. E.g. the SMT-LIB function `(bvadd x y)` for the bitvector addition of `x` and `y` will be written as $\mathrm{bvadd}(x, y)$. while the expression `((_ extract 15 8) x)` for extracting bits 8 to 15 from the bitvector `x` will be written as $\mathrm{extract}_{8,15}(x)$.

Finally, SMT does not distinguish syntactically between formulæ and terms. Instead formulæ are terms of a special, boolean sort. Because SMT's booleans are terms, in contrast to formulæ in first-order logic, they can be used as arguments to functions. This makes it syntactically possible in SMT to define the ternary function ite (if-then-else, see [RT03]), which takes a boolean as its first argument and returns the second argument if the first argument is true and the third argument otherwise:

$$\forall c, x, y, z \left(x = \mathrm{ite}(c, y, z) \leftrightarrow c \wedge x = y \vee \neg c \wedge x = z\right) \tag{2.1}$$

Just like equality, this function is implicitly defined in SMT-LIB for all sorts.

As is common in SMT, we restrict ourselves to ground formulæ and make frequent use of uninterpreted constant symbols.

SMT has proven to be highly useful for a wide range of verification tools, with different tools using different theories. For LLBMC, out of the SMT theories standardized or suggested so far, the two theories that are the most relevant are the theory of

bitvectors and the theory of arrays. Furthermore, LLBMC's use of these theories is restricted to the quantifier free fragment. Because of the finite nature of the theory of bitvector and the theory of arrays (of bitvectors), the resulting formulæ are decideable. However, this comes at the cost of only being able to analyze bounded fragments of programs.

2.1.3 The Theory of Bitvectors

The theory of bitvectors is concerned with modeling vectors of bits and operations on these vectors in logic. The functions defined in this theory are based on the core set of operations supported by the arithmetic logic unit of a computer's CPU. These are bitwise logic operations, such as bitwise conjunction and disjunction, but also arithmetic operations, such as addition, subtraction, multiplication, et cetera.

Due to the limited number of bits in each bitvector, these arithmetic operations have subtle differences compared their regular definition for mathematical integers. E.g. the expression $x + 1 > x$ is always true for mathematical integers, but might not be true if $+$ and $>$ are interpreted as operations on bitvectors. This is because if x is the largest representable value for a given bitvector type, the additional will wrap around and result in the smallest representable bitvector for $x + 1$. This is obviously smaller than x itself, making $x + 1 > x$ false for this particular value.

Most implementations of bitvector solvers apply bit-blasting in one way or the other. The idea behind bit-blasting is to reduce a formula in the theory of bitvectors to an equisatisfiable formula in propositional logic.[1] This formula can then be solved with an off-the-shelf SAT solver. For this, each bitvector variable is represented by a vector of separate propositional logic variables. Functions and predicates on the bitvectors are then replaced by propositional logic formulæ modeled after the circuit of the function or predicate in question. Bitvector addition, for example, is usually bit-blasted to a formula that mimics the circuit of a ripple-carry adder.

Pure bit-blasting does not make use of syntactically recognizable tautologies on the bitvector level such as $x + y = y + x$. Instead with bit-blasting alone, the circuit representation of $x + y$ and that of $y + x$ would be generated and the resulting bit-vectors would be compared to each other bit by bit. Because of this, bit-blasting is usually combined with formula transformations, that try to identify these cases and simplify them before bit-blasting happens.

More importantly though, modern bitvector solvers try to avoid bit-blasting the whole formula up-front, as this might not even be necessary to decide satisfiability but can be very costly. Instead, an over-approximation of the real formula is passed to the internal SAT solver and if the solver finds a model, it is checked against the background theory for bitvectors. If the model is not valid with respect to the theory the necessary circuitry to rule out this particular model is bit-blasted and the SAT solver is run again. One such, introduced by Brummayer and Biere [BB09] for bitvector solvers, is called lemmas on demand.

[1] Each bit in each bitvector in the bitvector formula is identified with a variable in the propositional logic formula.

2.1.4 The Theory of Arrays

The second SMT theory highly relevant for this thesis is the theory of arrays. This theory models arrays and contains three sorts, the sort I for indices, the sort E for elements, and the binary sort $\sigma_A(\sigma_I, \sigma_E)$ for arrays mapping from σ_I to σ_E. For brevity we will use σ_A instead of $\sigma_A(\sigma_I, \sigma_E)$, if σ_I and σ_E are obvious. The theory has only two functions,

$$\text{select} : \sigma_A \times \sigma_I \to \sigma_E$$

for retrieving the value stored at a specific index in the array, and

$$\text{store} : \sigma_A \times \sigma_I \times \sigma_E \to \sigma_A$$

for updating the element at a certain index in the array. The theory is concisely described by McCarthy's [McC62] axioms:

$$\forall a, i_1, i_2, e \left(i_1 = i_2 \to \text{select}(\text{store}(a, i_1, e), i_2) = e\right) \tag{2.2}$$

$$\forall a, i_1, i_2, e \left(i_1 \neq i_2 \to \text{select}(\text{store}(a, i_1, e), i_2) = \text{select}(a, i_2)\right) \tag{2.3}$$

$$\forall a_1, a_2 \left(a_1 = a_2 \leftrightarrow \forall i \left(\text{select}(a_1, i) = \text{select}(a_2, i)\right)\right) \tag{2.4}$$

The first axiom states that if one writes to an array at a specific index and then reads from the result array at the same index, then the result is equal to the previously written element. The second axiom states, that writing to a specific index does not change elements subsequently read from a different index. The third axiom specifies that two arrays are equal if all elements are equal. this property is called *extensionality*. Note that not all SMT solvers support extensionality, e.g. STP, LLBMC's preferred SMT solver, never has while Boolector did and Boolector 2 at the time of writing does not.

A simple and straightforward decision procedure for the theory of arrays (without extensionality) can be implemented in two steps. First as a term rewriting system, which "moves" all select operations over the store operation that makes up its first argument, by exhaustively replacing all $\text{select}(\text{store}(a, i_1, e), i_2)$ by $\text{ite}(i_1 = i_2, e, \text{select}(a, i_2))$. All store functions can now be removed from the formula. Then, in a second step, all remaining select functions are replaced by uninterpreted terms $r_1, r_2, \cdots \in \sigma_E$ and for all pairs (r_i, r_j) the constraint $i = j \to r_i = r_j$ is added to the formula.

The theory of arrays is used in LLBMC to model memory and accesses to memory. It is notable, that LLBMC does not have a hierarchical memory model, but uses a single array to model the whole range of addressable memory.

2.1.5 Temporal Logic

Temporal logic provides means to reason about the passage of time without having to talk about time or points in time explicitly. Temporal logic is best suited for reasoning about concurrent systems but can also be used for sequential program. In this thesis we restrict ourselves to linear temporal logic (LTL), as is common for bounded model checking.

We assume the reader is familiar with LTL and will content ourselves with a short introduction into the notational conventions used in this thesis. Using Backus-Naur form, we defined an LTL formula as follows:

$$\phi, \psi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi \vee \psi \mid \phi \wedge \psi \mid \mathsf{X}\,\phi \mid \mathsf{F}\,\phi \mid \mathsf{G}\,\phi \mid \phi\,\mathsf{U}\,\psi \mid \phi\,\mathsf{R}\,\psi$$

Here, p is a proposition or a predicate, the symbols $\top$, $\bot$, $\neg$, $\wedge$, $\vee$ are defined as usual. Furthermore, $\mathsf{X}\,\phi$ indicates ϕ holds in the next state, $\mathsf{F}\,\phi$ indicates ϕ holds in some future state, $\mathsf{G}\,\phi$ indicates ϕ holds in all future states, $\phi\,\mathsf{U}\,\psi$ indicates ϕ holds until ψ holds and ψ is eventually true, and $\phi\,\mathsf{R}\,\psi$ indicates ϕ holds until ψ holds or ϕ holds forever.

Temporal logic makes it possible to express safety and liveness properties concisely. *Safety properties* assert that "nothing bad (ever) happens". In temporal logic a safety property is expressed as $\mathsf{G}\,\neg\phi$ where ϕ describes an error state. while *liveness properties* assert that "something good eventually happens". In temporal logic, a liveness property is express as $\mathsf{F}\,\phi$, where ϕ describes a state that should eventually be reached.

Another way to distinguish these properties is by the properties of the counterexamples required. For safety properties, finite counterexamples are sufficient to show that they do not hold, while liveness properties require infinitely long counterexamples shaped like a lasso.

2.1.6 Term Rewriting Systems

Term rewriting is a major part of this thesis, as can be seen in chapters 4 and 6. Term rewriting provides a formalism for describing how terms can be transformed, often to retrieve simpler but equivalent terms. Notational conventions used in this book are loosely based on the conventions introduced by Bündgen [Bün98]. Because we apply term rewriting to first-order logic terms, we will use terms and functions as defined in section 2.1.1.

We will use $l \longrightarrow r$ to indicate a *term rewriting rule*, where the term l can be rewritten to r. For example, we can provide a more formal description of the first part of the decision procedure for the theory of arrays hinted at in section 2.1.4, by using the ite function defined in equation (2.1) to combine the theory of array's axioms in equations (2.2) and (2.3) into a single rewrite rule

$$\mathrm{select}(\mathrm{store}(a, i_1, e), i_2) \longrightarrow \mathrm{ite}(i_1 = i_2, e, \mathrm{select}(a, i_2)). \tag{2.5}$$

Applying this rewrite rule to the term

$$y = \mathrm{select}(\mathrm{store}(\mathrm{store}(a, k, y), j, x), i))$$

once, we obtain the term

$$y = \mathrm{ite}(j = i, x, \mathrm{select}(\mathrm{store}(a, k, y), i)).$$

Note how the subterm matching the left hand side of the rewrite rule was replaced by the right hand side of the rewrite rule. We can apply the same rewrite rule again to obtain

$$y = \mathrm{ite}(j = i, x, \mathrm{ite}(k = i, y, \mathrm{select}(a, i))).$$

The rewriting now terminates, because even though the term still contains a select, we cannot apply the rewrite rule on it because its first argument is not a store and the subterm does not match the left hand side anymore.

We will call a set of rewrite rules a *term rewriting system* and follow Bündgen's definition in [Bün98, p. 31]. A term rewriting system is applied to a term (or formula) by applying all rewrite rules in the term rewriting system on the term (respectively formula) and all of its subterms until no further rule can be applied.

We will frequently use *conditional term rewriting rules*. Such rules can only be applied if a given condition is met. We will write $l \longrightarrow r \,;\, c$ to indicate that the rewrite rule $l \longrightarrow r$ can be applied only if the condition c holds.

For example, the rewrite rule in equation (2.5) can be supplemented by the following two rules:

$$\begin{aligned} \mathrm{select}(\mathrm{store}(a, i_1, e), i_2) &\longrightarrow e \,;\, i_1 = i_2 \\ \mathrm{select}(\mathrm{store}(a, i_1, e), i_2) &\longrightarrow \mathrm{select}(a, i_2) \,;\, i_1 \neq i_2. \end{aligned}$$

With this rule we can rewrite the term $\mathrm{select}(\mathrm{store}(a, 0, e), 0)$ to e instead of $\mathrm{ite}(0 = 0, e, \mathrm{select}(a, i))$.

We will frequently be able to evaluate the conditions for the rewrite rules in this dissertation statically. For example, all rewrite rules in chapter 4 only depend on syntactic properties of the program under verification. Because of this, for any given program a finite set of non-conditional rewrite rules, which is sufficient to rewrite that specific program, can be derived from a conditional rewrite rule. We will refer to this kind of conditional rewrite rule as a *term rewriting rule schema*.

For convenience we will occasionally use a syntax similar to term rewriting rules to construct sets of similarly structured rewrite rules from a template. For example, given the template

$$\mathrm{ordered}(a, b, c) \longrightarrow a < b \wedge b < c \tag{2.6}$$

we can rewrite the rewrite rule

$$\mathrm{foo}(a, b, c, d, e) \longrightarrow \mathrm{ordered}(a, b, c) \wedge \mathrm{ordered}(c, d, e) \tag{2.7}$$

into the rewrite rule

$$\mathrm{foo}(a, b, c, d, e) \longrightarrow a < b \wedge b < c \wedge c < d \wedge d < e. \tag{2.8}$$

Instead of applying the rule in equation (2.7) to the term and afterwards the rule in equation (2.6) on the resulting term, we apply the second rule to the first and then apply the resulting rule to the term. The result is the same, though it reduces the overall number of rewrite operations for many formulæ.

We use such templates in multiple places for different reasons. For example, we introduce such a template in equation (4.1) to work around the polymorphism coming from LLVM's concept of a value, without having to introduce a large number of nearly redundant rules for all subtypes of these values, such as instructions, basic blocks, or constants. Furthermore, we present a meta rule in equation (4.39) to handle disjunction of more than two arguments where the number of arguments is known in advance for every instance but might be different each time. Finally, we introduce two such rules in equation (5.1) simply to avoid having to add the functions $\mathrm{contains}$ and $\mathrm{disjoint}$ to the term rewriting system.

2.1.7 Graphs

This dissertation uses two different kinds of graphs prominently, control flow graphs (see section 3.5) and call graphs (see section 3.6). Control flow graphs describe the control flow between basic blocks in a single function, while call graphs describe control flow between functions. These two types of graphs are used in LLBMC for example to implement the bounding of the control flow from which the method software bounded model checking has its name.

Graphs are defined as usual:

Definition 2.1 (Graph). *A graph is an ordered tuple* (V, E)*, where* V *is a set of vertices and* E *a set of edges, with* $E \subseteq (V \times V)$*.*

A graph can be directed or undirected:

Definition 2.2 (Directed graph). *A directed graph is a graph* (V, E)*, where each* $e \in E$ *is a pair.*

Consequently, an undirected graph is a graph where each $e \in E$ is an unordered pair. However, in the following, we will use directed graphs exclusively and any graph mentioned in this thesis may be assumed to be directed.

Figure 2.1 shows an example for a graphical representation of a directed graph with three vertices (v_1, v_2, v_3) and four edges between them ((v_1, v_2), (v_1, v_3), (v_2, v_3), and (v_3, v_1)).

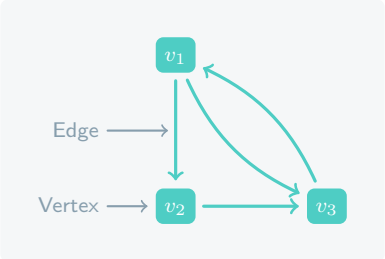


Figure 2.1: Example of a directed graph

When talking about calling relations between functions and when discussing counterexamples we will require walks over graphs:

Definition 2.3 (Walk). *Given a graph* (V, E)*, a walk* $w = (v_1, e_1, \ldots, e_{n-1}, v_n)$ *is an alternating sequence of vertices (*$v_i \in V$*) and edges (*$e_i \in E$*). Its first and last element are vertices and each edge connects its predecessor and its successor vertex:* $e_i = (v_i, v_{i+1})$*.*

The graph in figure 2.1 contains the walk $(v_1, (v_1, v_3), v_3, (v_3, v_1), v_1, (v_1, v_2), v_2)$, while $(v_1, (v_1, v_2), v_2, (v_2, v_1), v_1)$ is not a walk in this graph, because (v_2, v_1) is not an edge in this graph and $(v_1, (v_2, v_3), v_3)$ is not a walk at all.

We furthermore require the definition of a cycle:

Definition 2.4 (Cycle). *Given a graph* (V, E)*, a cycle is a walk* $(v_1, e_1, \ldots, e_{n-1}, v_n)$*, where* $v_1 = v_n$*.*

In the example in figure 2.1, $(v_1, (v_1, v_3), v_3, (v_3, v_1), v_1)$ is a cycle.

This definition immediately leads to one of the most important graph related definitions concerning software bounded model checking:

Definition 2.5 (Acyclic graph). *An acyclic graph is a graph that does not contain cycles.*

The graph in figure 2.1 is, for example, not acyclic.

Given the definition of a walk in definition 2.3 we can now define a connected graph:

Definition 2.6 (Connected graph). *A connected graph (V, E) is a graph, where for all $v_1, v_2 \in V$ there is a either a walk $w = (v_1, \ldots, v_2)$, or a walk $w' = (v_2, \ldots, v_1)$.*

An even stronger restriction on graphs is the tree:

Definition 2.7 (Undirected tree). *An undirected tree is a graph in which all pairs of vertices are connected by exactly one path.*

We can now define a directed tree with an explicitly named root:

Definition 2.8 (Directed, rooted tree). *The tuple (V, E, r) is a directed, rooted tree, if (V, E) is directed graph and $\forall u \in V \left((u, r) \notin E\right)$. r is called the* root *of the tree.*

figure 2.2 shows an example of a directed, rooted tree.

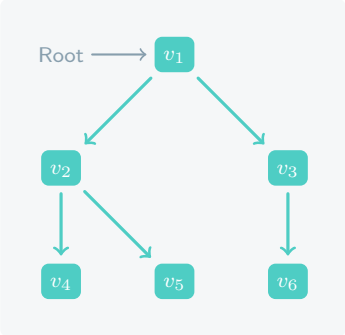


Figure 2.2: Example of a directed, rooted tree (V, E, v_1)

2.2 Related Work and State of The Art

Software bounded model checking, the primary focus of this dissertation, is just one small single step in the research on software verification done over many years. This section provides an overview of the historical context of software bounded model checking as well as insight into the current state of the art concerning this verification method.

2.2.1 A Brief History of Formal Software Verification

Already in the earliest years of computer science as an academic discipline, formal mathematic proofs of a program's correctness were considered desirable. In 1949 Alan Turing asks in his paper "Checking a large routine" "How can one check a routine in the sense of making sure that it is right" (Turing [Tur49, page 67]) and consequently introduces an approach on how to do exactly this.

Interest in formal methods for software verification further rose in the 1960s with McCarthy [McC62] introducing a basis for a mathematical theory of computation and Naur [Nau66] presenting so called snapshots for conducting semi-formal proofs on programs. Roughly at the same time, but mostly independently of each other, Robert Floyd and C. A. R. Hoare approached the same issue in a more formal and axiomatic way, Floyd [Flo67] for flowcharts and Hoare [Hoa69] for program code. Both provided formal semantics for a programming language's constructs and thereby made formal reasoning about said programs possible.

This laid the foundation for what became later known as *Floyd-Hoare logic*: a formal system for reasoning accurately about the correctness of programs. For several years, Floyd-Hoare style verification methods were predominant in this area of research. The methods used at this time, however, were highly theoretical and relied heavily on theorem proving, which requires a lot of ingenuity from the user.

In the 1970s Pnueli [Pnu77] first described a temporal logic of programs. He applies temporal logic, which was previously predominantly used in philosophy, to software systems. Instead of reasoning about the program's input and output, Pnueli argues for reasoning about the ordering of events in time. Temporal logic makes it possible to do so without introducing time explicitly, and thereby allows more elaborate reasoning about parallel and reactive systems. The approach is particularly well-suited for typical desired properties of concurrent systems, like mutual exclusion or the absence of deadlocks. Pnueli, though, still advocated hand-constructed proofs. This only changed with the advent of model checking.

2.2.2 Explicit State Model Checking

When Clarke and Emerson acquainted themselves with Pnueli's work on temporal logic in 1981, they realized that the proof-theoretic approach can be replaced, in suitable cases, by a model-theoretic one. This is important in so far as this way, proof construction, which is tedious and requires a fair amount of ingenuity, is not required anymore. This led to the invention of *model checking* [CE81].[2] The term model checking nowadays refers to a diverse number of related methods. The classic approach introduced in [CE81] and discussed in the following is nowadays commonly referred to as *explicit state model checking*.

In short, explicit state model checking refers to the problem of showing that a given structure (usually a Kripke structure) is a model for a given formula (usually provided in temporal logic). explicit state model checking represents a system's state space as a so called *Kripke structure*. A Kripke structure is essentially a directed graph whose nodes represent system states and whose edges represent state transitions.

[2] The concept was discovered independently by Queille and Sifakis [QS82].

Additionally, a labeling function maps states to sets of atomic propositions that hold for each state. These propositions are used in temporal logic formulæ to specify the desired properties of the system. Each temporal logic modality can be characterized as a fix point of a monotonic functional over the set of propositions over the set of states. In [CE81] multiple decision-procedures for the model checking problem are proposed, all based on the calculation of said fix point, with the best-performing algorithm being closely related to Tarjan's algorithm for finding strongly connected components.

Initially model checking was applied primarily on abstract state machines (e.g. models of network protocols), but it was also used for hardware verification as early as [MC85]. Since then, research on explicit state model checking has progressed considerably, with the SPIN model checker, presented in [Hol97], being one of the most prominent examples for modern explicit state model checkers. SPIN is also notable in that it supports embedded C code as part of the model specifications, and also supports model checking of C programs, as presented in [Hol00].

The key disadvantage is the so called *state space explosion*: For realistic systems, the number of states grows exponentially, which makes reasoning about these systems infeasible. For example a process with 4 32-bit registers would already have 2^{128} states, which is infeasible to handle with an explicit state model checker. While various approaches were invented to counteract this, e.g. by making use of symmetry, the state space explosion is still by far the biggest limitation of explicit state bounded model checking.

2.2.3 Symbolic Model Checking

A major break-through in extending the number of states a model checker can handle was *symbolic model checking*, introduced by Burch et al. [Bur+90].

In symbolic model checking the set of reachable states is represented by a characteristic boolean function. This function is represented using *binary decision diagrams* (*BDD*).[3] This way the Kripke structure does not need to be built, and this approach thereby avoids the space consumption of the Kripke structures. Efficient operations on BDDs for negation, conjunction, substitution and quantification enable a far more efficient fix point calculation, allowing model checking of systems with up to 10^{20} states and beyond.

Examples for successful implementations of symbolic model checking include the pioneering SMV, presented by McMillan [McM93], as well as the newer NuSMV, presented by Cimatti et al. [Cim+99]. The latter being a reengineering, reimplementation and extension of the former. Furthermore, symbolic model checking was applied successfully in the industrial environment. As reported by Fix [Fix08], the method is used at Intel for finding bugs in their system design since 1995. In particular in CPU design, symbolic model checking found defects in the design that otherwise would either have been found much later in the process (making fixing these defects considerably more expensive) or that even might have slipped through validation and verification and ended up in the finished product. The use of symbolic model checking at Intel was, to no small part, driven by the fact that the Pentium

[3] Strictly speaking it is reduced ordered binary decision diagrams (ROBDD).

FDIV bug from 1994, which cost Intel an estimated $400-500 million and made the company the object of ridicule around the world, could likely haven been prevented by the use of model checking.

While the state space explosion problem is reduced considerably with symbolic model checking, it is not eliminated. The ordering of variables has a huge impact on the size of a BDD and it can be hard to find a good ordering for a given formula. But even more troubling is the fact that for some problems no efficient encoding exists at all. And these cases are highly relevant, in particular for model checking of software systems, with a prominent example being integer multiplication. As Bryant [Bry86] proved, no good variable ordering exists for integer multiplication.

2.2.4 Bounded Model Checking

An approach to mitigate the state space explosion proposed by Biere et al. [Bie+99] is called *bounded model checking* (*BMC*). Bounded model checking uses SAT solving instead of BDDs at its core and checks bounded traces through the system.

Bounded Model Checking works by unrolling the system k-times. This means for a system with a state representable by n bits, kn bits are allocated in total, with n bits for each of the k unrolled copies of the state. The method then adds constraints, which encode the transition relation for each pair of successive states, to the propositional logic formula. Additionally, the desired properties are negated and expressed in propositional logic and added to the formula. A SAT solver is then used to check for satisfiability of the resulting formula. If the formula is satisfiable, the generated model can be translated into a sequence of states that make up a counterexample. However, if the formula is unsatisfiable for a given bound k, the bound is increased by one until either a counterexample is found or a sufficiently large bound is reached. Biere et al. [Bie+99] introduces various methods for estimating an upper bound for the bound.

The method has a number of advantages over symbolic model checking. For once, it suffers less from the space explosion problem than BDD based methods do. In addition, BMC finds counterexamples quickly. This is to some degree because at their core SAT solvers follow a depth-first approach, but this is also due to the highly optimized SAT solvers available today. By starting with short paths through the program and gradually increasing their length by one, BMC can also guarantee that it finds shortest counterexamples. This is important because shorter counterexamples are in general easier to understand than longer ones. Last but not least, unlike with BDDs, no manual variable ordering is required.

Due to the state space explosion problem, "[m]odel checking is often used for finding logical errors ('falsification') rather than for proving that they do not exist ('verification')" [BAS02]. Furthermore, "[i]n practical application, checking of safety properties is prevalent" [BAS02]. Bounded Model Checking continues this trend in that it is best suited for falsification of safety properties.

2.2.5 Software Bounded Model Checking

After the success of bounded model checking for hardware systems, the application of bounded model checking to software systems seemed to be the logical next step. Unfortunately, software systems in general have a markedly different structure to hardware systems. Hardware systems mostly have a well-defined state which consists of the CPU registers and the system's memory, as well as single control loop that can be unwound easily. In contrast, software systems in general have nested loops and potentially recursive function calls as well as a dynamically growing and shrinking state space resulting from heap and stack based memory allocation.

Nonetheless, bounded model checking can be applied to software, though the approach differs significantly from hardware bounded model checking. The first tool to implement *software bounded model checking* was CBMC, the C Bounded Model Checker. The tool was first presented by Clarke et al. [CKL04] and is still actively developed at the University of Oxford under supervision of Daniel Kröning. The following description of CBMC's mode of operation is based heavily on the publication mentioned above and the slightly more in-depth technical report [CKY03].

CBMC transforms a C program into a CNF formula in seven steps:

1. The program is preprocessed using a regular C preprocessor eliminating all preprocessor directives, such as `#define` and `#include`.
2. The program is transformed into an equivalent but simplified structure, e.g. `break` and `continue` are replaced by equivalent `goto` statements.
3. Structured loops are unwound, i.e. the loop body is repeated n times, each copy of the body being guarded by appropriate `if` statements. An unwinding assertion is inserted after the last copy of the loop body to ensure the selection of an insufficiently large bound n is detected.
4. Unstructured loops, i.e. those using backwards `goto` statements are unwound similarly to structured loops.
5. Function calls are expanded. Recursive calls are unwound up to a certain bound. An inlining assertion is inserted similarly to the one for loop unwinding. The resulting program consists solely of assignments, `if` statements, assertions, labels, and forward jumping `goto`s.
6. The program is transformed into single static assignment form. In this form, each variable is only assigned once, on declaration, making each variable essentially a constant value. This form is syntactically already mostly identical to a bitvector formula.
7. Finally, the bitvector formula is bit-blasted to a CNF formula.

The CNF formula is then passed to a SAT solver, in this case MiniSat [ES03], for solving, and if the formula is satisfiable, the model found by the SAT solver is translated back into a counterexample for the C program.

CBMC was quite successful and inspired development on SMT-CBMC [AMP06] is a prototypical adaptation of CBMC which uses SMT solvers as their back end instead of SAT solvers. Work on SMT-CBMC is, to the best of the author's knowledge,

discontinued. In the same year F-Soft was presented to the public in [GG06]. F-Soft, like SMT-CBMC, uses SMT solvers. The tool is conceptually closer to more traditional model checkers than CBMC and its derivatives, but improves on previous work on traditional model checking in the extraction and efficient use of high-level information.

In 2009 ESBMC, a second adaptation of CBMC to use SMT solvers, was presented to the public by Cordeiro et al. [CFM09]. In contrast to SMT-CBMC, it is still actively developed and a considerable number of research papers are based on it. For example the tool was extended to support C++, as described by Ramalho et al. [Ram+13].

In 2010 LLBMC was presented to the public by Sinz et al. [SFM10]. Like SMT-CBMC and ESBMC, LLBMC is based on SMT solvers, but in contrast to the former two, LLBMC uses a compiler framework as a front end. LLBMC is the focus of the present thesis and will be presented in-depth in chapter 3.

Another project that took a similar approach to LLBMC was LAV, published by Vujosevic-Janicic and Kuncak [VK12]. LAV followed LLBMC's example in using LLVM as its front end and an SMT-solver as its back end. LAV differs from LLBMC in that it incrementally extends the context in which each LLVM-IR instruction is checked. At first, LAV checks the instruction by itself, and only if the instruction is not safe by itself is it checked in the context of the containing basic block. If necessary, the checking context can then be extended to include the whole function. Unfortunately, LAV never competed in the International Software Verification Competition, therefore no direct comparison between LLBMC and LAV performance exists.

The tool FAuST by Riener and Fey [RF12] implements software bounded model checking on top of LLVM-IR and SMT solvers. FAuST thereby closely follows LLBMC's approach. FAuST extends this by an application layer which makes it easy to use SBMC for specific use cases such as test case generation.

Interestingly, with the exception of F-Soft, for none of these tools temporal logic seems to play the prominent role that it did for the more traditional model checking techniques. This is certainly because these tools only support checking safety properties ($\mathsf{G}\,\phi$). Because of this, the inventor of model checking, Clarke, would not consider these tools model checkers at all, as he explains in Clarke [Cla08].

Case studies, like those presented by Post [Pos09] and Post et al. [Pos+09], have demonstrated the feasibility of Software Bounded Model Checking for real world embedded software. This includes application of the method on an implementation of the AES crypto system, multiple versions of the Linux operating system kernel, and several software products from the automotive industry. This is even true for the verification of functional properties, e.g. for the Advanced Encryption Standard (AES) in [Pos09], but also for automotive code by Post et al. [Pos+09].

Current research on software bounded model checking seems to focus primarily on applying the method on new use cases, as well as on ways to better cope with the inherent limitations of the method caused by the boundedness.

One approach at going past the bounds of bounded model checking attempts to use k-induction for this. k-induction was invented by Sheeran et al. [SSS00] and adapted to software bounded model checking by Donaldson et al. [DKR11; Don+11]. A core limitation in bounded model checking comes from the fact that all traces that reach

a loop bound are cut off at this point. This means checking a loop that needs to run at least n times with a bound $k < n$ will make the code after the loop unreachable. k-induction tries to solve this problem, by unrolling the loop $2k$ times, and adding a special state transition after the first k iterations. This transition leads to a set of states for which nothing is known about the part of the state modified in the loop. Execution is assumed to continue as usual afterwards. Obviously, this approach introduces spurious counterexamples. In an attempt to reduce this problem, Rocha et al. [Roc+15] combined k-induction with invariants for checking properties in bounded and unbounded loops.

Concurrency is another hot topic in software bounded model checking. Work on this started as early as 2005 by Rabinovitz and Grumberg [RG05]. Later on Morse et al. [Mor+11] introduced context-bounded model checking. The method limits the number of contexts (and thereby context switches) executed. The idea being once again to make use of the small scope hypothesis for finding concurrency bugs quickly. More recent work includes CSeq, published by Fischer et al. [FIP13], which also interleaves contexts, and lazy CSeq [Inv+14], which was hugely successful in the concurrency track of the International Software Verification Competition 2014.

Further related work includes an approach to modularization of software bounded model checking presented by Hashimoto and Nakajima [HN09], as well as Frankenbit, presented by Gurfinkel and Belov [GB14], which is a combination of LLBMC-based software bounded model checking and UFO (see [Alb+12]), a framework for abstraction- and interpolation-based software verification.

Chapter 3

LLBMC: An Efficient Implementation of SBMC

This chapter provides an overview of LLBMC in section 3.1, gives an introduction into LLVM's intermediate representation in section 3.2, shows how properties on the C language level are translated to LLVM-IR in section 3.3. Furthermore, the chapter shows how compiler optimization passes can be used for the benefit of software bounded model checking in section 3.4, and it gives an introduction into control flow graphs (see section 3.5) and call graphs section 3.6. Finally the chapter introduces LLBMC's intermediate logic representation in section 3.7.

3.1 An Overview of LLBMC

LLBMC is a software bounded model checker designed for the verification of safety-critical embedded systems. Its primary focus lies on the verification of runtime properties, such as undefined behavior in C, though it also allows verification of functional properties to some degree.

LLBMC was developed at the Karlsruhe Institute of Technology, more precisely at the research group "Verification Meets Algorithm Engineering" at the Institute for Theoretical Computer Science. Development on LLBMC started mid-August 2009 under supervision of Dr. Carsten Sinz. Initial work on LLBMC was supported in part by the "Concept for the Future" of the Karlsruhe Institute of Technology within the framework of the first German Excellence Initiative.

Development on LLBMC was inspired by the software bounded model checker CBMC. CBMC was invented by Clarke et al. [CKL04] at Carnegie Mellon University. The tool was primarily developed by Daniel Kröning, who is supervising ongoing development of CBMC at the University of Oxford. A more in-depth description of CMBC is provided in section 2.2.5. In short, CBMC relies heavily on source-level transformations via `goto-cc`, a preprocessor which compiles C/C++ programs into a subset of C/C++ and makes heavy use of `goto` statements instead of structured control flow. The resulting program is then converted into a bitvector formula, which in turn

is bit-blasted to a propositional logic formula and converted to conjunctive normal form. This formula is then solved by an off-the-shelf SAT solver.

Since publication of [CKL04], the first work on CBMC in 2004, two new frameworks emerged: the SMT-LIB standard, which accelerated adoption of SMT in the research community, and the compiler framework LLVM by Lattner and Adve [LA04] which turned from a research project into a versatile and widely adopted technology.

LLVM is an open source compiler framework supporting ahead-of-time compilation as well as just-in-time compilation for a wide range of languages and architectures [LA04]. In addition it provides a clean architecture throughout and good standards support, while being known for its above-average documentation. The LLVM community is also pronouncedly friendly towards research projects, with more than 240 papers published between 2002 and 2015 based on the framework[1]. This also resulted in a large number of static analyses related research projects based on LLVM-IR [CDE08; CKC11; VK12; LGR11; McM10; RF12; RE11; Alb+12; GB14]. At the same time LLVM is increasingly used in various open source projects while also being embraced by various IT companies, including IBM and Apple. Especially Apple invested in the framework early on and these days, LLVM is the default compiler for Apple's OS X and iOS architectures.

Higher-level languages like C or C++ are often syntactically and semantically complex. Lower-level languages like LLVM's intermediate representation (LLVM-IR) are far simpler to parse and reason about. LLBMC makes use of LLVM's compiler front end and middle end to turn C and C++ code into LLVM-IR code. Thereby the complexity of LLBMC's own front end is greatly reduced.

The other new technology that emerged in the mean-time, SMT, was equally important to the development of LLBMC. The idea behind SMT is explained in section 2.1.2. Implementing efficient procedures for encoding programs as satisfiability problems, e.g. by bit-blasting, is time consuming, error prone, and often restricts a static analysis tool to a single encoding strategy. Using SMT Solvers instead allows using higher-level theories, such as the theory of bitvectors or the theory of arrays. This hides the concrete implementation behind a standardized abstraction layer.

Using LLVM and SMT in combination tremendously reduces the gap between higher-level languages like C and the propositional logic formula at the core of a software bounded model checker. These technologies thereby also greatly facilitate one of the core design goals of LLBMC: a clean and extensible software architecture.

3.1.1 A Multi-Layered Architecture

LLBMC consists of eleven internal *components*[2] (see table 3.1) and depends on a number of external components, e.g. from the LLVM compiler framework and multiple SMT solvers.

All internal and external components are arranged in four *architectural layers*, as can be seen in figure 3.1. Components in higher layers may only depend on components

[1] `http://llvm.org/pubs/`

[2] While the term modules is more fitting here, given the lack of a specified interface and substitutability, it was avoided due to the danger of confusion, considering the same term is already being used in LLVM for a different concept.

Call Graph	Call graph representation and construction.
Configuration	Translation of command line options to configuration settings of individual components.
Encode	Encoding of LLVM-IR in a call graph and an ILR formula.
ILR	Language definition of the Intermediate Logic Representation.
Optimize	Compiler optimizations implemented as a set of LLVM optimization passes.
Output	User friendly printing of LLBMC's results.
Simplify	Term rewriting based formula simplification for ILR formulæ.
SMT	Unified interface for driving multiple SMT solvers.
Solve	Classes related to solving ILR formulæ.
Tools	Higher-level algorithms, e.g. the model checking algorithm itself.
Utilities	Utility, support and helper libraries.

Table 3.1: LLBMC's internal components

directly below them, but may never depend on components in the same layer or above. The architectural layers are, from bottom to top, the utility and dependency layer, the data structure layer, the algorithmic layer, and the policy and input/output layer.

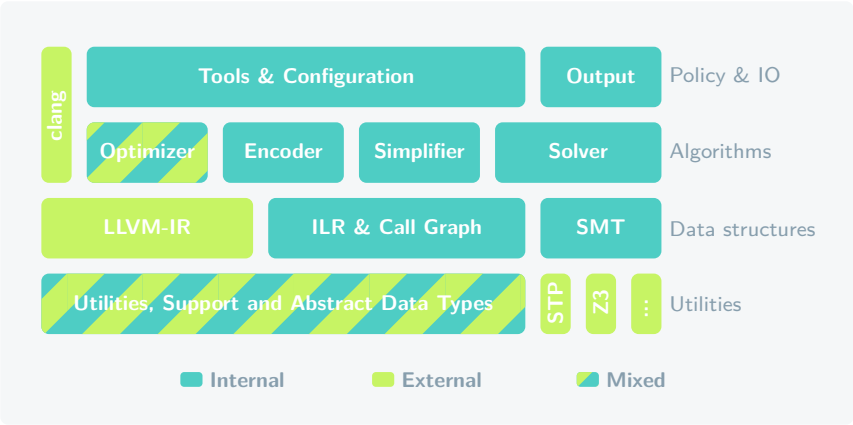


Figure 3.1: LLBMC's architectural layers

The utility and dependency layer contains external dependencies, e.g. SMT solvers such as STP, Boolector, Z3, CVC4, but also LLVM's many utility classes and abstract data types. The utilities in this layer are mostly generic and therefore not specific to LLBMC's core purpose.

The data structure layer contains the core data structures. The most important components here are the classes implementing LLBMC's Intermediate Logic Representation and call graph related classes. This layer also contains LLVM's Intermediate Representation and an abstraction layer for uniform access to all supported SMT

solvers.

The algorithmic layer contains, as the name indicates, most of LLBMC's core algorithms. This includes various compiler optimization passes, a component for encoding LLVM-IR in ILR, a component for simplification of ILR by term rewriting, and a component for translating ILR to SMT as well as for using SMT solvers to solve ILR formulæ.

The policy and IO layer provides translation from command line options to the various configuration options of the algorithmic layers. It also contains LLBMC's higher-level application programming interface. Finally, this layer also translates the tool's output back into a user readable format.

LLBMC also consists of five *language layers* (see figure 3.2), which are orthogonal to its architectural layers. Each of these layers is built around one language, be it a programming language or logic one. The five layers correspond to the languages C/C++, LLVM-IR, ILR, SMT, and SAT. As important as the layers themselves are the transitions and translations between the layers and their languages.

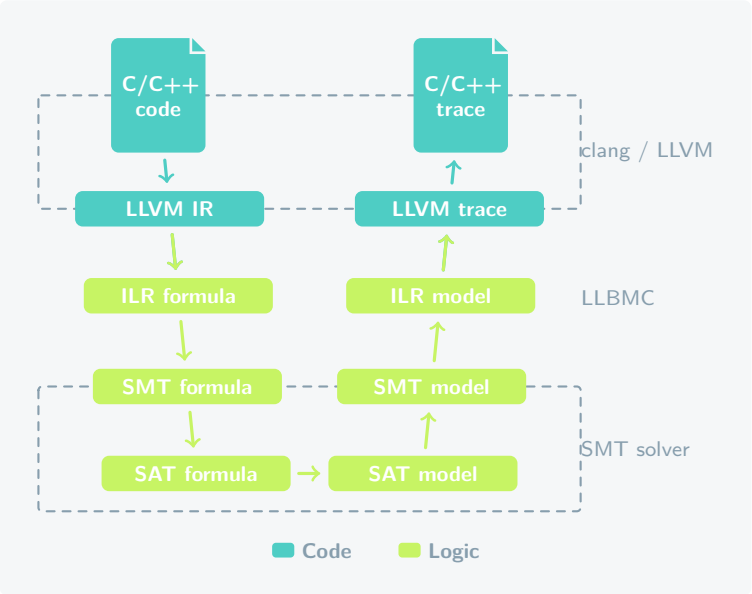


Figure 3.2: LLBMC's language layers

Figure 3.2 gives a visual overview of the different layers in LLBMC. The figure illustrates how LLBMC translates the input program and the properties to be checked step-by-step from C/C++ to SAT. It also shows how the resulting satisfiability problem is solved using a SAT solver and how the found model, if one exists, is translated back into a counterexample on the C/C++ level.

The C layer is mostly implemented in clang, LLVM's C language front end. LLBMC uses a modified version of clang, which has a plugin interface added to its code generator (see section 3.3). Because of this, LLBMC itself does not require a separate code generator, but relies on a number of such plugins for its code generation instead.

LLBMC's LLVM-IR layer contains LLVM's implementation of its intermediate rep-

resentation and LLVM's built-in optimization passes. Furthermore, LLBMC adds its own optimization passes and adapts some of LLVM's passes for use in LLBMC (see section 3.4). The LLVM-IR layer is connected to the ILR layer below it, by the Encoder component, which encodes LLVM-IR formulæ in ILR (see chapter 4).

The ILR layer contains the classes implementing ILR itself (see section 3.7), as well as the term rewriting system for simplification (see section 6.1). Additionally, this layer contains control flow and call graph related classes (see sections 3.5 and 3.6), though these classes are mostly closely tied to the Encoder component. An ILR solver based on SMT, including translation from the ILR layer to the SMT layer, is contained in the Solver component.

The SMT layer is mostly internal to the SMT solvers, as is the SAT layer below it. LLBMC only contributes a thin wrapper around the various supported SMT solvers in the Solver component.

Not all components in LLBMC are associated with one of these language layers. The Tools component provides higher-level control and configuration of the algorithms and does so for most language layers. Similarly classes from the Utility component may be used in any language layer.

3.1.2 Model Checking Algorithm

LLBMC's higher-level model checking algorithm is strictly sequential with five stages executed one after the other. Because of this, this algorithm is rather simple, as can be seen in listing 3.1. The algorithm takes an IR module p containing the system under verification as its first argument. The second argument, the entry point e, is a function in p that serves as an entry point into the system. The final two arguments, b_l and b_c are integers indicating maximum number of loop iterations as well as the maximum function call depth.

```
1  function MODELCHECK(p, e, b_l, b_c)
2      p ← OPTIMIZE(p)
3      p ← UNROLL(p, b_l)
4      g ← CALLGRAPH(p, e, b_c)
5      φ ← ENCODE(p, g)
6      φ ← SIMPLIFY(φ)
7      r, m ← SOLVE(φ)
8      if r then
9          c ← COUNTEREXAMPLE(p, φ, m)
10         return c
11     else
12         return null
13     end if
14 end function
```

Listing 3.1: Core model checking algorithm

The function OPTIMIZE takes a program p and runs the LLVM optimization passes presented in section 3.4 on all functions in p. While some passes are supposed to

increase performance of LLBMC, other passes are used to extend language support by replacing unsupported instructions by equivalent supported instructions, e.g. a switch instruction by a series of basic blocks and branch instructions.

The function UNROLL takes a module p and a loop bound b_l and unrolls all loops in p at most b_l times. If LLVM's scalar evolution analysis finds a sufficiently large loop bound smaller than b_l, then that loop bound is used instead. The function uses a modified version of LLVM's loop unroll optimization pass. The passes' adaptations make it possible to unroll a wider variety of loops that confuse the regular loop unrolling pass. This is necessary because LLVM's loop unroll pass misses unrolling loops with certain unusual basic block structures.

The function CALLGRAPH takes a program p, an entry point e, and a call depth bound b_c and creates a call-site-sensitive call graph (see section 3.6) for p, rooted at e with depth b_c.

The function ENCODE takes a program p and a call graph g and creates an ILR formula φ encoding the program and all properties to be checked into the formula. Because loops were already bounded in UNROLL and function call depth is bound during call graph generation, encoding is guaranteed to terminate and produce a finite formula. Encoding is presented in depth in chapter 4.

The function SIMPLIFY takes an ILR formula φ and returns the same formula simplified using the term rewriting system presented in section 6.1. Due to LLBMC's high context sensitivity introduced during the call graph generation, these simplifications are particularly effective in improving performance of software bounded model checking.

In the next step, the function SOLVE takes an ILR formula φ and returns a pair (r, m) where r is true if and only if φ is satisfiable and m is a model for φ, if r is true and undefined otherwise. This part of LLBMC is primarily described in section 6.2.

Finally, a counterexample is generated from the model m for the program p. It is notable, that the counterexample is based on the optimized and unrolled program and not on the original input program. however this poses no problem to the user, as the counterexample is usually not shown directly. Instead it is mapped back to the source language using the compiler's debug information. While information can get lost during optimization experience shows that the counterexample remains readable.

3.2 LLVM and Its Intermediate Representation

LLVM, like most compilers and compiler frameworks, can be divided into three parts: front end, back end, and middle end (see figure 3.3). The front end allows the compiler to support a large number of programming languages, including C, C++, Objective-C, Swift, Haskell, Python and GLSL. The compiler's back end enables the compiler to support multiple target architectures, e.g. x86, ARM and IA64. The middle end provides a framework for language and architecture independent code optimizations and thereby ties front end and back end together.

LLVM's middle end is built around its intermediate representation (LLVM-IR). This

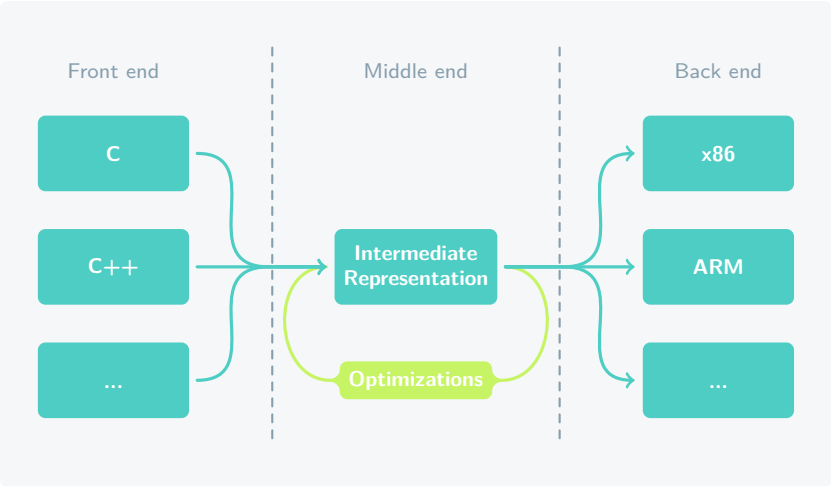


Figure 3.3: Common compiler architecture

intermediate representation is a low-level, typed, machine-independent assembler language designed primarily for use in compiler optimizations. An extensive description of LLVM-IR is available in the official language reference manual[3]. LLVM-IR is also LLBMC's primary input language. Because of this, a short and informal introduction into the language is provided in this section.

Note, though, that this thesis describes a simplified dialect of LLVM-IR, because full LLVM-IR has many features that are not relevant for this dissertation, e.g. garbage collection, exception handling, vector operations, address spaces, function calling conventions, function attributes, packed structures, and values of aggregate types. Any instructions or variants of instruction related to these concepts are not handled in this thesis. Whenever the term LLVM-IR is used in this thesis, this simplified variant is meant, unless explicitly stated differently. However, LLBMC's actual support of LLVM-IR, while it does not cover all of real LLVM-IR, goes considerably beyond this simplified variant, e.g. by covering an extensive number of additional instruction and value types. With this extensive support, LLBMC can process nearly any C or C++ that does not use exceptions or multi-threading.

A first look at an exemplary LLVM-IR program can be found in listing 3.2 on the following page and the corresponding C program in listing 3.3. As can be seen in this example, an LLVM-IR program (also called module) consists of a set of functions, each prefixed by the keyword `define`, followed by the function header and completed with the function body enclosed in curly braces (`{` and `}`). Each function consists of a sequence of basic blocks, each prefixed by its name, with each basic block in turn consisting of a sequence of instructions, each on a separate line.

Functions, basic blocks, instructions, and most other objects in LLVM-IR are derived from the same basic concept, which we will refer to as a *value object* (or simply *value*). Each separate class of objects, e.g. instruction or basic block, will be referred to as *value class*.

[3] `http://llvm.org/docs/LangRef.html`

```
define i32 @fib(i32 %n) {
entry:
  switch i32 %n, label %if.else3 [
    i32 0, label %return
    i32 1, label %if.then2
  ]

if.then2:
  br label %return

if.else3:
  %sub = add nsw i32 %n, -1
  %call = call i32 @fib(i32 %sub)
  %sub4 = add nsw i32 %n, -2
  %call5 = call i32 @fib(i32 %sub4)
  %add = add nsw i32 %call5, %call
  br label %return

return:
  %0 = phi i32 [ %add, %if.else3 ], [ 1, %if.then2 ], [ 0, %entry ]
  ret i32 %0
}

define i32 @main(i32 %argc, i8** %argv) {
entry:
  %call = call i32 (...)* @__llbmc_nondef_int()
  %call1 = call i32 @fib(i32 %call)
  ret i32 %call1
}

declare i32 @__llbmc_nondef_int(...)
```

Listing 3.2: The Fibonacci function in LLVM-IR

```
int __llbmc_nondef_int();

int fib(int n)
{
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}

int main(int argc, char **argv)
{
  int x = __llbmc_nondef_int();
  return fib(x);
}
```

Listing 3.3: The Fibonacci function in C

3.2.1 Values and Types

LLVM-IR a typed language, meaning all values have a *type*. he simplest type in LLVM-IR is the void type (`void`), which is used wherever there is a type required syntactically, but semantically none is needed. This is used for example as the return type of functions that do not return a value. The language also has integer types of any bitwidth. Names for these types follow the pattern `iN`, where `N` is the type's bitwidth (e.g. `i32` for a 32-bit integer). Pointer types adhere to the pattern `T*`, where `T` stands for the pointed-to type (e.g. `i32*` indicates a pointer to a 32-bit integer). LLVM-IR also has two kinds of aggregate types, array types and struct types. Array types are of the pattern `[N x T]`, where <N> indicates the number of elements and <T> indicates the element type (e.g. `[40 x i16]` for an array containing 40 16-bit integers). Structure types are indicated by curly braces (`{i1, i32}`, for a pair of a 1-bit integer and a 32-bit integer). Last but not least, function types are indicated by parentheses around the parameter types (`i16 (i32)` for a function taking a 32-bit integer and returning a 16-bit integer). Table 3.2 shows an overview of the types mentioned above. Note thate floating point types, are not part of this subset of LLVM, though they basic support for these types is implemented in LLBMC.

Integers	`iN`, where `N` is a positive number.
Integer types, e.g. `i1`, `i8`, `i32`, etc.	
Pointers	`T*`, where `T` is a type.
E.g. `i32*` for a pointer to a 32-bit integer.	
Structures	`{X, ...}`, where `X` is a type.
Structures, e.g. `{i32, i32}` for a pair of 32-bit integers.	
Arrays	`[N x T]`, where `N` is a number and `T` is a type.
Arrays, e.g. `[4 x i32]` for an array of four 32-bit integer values.	
Functions	`R (X, ...)`, where `R` and `X` are types.
Function types, e.g. `void (i32)` for a function that takes a 32-bit integer and does not return a value.	
Void	`void`
Placeholder type used when a type is syntactically expected, but not semantically, e.g. for functions without return value.	

Table 3.2: LLVM-IR types

Furthermore, LLVM-IR's structure types can also be "identified". *Identified structure types* are defined at the top-level of an IR file and always named. They are assigned to a %-prefixed value: `%typename = type { <type list> }`. The code in listing 3.6 contains an example for an identified type in line 1.

A few particularities are notable about LLVM-IR's type system, for example, the language does not have a dedicated boolean type, but uses an integer type with a bitwidth of one (`i1`) instead. Furthermore, LLVM-IR does not have distinct signed and unsigned types. Instead of this, LLVM-IR provides signed and unsigned versions for its arithmetic instructions, whereever needed. Finally, LLVM-IR allows for integer

types of nearly any bitwidth, so `i33` is a legal, though unusual, type and does actually occur in practice.

There is a considerable number of other types in LLVM-IR which are not listed in this thesis, e.g. floating point types and vector types. Similarly, not all language features, instructions and instruction variants are covered here for reasons of brevity and clarity. This includes any instructions related to vector types, inheritance, garbage collection, floating point types, arrays of size zero, and virtual register that contain aggregate types, just to name the more prominent features.

Locally defined values and types in LLVM are prefixed with the sigil `%`. In contrast to this, globally defined values, e.g. global variables and functions, are prefixed with the sigil `@`.

LLVM-IR also has *integer constants*, which are simply used in-line, e.g. as in `%0 = add i32 3, 4`, which adds the constants 3 and 4 and stores the result in the virtual register `%0`.

Global variables are realized in LLVM-IR as *pointer constants*, e.g. the line `@x = global i32 0` in a module's scope defines `@x` to be a pointer to a global variable which is zero-initialized. The global variable is then used via `load` and `store` instructions, e.g. `%1 = load i32 @x` loads the current value of the global variable `@x` into the virtual register named `%1`.

3.2.2 Instructions

An *instruction* in LLVM-IR is an atomic unit of execution performing a single operation. An instruction has an *opcode* indicating what it does, e.g. `add` for integer arithmetic addition, and performs this operation on zero or more *operands*, each one of these being a value itself. If the operation has a *result*, then this result is assigned to a target *virtual register*. LLVM-IR provides an unlimited number of such virtual registers[4] and all virtual registers are named.[5] Certain instructions may also have one or more *instruction flags* set, which modify the instructions' semantics slightly.

Figure 3.4 shows an add instruction as an example with all its components labeled. In this example, the `nsw` indicates that no signed wrap-around is expected to occur.

The term instruction is often used to refer to an instruction's target register as well as to the operation itself, which is possible because an instruction and its target register are inseparable. This is because of, like most compiler intermediate representations too, LLVM-IR is in *static single assignment form* (*SSA*) [RWZ88]. This means that every virtual register is only ever assigned once and it is guaranteed to be defined before it is used. Code that is not in SSA form can be transformed into this form by versioning all variables, as can be seen by the example in listing 3.4. For this, every time any variable is assigned a new value, a new variable is introduced instead and all subsequent references to the old variable are immediately replaced by references to this new variable. These steps are repeated until no variable is assigned more than once.

[4] Once the program is compiled to architecture specific code these virtual registers will be allocated to a limited number of real registers.

[5] If an instruction is not explicitly named, it is automatically assigned a unique number as a name.

Figure 3.4: Exemplary add instruction and its components

```
1 int g;
2
3 int f(int a) {
4   int x;
5   x = a;
6   g = x;
7   x = 1;
8   return x;
9 }
```

(a) Before SSA transformation

```
1 int g;
2
3 int f(int a) {
4   int x0, x1;
5   x0 = a;
6   g  = x0;
7   x1 = 1;
8   return x1;
9 }
```

(b) After SSA transformation

Listing 3.4: Transformation to static single assignment form

All LLVM-IR instructions relevant for this thesis are introduced in tables 3.3 to 3.7 and 3.9. The instructions are introduced using so called instruction patterns.

Definition 3.1 (Instruction pattern). *An instruction pattern is a string describing LLVM-IR instructions. In an instruction pattern,*

- *square brackets ([and]) enclose optional elements,*
- *angle brackets (< and >) enclose placeholders for references to values or types,*
- *vertical bars (|) separate alternatives,*
- *curly braces followed by a star ({ and }*) indicate sequences of zero or more elements,*
- *and curly braces followed by a plus ({ and }+) indicate sequences of one or more elements.*

If a list contains more than one value, elements are separated by a comma, which is not shown here explicitly.

For easier readability, LLVM-IR code itself is set in the color ●, LLVM-IR's opcodes are highlighted in ●, and characters belonging to the pattern language are set in ●.

For instance, the pattern

```
<r> = add|sub [nuw] [nsw] <ty> <op1>, <op2>
```

represents all integer addition and subtraction instructions, with and without the

flags `nsw` and `nuw`. Similarly, the pattern

```
<res> = call <ty> <fptr>({<ty> <arg>}*)
```

represents all call instructions with any number of arguments. Instruction patterns are not only used to present LLVM's instruction set in this section, but also to reason about these instructions later on. For this, pattern matching, which is inspired by regular expressions, is used.

Definition 3.2 ($\cdot \sim [\![\cdot]\!]$). *The predicate $s \sim [\![p]\!]$ is true, if and only if the instruction s matches the instruction pattern p.*

For example if s is the instruction shown in figure 3.4 on page 39, then

$$s \sim [\![\texttt{<r> = add [nuw] nsw <t> <op1>, <op2>}]\!]$$

is true.

3.2.3 Undefined Values and Undefined Behavior

LLVM-IR provides a special `undef` value, which can be used anywhere a constant is expected and which is used to indicate that a value is undefined. Compiler optimizations may assume this to be any value suitable in order to generate the best performing code. For example, any reference to the instruction `%0 = add i32 %x, undef` by another instruction may replaced by `undef`, which may then again open up new optimization opportunities.

A related, though markedly different subject is that of *undefined behavior*. Undefined behavior[6] is identical to C's notion of undefined behavior: If undefined behavior occurs, anything may happen. Like undefined values, undefined behavior opens up optimization opportunities for the compiler.

A concept related to undefined behavior, but more specific to LLVM-IR are *poison values*. "Poisonousness" can roughly be seen as an additional bit of information attached to all values. It is caused, for example, by certain cases of arithmetic overflow, and it is propagated to all users of a poisonous value, e.g. from instruction to instruction, but also to and from individual memory locations. If a poisonous instruction has an observable behavior it is undefined behavior. Undefined behavior in LLVM-IR often occurs indirectly via poison values. Even though poison values do solve some problems, they seem to introduce new ones as well[7], and as of the time of writing discussions in the LLVM community are ongoing about these concepts. Because of the ill-defined behavior of poison values, in LLBMC, the generation of poison values is already treated as undefined behavior.

3.2.4 LLVM's Instruction Set

LLVM-IR provides the usual set of *arithmetic instructions* for addition, multiplication, division, remainder calculation for signed and unsigned integers as well as floating

[6] In contrast to the C standard, LLVM's language reference manual does indeed not define undefined behavior.

[7] See also Dan Grohman's post to the LLVM mailing list on the whys and hows of poison values: "The nsw story" (http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-November/045730.html)

point numbers (see table 3.3). Instead of signed and unsigned integer types, LLVM-IR has signedness-neutral types and signed and unsigned operations. For example, `sdiv` and `udiv` are separate instruction for signed and unsigned division respectively. All arithmetic instructions have the same type for their result as for their operands.

`<res> = add|sub|mul [nuw] [nsw] <ty> <op1>, <op2>`
Addition, subtraction and multiplication <op1> and <op2>, both of type <ty>.
`<res> = udiv|sdiv [exact] <ty> <op1>, <op2>`
Unsigned and signed division of <op1> by <op2>, both of type <ty>.
`<res> = urem|srem <ty> <op1>, <op2>`
Unsigned and signed remainder of the division of <op1> by <op2>, both of type <ty>.

Table 3.3: Arithmetic instructions

The flags `nuw` and `nsw` indicate that no unsigned respectively signed overflow is expected to happen. If this happens anyways, the result is a poison value. Similarly, the keyword `exact` for division operations indicates, that the division's remainder is expected to be zero, and the result is a poison value if not.

LLVM-IR also has three *shift instructions*, including shift left and arithmetic and logical shift right. Shift instructions have similar flags to arithmetic instructions. These flags are mostly used when shift instructions are used as cheaper alternatives for arithmetic instructions that have one or more of these flags set. Note that while the flags `nsw`, `nuw`, and `exact` only cause poison values, shifts can also produce undefined behavior, if the value of the second operand is larger than the first operands bitwidth. Furthermore, LLVM-IR has *bitwise instructions* for bitwise conjunction, disjunction, and exclusive disjunction (see table 3.4). Logical negation is emulated using an `xor` instruction whose second argument has all bits set to 1. Note, that logical equality is already covered by the `icmp` instruction shown in table 3.8.

`<res> = shl [nuw] [nsw] <ty> <op1>, <op2>`
Left shift of <op1> by<op2>, both of type <ty>.
`<res> = lshr|ashr [exact] <ty> <op1>, <op2>`
Logical and arithmetic shift of <op1> by <op2>, both of type <ty>.
`<res> = and|or|xor <ty> <op1>, <op2>`
Bitwise logical and, or, and exclusive or of <op1> and <op2>, both of type <ty>.

Table 3.4: Bitwise instructions

Because LLVM-IR is a strongly typed language, it needs to provide a large number of type *conversion instructions* (see table 3.5). The instruction `trunc` truncates an integer to an integer with a shorter bitwidth. Information in the most significant bits is lost. The instruction `zext` extends an integer to a larger bitwidth while keeping its unsigned value. The instruction `sext` does the same, but retains the argument's signed value. The `ptrtoint` and `intotoptr` are used for conversion

between pointers and integers. Both instructions implicitly perform zero extensions or truncations as required. Pointer to pointer conversion can be done with `bitcast`, though both pointer types must have the same bitwidth.

`<res> = trunc <ty> <op> to <ty2>`
Truncation of the integer `<op>` of type `<ty>` to the bitwidth of `<ty2>`.
`<res> = zext|sext <ty> <op> to <ty2>`
Zero extension and signed extension of the integer `<op>` of type `<ty>` to the bitwidth of `<ty2>`.
`<res> = ptrtoint|inttoptr <ty> <op> to <ty2>`
Conversion of the pointer or integer and integer `<op>` of type `<ty>` to an integer or pointer of type `<ty2>`.
`<res> = bitcast <ty> <op> to <ty2>`
Conversion of the pointer `<op>` of type `<ty>` to pointer of type `<ty2>`.

Table 3.5: Conversion instructions

All *memory-related instructions* are listed in table 3.6. All stack memory is allocated in LLVM-IR via the `alloca` instruction and deallocated automatically. Memory access is done using `load` and `store`. Both of these instructions may have a `volatile` flag, which indicates that the value stored at this memory location might change anytime, e.g. through a different thread, process or even the hardware itself. The `volatile` flag disallows certain compiler optimizations, e.g. joining of otherwise redundant `load` operations or reordering of `store` operations.

`<res> = alloca <ty> [, <t> <num>]`
Allocates `<num>` elements of type `<ty>` on the current stack frame, or a single element, if `<num>` is not given. `<t>` is `<num>`'s type.
`<res> = load [volatile] <ty>, <ty>* <ptr>`
Loads the value stored at address `<ptr>` interpreting the bit pattern as `<ty>`.
`store [volatile] <ty> <op>, <ty>* <ptr>`
Stores `<op>` at the address `<ptr>` in memory.
`<res> = getelementptr <ty>, <ty>* <ptrval>[, {<ity> <idx>}+]`
Type safe pointer arithmetic for sub-element access in arrays and structs.

Table 3.6: Memory related instructions

The `getelementptr` instruction is used for type-safe, architecture independent, and optimizable address calculation. It takes a base pointer argument to a structure or array and any number of arguments for indexing into this aggregate type. The first index indexes on the pointer value itself. Subsequent indices are used to index into sub-elements of the aggregate type.

The first index argument often causes considerable confusion.[8] This is because it is

[8] See `http://llvm.org/docs/GetElementPtr.html` for an explanation of `getelementptr`.

zero in the most common cases, e.g. for global or stack local variable that are not arrays, which is highly confusing when encountering this instruction for the first time. The reason for the confusion is most likely because LLVM-IR and C use pointers differently. In C, given a pointer `p`, `*p` dereferences the pointer. The expression `*p` is equivalent to `p[0]`, as `p[i]` is merely syntactic sugar for `*(p + i)`. LLVM-IR's `getelementptr` simply does not have this syntactic sugar, so it always requires an explicit `0`.

Another consequence of `getelementptr`'s definition is that without an index argument it is a NOOP, a `getelementptr` with a single index argument is simple pointer arithmetics, and only a `getelementptr` with more than one index argument does what the name says, namely retrieving a pointer to an element of an object of aggregate type.

```
1  struct RT {
2    char A;
3    int B[10][20];
4    char C;
5  };
6  struct ST {
7    int X;
8    double Y;
9    struct RT Z;
10 };
11
12 int *foo(struct ST *s) {
13   return &s[1].Z.B[5][13];
14 }
```

Listing 3.5: Example C input for `getelementptr`

The C code in listing 3.5 is taken from LLVM's language reference manual on `getelementptr`. Line 13 of the example nicely illustrates various kinds of address calculations nested into each other such as one dimensional and two dimensional array indexing and indexing into structures. Listing 3.6 shows the corresponding IR code, which combines all of the example's address calculations into a single `getelementptr` instruction.

```
1 %RT = type { i8, [10 x [20 x i32]], i8 }
2 %ST = type { i32, double, %RT }
3
4 define i32* @foo(%ST* %s) {
5 entry:
6   %i = getelementptr %ST, %ST* %s, i64 1, i32 2, i32 1, i64 5, i64 13
7   ret i32* %i
8 }
```

Listing 3.6: Example IR for `getelementptr`

LLVM's `select` is similar to C's `?:`-operator. If the first operand is true, the instruction returns the second operand's value, otherwise it returns the third operand's value. The first operand must be of type `i1`, the second and third operands as well as the instruction itself may be of any integer type, but must all be of the same

type.

The `phi` instruction makes SSA form work for basic blocks with multiple predecessors. A phi instruction has a list of pairs of basic blocks and instructions as operands. Depending on which basic block was executed immediately before the current basic block, the phi instruction returns the value of the instruction that is paired with this predecessor basic block (see line 20 in listing 3.2 on page 36 for an example of its use). A `phi` instruction is usually not translated to machine code directly. Instead, the compiler tries to allocate all virtual registers which are operands of the same phi instruction in the same hardware register or, if that is not possible, stores those values at the same memory location.

The `call` instruction is used to pass control flow to another function. A `call`'s first operand, before the opening parenthesis, is a pointer to a function, the remaining operands are arguments passed to the function. The instruction comes in two variants, one of type `void` which does not return a value, and one for any other type but `void`.

`<res> = call <ty> <fptr>({<ty> <arg>}*)`
Call of the function at address `<fptr>` with the list of arguments `<arg>` returning a value of type `<ty>`.
`call void <fptr>({<ty> <arg>}*)`
Call of the function at address `<fptr>` with the list of arguments `<arg>` return no value.
`<res> = icmp <cond> <ty> <op1>, <op2>`
Compare the integers `<op1>` and `<op2>` with `<cond>` interpreted as in table 3.8.
`<res> = phi <ty> {[<val>, <label>]}+`
Return `<val>`, if the basic block labeled `<label>` was executed immediately before the current basic block.
`<res> = select i1 <cond>, <ty> <val1>, <ty> <val2>`
Return `<val1>` if `<cond>`, `<val2>` otherwise.

Table 3.7: Miscellaneous instructions

Table 3.8 shows the different variations of the `icmp` instruction, where $\cdot_u$ indicates interpretation of the operation's operands as unsigned integers and $\cdot_s$ indicates interpretation of the operands as signed integers.

The set of instructions in LLVM-IR itself is fixed. Nonetheless, LLVM-IR is extensible due to its so called intrinsic functions. *Intrinsic functions* are functions that have well known names (starting with `llvm.`) and semantics. Most intrinsics are subject to additional constraints, making their use transparent to LLVM's optimization passes. All intrinsics along with their constraints are extensively documented in the language reference manual.

`<res> = icmp eq <ty> <op1>, <op2>`	yields true iff <op1> $=$ <op2>.
`<res> = icmp ne <ty> <op1>, <op2>`	yields true iff <op1> $\neq$ <op2>.
`<res> = icmp ugt <ty> <op1>, <op2>`	yields true iff <op1> $>_u$ <op2>.
`<res> = icmp uge <ty> <op1>, <op2>`	yields true iff <op1> $\geq_u$ <op2>.
`<res> = icmp ult <ty> <op1>, <op2>`	yields true iff <op1> $<_u$ <op2>.
`<res> = icmp ule <ty> <op1>, <op2>`	yields true iff <op1> $\leq_s$ <op2>.
`<res> = icmp sgt <ty> <op1>, <op2>`	yields true iff <op1> $>_s$ <op2>.
`<res> = icmp sge <ty> <op1>, <op2>`	yields true iff <op1> $\geq_s$ <op2>.
`<res> = icmp slt <ty> <op1>, <op2>`	yields true iff <op1> $<_s$ <op2>.
`<res> = icmp sle <ty> <op1>, <op2>`	yields true iff <op1> $\leq_s$ <op2>.

Table 3.8: Variations of the `icmp` instruction

3.2.5 Basic Blocks and Terminators

The instructions in an LLVM-IR program are grouped in so called *basic blocks*. LLVM itself does not provide a definition of a basic block, but a sufficiently suitable definition can be found in [All70], though:

Definition 3.3 (Basic block). *A basic block is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed). It may of course have many predecessors and many successors and may even be its own successor.*

Every basic block in LLVM-IR has a *label* that uniquely identifies the basic block in the containing function. In the following, by convention, the term b refers to basic block with the name B, b_1, to the basic block labeled $B1$, etc.

All instructions in a basic block are executed in order, starting with the first instruction and ending with the last instruction. If an instruction in a basic block is executed, the subsequent instruction will be executed next (except for `call` instructions), unless the former instruction terminates the program as a whole, e.g. by calling C's `exit()` directly or indirectly. This property greatly simplifies various code analyses used in compilers.

There are some restrictions on how instructions can be used in basic blocks, with the most notable exception concerning the instructions `ret`, `br`, `switch`, `indirectbr`, `invoke`, `resume`, and `unreachable` (see table 3.9). These instructions are called *terminators*. In a well-formed LLVM-IR program, only one of these instructions may be the last instruction of a basic block, and none of these instructions may occur anywhere else. Terminators play an important role in defining the control flow between basic blocks.

Definition 3.4 ($\mathrm{term}_{\mathtt{B}}$). *The function* $\mathrm{term}_{\mathtt{B}} : \mathtt{B} \to \mathtt{I}$ *returns a basic block's last instruction. This instruction is called the terminator.*

The `ret` instruction returns control to the the calling function, `br` is for direct conditional and unconditional branches, and `switch` acts similar to C's switch-case

`ret void`
A `ret` instruction with type `void` and no argument returns control to the calling function without returning a value.
`ret <ty> <op>`
A `ret` instruction with a non-void type returns control to the calling function and also returns the value `<op>`.
`br label <dst>`
A `br` instruction with only a single argument of type `label` unconditionally passes control flow to the basic block labeled `<dst>`.
`br i1 <cond>, label <then>, label <else>`
A `br` instruction with a condition argument of type `i1` passes control flow to the basic block labeled `<then>` if the condition `<cond>` is true and to the basic block labeled `<else>` otherwise.
`switch <ty> <op>, label <def> [{<ty> <val>, label <dst>}*]`
The `switch` instruction has the same purpose as the switch statement in C and C++. Apart from the control variable `op` and the default target `def`, `switch` has pairs of constants (`val`) and basic block labels (`dst`). For each `val`, if `op` equals `val`, the basic block paired with `val` executed.
`indirectbr <ty>* <address>, [{label <dst>}+]`
The `indirectbr` instruction branches to the basic block stored at the dynamically calculated `<address>`. The address must be one of labels listed as `<label>`, otherwise the behavior is undefined.
`unreachable`
The `unreachable` instruction indicates, that the end of the basic block is not expected to be reached. Behavior is undefined if this happens nonetheless and it is the compiler's responsibility to ensure this does not happen.

Table 3.9: Terminator instructions

statement. The instruction `indirectbr` allows for the branch target to be dynamically calculated, but it must be one of the basic blocks listed in the set of possible branch targets instruction. This set is required for LLVM's control flow analyses to work correctly. The instruction `unreachable` is not expected to be executed. It is undefined behavior, and likely a compiler bug, if this happens nonetheless.

A basic block can be used as an operand in `br` and `switch` instructions. In this case, the value uniquely identifies the basic block in the function. A basic block's address can be retrieved with the `blockaddress` constant, which returns an `i8` pointer. This pointer can then be used in a `indirectbr` instruction.

Reasoning about instructions in basic blocks requires two auxiliary functions:

Definition 3.5 ($\mathrm{first}_{\mathtt{B}}$). *The function $\mathrm{first}_{\mathtt{B}} : \mathtt{B} \to \mathtt{I}$ returns a basic block's first instruction.*

Note that every well-formed basic block contains at least one instruction, making this function total. Similarly important is the following definition:

Definition 3.6 ($\mathrm{succ}_{\mathtt{B}}$). *The relation $\mathrm{succ}_{\mathtt{B}} : \mathtt{I} \times \mathtt{I}$ is true if and only if both instructions are in the same basic block and its second argument is a direct successor to its first argument.*

3.2.6 Modules and Functions

Real LLVM-IR *functions* have a wide range of function attributes, which are necessary to faithfully model a source function's properties from the many supported source languages and to provide hints for the compiler optimizations. For this thesis' purposes, function attributes are neglected and a simpler definition of a function suffices:

Definition 3.7 (Function). *A function (n, B, e) is a tuple of a name n, a sequence of basic blocks $B = (b_0, b_1, \ldots, b_m)$, and an entry block $e \in B$.*

Every LLVM-IR function f has a so called *entry basic block*, the only block in f that is externally reachable.

Definition 3.8 (entry). *Given a function f, $r_f = \mathrm{entry}(f)$ is f's entry block, the first element of f's sequence of blocks and the only one in f that is externally reachable.*

An *exit basic block* is a basic block that returns control flow to the calling function:

Definition 3.9 (exit). *The predicate exit on basic blocks is defined by*

$$\mathrm{exit}(b) \leftrightarrow \mathrm{term}_{\mathtt{B}}(b) \sim [\![\texttt{ret <ty> <op>}]\!] \vee \mathrm{term}_{\mathtt{B}}(b) \sim [\![\texttt{ret void}]\!].$$

LLVM-IR programs are structured in modules. Each module consists of a set of declarations of global variables and a set of function declarations and definitions[9]. Modules can be linked together into larger modules, e.g. multiple modules, each containing the code of one translation unit can be linked together to create a single

[9] As well as debug information, symbol table entries, and quite a few other things which are not relevant for this dissertation.

module containing the whole *program*. In the following, whole program analysis is performed, meaning that it is assumed that this linking process has already happened. Because of this, the terms module and program will be used interchangeably.

Definition 3.10 (Module). *A module $m = (F_m, G_m)$ is a pair of a set of function symbols F_m and a set of global variable symbols G_m.*

Note that just because LLVM-IR as a language is called architecture independent, this does not mean an LLVM-IR program is also architecture independent. It merely means LLVM-IR is sufficiently expressive to describe programs for many different architectures. Each program is still tied to one specific architecture.[10] Many of LLVM's optimizations, and LLBMC itself as well, require information about the target architecture. This includes, in particular, endianness of the architecture and alignment rules for various types. An LLVM-IR module encodes this information in so called *data layout* lines as a character string. A data layout line consists of multiple entries separated by a minus (`-`). Each entry encodes a fact about the target architecture, e.g. in the data layout line `target datalayout = "e-m:w-p:32:32-i64:64"`, `e` stands for little endianness and `E` for begin endianness, `m:w` indicates windows style mangling, while `p:32:32` encodes the fact that pointers have a bitwidth of 32-bit with an alignment of 32-bit, and `i64:64` indicates that the `i64`-type is 64-bit aligned.

3.3 Verification of Source Languages

LLVM-IR is a compiler intermediate representation and is therefore usually not the programming language in which the developer originally wrote the program under analysis. However, this source language can be translated to LLVM-IR easily, provided that the LLVM compiler framework supports the given source language. This section shortly discusses source languages of interest for embedded development with a focus on the C language, and shows how the translation to LLVM-IR must be adapted for sound LLVM-IR based source code analysis.

The most popular programming language in embedded system development is, with a comfortable lead, the C programming language[11]. C was invented in the early 70's by Dennis Ritchie at Bell Laboratories for the Unix operating system, and was, from the beginning, designed as a machine-oriented high-level language. C's success in the embedded world is mostly due to its well-suited mix of high and low-level language features, but equally important are its widespread availability and a plethora of well established and certified tools and tool chains. Finally, embedded software development projects are oftentimes tied closely to the chip for which the software is developed. Chip vendors often provide their own tool-chains, including custom-tailored compilers and debuggers with the chip, which often only support C.

Assembly languages are used in many embedded projects, though usually only for specific functions where high-level languages cannot be used and therefore nearly

[10] This is in contrast to languages like Java, which are tied to a virtual machine which makes it possible to run the same Java bytecode unmodified on any number of architectures. Though it could also be argued that Java actually runs on no real architecture at all.

[11] TIOBE Index for March 2014 lists C as the most popular language among developers in general. It is significantly more widespread in embedded systems development.

never as a project's primary language. In particular the combination of C and assembly occurs often.

While C is the most widespread high-level language in the embedded market, it is not the only such language in this area. In recent years, C++, C's object-oriented offspring, has seen increased use in the embedded context but has not yet reached C's popularity and the market share seems to be decreasing recently. Java, Basic, and C# are occasionally used in the embedded market, but none of these languages have a market share worth mentioning. Finally, the programming language Ada, while specifically designed for the Department of Defense's demanding requirements[12] concerning safety and security critical embedded code, is commonly considered the better language for embedded code, but has not nearly as many users as C.

Recently, new languages showed up on the embedded market, e.g. the Mozilla Foundation's Rust[13] programming language and Google's Go[14] language. These languages are explicitly targeted at the embedded market, avoiding many of the pitfalls and problems associated with the C language. Time will tell if these languages eventually manage to replace C.[15]

With newer and safer languages being available, one would expect that C is slowly replaced by these, but this does not seem to happen any time soon, as shown by a survey conducted by the publisher of Embedded Systems Programming.[16]

Because of C's continuing dominating role in the embedded software market, and because there is no indication of this changing in the near future, the verification of embedded software written in C are the primary concern of the remainder of this section.

3.3.1 Verifying Embedded C Code

C code has proven to be exceptionally hard to reason about formally, and there is even little consensus on desirable and undesirable properties, as already stated by Cuoq et al. [CKY12]. This is partly due to C's overly permissive cast operator, which is responsible for the fact that type information in C is inherently unreliable and can only cautiously be used for any kind of reasoning about the code. But this is also due to C's notion of undefined behavior that permeates the language's standards document.

The language was, from the beginning, designed for portability and efficiency. Efficiency, in this context, meaning that code can be written in C which runs comparably fast to hand-crafted assembly, and portability meaning that C code could be written that runs equally efficient on the wide range of computer architectures available at that time. How wide this range was becomes apparent when one considers the C standards definition of a byte as an

[12] http://www.dwheeler.com/steelman/steelman.htm
[13] https://rust-lang.org/
[14] https://golang.org/
[15] http://www.embedded.com/design/programming-languages-and-tools/4428704/Alternatives-to-C-C--for-system-programming-in-a-distributed-multicore-world
[16] http://www.embedded.com/electronics-blogs/programming-pointers/4372180/Unexpected-trends

> addressable unit of data storage large enough to hold any member of the basic character set of the execution environment. [ISOC99, section 3.6]

Note that the number of bits in a "byte" is intentionally not specified, as the number varied from computer architecture to architecture at the time the C language was standardized and was therefore not fixed to 8 bits, as it is now. Furthermore, while some computer architectures used two's complement for binary signed integer representation, others used — the now obsolete — ones' complement instead.

A language designed to be nearly as efficient as hand-crafted assembly on a wide range of architectures needs to be extremely flexible. For C this was made possible by either not defining corner cases where different architectures might behave differently, or by explicitly leaving the behavior to be defined by the implementation.[17]

In the embedded market this is made even worse due to the fact that compiler vendors often define and implement language extensions that go even beyond the freedom provided by the standard itself.

As already noted in chapter 1, undefined behavior is a cause of errors and therefore LLBMC focuses on proving the absence of undefined behavior or, if this is not possible, finding counter examples for cases of undefined behavior. At the same time, however, undefined behavior also makes the process of verification itself harder.

3.3.2 Undefined Behavior in C

Undefined Behavior and its close relatives implementation-defined and unspecified behavior[18] are deeply embedded in the C standard. The concept is mentioned throughout the standard and in addition, Annex J of the standard, which explicitly states that it does not claim to be complete, lists 54 cases of unspecified behavior, 191 cases of undefined behavior, and 112 cases of implementation-defined behavior.

Unspecified behavior is defined in the standard as:

> Use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance [ISOC99, section 3.4.4]

For a complete analysis of the code, all of those possibilities need to be analyzed.[19]Complexity increased furthermore by the fact the any implementation may opt for a different behavior each time. This causes an exponential blow-up in the size of the search space.

An often cited example for this is the ordering in which function arguments are evaluated:

[17] A C implementation can be seen as the union of a specific compiler and its target architecture.

[18] There is a fourth kind of behavior, *locale specific behavior*, though it is rarely relevant in the embedded contexts.

[19] For example, the order of the evaluation of function arguments is unspecified. Therefore, one needs to analyze all possible orderings. Because the ordering may be different for each function call, the complexity scales with the product of the number of function calls and the faculty of the number of arguments per function. Alternatively, one could prove that the program's behavior is independent of the ordering of the evaluation of a function's arguments, so that analyzing a single ordering is sufficient for this function.

> The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, [...] [ISOC99, section 6.5.2.2]

The evaluation of one such argument might depend on the evaluation of any other argument, simply because the evaluation of an argument might modify the state the program is in. Analyze all program executions for all orderings, while not making verification impossible, certainly makes it infeasible for any reasonably sized program.

Implementation-defined behavior is defined as

> unspecified behavior where each implementation documents how the choice is made [ISOC99, section 3.4.4]

One prominent case of implementation-defined behavior are the sizes of integer types. An implementation must decide on a size for each integer type and record this decision in the implementation's documentation.

Finally, *undefined behavior* poses an entirely different challenge for the verification of C code. It is defined in the standard as

> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements [ISOC99, section 3.4.3]

One of the most well known examples for this is C's handling of signed integer overflows:

> If an exceptional condition occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined. [ISOC99, section 6.5]

When the C standard was designed, this was primarily done to ensure that C can be implemented efficiently on two's complement architectures, ones' complement architectures, and even sign-and-magnitude architectures. Though all modern computer architectures use two's complement, this undefined behavior is still used by compilers to generate more optimized code. Consider for example the following code fragment:

```
1  for (i = 0; i <= N; ++i) {
2    // do something
3  }
```

If `i` is of type `unsigned int`, the compiler must take into account that `N` might be `UINT_MAX` and the loop might be an infinite loop. In contrast, if `i` is `int`, the implementation may assume that the loop is executed exactly N+1 times, which opens up optimization opportunities.

Most code analysis tools which approach the issue of implementation-defined behavior try to provide a relevant subset of real-life implementations. For example, the bitwidth of types is often configurable while two's complement is usually assumed. Many tools also follow a specific implementation on which they are often based. In the case of LLBMC this is LLVM, which is mostly compatible to GCC and

MSVC. Furthermore, because LLVM is open source, additional support for other implementations can easily be added if required.

To avoid the blow-up in size of the search space caused by unspecified-behavior, code analysis tools usually assume one specific behavior, e.g. evaluation of function arguments from left to right, even though implementations do not guarantee this behavior.

Proving or disproving undefined behavior is LLBMC's main goal, and at the same time a major challenge in building a code analyzer. This is in particular true, if the code analyzer operates on a compiler intermediate representation, like LLBMC does.

3.3.3 Translating C to LLVM-IR

One of LLBMC's primary purposes is the detection of undefined behavior in C programs. While *clang*, LLVM's C language front end, faithfully transforms C code to LLVM-IR, the same is not guaranteed for undefined behavior. Contrariwise, the compiler is explicitly allowed to make use of undefined behavior to improve performance of the generated code. In consequence, this means the compiler needs to be adapted to translate undefined behavior from C to LLVM-IR explicitly so that it becomes available in the LLVM-IR code for use by LLBMC. The following section shows how this is done in LLBMC, though first a short introduction to clang is necessary.

Figure 3.5 roughly illustrates how clang translates C code to LLVM-IR code. The process starts with the driver which processes the command line options and passes them on to the front end that controls the remaining compilation process. The front end first runs the lexer, which turns the character stream from the source file into a token stream. Note that clang does not have a separate preprocessor, but has preprocessing integrated into the lexer instead. Macro definitions are turned into token sequences and stored for later use. Macro expansions are then handled by injecting the previously generated token sequence into the token stream. This is the technical foundation for clang's exceptionally user-friendly compiler error messages. Next in line is the parser, which in turn drives clang's semantic analysis module. The result of this is the abstract synatx tree (AST). Finally, the code generator turns the AST into LLVM-IR. In normal operation, clang would now execute a user-configured set of optimizations and then generate binaries for the target architecture, but the option `-emit-llvm` in combination with `-c` tells clang to forgo optimizations and to output LLVM-IR bitcode instead of binaries.

The result of executing clang with these options on the C code in listing 3.7a can be seen in listing 3.7b. Note how undefined behavior is handled by clang in this example: The signed addition in line 2, which causes undefined behavior on overflow, translates to the `add` instruction in line 9 of the output. This instruction has the `nsw` flag set, and can therefore similarly cause undefined behavior in the case of an overflow.[20]

[20] The stack memory allocations (`alloca`), and loading and storing seems superfluous, but are removed later on during optimization. The instructions are generated by clang, because on the C level, the function parameters `x` and `y` are l-values. This means they do not only have a value, but also an address, which can be passed to functions, if required. In this particular example, this is not necessary, because the function does not make any use of its parameters' addresses. However, clang does not know this at this stage of code generation, so it generates the instruction just in case. Concerning performance, the compiler relies on the fact that these superfluous memory accesses

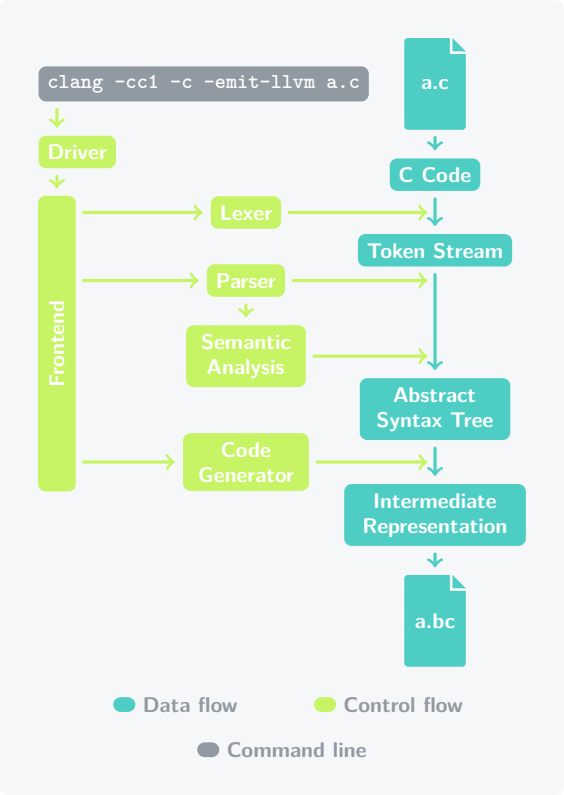


Figure 3.5: clang's architecture

```c
int add(int x, int y) {
  return x+y;
}
```

(a) Input C program

```
define i32 @add(i32 %x, i32 %y) {
entry:
  alloca i32 %x_addr
  alloca i32 %y_addr
  store i32 %x, i32* %x_addr
  store i32 %y, i32* %y_addr
  %0 = load i32, i32* %x_addr
  %1 = load i32, i32* %y_addr
  %2 = add i32 nsw %0, %1
  ret i32 %2
}
```

(b) Output IR program

Listing 3.7: C and IR code for `@add`

Indeed many potential causes of undefined behavior in a source program translate to similar sources of undefined behavior in the IR program, because C and LLVM-IR have a similar notion of undefined behavior. These similarities are hardly surprising, as information about undefined behavior needs to be present in LLVM-IR in order for LLVM to be able to make use of undefined behavior during optimization later on. Furthermore, because LLVM is not allowed to introduce new undefined behavior whenever there is undefined behavior on the LLVM-IR level, it is guaranteed that this stems from undefined behavior on the C level. Practical experience with LLBMC has shown that checking for undefined behavior on the LLVM-IR level works surprisingly well for finding undefined behavior on the C level, though it has also shown that it is not quite reliable.

Compiler optimizations in particular pose a problem when relying on undefined behavior in LLVM-IR. This is because optimizations might remove the add instruction entirely, if it is not needed or might remove the `nsw` flag from the instruction. The problem can be avoided if optimizations are disabled, but this causes severe performance penalties due to superfluous memory accesses for the code analysis and should therefore be avoided.

This problem can be solved more elegantly by *instrumentation* of the IR code with explicit checks for undefined behavior after the LLVM-IR code is generated and before optimizations are performed on the code. The compiler is then not allowed to change or remove this instrumentation during later optimization, and therefore optimization of the code can be done without losing any relevant information. Note, though, that the instrumentation itself might make some optimizations impossible.

```
1  define i32 @add(i32 %x, i32 %y↩
       ) {
2  entry:
3    alloca i32 %x_addr
4    alloca i32 %y_addr
5    store i32 %x, i32* %x_addr
6    store i32 %y, i32* %y_addr
7    %0 = load i32, i32* %x_addr
8    %1 = load i32, i32* %y_addr
9    call void %check.saddo.i32(↩
         i32 %0, i32 %1)
10   %2 = add i32 nsw %0, %1
11   ret i32 %2
12 }
```

(a) `@add` calling `@assert.saddo.i32`

```
1  define void @check.saddo.i32(↩
       i32 %x, i32 %y) {
2  entry:
3    %r = add %x, %y
4    %xneg = icmp slt i32 %x, 0
5    %yneg = icmp slt i32 %y, 0
6    %rneg = icmp slt i32 %r, 0
7    %xpos = xor i1 %xneg, 1
8    %ypos = xor i1 %yneg, 1
9    %rpos = xor i1 %rneg, 1
10   %0 = and i1 %xneg, %yneg
11   %1 = and i1 %0, %rpos
12   %2 = and i1 %xpos, %ypos
13   %3 = and i1 %2, %rneg
14   %4 = or i1 %1, %3
15   %5 = zext i1 %4 to i32
16   call void assert(i32 %5)
17   ret void
18 }
```

(b) Implementation of `assert.saddo.i32`

Listing 3.8: Instrumentation of `@add`

Listing 3.8a shows an example of what the instrumented IR code for the example in listing 3.7a could look like.[21] In this example, an additional checking function is

will be removed during optimization, in particular by the `mem2reg` optimization pass, later on.

[21] The implementation of the checking function in this example is inspired by [Bru10].

added to the module, which implements the check for an arithmetic overflow, and a call to this function is added before the addition's `add` instruction.

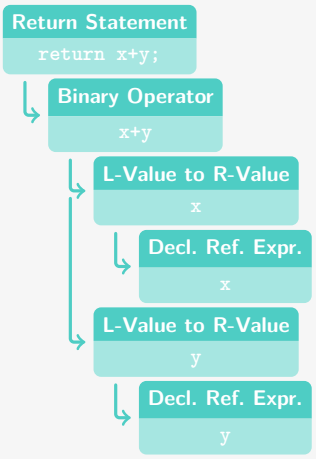
Even though instrumenting the IR code is an improvement over relying on LLVM-IR's undefined behavior, this is not sufficient to cover all cases of undefined behavior in a source file. The C standard established the following about pointer to integer conversion:

> Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. [...] [ISOC99, section 6.3.2.3]

In contrast to this, LLVM's `ptrtoint` instruction unconditionally performs a truncation if the value is not representable in the integer type, and therefore cannot cause undefined behavior. This possible source of undefined behavior clearly gets lost during code generation. A tool still trying to check for this particular case of undefined behavior would have to guess where to insert checks, which would hardly be reliable.

```
1 void add(int x, int y) {
2   return x+y;
3 }
```

(a) Input C program



(b) Parsed AST

Figure 3.6: AST for the recurring example

This leaves the modification of clang's code generator to take care of instrumenting LLVM-IR with undefined behavior as the only truly viable option for implementing code instrumentation.

Because clang's code generator is already rather complex, a minimally invasive, plug-in based solution was chosen.

Figure 3.7 shows how clang traverses the abstract syntax tree (shown in figure 3.6b) during code generation of the statement highlighted in figure 3.6a. In this example, the code generator starts with the return statement itself, recursively descends into its subexpressions, and emits instructions where necessary.

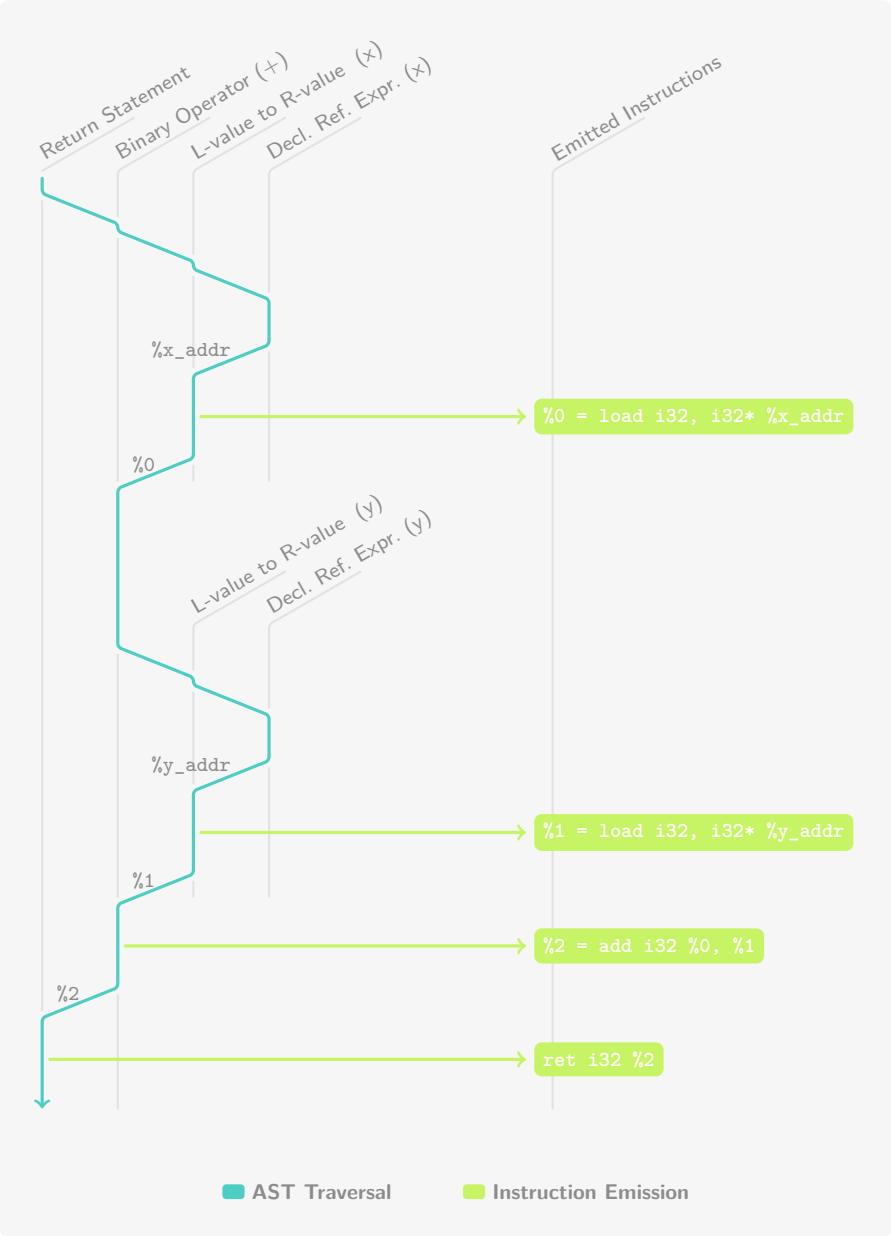


Figure 3.7: clang's code generator

The two inner-most (in the figure the right-most) expressions are so called declaration reference expressions. As the name indicates, these expressions are used to reference previously declared variables, in this case the integer variables `x` and `y`.

Because these variables are l-values, but the addition operator expects r-values as arguments, the arguments are converted from l-values to r-values first. According to the C standard, in C such conversions are implicit, trivial and always possible. In contrast, in clang's AST, the conversion are made explicit using dedicated AST nodes. Finally, in the LLVM-IR code, these conversions show up as `load` instructions.

Eventually, an `add` and a `ret` instruction are emitted for the binary operator + and the return statement, respectively. This concludes code generation of the return statement and its subexpressions.

LLBMC's instrumentation mechanism is implemented on top of a plugin system for clang's code generator. The system is minimally invasive and only touches a handful of locations in clang's code base. It installs hooks in the code generator at the beginning and end of the code generator of each statement, expression, and declaration. Even though instrumentation was the driving factor behind the development of this system, it is kept sufficiently generic to be used for other purposes. Figure 3.8 illustrates how hooks are called for the recurring example.

The instrumentation mechanism primarily makes use of the hooks called at the end of code generation of expressions to insert checking instructions into the IR. Figure 3.9 shows how this is done for the example above.

The plugin based approach has proven itself versatile enough to also support more difficult expressions, such as compound assignment. Figure 3.10 shows an example for the instrumentation of the expression `x += y`.

The C standard defines the semantics of *compound assignment* as follows:

> A *compound assignment* of the form **E1** *op*= **E2** differs from the simple assignment expression **E1** = **E1** *op* **(E2)** only in that the lvalue **E1** is evaluated only once. [ISOC99, section 6.5.16.2]

For the example above, this singular evaluation of the left-hand side operand can be observed in clang's traversal of the AST as a single visitation of the declaration reference expression for `x`. Because the left-hand side operand is an L-value not an R-value, there is no explicit L-value to R-value conversion node in the AST, and consequently, the `load` instruction, which is nonetheless required, has to be emitted by the compound assignment operator itself. This results in three instructions being emitted one after the other, the `load` instruction itself as well as the `add` and the `store` instruction.

The instrumentation should be added after the expression's arguments are done, but before the `store` instruction is done, due to that instruction's side effects. Therefore, the perfect place for instrumentation would be right after the `load` instruction and before the `add` instruction. Unfortunately, the plugin system doesn't provide a hook in between these instructions, and adding such a hook would make the plugin system far more invasive. As a compromise, the instrumentation therefore emits a second `load` instruction, identical to the one emitted by clang itself later on, so the value of `x` is loaded from memory and can be passed to the checking function. The two

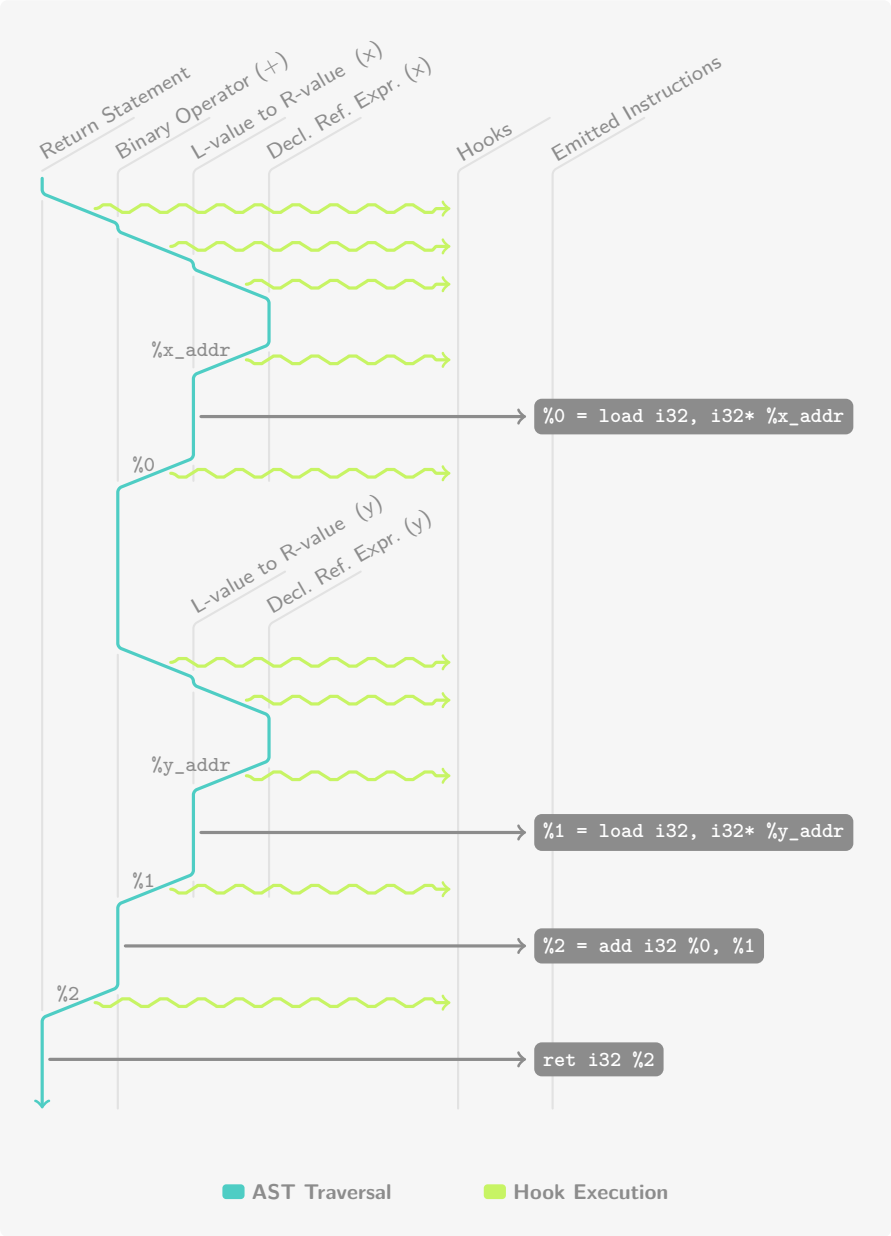


Figure 3.8: Callbacks for clang's code generator

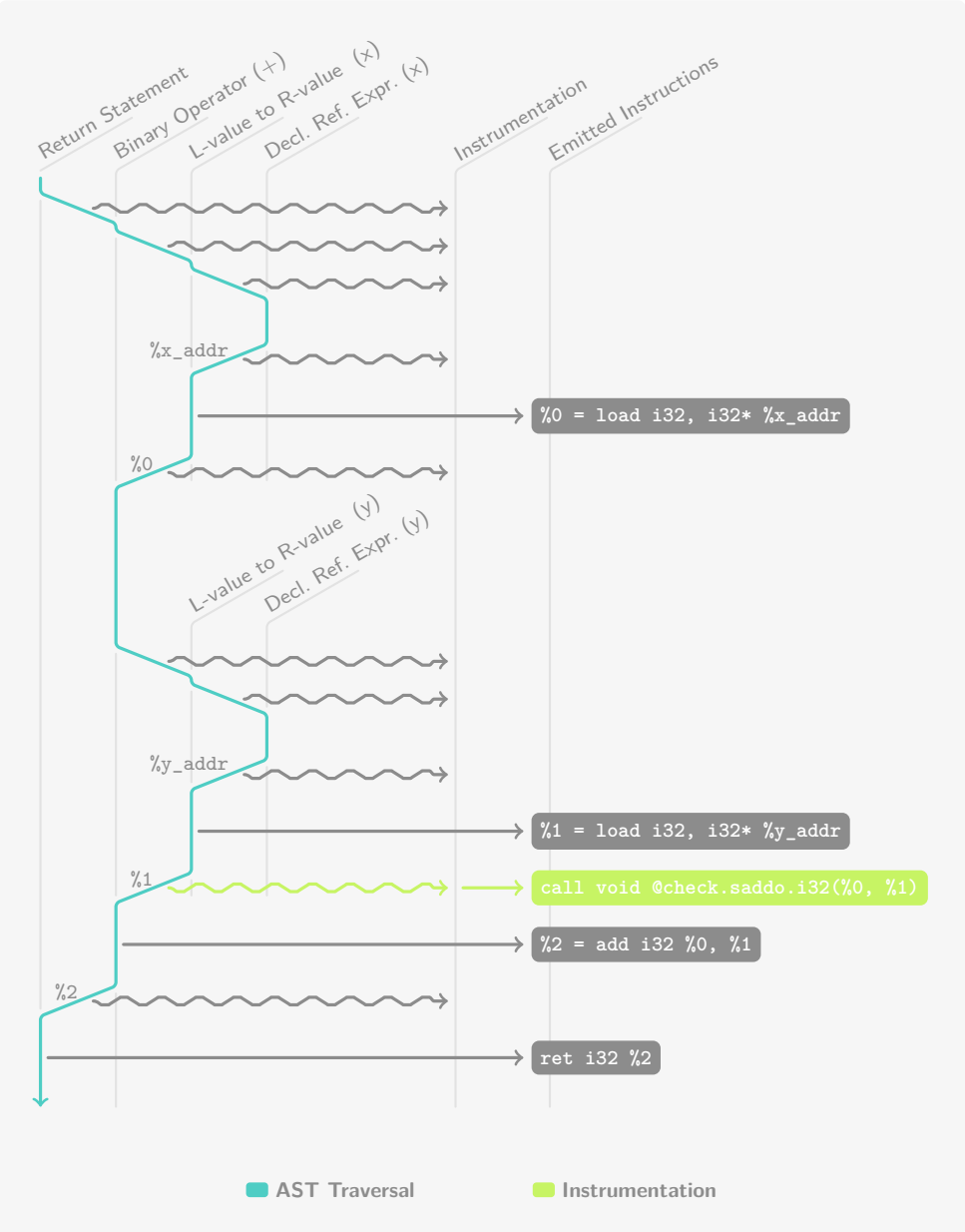


Figure 3.9: Instrumentation of clang's code generator

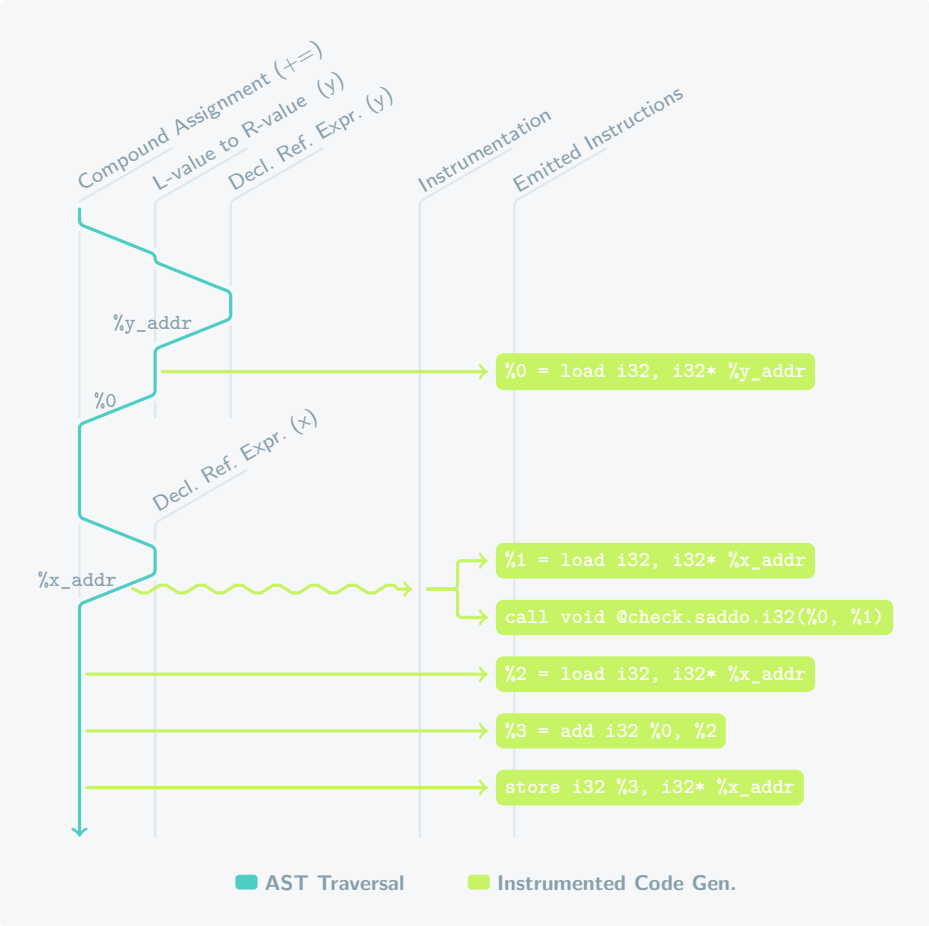


Figure 3.10: Instrumented code generation of `x += y;`

redundant `loads` can be easily merged into a single load instruction later on during optimization.

Section 6.3.4 shows a detailed example of how a C program is first translated to LLVM-IR, then to ILR, then to SMT, and finally how a counterexample is generated from this. The example also includes optimized and unoptimized LLVM-IR code, as well as instrumented variants of each.

More recent versions of clang contain `usan`, the undefined behavior sanitizer. This feature allows instrumentation of the clang code generator in a similar way to the approach described here. This provides a good starting point for checking undefined behavior if insufficient resources are available to implement instrumentation as described above. The sanitizer however is closely interleaved with clang's code generator and therefore highly intrusive and hard to modify.

3.4 Compiler Optimization Passes in LLBMC

Compiler optimization passes are a core feature of LLVM. The number of optimization passes provided by LLVM is continuously growing with currently more than 50 passes available. Each one of these is making use of different optimization opportunities to modify LLVM-IR code to improve the output binary's runtime or sometimes memory consumption. Additionally to these passes, LLBMC contains an additional set of LLVM-IR transformations, most of which are not targeted at optimization but at extending LLBMC's support for more exotic LLVM-IR language features.

Interestingly, many of these optimizations not only improve runtime on the target architecture, but they are also often, though not always, beneficial to the runtime of the SMT solver solving the formulæ generated from this code using LLBMC. This is because SMT solvers use bit-blasting to lower-level bitvector operations to propositional formulæ, and the generated formulæ closely match the circuits of real architectures.

Using compiler optimizations in a code analysis tool has to be handled with care, though. What is merely a missed optimization opportunity for a compiler might cause unsoundness or incompleteness for a static analysis tool, e.g. when loop unrolling fails because a loop has not the expected structure, the compiled code will at worst run slightly slower but a static analysis tool relying on loop unroll, such as LLBMC, might not be able to analyze the code at all.

3.4.1 Optimizations for Performance Improvement

Many optimizations in LLVM-IR can be used in LLBMC to improve performance. Experience has shown that many things which are expensive for a real processor are similarly expensive for an SMT solver.

One basic, exemplary optimization pass, `instcombine`, replaces one or more expensive instructions by one or more cheaper instructions, as can be seen in listing 3.9. The fact that the right hand side argument of the multiplication in listing 3.9a is the constant 3 is used to replace the multiplication by a shift operation and an addition, as can be seen in listing 3.9b.

```
define i32 @times3(i32 %x) {
  %0 = mul i32 %x, 3
  ret %0
}
```

(a) Code before optimization

```
define i32 @times3(i32 %x) {
  %0 = shl i32 %x, 2
  %1 = add i32 %0, %x
  ret %1
}
```

(b) Code after optimization

Listing 3.9: Example for instruction simplification optimization

The single most imprtant pass for LLBMC's performance is LLVM's built-in `mem2reg` pass. This pass lifts values stored in memory to virtual registers where possible. This reduces the number of memory read and write operations considerably, which in turn reduces the load on the SMT solver's implementation of the theory of arrays as well.

Consider the following trivial C function:

```
  int double(int x) {
    return x*x;
  }
```

This program is compiled to the IR code in listing 3.10a, and can then be optimized to the code in listing 3.10b. While similar optimizations are done by SMT solvers, it is paramount to do these optimizations early in order to avoid an intermediate blow-up of the formula's size.

```
define i32 @double(i32 %x) {
  %x_addr = alloca i32
  store i32 %x, %x_addr
  %0 = load i32 %x_addr
  %1 = add i32 %0, %0
  ret %1
}
```

(a) Code before optimization

```
define i32 @double(i32 %x) {
  %1 = add i32 %x, %x
  ret %1
}
```

(b) Code after optimization

Listing 3.10: Example for `mem2reg` optimization

The `alloca`, `store`, and `load` instructions are inserted by the clang code generator because at the time this code is emitted, the generator does not yet know that the address of `x` will never be used for anything else but loading and storing of `x` in the local function. Once the whole function is emitted, the `mem2reg` pass can analyze the function and can then remove the redundant memory access operations.

3.4.2 Optimizations for Extending Language Support

For programs which contain certain language constructs, the following compiler optimizations are required by LLBMC before encoding (see chapter 4) can happen.

This is because LLBMC's translation to ILR does not support all of LLVM's language features, so a preprocessing step is necessary to remove these.

LLVM's optimization pass `lowerindirectbranch` replaces any `indirectbr` by an equivalent construct of direct, conditional branches (`br`). This is straightforward, because `indirectbr` contains a list of allowed branch targets, so one `br` per unique entry in this list suffices.

LLVM's pass `lowerswitch` has a similar purpose as `lowerindirectbranch` though instead of replacing `indirectbr` instructions, it replaces `switch` instructions. Again, because the conditions and associated branch targets are explicitly listed in the `switch` instruction, this pass is straightforward.

LLBMC's `lowerindirectcall` pass replaces indirect calls by a switch over the set of possible call targets, with each case in the switch calling a single target. The set of possible targets is over-approximated by the set of functions with a matching signature. This can be refined using LLVM's alias analysis, though this currently does not happen in LLBMC.

LLBMC's `generalunroll` optimization pass is derived from LLVM's `unroll` pass but allows unrolling in corner cases which LLVM itself does not support. Note that this pass still relies on LLVM for detection of loops to unroll. This detection fails on loops with multiple entry points, e.g. as used in the so called Duff's device.

The pass `scalarrepl`, implemented in LLVM, replaces aggregates values by a set of scalar values. This reduces the need for support of aggregates in LLBMC.

LLBMC's `lowergep` replaces a `getelementptr` with more than 2 index arguments by a sequence of shorter ones. The code in listing 3.11 is the lowered form of the code in listing 3.6.

```
1 %RT = type { i8, [10 x [20 x i32]], i8 }
2 %ST = type { i32, double, %RT }
3
4 define i32* @foo(%ST* %s) {
5   %t1 = getelementptr %ST, %ST* %s, i32 1
6   %t2 = getelementptr %ST, %ST* %t1, i32 0, i32 2
7   %t3 = getelementptr %RT, %RT* %t2, i32 0, i32 1
8   %t4 = getelementptr [10 x [20 x i32]], [10 x [20 x i32]]* %t3, i32 ↩
        0, i32 5
9   %t5 = getelementptr [20 x i32], [20 x i32]* %t4, i32 0, i32 13
10  ret i32* %t5
11 }
```

Listing 3.11: Example for lowering of `getelementptr`

Finally, LLVM's `simplifycfg` pass simplifies the program's control flow graph and thereby ensures that only the entry basic block has no predecessor.

3.5 Control Flow Graphs

Control flow graphs are graphs describing the order in which a program's or function's basic blocks, and thereby also its instructions, are executed. They are an important

concept in compiler construction and used in many different compiler optimizations. LLBMC leverages LLVM's support for control flow graphs for loop unrolling but also the optimal order in which basic blocks are encoded.

In "Control Flow Analysis", Allen describes a *control flow graph* (*CFG*): "A control flow graph is a directed graph in which the nodes represent basic blocks and edges represent control flow paths " [All70].

While this provides a good starting point for a definition of a control flow graph, adaptations are required for one that is useful in the context of LLBMC bounded model checking. For example, LLBMC's definition for control flow graphs is based on functions instead of whole programs. Before an adapted definition can be introduced, it is necessary to define the term control flow edge first, though:

Definition 3.11 (Control flow edge). *Given a function f, a pair of basic blocks (b_0, b_1), $b_0, b_1 \in f$, where b_1 is labeled* `<b1>`*, is a control flow edge, if and only if*

- $\mathrm{term}_{\mathtt{B}}(b_0) \sim [\![$`br label <b1>`$]\!]$, or
- $\mathrm{term}_{\mathtt{B}}(b_0) \sim [\![$`br i1 <i1>, label <b1>, label <b2>`$]\!]$, or
- $\mathrm{term}_{\mathtt{B}}(b_0) \sim [\![$`br i1 <i1>, label <b2>, label <b1>`$]\!]$.

b_0 is called the predecessor, and b_1 is called the successor and the predicate $\mathrm{succ}_{\mathtt{F}}(b_0, b_1)$ is true.

Out of all of LLVM's terminator instructions introduced in table 3.9, only `br` occurs in this definition. Other terminators that pass control flow from one basic block to another, such as `switch` or `indirectbr`, are not considered here. This is possible, because it can be assumed, that the code transformations shown in section 3.4.2 were applied first and therefore none of these terminators occur in the code. The definition of a control flow edge leads to the following definition of a control flow graph:

Definition 3.12 (Control flow graph). *Given a function f, a control flow graph (V_f, E_f) for function f is a directed graph in which each node represents a basic block in f and the edges represent control flow edges between these basic blocks.*

In the context of LLBMC, a more specialized kind of control flow graph is used:

Definition 3.13 (Rooted control flow graph). *Given a function f, a rooted control flow graph (V_f, E_f, r_f) is a rooted directed graph, where (V_f, E_f) is a control flow graph and r_f is the only externally reachable basic block.*

In LLVM, a control flow graph's root is called the entry block.

3.5.1 Graphical Illustration of Control Flow Graphs

Figure 3.11 shows a graphical representation of a basic block, useful for displaying a basic block as a node in a control flow graph. We will use this style of illustration throughout the thesis. In these illustrations, an ellipsis indicates that this part of an instruction is not relevant for the example. Additionally, names of basic blocks as well as locally and globally defined values are assumed to be unique in the example in which they are used. In these illustrations, names always consist of a single letter which is optionally followed by a number. If such a value is referred to outside of

the example, in particular in mathematical formulæ, the number will be displayed as a subscript to the letter, e.g. the basic block labeled `b1` will be referred to as b_1.

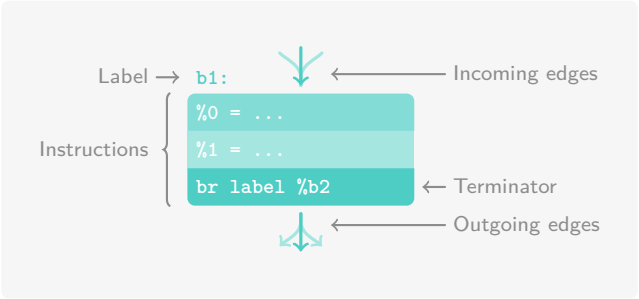


Figure 3.11: Graphical illustration of a basic block b_1

3.5.2 Bounded Control Flow Graphs

In LLBMC there are two places at which the bound, from which the method Software Bounded Model Checking has its name, comes into play. The first is the bounding of the call graph, and the second the bounding of the control flow graph, which is discussed here:

Definition 3.14 (Back Edge Set). *Given a rooted control flow graph (V, E, r), a set of edges B is a back edge set if B is the smallest possible set so that $(V, E \setminus B, r)$ is a connected, directed acyclic tree.*

The above definition immediately leads to the following definition of a back edge:

Definition 3.15 (Back Edge). *Given a rooted control flow graph (V, E, r), and a back edge set $B \subset E$, each edge $b \in B$ is called a back edge.*

Which in turn leads to the following definition:

Definition 3.16 (Bounded Control Flow Graph). *Given a function f and a control flow graph (V_f, E_f, r_f) and a back edge set $B_f \subseteq E_f$, The graph $(V_f, E_f \setminus B_f, r_f)$ is called the bounded control flow graph of (V_f, E_f, r_f).*

A program which is modified so that its control flow graph is reduced to its bounded control flow graph (by modifying branch instructions accordingly) is entirely loop free. Such a loop-free program can be translated into an SMT formula straightforwardly by symbolic execution, though the semantics of the resulting program have changed relative to the original program. More of the that program's semantics can be retained by unrolling loops before applying the above transformation. If a maximal number of loop iterations is known for each loop in the program, the unrolling can be done so that the program's semantics do not change. This is guaranteed to be true because the edges which are removed are known to never be executed. Unrolling in LLBMC is done as shown in section 3.4.2.

3.6 Call Graphs

Call graphs are directed graphs which represent the calling relationships between functions. Nodes in a call graph represent functions and the edges of a call graph represent function calls. Call graphs are an important data structure in compilers and software analysis tools and indispensable for a wide range of compiler optimizations.

CBMC and its derivatives use function inlining and loop unrolling to ensure that every variable is assigned only once during any execution of the program.[22] As a consequence, the program's call graph is only implicitly given by the single, large resulting function. In contrast, LLBMC uses explicit call graphs for the same purpose. This reduces the tool's memory footprint but also adds more flexibility when enhancing the core SBMC algorithm. For example, function summarization is easily added to LLBMC by replacing regular nodes in the call graph which represent the calling of a function, by special nodes which represent the summarization of the function. Instead of inlining a function's body the summarization is used instead. DAG inlining, as presented by Lal and Qadeer [LQ15], can be implemented easily using an explicit call graph.

This section leads towards the definition of a call graph as used in LLBMC. The terms program, function and instruction are used as defined in section 3.2. Furthermore, the optimizations presented in section 3.4 ensures, that indirect function calls are replaced by direct function calls, so it is safe to assume no indirect function calls occur in the program.

There is a wide variety of different kinds of call graphs. For the purpose of this thesis, we can distinguish three core properties of call graphs: Call graphs can have different levels of precision and context-sensitivity, and they can be either static or dynamic.

A call graph is called *dynamic* if it was generated from the results of an instrumented run of the program and it is called *static* otherwise. Because LLBMC is a static code analysis tool, this thesis exclusively deals with static call graphs. Dynamic call graphs are merely introduced for the sake of completeness.

We will also distinguish between precise, over-approximating, and under-approximating call graphs. A *precise call graph* contains an edge for a call if and only if a trace through the program exists which contains this call. An *over-approximating call graph* contains additional edges for which the associated call cannot be executed in reality. Finally, an *under-approximating call graph* lacks some edges of a precise call graph. Grove et al. [Gro+97] provide a formalism which embeds call graphs with different levels of precision in a lattice where the top element corresponds to the empty call graph and the bottom element corresponds to the complete call graph, which covers all possible executions.

Dynamic call graphs are often under-approximating, as they are generated by a limited number of program executions during which not necessarily all paths might have been taken. In contrast, static call graphs are often over-approximating, because computation of a precise call graph can be too expensive for many uses, in particular for the use in compilers. Instead of running a semantic analysis to determine if an

[22] This can also be seen as an inter-procedural extension of static single assignment form.

edge can actually be called, these call graphs simply assume that if a call exists in the program syntactically then it is also an edge in the call graph.

3.6.1 Context-Sensitivity in Call Graphs

The most important property of call graphs in the context of LLBMC is *context-sensitivity*. In a *context-insensitive call graph* every function is represented by exactly one node and given two functions `@f` and `@g`, the graph contains a directed edge between the nodes representing function `@f` and `@g`, if there is a function call from `@f` to `@g`. In contrast, in a *context-sensitive call graph* a single function might be represented by multiple nodes. Nodes that represent the same function differ by the context in which a function is called. In the most basic case this context is simply the immediately calling function, but multiple levels of calls can be taken into account, too.

Consider, for example, a program with the functions `@f`, `@g`, and `@h` where `@h` is called from functions `@f` and `@g`. In a context-insensitive call-graph, instead of having a single node for function `@h`, one could have one node for function `@h` as called from `@f` and another node for function `@h` as called from function `@g`. Such a context-sensitive call graph provides a more detailed view on the calling relations in the program and thereby can open up additional optimization opportunities for a compiler. Nonetheless, compilers often use static, over-approximating, context-insensitive call graphs, primarily because they can be generated quickly, and provide the best cost-benefit ratio. In the context of software bounded model checking, the reduced precision in a compiler's call graphs, is not a matter of missed optimization opportunities, but of correctness.

Figure 3.12 is the context-insensitive call graph for listing 3.12. Figure 3.13 shows the context-sensitive call graph of listing 3.12. Note that, in contrast to the context-insensitive call graph in figure 3.12, there are two nodes for `@h` and `@i` each.

```
1  define i32 @f() {
2  entry:
3    %fg = call i32 @g()
4    %fh = call i32 @h()
5    ret i32 %1
6  }
7
8  define i32 @g() {
9  entry:
10   %gh = call i32 @h()
11   %gi = call i32 @i()
12   ret i32 %1
13 }
```

```
14 define i32 @h() {
15 entry:
16   %h1 = call i32 @i()
17   %h2 = call i32 @i()
18   ret i32 %h2
19 }
20
21 define i32 @i() {
22 entry:
23   ret i32 0
24 }
25
26
```

Listing 3.12: Call graph example code

The notion of context-sensitivity can be generalized further. Instead of identifying a node in a context-sensitive call graph only by a function and its immediate caller, a sequence of functions can be used instead, where each element of the sequence

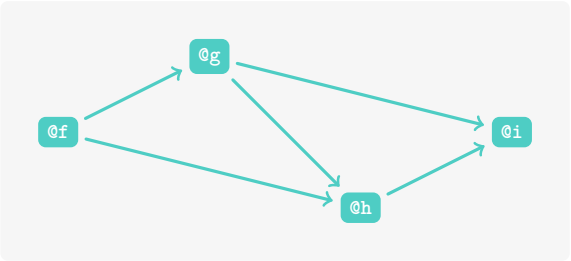


Figure 3.12: A context-insensitive call graph for listing 3.12

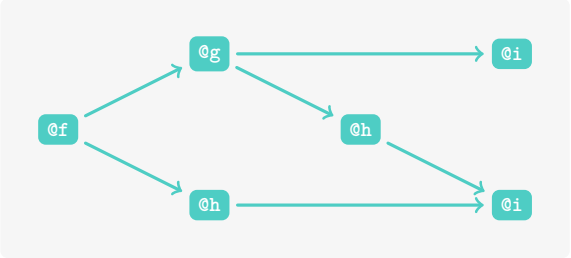


Figure 3.13: A context-sensitive call graph for listing 3.12

contains a call to its successor (except, of course, for the last element in the sequence). In order to be able to express this we define a context to be such a sequence:

Definition 3.17 (Context). *Given a set of functions F and a function $f \in F$, a sequence $c = (f_1, \ldots, f_n)$ (with all $f_i \in F$) is called a context of f if f_n contains a call to f, and for each pair f_i, f_{i+1} in c each f_i contains a call to f_{i+1}.*[23]

We use $|c|$ to indicate the length of a context and we define the empty sequence to be a context of every function. Contexts are inspired by the concept of contours as presented by Grove et al. [Gro+97] (see also [Shi90]).

With this definition of a context we can now define context-sensitive call graphs:

Definition 3.18 (Context-sensitive call graph). *A context-sensitive call graph is a graph $G = (V, E)$ where each $v \in V$ is labeled with a sequence $(f_1, \ldots, f_n, f)$, where $(f_1, \ldots, f_n)$ is a context of f and for all $(v, u) \in E$: if u is labeled $(f_1, \ldots, f_n, f)$ then v is labeled $(\ldots, f_1, \ldots, f_n)$.*

A context-insensitive call graph can be seen as a context sensitive call graph (V, E) where $|v| = 1$ for all $v \in V$. The most common type of context-sensitive call graphs used by compilers is a context-sensitive call graph where $|v| = 2$ for all $v \in V$. In addition, we will call a context-sensitive call graph a *fully context-sensitive call graph* if for all $(f_1, \ldots, f_n, f) \in V$, there is no context c of f with $c = (f_0, f_1, \ldots, f_n)$. Intuitively this means the context for every node reaches back all the way to the program's entry point.

In chapter 4 we assume that the program has a single, unique entry point. If a library with multiple entry points is to be verified, LLBMC needs to be run multiple times,

[23] A C,C++, or LLVM-IR program's entry point is the `main` function. Libraries may have multiple entry points.

once for each entry point. A call graph can then be generated based on the slice of the program reachable from this entry point. DAG inlining, introduced by [LQ15], is a different approach which handles multiple entry points natively.

In chapter 4, we use call graphs to guide the encoding algorithm. An infinite large call graph would result in non-termination of the algorithm. To ensure the call graph and therefore the encoding's runtime are finite, the call graph needs to be bounded artificially:

Definition 3.19 (Bounded call graph). *Given a program p and its rooted, fully context-sensitive call graph $G = (V, E, r)$, we call $G^n = (V^n, E^n, r^n)$ the n-bounded call graph of G, if $V^n = V \setminus \{v \in V : |v| > n\}$, $E^n = E \setminus \{(v, u) : v \notin V^n \vee u \notin V^n\}$, and $r = r^n$.*

3.6.2 Call-Site-Sensitive Call Graphs

CBMC uses function inlining to ensure each variable is assigned a value at most once during execution of the program. This is necessary for CBMC because it encodes each instruction as a separate term. However, LLBMC does not do inlining and therefore cannot have a one-to-one relation between instructions and terms. Instead, LLBMC associates each pair (c, i) of a context c and an instruction i with a unique term.

A context-sensitive call graph as described above is not sufficient for this because such a graph does not distinguish between two calls to a function which are from different call-sites inside another function.[24] Consider the example in listing 3.13: The function `@g` is called twice in `@f`, where the first call is safe, but the second call is not.

```
define void @f() {
  %0 = call i32 @g(i32 4)
  %1 = call i32 @g(i32 0)
}
```

```
define i32 @g(i32 %x) {
  %res = udiv i32 10, %x
  return i32 %res
}
```

Listing 3.13: Example showing the importance of call sites

A call-site-sensitive call graph requires a slightly different definition of a context:

Definition 3.20 (Call-site-sensitive context). *Given a function f, a sequence $(f_1, i_1, f_2, \ldots, f_n, i_n)$ is a call-site-sensitive context, if each f_j is a function and each i_j a* `call` *instruction in f_j which calls f_{j+1}.*

We will call i_j the context's *call site* and $(f_1, i_1, \ldots, f_{n-1}, i_{n-1})$ the context's *parent*. We use $\mathrm{callsite}(c)$ to refer to c's call site and $\mathrm{parent}(c)$ to refer to c's parent. Note that both functions are only partially defined. We will furthermore use $\mathrm{fun}(c)$ to refer to the function called by c's last instruction.

Definition 3.21 (Call-site-sensitive call graph). *A call-site sensitive call graph (V, E) is a context-sensitive call graph where each node $v \in V$ is labeled with the sequence $(f_1, i_1, \ldots, f_n, i_n, f)$ and $(f_1, i_1, \ldots, f_n, i_n, f)$ is a context of f.*

[24] A call-site is the function call's `call` instruction.

Figure 3.14 shows the call-site-sensitive call graph of listing 3.12.

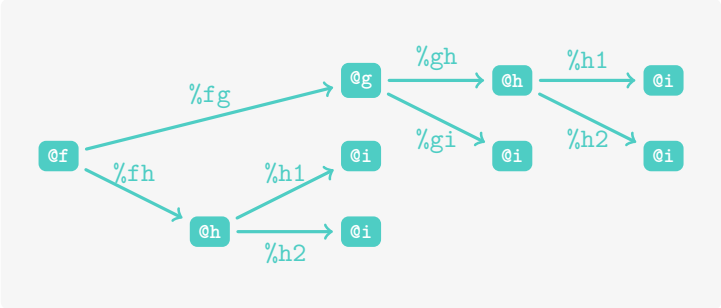


Figure 3.14: A call-site-sensitive call graph for listing 3.12

The definitions of bounded, rooted, and fully call-site-sensitive call graphs follow the corresponding definitions for context-sensitive call graphs. In the following, we use bounded, rooted, fully call-site-sensitive call graphs. Because of the loop unrolling, every instruction in every context is guaranteed to be called only once. Furthermore, because we assume there are no indirect function calls so every call goes to a specific function. Finally, due to the construction of a call-site-sensitive graph, in such a context every `call` instruction leads to a distinct context. These properties together guarantee, that for any execution of the program any pair (c, i) of a context c and an instruction i is executed at most once. This means such a call graph can be used in place of syntactic inlining. This is shown in more detail in chapter 4.

3.7 LLBMC's Intermediate Logic Representation

The *Intermediate Logic Representation* (*ILR*) is LLBMC's primary internal language and at the same time an SMT theory. It is primarily used in LLBMC for term rewriting and thereby acts as an intermediate step in the translation of an LLVM-IR program into a bitvector and array formula. Its design took inspiration from all three of LLVM-IR (see section 3.2), SMT-LIB (see section 2.1.2), and term rewriting systems (see 2.1.6). The language is designed to be sufficiently expressive to make encoding of bounded fragments of LLVM-IR programs simple, to be easily translatable to SMT-LIB's quantifier-free logic of arrays and bitvectors (QF_ABV), and to be well-suited for term rewriting.

LLVM-IR's instruction set and the functions defined in the theory of bitvectors already share many similar operations, including arithmetic operations, shift and bitwise logic operations, truncation and extension operations, as well as extraction and insertion of subranges of bitvectors. On top of this, LLVM-IR's memory related operations can be mapped straightforwardly to functions from SMT-LIB's theory of array. Of course, these commonalities were taken into account during the design of many of ILR's functions, which often have matching counterparts in LLVM-IR instructions as well as QF_ABV.

Because ILR is intended to encode LLVM-IR programs, ILR, like LLVM-IR itself, is a typed language with a type system closely modeled after LLVM-IR's type system. Furthermore, the functions mentioned above are supplemented by a considerable

number of functions that encode desired and undesired properties of these programs, e.g. different kinds of arithmetic overflow. This way, LLVM-IR's notion of undefined behavior and its instruction flags can be encoded, too.

Unlike LLVM-IR, and unlike Boogie [Bar+05], a popular intermediate representation for verification, ILR is not a language for describing imperative programs but a logic language. Consequently, ILR has no notion of control flow, and no implicit program state. Instead, an LLVM-IR program's control flow and program state are encoded explicitly in an ILR formula. While LLVM-IR supports recursive types[25] and LLBMC's implementation supports this as well, ILR as presented here does not do so for reasons of readability.

As exemplified by SMT-LIB, ILR does not distinguish between terms and formulæ. The role of formulæ in first-order logic is assumed by terms of a dedicated boolean sort and predicates are replaced by functions of this sort. Like SMT-LIB, ILR does not pose any constraints on the bitwidth of a bitvectors, while LLVM-IR restricts the bitwidth to $2^{23} - 1$ bits. However, for realistic problems this is not relevant, as such large types do not occur in practice.

The comprehensive type system and the type conversion functions are one major differentiating point between ILR and QF_ABV. For example, unlike QF_ABV, ILR differentiates bitvector types not just by their bitwidth but also by their use, e.g. as integers or as pointer. Additionally, while SMT-LIB only supports single byte read and write operations, ILR uses a flat memory model with a single, large array for the entire memory space and therefore requires support for read and write operations with any fixed-number of bytes.

Note that because ILR uses schemata to describe sorts, functions, and axioms, it does not have a finite set of sorts, functions, and axioms. However, because LLVM-IR programs are finite, LLBMC's call graphs are finite, too, and because LLBMC's term rewriting systems terminates, any program can be encoded in a formula using finite subsets.

The primary focus of this section is the introduction of ILR and it sorts and functions. In the following ILR's function schemata are introduced in tables, such as table 3.13. Functions and predicates that are not part of ILR itself, but only used for the axiomatization of ILR functions, are called auxiliary functions and predicates and are introduced in definitions throughout the section. Note that the set of sorts and functions introduced here comprise just the core of the ILR language, as ILR is an extensible language. The language is extended in chapter 4 by function symbols used for the encoding of LLVM-IR programs in ILR and in chapter 5 with features for dynamic memory management.

Note that because ILR is only used as an intermediate representation and is never written to disc or the screen, it currently does not have a concrete syntax. We use the syntax of first-order logic for ILR's abstract syntax.

Note that ILR is not fully specified, similar to how QF_ABV is not fully specified (see [KRW09]). For example, the result of a division by zero is unspecified in ILR and QF_ABV, and may be implemented any way, as long as $x = y \rightarrow \frac{x}{0} = \frac{y}{0}$.

[25] Recursive types are types that refer to themselves, e. g. a structure containing a pointer to another structure of the same type.

3.7.1 Sorts

ILR's sorts are listed in table 3.10. Like SMT-LIB, ILR allows for derived sorts, which means sorts can have an arity larger than zero and may take natural numbers or other sorts as arguments. For example, the pointer sort has an arity of one and may take, an integer sort as its argument. For sake of simplicity, we will not consider recursive sorts in this thesis.[26]

Sort Family	Pattern	Examples
Boolean	bool	bool
Integer	i$\langle N \rangle$	i1, i8, i32, i64
Pointer	$\langle S \rangle$*	i32*
Array	[$\langle N \rangle$ x $\langle S \rangle$]	[4 x i32]
Structure	$\{\langle S \rangle, \ldots\}$	{i32, i8*}
Memory State	m$\langle N \rangle$	m16, m32 m64

Table 3.10: ILR sorts and sort schemata
($\langle N \rangle$ act as a placeholder for a positive integer and $\langle S \rangle$ for any sort except memory states)

Like LLVM-IR but unlike C, ILR does not have a small, fixed set of sorts and functions, but whole families of related sorts and functions instead. For example, where C has a fixed number of integer sorts[27], ILR has integer sorts with arbitrary bitwidths. Because of this, this section primarily presents *axiom schemata*, which need to be instantiated first to retrieve concrete axioms. Consequently, these schemata which make use *sort placeholders* instead of concrete sorts. For example, the sort placeholder $\mathcal{I}$ stands for any integer sort and an axiom schema containing $\mathcal{I}$ can be instantiated by substituting all occurrences of $\mathcal{I}$ with a concrete integer sort, e.g. i8. Table 3.11 introduces the sort placeholders used in this section and the sorts each one may be instantiated with.For convenience, we will apply the patterns shown in table 3.10 on the sort placeholders from table 3.11 to express relations between sort placeholders. For example, $\mathcal{I}$* indicates a placeholder for a pointer sort pointing at an object of sort $\mathcal{I}$. For example, if $\mathcal{I}$ is instantiated as i8, $\mathcal{I}$* must be instantiated as i8*.

Core sorts in ILR are the integer sorts (e.g. i8) and the pointer sorts (e.g. i8*). In LLVM, integer and pointer types are so called simple value types. For consistency with LLVM, these sorts will be called the *simple value sorts* in ILR. Some axiom schemata can be instantiated for integers and pointers alike. These axioms will use the dedicated placeholders $\mathcal{V}$, $\mathcal{V}_1$, and $\mathcal{V}_2$.

Array and structure sorts refer to the same concepts in C and LLVM-IR. These are called *aggregate sorts*. Note that, while LLBMC supports aggregate sorts, ILR, as presented in this thesis, does not allow these sorts for reasons of readability. These sorts are only used for constructing appropriate pointer sorts and for use in pointer arithmetics based on these sorts.

[26] Recursive sorts are sorts which contain pointers to themselves. Of course, LLBMC's implementation allows for recursive sorts.

[27] Namely `char`, `short`, `int`, `long`, long long and their unsigned counterparts.

Sort Placeholder	Instantiatable Sorts
$\mathcal{I}$, $\mathcal{I}_1$, $\mathcal{I}_2$	Integer sorts
$\mathcal{P}$, $\mathcal{P}_1$, $\mathcal{P}_2$	Pointer sorts
$\mathcal{A}$	Array sorts
$\mathcal{S}$	Structure sorts
$\mathcal{M}$	Memory state sorts
$\mathcal{V}$, $\mathcal{V}_1$, $\mathcal{V}_2$	Simple value sorts
$\mathcal{N}$	Native sorts
$\mathcal{U}$	All sorts

Table 3.11: Sort placeholders used in the following and their possible instantiations

Because simple value sorts and aggregate sorts both are also present in LLVM-IR, these sorts will be called *native sorts*. Native sorts are primarily used to indicate that something can be pointed at with a pointer. Consequently, memory state sorts are not native sorts, as neither do they exist in LLVM nor can the be pointed at.

Sort Family	Variables
Boolean	b, b_1, b_2, ...
Integer	x, y, z, x_1, x_2, ...
Pointer	p, p_1, p_2, ...
Memory state	m, m_1, m_2, ...
Integer or Pointer	v, v_1, v_2, ...
All sorts	u, u_1, u_2, ...

Table 3.12: Variable naming conventions

Not all sorts in ILR are matched by LLVM types. In contrast to LLVM-IR, ILR has a dedicated boolean sort. Furthermore, ILR has a sort representing memory states. LLVM-IR and SMT-LIB have notably different uses of the word array. ILR follows LLVM-IR's lead here, with its array sort matching LLVM-IR's array type. SMT's array sort is matched by ILR's memory sort, with the differing name being chosen to avoid confusion with LLVM-IR inspired array sort. While SMT-LIB's array sort takes two arguments, the index sort and the element sort, ILR's memory sort only takes a single argument, the index sort, while the element sort is always i8.

3.7.2 Booleans

ILR defines a dedicated boolean sort bool. Having first-order logic in mind this might seem redundant, but this is important for LLBMC because the sort makes it syntactically possible to define a select function which is similar to C's ?:-operator.[28]

[28] Functions in first-order logic take terms as arguments, never formulæ, so select would not be legal syntax without a dedicated boolean sort. The term $\mathrm{select}(b, x_1, x_2)$ can be "simulated" by

Functions related to the boolean sort are listed in table 3.13.

Symbol : Signature	Interpretation
$T : \rightarrow \text{bool}$	Boolean true
$F : \rightarrow \text{bool}$	Boolean false
$\text{eq} : \text{bool} \times \text{bool} \rightarrow \text{bool}$	Logical equivalence
$\text{and} : \text{bool} \times \text{bool} \rightarrow \text{bool}$	Logical conjunction
$\text{or} : \text{bool} \times \text{bool} \rightarrow \text{bool}$	Logical disjunction
$\text{not} : \text{bool} \rightarrow \text{bool}$	Logical negation

Table 3.13: Boolean ILR functions

Definition 3.22 ($\langle\cdot\rangle$). *Given a boolean b, the predicate $\langle b\rangle$ is true if $b = T$ and false otherwise.*

This definition allows to formalize the semantics of functions related to the boolean sort:

$$\langle T\rangle \leftrightarrow \top \tag{3.1a}$$
$$\langle F\rangle \leftrightarrow \bot \tag{3.1b}$$
$$\forall b_1, b_2 \left(\langle \text{eq}(b_1, b_2)\rangle \leftrightarrow (\langle b_1\rangle \leftrightarrow \langle b_2\rangle)\right) \tag{3.1c}$$
$$\forall b_1, b_2 \left(\langle \text{and}(b_1, b_2)\rangle \leftrightarrow (\langle b_1\rangle \wedge \langle b_2\rangle)\right) \tag{3.1d}$$
$$\forall b_1, b_2 \left(\langle \text{or}(b_1, b_2)\rangle \leftrightarrow (\langle b_1\rangle \vee \langle b_2\rangle)\right) \tag{3.1e}$$
$$\forall b_1 \left(\langle \text{not}(b)\rangle \leftrightarrow \neg\langle b\rangle\right) \tag{3.1f}$$

For brevity, we will at times use terms of sort bool as if they were formulæ, though only when referring to them in the running text.

3.7.3 Integers and Pointers

ILR's integers are nearly identical to QF_ABV's bitvectors. Nonetheless, in the context of ILR, the term integer is used instead of bitvector, in order to stay in line with LLVM's terminology here. To avoid confusion with mathematical integers, those are referred to as mathematical integers explicitly.

Definition 3.23 (Integer). *$x = (x_{n-1}, \ldots, x_0)$ is an integer, where $x_i \in \{0, 1\}$ for $0 \leq i < n$.*

Because LLBMC aims to verify low-level software, it requires faithful handling of various operations on pointers that occur in such code, e.g. bit-stuffing. Because of this, a pointer's bit pattern is highly relevant for verification, and therefore LLBMC cannot handle pointers in an abstract way, even thought desirable for performance reasons, but needs to treat them as bitvectors, too:

replacing it by an uninterpreted constant symbol x of the appropriate sort and adding $b \rightarrow x = x_1$ and $\neg b \rightarrow x = x_)$ as top-level constraints to the formula. This is undesirable in the context of LLBMC's reliance on term rewriting, though.

Definition 3.24 (Pointer). $p = (p_{n-1}, \ldots, p_0)$ *is a pointer, where* $p_i \in 0, 1$ *for* $0 \leq p < n$.

Many of the axioms related to integers and pointers presented below depend on the ability to reason about single bits in an integer or pointer. For this an auxiliary bit extraction predicate is provided:

Definition 3.25 (Bit extraction). *Given an integer or pointer* $b = (b_{n-1}, \ldots, b_0)$, *the predicate* $\cdot[i]$ *with* $0 \leq i < n$ *is defined by* $b[i] \leftrightarrow b_i = 1$.

Finally, the number of bits in an integer is equally important, requiring the following definition:

Definition 3.26 (Bitwidth). *Given an integer or pointer* $b = (b_{n-1}, \ldots, b_0)$, $|b| = n$ *is called* b*'s bitwidth.*

We will also extend the use of the operator $|\cdot|$ to simple value sorts and sort placeholders, with the obvious meaning.

Symbol : Signature	Interpretation
$\mathrm{and}_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Bitwise logical conjunction
$\mathrm{or}_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Bitwise logical disjunction
$\mathrm{xor}_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Bitwise exclusive or
$\mathrm{not}_{\mathcal{I}} : \mathcal{I} \to \mathcal{I}$	Bitwise negation

Table 3.14: Bitwise ILR functions

We can now introduce the first set of ILR's integer related functions, the bitwise operations. These operations are closely modeled after LLVM-IR's bitwise operations, though the instructions' flags (`nsw`, `uw`, and `exact`) are omitted here. Suitable ILR functions for encoding these flags are presented in table 3.18. The bitwise operations include and, or, and xor, as well as not for bitwise logical negative (see table 3.14). While the former are present in LLVM-IR, too, the latter is not. These operations are defined as usual:

$$\forall x, y, i \left(0 \leq i < |x| \to (\mathrm{and}_{\mathcal{I}}(x, y)[i] \leftrightarrow x[i] \wedge y[i])\right) \tag{3.2a}$$

$$\forall x, y, i \left(0 \leq i < |x| \to (\mathrm{or}_{\mathcal{I}}(x, y)[i] \leftrightarrow x[i] \vee y[i])\right) \tag{3.2b}$$

$$\forall x, y, i \left(0 \leq i < |x| \to (\mathrm{xor}_{\mathcal{I}}(y, z)[i] \leftrightarrow \neg(x[i] \leftrightarrow y[i]))\right) \tag{3.2c}$$

$$\forall x, y, i \left(0 \leq i < |x| \to (\mathrm{not}_{\mathcal{I}}(x)[i] \leftrightarrow \neg(x[i]))\right) \tag{3.2d}$$

Additionally to bitwise operations, ILR's integers support arithmetic operations, many of which can be defined by interpreting an integer's bit pattern as a natural number:

Definition 3.27 (Binary encoding). $\langle\cdot\rangle^u_{\mathcal{V}} : \mathcal{V} \to \mathbb{N}$ *is the binary encoding (unsigned encoding) of a mathematical integer as an ILR integer or pointer with bitwidth* n*:*

$$\langle v \rangle^u_{\mathcal{V}} = \sum_{i=0}^{n-1} v_i 2^i$$

Similarly, an integer's bit pattern can be interpreted as a mathematical integer using two's complement encoding:

Definition 3.28 (Two's complement encoding). $\langle\cdot\rangle^s_{\mathcal{V}} : \mathcal{V} \to \mathbb{Z}$ *is the signed encoding (two's complement) of a mathematical integer as an ILR integer with bitwidth* n*:*

$$\langle v\rangle^s_{\mathcal{V}} = -v_{n-1}2^{n-1} + \sum_{i=0}^{n-2} v_i 2^i$$

ILR uses truncated division for signed integer division, as in C, LLVM-IR and SMT'S theory of bitvectors. This is in contrast to the use of floored division by Knuth [Knu73]. Truncated division is based on the following definition of truncation:

Definition 3.29 (Truncation). *The function* int *truncates a real number to its integer part:*

$$\mathrm{int}(x) = \begin{cases} \lfloor x \rfloor & \text{if } x \geq 0 \\ \lceil x \rceil & \text{otherwise} \end{cases}$$

Symbol : Signature	Operation
$(x)_{\mathcal{I}} : \to \mathcal{I}$	Integer constant
$(x)_{\mathcal{P}} : \to \mathcal{P}$	Pointer constant

Table 3.15: Integer and pointer constants

Integer and pointer constants are functions with arity zero, as shown in table 3.15, e.g. $(42)_{i32}$ for the number 42 as a 32-bit number, $(-3)_{i8}$ for the number -3 as an 8-bit integer, and $(0x0)_{i32*}$ for a null pointer pointing at a 32-bit integer.

$$\forall x \left(\langle (x)_{\mathcal{I}} \rangle^u_{\mathcal{I}} \equiv \mathrm{int}(x) \pmod{2^{|\mathcal{I}|}} \right) \tag{3.3a}$$

$$\forall x \left(\langle (x)_{\mathcal{P}} \rangle_{\mathcal{P}} \equiv \mathrm{int}(x) \pmod{2^{|\mathcal{P}|}} \right) \tag{3.3b}$$

Note that in equations (3.3), x may be any expression that can be evaluated to a real number. Furthermore, any two constants $(x)_{\mathcal{I}}$ and $(y)_{\mathcal{I}}$ for which

$$\mathrm{int}(x) \equiv \mathrm{int}(y) \pmod{2^{|\mathcal{I}|}}$$

are treated syntactically as the same constant. This means for any integer sort $\mathcal{I}$, there are exactly $2^{|\mathcal{I}|}$ different constant functions. This is important for the effectiveness of simplifications (see section 6.1) and term sharing (see section 3.7.7). For example, $(3)_{\mathcal{I}}$, $(\pi)_{\mathcal{I}}$, and $(10/3)_{\mathcal{I}}$ all represent the same function. The same is true for any pointer sort $\mathcal{P}$.

ILR provides a number of arithmetic functions including addition, subtraction, multiplication, division, and remainder calculation. If signed and unsigned differ for an operation, both variants are present. Note that signed and unsigned multiplication do not differ in the lower half of the result. Because ILR truncates the higher half of the result, a single multiplication function therefore suffices. Table 3.16 lists ILR's arithmetic functions.

Symbol : Signature	Interpretation
$\mathrm{add}_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Addition
$\mathrm{sub}_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Subtraction
$\mathrm{mul}_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Multiplication
$\mathrm{div}^u_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Unsigned division
$\mathrm{rem}^u_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Unsigned remainder
$\mathrm{div}^s_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Signed division
$\mathrm{rem}^s_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Signed remainder

Table 3.16: Arithmetic ILR functions

To be consistent with LLVM-IR and QF_ABV, division is defined using truncation, and neither using the floor operator nor according to the Euclidean definition. Notable is also the behavior for a division by zero: as in SMT-LIB's, ILR does not define the result of a division by zero, however $\forall x, y \left(x = y \to {}^{x}/_{0} = {}^{y}/_{0}\right)$ must be true. LLBMC's implementation currently returns -1 if $x < 0$ and the largest possible integer if x is positive. However it is intended to be configurable in the future.

For the definition of ILR's remainder function we first require the rem operator:

Definition 3.30 (rem). *Given two integers a and b, two integers q and r exist so that $a = qb + r \wedge 0 \leq r < |b|$. $r = a \operatorname{rem} b$ is called the remainder of a divided by b.*

The following axioms define ILR's arithmetic functions:

$$\forall x, y \left(\langle \mathrm{add}_{\mathcal{I}}(x, y)\rangle^u_{\mathcal{I}} \equiv (\langle x\rangle^u_{\mathcal{I}} + \langle y\rangle^u_{\mathcal{I}} \pmod{2^{|\mathcal{I}|}})\right) \tag{3.4a}$$

$$\forall x, y \left(\langle \mathrm{sub}_{\mathcal{I}}(x, y)\rangle^u_{\mathcal{I}} \equiv (\langle x\rangle^u_{\mathcal{I}} - \langle y\rangle^u_{\mathcal{I}} \pmod{2^{|\mathcal{I}|}})\right) \tag{3.4b}$$

$$\forall x, y \left(\langle \mathrm{mul}_{\mathcal{I}}(x, y)\rangle^u_{\mathcal{I}} \equiv (\langle x\rangle^u_{\mathcal{I}} \times \langle y\rangle^u_{\mathcal{I}} \pmod{2^{|\mathcal{I}|}})\right) \tag{3.4c}$$

$$\forall x, y \left(\langle \mathrm{div}^u_{\mathcal{I}}(x, y)\rangle^u_{\mathcal{I}} \equiv \mathrm{int}(\langle x\rangle^u_{\mathcal{I}} \,/\, \langle y\rangle^u_{\mathcal{I}}) \pmod{2^{|\mathcal{I}|}}\right) \tag{3.4d}$$

$$\forall x, y \left(\langle \mathrm{div}^s_{\mathcal{I}}(x, y)\rangle^s_{\mathcal{I}} \equiv \mathrm{int}(\langle x\rangle^s_{\mathcal{I}} \,/\, \langle y\rangle^s_{\mathcal{I}}) \pmod{2^{|\mathcal{I}|}}\right) \tag{3.4e}$$

$$\forall x, y \left(\langle \mathrm{rem}^u_{\mathcal{I}}(x, y)\rangle^u_{\mathcal{I}} = \langle x\rangle^u_{\mathcal{I}} \operatorname{rem} \langle y\rangle^u_{\mathcal{I}}\right) \tag{3.4f}$$

$$\forall x, y \left(\langle \mathrm{rem}^s_{\mathcal{I}}(x, y)\rangle^s_{\mathcal{I}} = \langle x\rangle^s_{\mathcal{I}} \operatorname{rem} \langle y\rangle^s_{\mathcal{I}}\right) \tag{3.4g}$$

Symbol : Signature	Interpretation
$\mathrm{shl}_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Shift left
$\mathrm{shr}^u_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Logical shift right
$\mathrm{shr}^s_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$	Arithmetic shift right

Table 3.17: Shift ILR functions

In contrast to C and LLVM-IR, where shift operations can cause undefined behavior for a number of different reasons, shift operations in ILR, like in SMT-LIB, are always

well-defined:

$$\forall x, y \left(\langle \mathrm{shl}_{\mathcal{I}}(x, y) \rangle^u_{\mathcal{I}} \equiv (\langle x \rangle^u_{\mathcal{I}} \times 2^{\langle y \rangle^u_{\mathcal{I}}}) \pmod{2^{|\mathcal{I}|}} \right) \tag{3.5a}$$

$$\forall x, y \left(\langle \mathrm{shr}^s_{\mathcal{I}}(x, y) \rangle^s_{\mathcal{I}} \equiv \mathrm{int}(\langle x \rangle^s_{\mathcal{I}} \,/\, 2^{\langle y \rangle^s_{\mathcal{I}}}) \pmod{2^{|\mathcal{I}|}} \right) \tag{3.5b}$$

$$\forall x, y \left(\langle \mathrm{shr}^u_{\mathcal{I}}(x, y) \rangle^u_{\mathcal{I}} \equiv (\langle x \rangle^u_{\mathcal{I}} \,/\, 2^{\langle y \rangle^u_{\mathcal{I}}}) \pmod{2^{|\mathcal{I}|}} \right) \tag{3.5c}$$

Symbol : Signature	Interpretation
$\mathrm{addo}^u_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Unsigned addition overflow
$\mathrm{addo}^s_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Signed addition overflow
$\mathrm{subo}^u_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Unsigned subtraction overflow
$\mathrm{subo}^s_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Signed subtraction overflow
$\mathrm{mulo}^u_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Unsigned multiplication overflow
$\mathrm{mulo}^s_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Signed multiplication overflow
$\mathrm{divo}^s_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Signed division overflow
$\mathrm{xdivz}_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Signed division overflow
$\mathrm{divx}^u_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Unsigned division exactness
$\mathrm{divx}^s_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Signed division exactness
$\mathrm{sho}_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Shift overflow
$\mathrm{shrx}^s_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Arithmetic shift right exactness
$\mathrm{shrx}^u_{\mathcal{I}} : \mathcal{I} \times \mathcal{I} \to \mathrm{bool}$	Logical shift right exactness

Table 3.18: Checking ILR functions

All functions introduced so far have direct counterparts in LLVM-IR. In contrast, the functions presented in table 3.18 are inspired by [Bru10] are exclusive to ILR and are dedicated to detecting undesired behavior in LLVM-IR programs. For example, $\mathrm{addo}^u_{\mathcal{I}}$ encodes whether an unsigned addition of integers of sort $\mathcal{I}$ overflows, and it is used together with $\mathrm{add}_{\mathcal{I}}$ to fully encode the semantics of an `add` instruction that has the `nuw` flag set. Other checking functions encode undefined behavior that is unrelated to flags, e.g. sho for detecting shifts by more than the bitwidth and divz for division by zero.

The axiomatization of most of these functions requires the definition of further auxiliary functions:

Definition 3.31 ($\mathrm{intmax}_{\mathcal{I}}$ and $\mathrm{intmin}_{\mathcal{I}}$). *$\mathrm{intmax}^u_{\mathcal{I}}$ is the largest unsigned integer of type $\mathcal{I}$, while $\mathrm{intmin}^u_{\mathcal{I}}$ is the smallest such integer. $\mathrm{intmax}^s_{\mathcal{I}}$ is the largest signed integer of type $\mathcal{I}$ and again $\mathrm{intmin}^s_{\mathcal{I}}$ is the smallest such integer.*

Given these definitions, the functions introduced in table 3.18 can now be defined

with the following axioms:

$$\forall x, y \left(\langle \text{addo}^u_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^u_{\mathcal{I}} + \langle y\rangle^u_{\mathcal{I}} > \text{intmax}^u_{\mathcal{I}}\right) \tag{3.6a}$$

$$\forall x, y \left(\langle \text{addo}^s_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^s_{\mathcal{I}} + \langle y\rangle^s_{\mathcal{I}} > \text{intmax}^s_{\mathcal{I}} \vee \langle x\rangle^s_{\mathcal{I}} + \langle y\rangle^s_{\mathcal{I}} < \text{intmin}^u_{\mathcal{I}}\right) \tag{3.6b}$$

$$\forall x, y \left(\langle \text{subo}^u_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^u_{\mathcal{I}} - \langle y\rangle^u_{\mathcal{I}} < \text{intmin}^u_{\mathcal{I}}\right) \tag{3.6c}$$

$$\forall x, y \left(\langle \text{subo}^s_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^s_{\mathcal{I}} - \langle y\rangle^s_{\mathcal{I}} < \text{intmin}^u_{\mathcal{I}} \vee \langle x\rangle^s_{\mathcal{I}} - \langle y\rangle^s_{\mathcal{I}} > \text{intmax}^s_{\mathcal{I}}\right) \tag{3.6d}$$

$$\forall x, y \left(\langle \text{mulo}^u_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^u_{\mathcal{I}}\langle y\rangle^u_{\mathcal{I}} < \text{intmax}^u_{\mathcal{I}}\right) \tag{3.6e}$$

$$\forall x, y \left(\langle \text{mulo}^s_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^s_{\mathcal{I}}\langle y\rangle^s_{\mathcal{I}} < \text{intmin}^u_{\mathcal{I}} \vee \langle x\rangle^s_{\mathcal{I}}\langle y\rangle^s_{\mathcal{I}} > \text{intmax}^s_{\mathcal{I}}\right) \tag{3.6f}$$

$$\forall x, y \left(\langle \text{divo}^s_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^s_{\mathcal{I}} = \text{intmax}^s_{\mathcal{I}} \wedge \langle y\rangle^s_{\mathcal{I}} = -1\right) \tag{3.6g}$$

$$\forall x, y \left(\langle \text{divz}_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle y\rangle^s_{\mathcal{I}} = 0\right) \tag{3.6h}$$

$$\forall x, y \left(\langle \text{divx}^u_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^u_{\mathcal{I}} \text{ rem } \langle y\rangle^u_{\mathcal{I}} = 0\right) \tag{3.6i}$$

$$\forall x, y \left(\langle \text{divx}^s_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^s_{\mathcal{I}} \text{ rem } \langle y\rangle^s_{\mathcal{I}} = 0\right) \tag{3.6j}$$

$$\forall x, y \left(\langle \text{sho}_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle y\rangle^u_{\mathcal{I}} \geq |\mathcal{I}|\right) \tag{3.6k}$$

$$\forall x, y \left(\langle \text{shrx}^s_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^s_{\mathcal{I}} \text{ rem } 2^{\langle y\rangle^s_{\mathcal{I}}} = 0\right) \tag{3.6l}$$

$$\forall x, y \left(\langle \text{shrx}^u_{\mathcal{I}}(x,y)\rangle \leftrightarrow \langle x\rangle^u_{\mathcal{I}} \text{ rem } 2^{\langle y\rangle^u_{\mathcal{I}}} = 0\right) \tag{3.6m}$$

Note that the above definitions are not suited for an actual implementation for detecting these overflows. Optimized approaches are presented by Brummayer [Bru10], Warren [War02], and Schultey et al. [Sch+00].

Symbol : Signature, with constraints	**Interpretation**
$\text{ext}^u_{\mathcal{I}_1,\mathcal{I}_2} : \mathcal{I}_1 \to \mathcal{I}_2$, with $\lvert\mathcal{I}_1\rvert > \lvert\mathcal{I}_2\rvert$	Integer zero extension
$\text{ext}^s_{\mathcal{I}_1,\mathcal{I}_2} : \mathcal{I}_1 \to \mathcal{I}_2$, with $\lvert\mathcal{I}_1\rvert > \lvert\mathcal{I}_2\rvert$	Integer signed extension
$\text{trunc}_{\mathcal{I}_1,\mathcal{I}_2} : \mathcal{I}_1 \to \mathcal{I}_2$, with $\lvert\mathcal{I}_1\rvert < \lvert\mathcal{I}_2\rvert$	Integer truncation
$\text{inttoptr}_{\mathcal{I},\mathcal{P}} : \mathcal{I} \to \mathcal{P}$	Integer to pointer conversion
$\text{ptrtoint}_{\mathcal{P},\mathcal{I}} : \mathcal{P} \to \mathcal{I}$	Pointer to integer conversion
$\text{bitcast}_{\mathcal{P}_1,\mathcal{P}_2} : \mathcal{P}_1 \to \mathcal{P}_2$, with $\lvert\mathcal{P}_1\rvert = \lvert\mathcal{P}_2\rvert$	Pointer to pointer conversion

Table 3.19: Conversion functions

Conversion operations (see table 3.19) are straightforward:

$$\forall x \left(\langle \text{ext}^u_{\mathcal{I}_1,\mathcal{I}_2}(x)\rangle^u_{\mathcal{I}_2} = \langle x\rangle^u_{\mathcal{I}_1}\right) \tag{3.7a}$$

$$\forall x \left(\langle \text{ext}^s_{\mathcal{I}_1,\mathcal{I}_2}(x)\rangle^s_{\mathcal{I}_2} = \langle x\rangle^s_{\mathcal{I}_1}\right) \tag{3.7b}$$

$$\forall x \left(\langle \text{trunc}_{\mathcal{I}_1,\mathcal{I}_2}(x)\rangle^u_{\mathcal{I}_2} \equiv \langle x\rangle^u_{\mathcal{I}_1} \pmod{2^{|\mathcal{I}_2|}}\right) \tag{3.7c}$$

$$\forall x \left(\langle \text{inttoptr}_{\mathcal{I},\mathcal{P}}(x)\rangle_{\mathcal{P}} \equiv \langle x\rangle^u_{\mathcal{I}} \pmod{2^{|\mathcal{P}|}}\right) \tag{3.7d}$$

$$\forall p \left(\langle \text{ptrtoint}_{\mathcal{P},\mathcal{I}}(p)\rangle^u_{\mathcal{I}} \equiv \langle p\rangle_{\mathcal{P}} \pmod{2^{|\mathcal{I}|}}\right) \tag{3.7e}$$

$$\forall p \left(\langle \text{bitcast}_{\mathcal{P}_1,\mathcal{P}_2}(p)\rangle_{\mathcal{P}_2} = \langle p\rangle_{\mathcal{P}_1}\right) \tag{3.7f}$$

The functions ext^u for zero bitvector extension, ext^s for signed bitvector extension, and trunc for bitvector truncation are defined as expected. The function $\mathrm{ptrtoint}$ reinterprets the pointer's bit pattern as an integer. If necessary, the value is truncated or zero extended. Its counterpart $\mathrm{inttoptr}$ operates in the same way. Finally, $\mathrm{bitcast}$ reinterprets a pointer as another type without changing the bit pattern. Note that this operation requires that both pointers have the same bitwidth.

Symbol : Signature	Interpretation
$\mathrm{eq}_{\mathcal{V}} : \mathcal{V} \times \mathcal{V} \to \mathrm{bool}$	Equality relation
$\mathrm{ne}_{\mathcal{V}} : \mathcal{V} \times \mathcal{V} \to \mathrm{bool}$	Inequality relation
$\mathrm{gt}^u_{\mathcal{V}} : \mathcal{V} \times \mathcal{V} \to \mathrm{bool}$	Unsigned-greater-than relation
$\mathrm{ge}^u_{\mathcal{V}} : \mathcal{V} \times \mathcal{V} \to \mathrm{bool}$	Unsigned-greater-or-equals relation
$\mathrm{lt}^u_{\mathcal{V}} : \mathcal{V} \times \mathcal{V} \to \mathrm{bool}$	Unsigned-less-than relation
$\mathrm{le}^u_{\mathcal{V}} : \mathcal{V} \times \mathcal{V} \to \mathrm{bool}$	Unsigned-less-or-equals relation
$\mathrm{gt}^s_{\mathcal{V}} : \mathcal{V} \times \mathcal{V} \to \mathrm{bool}$	Signed-greater-than relation
$\mathrm{ge}^s_{\mathcal{V}} : \mathcal{V} \times \mathcal{V} \to \mathrm{bool}$	Signed-greater-or-equals relation
$\mathrm{lt}^s_{\mathcal{V}} : \mathcal{V} \times \mathcal{V} \to \mathrm{bool}$	Signed-less-than relation
$\mathrm{le}^s_{\mathcal{V}} : \mathcal{V} \times \mathcal{V} \to \mathrm{bool}$	Signed-less-or-equals relation

Table 3.20: Comparison functions

The usual set of comparison operations is supported for integers and pointers. The functions are listed in table 3.20, with their semantics given by:

$$\forall x, y \left(\langle \mathrm{eq}_{\mathcal{V}}(x, y) \rangle \leftrightarrow \langle x \rangle^u_{\mathcal{V}} = \langle y \rangle^u_{\mathcal{V}} \right) \tag{3.8a}$$
$$\forall x, y \left(\langle \mathrm{ne}_{\mathcal{V}}(x, y) \rangle \leftrightarrow \langle x \rangle^u_{\mathcal{V}} \neq \langle y \rangle^u_{\mathcal{V}} \right) \tag{3.8b}$$
$$\forall x, y \left(\langle \mathrm{gt}^u_{\mathcal{V}}(x, y) \rangle \leftrightarrow \langle x \rangle^u_{\mathcal{V}} > \langle y \rangle^u_{\mathcal{V}} \right) \tag{3.8c}$$
$$\forall x, y \left(\langle \mathrm{ge}^u_{\mathcal{V}}(x, y) \rangle \leftrightarrow \langle x \rangle^u_{\mathcal{V}} \geq \langle y \rangle^u_{\mathcal{V}} \right) \tag{3.8d}$$
$$\forall x, y \left(\langle \mathrm{lt}^u_{\mathcal{V}}(x, y) \rangle \leftrightarrow \langle x \rangle^u_{\mathcal{V}} < \langle y \rangle^u_{\mathcal{V}} \right) \tag{3.8e}$$
$$\forall x, y \left(\langle \mathrm{le}^u_{\mathcal{V}}(x, y) \rangle \leftrightarrow \langle x \rangle^u_{\mathcal{V}} \leq \langle y \rangle^u_{\mathcal{V}} \right) \tag{3.8f}$$
$$\forall x, y \left(\langle \mathrm{gt}^s_{\mathcal{V}}(x, y) \rangle \leftrightarrow \langle x \rangle^s_{\mathcal{V}} > \langle y \rangle^s_{\mathcal{V}} \right) \tag{3.8g}$$
$$\forall x, y \left(\langle \mathrm{ge}^s_{\mathcal{V}}(x, y) \rangle \leftrightarrow \langle x \rangle^s_{\mathcal{V}} \geq \langle y \rangle^s_{\mathcal{V}} \right) \tag{3.8h}$$
$$\forall x, y \left(\langle \mathrm{lt}^s_{\mathcal{V}}(x, y) \rangle \leftrightarrow \langle x \rangle^s_{\mathcal{V}} < \langle y \rangle^s_{\mathcal{V}} \right) \tag{3.8i}$$
$$\forall x, y \left(\langle \mathrm{le}^s_{\mathcal{V}}(x, y) \rangle \leftrightarrow \langle x \rangle^s_{\mathcal{V}} \leq \langle y \rangle^s_{\mathcal{V}} \right) \tag{3.8j}$$

3.7.4 Miscellaneous

A set of miscellaneous functions, which do not belong to any group in particular, are listed in table 3.21.

As already mentioned, select is the primary reason why ILR has a dedicated boolean

Symbol : Signature, with constraints	**Interpretation**
$\text{select}_{\mathcal{U}} : \text{bool} \times \mathcal{U} \times \mathcal{U} \to \mathcal{U}$	Select
$\phi_{\mathcal{U}} : (\mathcal{U} \times \text{bool})^n \to \mathcal{U}$	Phi
$\text{concat}_{\mathcal{I}_1,\mathcal{I}_2} : \mathcal{I}_1 \times \mathcal{I}_2 \to \mathcal{I}$, with $\lvert\mathcal{I}\rvert = \lvert\mathcal{I}_1\rvert + \lvert\mathcal{I}_2\rvert$	Concatenation
$\text{extract}_{\mathcal{I}_1,i,j} : \mathcal{I}_1 \to \mathcal{I}_2$, with$\lvert I_1\rvert \geq \lvert I_2\rvert = j - i + 1$	Extraction

Table 3.21: Miscellaneous functions

sort. Its semantics are inspired by C's `?:`-operator:

$$\forall v_1, v_2 \left(b \to \text{select}_{\mathcal{V}}(T, v_1, v_2) = v_1\right) \tag{3.9a}$$
$$\forall v_1, v_2 \left(\neg b \to \text{select}_{\mathcal{V}}(F, v_1, v_2) = v_2\right) \tag{3.9b}$$

The ϕ function is the variadic sibling of the select function. It is inspired by LLVM's `phi` instruction, which is used to select the 'right' value, depending on which control flow edge was taken:

$$\forall u_1, b_1, \ldots, u_n, b_n \left(b_i \to \phi_{\mathcal{U}}(u_1, b_1, \ldots, u_n, b_n) = u_i\right) \tag{3.10a}$$
$$\forall u_1, b_1, \ldots, u_n, b_n \left(\forall b_i \left(\neg b_i\right) \to \phi_{\mathcal{U}}(u_1, b_1, \ldots, u_n, b_n) = u_n\right) \tag{3.10b}$$

Note that the formula is unsatisfiable if two b_i are true at the same time and the corresponding u_i cannot have the same value. The conditions are meant to be mutually exclusive and construction of ILR formulæ in LLBMC ensures this is true. While the ϕ function is shown as variadic here, it could equally well be defined as a set of functions with different arity.

Concatenation of bitvectors is based on SMT-LIB's bitvector function of the same name:

$$\forall i, x_1, x_2 \left(i < \lvert\mathcal{I}_1\rvert \to \text{concat}_{\mathcal{I}_1,\mathcal{I}_2}(x_1, x_2)[i] \equiv x_1[i]\right) \tag{3.11a}$$
$$\forall i, x_1, x_2 \left(i \geq \lvert\mathcal{I}_1\rvert \to \text{concat}_{\mathcal{I}_1,\mathcal{I}_2}(x_1, x_2)[i] \equiv x_2[i - \lvert\mathcal{I}_1\rvert]\right) \tag{3.11b}$$

The same holds for extraction of ranges of bits from an integer:

$$\forall i, k, j, x \left(i \leq k \leq j \to \text{extract}_{\mathcal{I},i,j}(x)[k - i] = x[k]\right) \tag{3.12}$$

E.g. $\text{extract}_{\text{i32},8,15}(x)$ extracts the second least-significant byte of x.

3.7.5 Memory

Memory related operations are considerably more complex than most other operations. This is to some part because low-level memory operations require various bits of knowledge about the target architecture, including data layout constraints (e.g. alignment) and the architecture's native bitwidth:[29]

[29] LLVM's data layout is introduced briefly in section 3.2

Definition 3.32 (Pointer bitwidth). *The function* $\mathrm{pointerwidth}^a$ *stands for the pointer width on architecture* a*.*

For example, for the x86 architecture, $\mathrm{pointerwidth}^{\text{x86}} = 32$. This definition leads immediately to the following extension of the definition of a bitwidth:

Definition 3.33 (Bitwidth). *The operator* $|\mathcal{N}|^a$ *stands for the width of sort* $\mathcal{N}$ *on architecture* a*.*

- *For all integer sorts* $\mathcal{I}$*,* $|\mathcal{I}|^a = |\mathcal{I}|$*,*
- *for all pointer sorts* $\mathcal{P}$*,* $|\mathcal{P}|^a = \mathrm{pointerwidth}^a$*,*
- *for all aggregate sorts* $\mathcal{A}$ *and* $\mathcal{S}$*,* $|\mathcal{A}|^a$ *and* $|\mathcal{S}|^a$ *are defined as the sum of the bitwidths of all elements.*

Note that this does not yet take padding or alignment into account, so the bitwidth can be seen as the minimum number of bits required to store a value.

Symbol : Signature	Interpretation
$\mathrm{load}^b_{\mathcal{V}} : \mathcal{M} \times \mathcal{V}* \to \mathcal{V}$	Big-endian load
$\mathrm{load}^l_{\mathcal{V}} : \mathcal{M} \times \mathcal{V}* \to \mathcal{V}$	Little-endian load
$\mathrm{store}^b_{\mathcal{V}} : \mathcal{M} \times \mathcal{V}* \times \mathcal{V} \to \mathcal{M}$	Big-endian store
$\mathrm{store}^l_{\mathcal{V}} : \mathcal{M} \times \mathcal{V}* \times \mathcal{V} \to \mathcal{M}$	Little-endian store

Table 3.22: Memory accessing ILR functions

ILR's memory accessing functions are listed in table 3.22. LLBMC treats endianness not as a property of the target architecture but as a property of each load or store operation. Because of this, two functions for load and store are defined each: load^b and store^b for big-endian memory accesses and load^l and store^l for little-endian memory accesses. This approach allows reasoning about big-endian, little-endian, mixed-endian[30], and bi-endian[31] architectures in a single formula. This is useful when comparing programs compiled for different architectures. We will use load instead of load^b or load^l if something holds for either endianness, e.g. when referring to loading and storing of a sort with bitwidth less than or equals eight.

As already mentioned above, only loading and storing of LLVM's simple value types (integers and pointers) are supported. Aggregate sorts are only ever used for deriving pointer sorts and for pointer arithmetics in the gep function introduced below.

While LLVM allows virtual registers containing integers of any bitwidth, a byte in memory is always fixed to contain 8 bits. This is relevant when accessing memory, e.g. because of endianness and when doing pointer arithmetics, e.g. due to alignment constraints. Single-byte loading and storing is inspired by McCarthy's theory of arrays (see section 2.1.4):

$$\forall m, p_1, p_2, v \left(p_1 = p_2 \to \mathrm{load}_{\mathrm{i8}}(\mathrm{store}_{\mathrm{i8}}(m, p_1, v), p_2) = v\right) \tag{3.13a}$$

$$\forall m, p_1, p_2, v \left(p_1 \neq p_2 \to \mathrm{load}_{\mathrm{i8}}(\mathrm{store}_{\mathrm{i8}}(m, p_1, v), p_2) = \mathrm{select}_{\mathrm{i8}}(m, p_2)\right) \tag{3.13b}$$

[30] Mixed-endian architectures have different endianness for different bitwidths.
[31] Bi-endian architectures can operate both in big-endian and little-endian mode.

Extensionality of arrays is defined as usual:

$$\forall m_1, m_2 \left(m_1 = m_2 \leftrightarrow \forall p \left(\mathrm{load}_{i8}(m_1, p) = \mathrm{load}_{i8}(m_2, p)\right)\right) \tag{3.14}$$

Like LLVM itself, LLBMC not only allows for loading and storing of single bytes but for integers and pointers with arbitrary bitwidths. The semantics of multi-byte loading and storing is mapped to that of multiple single-byte loading and storing. Note that in LLVM, a $\mathrm{load}_{\mathcal{I}}$ with $|\mathcal{I}| \not\equiv 0 \pmod 8$ is undefined if the memory at this location was not written using a store of the same type. Similarly, the values of the extra bytes for a `store` with a bitwidth smaller than eight are unspecified. Because LLVM-IR programs generated from C programs do not store values with a bitwidth that is not a multiple of eight we will spend the minimum amount of effort to handle these cases.

Semantics of constant sized, multi-byte reads can be derived from single byte reads by concatenating the result of a single-byte read and a second, complementary read:

$$\forall m, p \left(|\mathcal{I}| > 8 \rightarrow \mathrm{load}^b_{\mathcal{I}}(m, p) = \mathrm{concat}_{\mathsf{i8},\mathcal{I}}(\mathrm{load}_{\mathsf{i8}}(m, p), \mathrm{load}^b_{\mathcal{I}_1}(m, p+1))\right) \tag{3.15}$$

$$\forall m, p \left(|\mathcal{I}| > 8 \rightarrow \mathrm{load}^l_{\mathcal{I}}(m, p) = \mathrm{concat}_{\mathcal{I},\mathsf{i8}}(\mathrm{load}^l_{\mathcal{I}_1}(m, p+1), \mathrm{load}_{\mathsf{i8}}(m, p))\right) \tag{3.16}$$

Loads smaller than a single byte are realized by truncating the result of loading a whole byte:

$$\forall m, p \left(|\mathcal{I}| < 8 \rightarrow \mathrm{load}_{\mathcal{I}}(m, p) = \mathrm{trunc}_{\mathsf{i8},\mathcal{I}}(\mathrm{load}_{\mathsf{i8}}(m, p))\right) \tag{3.17}$$

Similarly, the effects of a constant sized, multi-byte store can be split into a single-byte store and a second, complementary store:

$$\forall m, p, x \left(\mathrm{store}^b_{\mathcal{I}}(m, p, x) = \mathrm{store}^b_{\mathcal{I}_2}(\,\mathrm{store}_{\mathsf{i8}}(m, p, \mathrm{extract}_{\mathcal{I},|\mathcal{I}|-8,|\mathcal{I}|-1}(x)), \tag{3.18}$$

$$p+1, \mathrm{extract}_{\mathcal{I},0,|\mathcal{I}|-9}(x))\right) \tag{3.19}$$

$$\forall m, p, x \left(\mathrm{store}^l_{\mathcal{I}}(m, p, x) = \mathrm{store}^l_{\mathcal{I}_2}(\,\mathrm{store}_{\mathsf{i8}}(m, p, \mathrm{extract}_{\mathcal{I},0,7}(x)), \tag{3.20}$$

$$p+1, \mathrm{extract}_{\mathcal{I},8,|\mathcal{I}|-1}(x))\right) \tag{3.21}$$

Storing of integers with a bitwidth smaller than eight is handled using zero extension:

$$\forall m, p, x \left(|\mathcal{I}| < 8 \rightarrow \mathrm{store}^b_{\mathcal{I}}(m, p, x) = \mathrm{store}^b_{\mathsf{i8}}(m, p, \mathrm{ext}^u_{\mathcal{I},\mathsf{i8}}(x))\right) \tag{3.22}$$

Loading and storing of pointers can be mapped to the loading and storing of integers of the same size:

$$\forall m, p_1, p_2 \left(|\mathcal{I}| = |\mathcal{P}|^a \rightarrow \mathrm{store}_{\mathcal{P}}(m, p_1, p_2) = \mathrm{store}_{\mathcal{I}}(m, p_1, \mathrm{ptrtoint}_{\mathcal{P},\mathcal{I}}(p_2))\right) \tag{3.23a}$$

$$\forall m, p_1, p_2 \left(|\mathcal{I}| = |\mathcal{P}|^a \rightarrow \mathrm{load}_{\mathcal{P}}(m, p_1, p_2) = \mathrm{inttoptr}_{\mathcal{I},\mathcal{P}}(\mathrm{load}_{\mathcal{I}}(m, p_1, p_2))\right) \tag{3.23b}$$

Pointer arithmetic is heavily dependent on the architecture's data layout rules.

In order to handle pointer arithmetic via LLVM's `getelementptr` for each native sort a constant symbol is required which represents the space in memory taken up by an array element of this sort. This is not necessarily equal to the size of the object itself, as this has to take alignment of the second element in the array into account. The numerical value of these constants can be retrieved from the LLVM libraries.

Definition 3.34 (Allocation bitwidth). *The function* $\mathrm{allocwidth}^a_{\mathcal{N}}$*, indicates the number of bits required to allocate sufficient space for an object of sort* $\mathcal{N}$ *on architecture* a *so that another such object can be placed after the first object with appropriate alignment.*

And similarly:

Definition 3.35 (Offset). *The function* $\mathrm{offset}^a_{\mathcal{S},n}$ *is the offset in bytes of* n^{th} *element in the structure sort* $\mathcal{S}$*.*

The $\mathrm{offset}^a_{\mathcal{S},n}$ function is the counterpart to GNU CC's macro `offsetof(s, f)`, which returns the offset of the field named `f` in the structure `s`.

Symbol : Signature	Interpretation
$\mathrm{add}_{\mathcal{P},\mathcal{I}} : \mathcal{P} \times \mathcal{I} \to \mathcal{P}$	Pointer addition
$\mathrm{sub}_{\mathcal{P},\mathcal{I}} : \mathcal{P} \times \mathcal{I} \to \mathcal{P}$	Pointer subtraction
$\mathrm{gep}^a_{\mathcal{P}_1,\mathcal{I}_1^n,\ldots,\mathcal{I}_n,\mathcal{P}_2} : \mathcal{P}_1 \times \mathcal{I}^n \to \mathcal{P}_2$	Pointer arithmetics

Table 3.23: Pointer arithmetic ILR functions

The functions $\mathrm{add}_{\mathcal{P},\mathcal{V}}$ and $\mathrm{sub}_{\mathcal{P},\mathcal{V}}$ are convenience functions for sort-correct pointer arithmetic operations. They convert a pointer to an integer, do integer arithmetics, and convert the resulting pointer back to the initial pointer sort.

$$\forall p, x \left(\mathrm{add}_{\mathcal{P},\mathcal{I}}(p,x) = \mathrm{inttoptr}_{\mathcal{I},\mathcal{P}}(\mathrm{add}_{\mathcal{I}}(\mathrm{ptrtoint}_{\mathcal{P},\mathcal{I}}(p), x))\right) \tag{3.24a}$$

$$\forall p, x \left(\mathrm{sub}_{\mathcal{P},\mathcal{I}}(p,x) = \mathrm{inttoptr}_{\mathcal{I},\mathcal{P}}(\mathrm{sub}_{\mathcal{I}}(\mathrm{ptrtoint}_{\mathcal{P},\mathcal{I}}(p), x))\right) \tag{3.24b}$$

gep is easily the most complex function in all of ILR. We will reason about gep as a variadic function, but in reality, for any program p, a largest number n can be found so that the variadic gep can be replaced by a sufficiently large series of gep functions with arities 1 to n.

The gep function takes as arguments a base pointer and a sequence of indices. A gep with an empty list of indices is a NOOP and returns the base pointer itself. For a gep with at least one index argument, the first one indexes the pointer itself. A gep with two or more index arguments is only valid, if the base pointer's sort is an aggregate sort. The second index operand indexes this array or structure sort.

Validity is extended analogously to gep functions with more index arguments.

$$\mathrm{gep}^a_{\mathcal{P}_1,\mathcal{P}_1}(p) = p \tag{3.25a}$$

$$\mathrm{gep}^a_{\mathcal{N}*,\mathcal{I},\mathcal{P}_2}(p,i) = \mathrm{add}_{\mathcal{N}*,\mathcal{I}}(p, \mathrm{mul}_{\mathcal{I}}(i, (\mathrm{allocwidth}^a_{\mathcal{N}})_{\mathcal{I}})) \tag{3.25b}$$

$$\begin{aligned}\mathrm{gep}^a_{\mathcal{A}*,\mathcal{I},\mathcal{I},\mathcal{N}*}(p,i_1,i_2) = \mathrm{add}_{\mathcal{N}*,\mathcal{I}}(&\mathrm{bitcast}_{\mathcal{A}*,\mathcal{N}*}(\\ &\mathrm{add}_{\mathcal{A}*,\mathcal{I}}(p, \mathrm{mul}_{\mathcal{I}}(i_1, (\mathrm{allocwidth}^a_{\mathcal{A}})_{\mathcal{I}}))),\\ &\mathrm{mul}_{\mathcal{I}}(i_2, (\mathrm{allocwidth}^a_{\mathcal{N}})_{\mathcal{I}}))\end{aligned} \tag{3.25c}$$

$$\begin{aligned}\mathrm{gep}^a_{\mathcal{S}*,\mathcal{I},\mathcal{I},\mathbb{N}*}(p,i_1,i_2) = \mathrm{add}_{\mathcal{N}*,\mathcal{I}}(&\mathrm{bitcast}_{\mathcal{S}*,\mathcal{N}*}(\\ &\mathrm{add}_{\mathcal{S}*,\mathcal{I}}(p, \mathrm{mul}_{\mathcal{I}}(i_1, (\mathrm{allocwidth}^a_{\mathcal{S}})_{\mathcal{I}}))),\\ &(\mathrm{offset}^a_{\mathcal{S},i_2})_{\mathcal{I}})\end{aligned} \tag{3.25d}$$

Just like a large `getelementptr` is split into multiple smaller ones in section 3.4, this can also be done with its ILR counterpart gep:

$$\begin{aligned}&\forall p, i_1, \ldots, i_n \left(\mathrm{gep}^a_{\mathcal{P},\mathcal{I}_1,\mathcal{I}_2,\mathcal{I}_3,\ldots,\mathcal{I}_n,\mathcal{P}_2}(p, i_1, i_2, i_3, \ldots, i_n) = \right.\\ &\left.\mathrm{gep}^a_{\mathcal{P}_1,\mathcal{I},\mathcal{I}_3,\ldots,\mathcal{I}_n,\mathcal{P}_2}(\mathrm{gep}^a_{\mathcal{P},\mathcal{I}_1,\mathcal{I}_2,\mathcal{P}_1}(p, i_1, i_2), (0)_{\mathcal{I}}, i_3, \ldots, i_n)\right)\end{aligned} \tag{3.26}$$

3.7.6 Instantiating ILR

As previously mentioned, ILR is not a language but a language schema. We will call ILR_x the x instantiation of ILR. To retrieve such an instantiation, we select sufficient instantiations of the sorts listed in table 3.10. Furthermore, we instantiate functions from tables 3.13 to 3.23, thereby restricting ourselves to those instantiations that use only the previously chosen sorts. The selected sorts and functions provide for ILR_x's signature. The same is then done for all necessary language extensions, e.g. from chapter 5 or chapter 4. The variadic functions ϕ and gep are then replaced each by a finite set of derived ϕ and gep functions with fixed arity. The set of instantiations of the axiom schemata related to these sorts and functions then make up the first-order logic theory ILR_x.

3.7.7 Sharing of ILR Terms

Due to LLBMC's extremely deep nesting of terms, term sharing is of major importance for LLBMC's memory consumption. *Term sharing* means that terms that are syntactically equal are represented by the same object in memory. LLBMC also uses this for performance improvements: Due to term sharing, structural equality can often be replaced by identity checks.

To reflect this, ILR terms can also be represented as sets of definitorial statements $l := r$, where r is a term and l is a newly defined alias for r. e.g. $i_2 := \mathrm{add}_{\mathrm{i32}}(i_0, i_1)$ indicates that i_2 is shorthand for $\mathrm{add}_{\mathrm{i32}}(i_0, i_1)$.

For example, given the set of terms

$$\begin{aligned} i_0 &\coloneqq (16)_{\text{i32}} \\ i_1 &\coloneqq x \\ i_2 &\coloneqq \text{add}_{\text{i32}}(i_0, i_1) \\ b_0 &\coloneqq \text{eq}_{\text{i32}}(i_0, i_2) \end{aligned}$$

the last line b_0 is the definitorial form of

$$\text{eq}_{\text{i32}}((16)_{\text{i32}}, \text{add}_{\text{i32}}((16)_{\text{i32}}, x)).$$

3.8 Summary and Outlook

LLBMC is a software bounded model checker based on the LLVM compiler framework and off-the-shelf SMT solvers. LLBMC's primary input language is LLVM's intermediate representation, but by using a minimally modified code generator it can also instrument LLVM-IR code with annotations for verification of runtime errors in C. LLBMC uses compiler optimizations to improve performance, in particular LLVM's stack promotion (`mem2reg`). LLBMC uses a slightly modified variant of LLVM's built-in loop unrolling but does not do function inlining on the LLVM-level. Instead, a call graph is maintained as a separate data structure. Finally, LLBMC has its own intermediate logic representation ILR which is closely related to LLVM-IR and fully axiomatized.

While LLBMC's implementation supports an even larger part of LLVM-IR than shown in this chapter, a few areas still remain in which support is currently lacking. Most notable are the lack of support for floating point operations, exception handling and runtime type information (RTTI). Furthermore, inline assembly is used in many embedded software development projects and support for it would be a welcome addition to LLBMC. Finally, in LLBMC function inlining and loop unrolling, the two core concepts in software bounded model checking, are handled in markedly different ways. A unified formalism based on combined call and control flow graphs would make the approach more flexible for future research.

Chapter 4

Encoding LLVM-IR in ILR

Encoding, in the context of LLBMC, is the process of creating an ILR formula that describes a bounded fragment of an LLVM-IR program's semantics and a set of safety properties for this program. This chapter provides an in-depth description of LLBMC's encoding.

There is a notable lack of detailed information about the encoding used in software bounded model checkers. CBMC's encoding is described by an example in [CKL04] and has changed considerably since then. ESBMC is based on CBMC with the encoding adapted for improved support of C++ (see [CFM09]). LAV's encoding ([VK12]) differs in that it first checks an instruction by itself, then in its containing basic block and finally in the function context. How this affects the structure of the final formula is not easily apparent. FAuST's approach to encoding is demonstrated using a toy example in [RF12]. None of the tools provide an in-depth look at the encoding. Lal and Qadeer [LQ15] present a markedly different approach for encoding function calls. Instead of the tree-like calling structure in CBMC or the explicit call tree in LLBMC, this approach works on a directed acyclic call graph, which shares nodes whenever possible. According to the author, this improves scalability of the tool considerably. Again, little is published about the tool's encoding beyond this. However, experience with LLBMC shows that minor details in the encoding can have considerable impact on the tool's performance and are therefore not to be underestimated. In some cases even minor details such as the order of the nesting of select terms can make the difference between solving a formula and getting a timeout.

4.1 Sorts, Functions, and Instruction Patterns

As described in section 3.7, ILR is an extensible language. The first such extension concerned with the encoding of LLVM-IR in ILR is presented in this chapter. The extension defines additional sorts for the LLVM-IR object types instructions, basic blocks, constants, functions, function arguments, and globals variables. Furthermore, the language extension also contains the sort `C` that represents contexts, which

are nodes in a call-site-sensitive call graph (see section 3.6). The sorts related to encoding are listed in table 4.1.

LLVM-IR Object Type	Sort Symbol
Basic blocks	B
Execution context	C
Functions	F
Function arguments	A
Global variables	G
Instructions	I
Integer constants	N
Programs	P

Table 4.1: Encoding related sorts

This chapter also introduces a set of function symbols, which are grouped in families of related symbols. Each symbol family encodes a different aspect about a program, e.g. the symbol family μ represents the memory state during execution of a program. Table 4.2 provides an overview of these families. The symbols for function families are decorated with the sort symbols shown in table 4.1 for the different members or subgroups in a family, e.g. $\varepsilon^{\mathtt{N}}$ indicates the evaluation of a constant and $\sigma^{\mathtt{I}}$ indicates the safety of an instruction. Furthermore, whenever a distinction between the state before or after execution of a certain LLVM-IR language object is needed, a decoration with a harpoon facing left ($\leftharpoonup$) is used to indicate the state before the object's execution (e.g. $\overleftharpoon{\eta}^{\mathtt{I}}$ for the execution condition before execution of an instruction), while it is decorated with a harpoon pointing right ($\rightharpoonup$) to indicate the state after its execution (e.g. $\overrightharpoon{\mu}^{\mathtt{B}}$ for the memory state after execution of a basic block).

Function Family	Symbol
Evaluation of values	ε
Memory state	μ
Execution condition	η
Instruction safety	σ
Stack state	τ

Table 4.2: Encoding related function families

The functions in family ε represent evaluation of values, such as instructions, constants or function arguments, e.g. the evaluation of an instruction i in a context c, is represented by $\varepsilon_t^{\mathtt{I}}(c, i)$. The function's sort is indicated by the subscript t, e.g. $\varepsilon_{\mathtt{i32}}^{\mathtt{I}}(c, i)$ has sort i32. All functions related to the evaluation of LLVM-IR values are listed in table 4.3.

Control flow is represent in LLBMC via so-called execution conditions (η). An

Symbol : Signature	Interpretation
$\varepsilon_t^{\mathtt{A}} : \mathtt{C} \times \mathtt{A} \to t$	Evaluation of an argument
$\varepsilon_t^{\mathtt{N}} : \mathtt{N} \to t$	Evaluation of a constant
$\varepsilon_t^{\mathtt{I}} : \mathtt{C} \times \mathtt{I} \to t$	Evaluation of an instruction

Table 4.3: Functions encoding the evaluation of values

execution condition collects all control flow decisions that happened so far during execution of the program in a term of sort bool. This includes information about which branches were taken and which functions were called. For example, $\overrightarrow{\eta}^{\mathtt{B}}(c, b)$ encodes the execution condition after execution of the basic block b in the context c, while $\overleftarrow{\eta}^{\mathtt{I}}(c, i)$ encodes the execution condition before execution of the instruction i in the context c. Most of the η-functions follow the same before/after pattern except for $\eta^{\mathtt{J}}(c, b_0, b_1)$, which encodes the execution condition of jumping from basic block b_0 to basic block b_1 in context c. All function symbols related to execution conditions are presented in table 4.4.

Symbol : Signature	Interpretation
$\overleftarrow{\eta}^{\mathtt{B}} : \mathtt{C} \times \mathtt{B} \to$ bool	Execution condition before a basic block
$\overrightarrow{\eta}^{\mathtt{B}} : \mathtt{C} \times \mathtt{B} \to$ bool	Execution condition after a basic block
$\overleftarrow{\eta}^{\mathtt{F}} : \mathtt{C} \times \mathtt{F} \to$ bool	Execution condition before a function
$\overrightarrow{\eta}^{\mathtt{F}} : \mathtt{C} \times \mathtt{F} \to$ bool	Execution condition after a function
$\overleftarrow{\eta}^{\mathtt{I}} : \mathtt{C} \times \mathtt{I} \to$ bool	Execution condition before an instruction
$\overrightarrow{\eta}^{\mathtt{I}} : \mathtt{C} \times \mathtt{I} \to$ bool	Execution condition after an instruction
$\overleftarrow{\eta}^{\mathtt{P}} : \mathtt{C} \times \mathtt{P} \to$ bool	Execution condition before a program
$\overrightarrow{\eta}^{\mathtt{P}} : \mathtt{C} \times \mathtt{P} \to$ bool	Execution condition after a program
$\eta^{\mathtt{J}} : \mathtt{C} \times \mathtt{B} \times \mathtt{B} \to$ bool	Execution at a control flow edge

Table 4.4: Functions encoding execution conditions

The family μ encodes memory states, or more precisely, the memory content. For example, $\overleftarrow{\mu}^{\mathtt{B}}(c, b)$ encodes the memory's contents before execution of basic block b in context c. The encoding of memory states follows the same pattern as that of execution conditions, with the notable exception that memory requires an additional subscript t which encodes the type of the memory, e.g. $\overleftarrow{\mu}^{\mathtt{B}}_{\mathsf{m32}}$ encodes a memory state for a 32-bit system. All function symbols related to the encoding of memory state are listed in table 4.5.

Stack related function symbols are listed in table 4.6. Note that we don't need $\overrightarrow{\tau}^{\mathtt{F}}$ and $\overrightarrow{\tau}^{\mathtt{P}}$. $\overrightarrow{\tau}^{\mathtt{P}}$ is always empty, as is $\overrightarrow{\tau}^{\mathtt{F}}$ for the main function. Due to stack unwinding, for any called function $\overrightarrow{\tau}^{\mathtt{F}}$ is the same as $\overleftarrow{\tau}^{\mathtt{I}}$ of the calling function.

An instruction can be unsafe, e.g. when it cause undefined behavior on the LLVM-IR level or C language level (see section 3.3), or when it represents the violation of a functional property. The function $\sigma^{\mathtt{I}}$ represents an instruction's safety, $\sigma^{\mathtt{F}}$ encodes

Symbol : Signature	Interpretation
$\overleftarrow{\mu}_t^{\mathtt{B}} : \mathtt{C} \times \mathtt{B} \to t$	Memory state before a basic block
$\overrightarrow{\mu}_t^{\mathtt{B}} : \mathtt{C} \times \mathtt{B} \to t$	Memory state after a basic block
$\overleftarrow{\mu}_t^{\mathtt{F}} : \mathtt{C} \times \mathtt{F} \to t$	Memory state before a function
$\overrightarrow{\mu}_t^{\mathtt{F}} : \mathtt{C} \times \mathtt{F} \to t$	Memory state after a function
$\overleftarrow{\mu}_t^{\mathtt{I}} : \mathtt{C} \times \mathtt{I} \to t$	Memory state before an instruction
$\overrightarrow{\mu}_t^{\mathtt{I}} : \mathtt{C} \times \mathtt{I} \to t$	Memory state after an instruction
$\overleftarrow{\mu}_t^{\mathtt{P}} : \mathtt{C} \times \mathtt{P} \to t$	Memory state before a program
$\overrightarrow{\mu}_t^{\mathtt{P}} : \mathtt{C} \times \mathtt{P} \to t$	Memory state after a program

Table 4.5: Functions encoding memory state

Symbol : Signature	Interpretation
$\overleftarrow{\tau}_t^{\mathtt{B}} : \mathtt{C} \times \mathtt{B} \to t$	Stack state before a basic block
$\overrightarrow{\tau}_t^{\mathtt{B}} : \mathtt{C} \times \mathtt{B} \to t$	Stack state after a basic block
$\overleftarrow{\tau}_t^{\mathtt{F}} : \mathtt{C} \times \mathtt{F} \to t$	Stack state before a function
$\overleftarrow{\tau}_t^{\mathtt{I}} : \mathtt{C} \times \mathtt{I} \to t$	Stack state before an instruction
$\overrightarrow{\tau}_t^{\mathtt{I}} : \mathtt{C} \times \mathtt{I} \to t$	Stack state after an instruction
$\overleftarrow{\tau}_t^{\mathtt{P}} : \mathtt{C} \times \mathtt{P} \to t$	Stack state before a program

Table 4.6: Functions encoding stack state

a function's safety, and $\sigma^{\mathtt{P}}$ a program's safety.

Symbol : Signature	Interpretation
$\sigma^{\mathtt{I}} : \mathtt{C} \times \mathtt{I} \to \text{bool}$	Safety of an instruction
$\sigma^{\mathtt{F}} : \mathtt{C} \times \mathtt{F} \to \text{bool}$	Safety of a function
$\sigma^{\mathtt{P}} : \mathtt{C} \times \mathtt{P} \to \text{bool}$	Safety of a program

Table 4.7: Function encoding the safety of instructions

In LLVM, the language objects, e.g. instructions, basic blocks, are all derived from the same type, the value. In many places in LLVM-IR more than one type of value can be used, e.g. instructions can have both constants and other instructions as operands. To reflect this in the encoding, the function symbol $\varepsilon_t^{\mathtt{V}}(c, v)$ is introduced.

LLBMC's encoding is formalized as a term rewriting system. The following set of rules describe how the evaluation of arbitrary LLVM-IR values is concretized for specific value subclasses, e.g. instructions or basic block:

$$\varepsilon_t^{\mathtt{V}}(c, v) \longrightarrow \varepsilon_t^{\mathtt{I}}(c, v)\ ;\ \text{v is an instruction} \tag{4.1a}$$
$$\varepsilon_t^{\mathtt{V}}(c, v) \longrightarrow \varepsilon_t^{\mathtt{A}}(c, v)\ ;\ \text{v is a function argument} \tag{4.1b}$$
$$\varepsilon_t^{\mathtt{V}}(c, v) \longrightarrow \varepsilon_t^{\mathtt{N}}(v)\ ;\ \text{v is a constant} \tag{4.1c}$$
$$\varepsilon_t^{\mathtt{V}}(c, v) \longrightarrow \varepsilon_t^{\mathtt{F}}(v)\ ;\ \text{v is a function} \tag{4.1d}$$
$$\varepsilon_t^{\mathtt{V}}(c, v) \longrightarrow \varepsilon_t^{\mathtt{B}}(v)\ ;\ \text{v is a basic block} \tag{4.1e}$$

Note that because regular many-sorted logic does not have polymorphism, these rules are, strictly speaking, syntactically invalid. This is because in many-sorted logic, the types for instructions and values are not related, so if $\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, v)$, which takes a value as its first argument, is a well-formed term, $\varepsilon^{\mathtt{I}}(c, v)$, which takes an instruction as its first argument, cannot be well-formed. An equivalent but valid approach, though it is slightly more complex, manages without the rewrite rules above. Instead, all other rules of the term rewriting system are interpreted not as individual rules but as rule schemata. The symbol $\varepsilon^{\mathtt{V}}$ in such a schema acts as a placeholder. Concrete rules are then derived by replacing all of these placeholders by one of the other ε-based function symbols.

4.1.1 Pattern Matching

Most rewrite rule schemata introduced in this chapter are conditional. Their conditions can be evaluated when the schemata are instantiated for a specific program. The rewrite rules resulting from this instantiation are therefore unconditional. Many of these conditions make use of pattern matching, which is based on instruction patterns, as defined in definition 3.1. In the rewrite rules described below, the pattern matching symbol $[\![\cdot]\!]$ is also used outside of patterns for the capturing of values occurring in the patterns[1]. For example, if

$$s \sim [\![\texttt{<r> = add [nuw] [nsw] <t> <op1> <op2>}]\!]$$

[1] This is inspired by the concept of capturing as it is used frequently in regular expressions.

Sort Family	Variables
Basic blocks	b, b_1, ...
Context	c, c_f, c_1, c_2
Functions	f
Function arguments	a, a_1, a_2, ...
Global variables	g
Instructions	i, i_1, ...
Integer constants	n
Program	p

Table 4.8: Variable naming conventions

occurs in a rewrite rule schema's condition and evaluates to true, then ⟦op1⟧ and ⟦op2⟧ can be used in the rewrite rule to refer to the instruction's first and second operand respectively. The concept of capturing also applies to types. Though in this case LLVM-IR types are also implicitly mapped to the matching ILR, i.e. in the above example, ⟦t⟧ captures the LLVM-IR type of the instruction and maps it to the corresponding ILR sort. Because ILR's type system is modeled closely after LLVM-IR's type system, this mapping is trivial.

An exemplary rewrite rule schema for the integer addition instruction `add` is encoded as follows:

$$\begin{aligned} \varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c, i) \longrightarrow \mathrm{add}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v0}]\!]), \varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v1}]\!]))\,; \\ i \sim [\![\texttt{<i> = add [nuw] [nsw] <t> <v0>, <v1>}]\!] \end{aligned} \tag{4.2}$$

This can be read as: if i is an `add` instruction, the instruction's result rewrites to the function add, with the function's first argument matching the instruction's first argument and the function's second argument matching the instruction's second argument. Here, `<t>` is a placeholder for an LLVM-IR type, `<v0>` and `<v1>` for values. The flags `nuw` and `nsw` are optional, so this rule matches independently of their presence.[2]

```
define i32 @f(i32 %x, i32 %y) {
entry:
  %0 = add i32 %x, %y
  %1 = zext i32 %x to i64
  %2 = add i64 %1, 42
  ret i32 %0
}
```

Listing 4.1: Example code for instruction pattern instantiation

The rewrite rule schema shown in equation (4.2) instantiated for the LLVM-IR

[2] Flags related to undefined behavior are encoded separately in section 4.5.

program in listing 4.1 with

$$
\begin{aligned}
a_1 &=; \texttt{\%x},\\
a_2 &=; \texttt{\%y},\\
i_0 &=; \texttt{\%0 = add i32 \%x, \%y},\\
i_1 &=; \texttt{\%1 = zext i32 \%x to i64}, \text{ and}\\
i_2 &=; \texttt{\%2 = add i64 \%1, 42}
\end{aligned}
$$

produces, among others, the following concrete rewrite rules:

$$
\begin{aligned}
\varepsilon^{\mathtt{I}}_{\mathsf{i32}}(c, i_0) &\longrightarrow \mathrm{add}_{\mathsf{i32}}(\varepsilon^{\mathtt{A}}_{\mathsf{i32}}(c, a_1), \varepsilon^{\mathtt{A}}_{\mathsf{i32}}(c, a_2))\\
\varepsilon^{\mathtt{I}}_{\mathsf{i64}}(c, i_2) &\longrightarrow \mathrm{add}_{\mathsf{i64}}(\varepsilon^{\mathtt{I}}_{\mathsf{i64}}(c, i_1), \varepsilon^{\mathtt{N}}_{\mathsf{i64}}(42))
\end{aligned}
$$

Furthermore, for a pattern with a sequence (indicated by `{` and `}*`, see section 3.2.2) in it such as

$$s \sim [\![\texttt{call <ty> <fptr>(\{<ty> <arg>\}*)}]\!]$$

`<ty>` matches to a sequence of sorts and `<arg>` matches to a sequence of terms and the i$^{\text{th}}$ sort in the sequence can be accessed using $[\![\texttt{ty}]\!]_i$, while the i$^{\text{th}}$ term can be accesses using $[\![\texttt{arg}]\!]_i$.

4.2 Symbolic Evaluation

Symbolic evaluation refers to the encoding of values in LLVM-IR as terms in ILR[3]. This applies to instructions, function arguments, constant integers, and constant pointers such as global variables.

4.2.1 Instructions

The term $\varepsilon^{\mathtt{I}}_{t}(c, i)$ represents the result of evaluating the instruction i in the context c. Because ILR is closely modeled after LLVM-IR, the encoding of individual instructions is mostly straightforward. All integer arithmetic operations are handled mostly the same way as the introductory example above:

$$
\begin{aligned}
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c, i) &\longrightarrow \mathrm{add}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v1}]\!]), \varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v2}]\!]))\ ;\\
&\quad\ i \sim [\![\texttt{<i> = add [nuw] [nsw] <t> <v1>, <v2>}]\!] \qquad (4.3a)\\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c, i) &\longrightarrow \mathrm{sub}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v1}]\!]), \varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v2}]\!]))\ ;\\
&\quad\ i \sim [\![\texttt{<i> = sub [nuw] [nsw] <t> <v1>, <v2>}]\!] \qquad (4.3b)\\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c, i) &\longrightarrow \mathrm{mul}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v1}]\!]), \varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v2}]\!]))\ ;\\
&\quad\ i \sim [\![\texttt{<i> = mul [nuw] [nsw] <t> <v1>, <v2>}]\!] \qquad (4.3c)\\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c, i) &\longrightarrow \mathrm{div}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v1}]\!]), \varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v2}]\!]))\ ;\\
&\quad\ i \sim [\![\texttt{<i> = udiv [exact] <t> <v1>, <v2>}]\!] \qquad (4.3d)
\end{aligned}
$$

[3] This is in contrast to symbolic execution, which is additionally concerned with control flow.

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow \mathrm{div}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \quad i \sim [\![\texttt{<i> = sdiv [exact] <t> <v1>, <v2>}]\!] \tag{4.3e}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow \mathrm{rem}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \quad i \sim [\![\texttt{<i> = urem <t> <v1>, <v2>}]\!] \tag{4.3f}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow \mathrm{rem}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \quad i \sim [\![\texttt{<i> = srem <t> <v1>, <v2>}]\!] \tag{4.3g}$$

Similarly to the arithmetic instructions, LLVM-IR's shift and bitwise instructions are converted straightforwardly. Again, shift operations can have `nuw`, `nsw`, and `exact` flags, which are ignored for now:

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow \mathrm{shl}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \quad i \sim [\![\texttt{<i> = shl [nuw] [nsw] <t> <v1>, <v2>}]\!] \tag{4.4a}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow \mathrm{shr}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \quad i \sim [\![\texttt{<i> = ashr [exact] <t> <v1>, <v2>}]\!] \tag{4.4b}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow \mathrm{shr}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \quad i \sim [\![\texttt{<i> = lshr [exact] <t> <v1>, <v2>}]\!] \tag{4.4c}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow \mathrm{and}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \quad i \sim [\![\texttt{<i> = and <t> <v1>, <v2>}]\!] \tag{4.4d}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow \mathrm{or}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \quad i \sim [\![\texttt{<i> = or <t> <v1>, <v2>}]\!] \tag{4.4e}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow \mathrm{xor}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \quad i \sim [\![\texttt{<i> = xor <t> <v1>, <v2>}]\!] \tag{4.4f}$$

Type conversion is again mapped one-to-one to the ILR counterparts:

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t2}]\!]}(c,i) \longrightarrow \mathrm{trunc}_{[\![\mathtt{t1}]\!],[\![\mathtt{t2}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t1}]\!]}(c,[\![\mathtt{v}]\!]))\,; \quad i \sim [\![\texttt{<i> = trunc <t1> <v> to <t2>}]\!] \tag{4.5a}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t2}]\!]}(c,i) \longrightarrow \mathrm{ext}^{u}_{[\![\mathtt{t1}]\!],[\![\mathtt{t2}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t1}]\!]}(c,[\![\mathtt{v}]\!]))\,; \quad i \sim [\![\texttt{<i> = zext <t1> <v> to <t2>}]\!] \tag{4.5b}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t2}]\!]}(c,i) \longrightarrow \mathrm{ext}^{s}_{[\![\mathtt{t1}]\!],[\![\mathtt{t2}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t1}]\!]}(c,[\![\mathtt{v}]\!]))\,; \quad i \sim [\![\texttt{<i> = sext <t1> <v> to <t2>}]\!] \tag{4.5c}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t2}]\!]}(c,i) \longrightarrow \mathrm{ptrtoint}_{[\![\mathtt{t1}]\!],[\![\mathtt{t2}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t1}]\!]}(c,[\![\mathtt{v}]\!]))\,; \quad i \sim [\![\texttt{<i> = ptrtoint <t1> <v> to <t2>}]\!] \tag{4.5d}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t2}]\!]}(c,i) \longrightarrow \mathrm{inttoptr}_{[\![\mathtt{t1}]\!],[\![\mathtt{t2}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t1}]\!]}(c,[\![\mathtt{v}]\!]))\,; \quad i \sim [\![\texttt{<i> = inttoptr <t1> <v> to <t2>}]\!] \tag{4.5e}$$

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t2}]\!]}(c,i) \longrightarrow \mathrm{bitcast}_{[\![\mathtt{t1}]\!],[\![\mathtt{t2}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t1}]\!]}(c,[\![\mathtt{v}]\!]))\,; \quad i \sim [\![\texttt{<i> = bitcast <t1> <v> to <t2>}]\!] \tag{4.5f}$$

LLVM-IR provides a single instruction for ten different kinds of integer comparisons using an additional flag to decide which comparison to execute. ILR differs in this respect in that it has separate functions for each of these:

$$\begin{aligned}
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{eq}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = icmp eq <t> <v1>, <v2>}]\!] && \text{(4.6a)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{ne}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = icmp ne <t> <v1>, <v2>}]\!] && \text{(4.6b)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{gt}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = icmp ugt <t> <v1>, <v2>}]\!] && \text{(4.6c)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{ge}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = icmp uge <t> <v1>, <v2>}]\!] && \text{(4.6d)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{lt}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = icmp ult <t> <v1>, <v2>}]\!] && \text{(4.6e)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{le}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = icmp ule <t> <v1>, <v2>}]\!] && \text{(4.6f)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{gt}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = icmp sgt <t> <v1>, <v2>}]\!] && \text{(4.6g)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{ge}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = icmp sge <t> <v1>, <v2>}]\!] && \text{(4.6h)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{lt}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = icmp slt <t> <v1>, <v2>}]\!] && \text{(4.6i)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{le}^{s}_{[\![\mathtt{t}]\!]}\,\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = icmp sle <t> <v1>, <v2>}]\!] && \text{(4.6j)}
\end{aligned}$$

The rewrite rule for `select` instructions shown in equation (4.7a) is mostly straightforward. The `phi` instruction is encoded similarly again, though this instruction is variadic and therefore its arguments need to be matched as a sequence. A `getelementr` is encoded similarly to a `phi` as it is variadic, too. Note, thought, that gep is specific to the current program's architecture a.

$$\begin{aligned}
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{select}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v3}]\!]))\,; \\
&\quad i \sim [\![\texttt{<i> = select i1 <v1>, <t> <v2>, <t> <v3>}]\!] && \text{(4.7a)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \phi_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v}]\!]_1),\vec{\eta}^{\mathtt{B}}(c,[\![\mathtt{l}]\!]_1),\ldots,\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v}]\!]_n),\vec{\eta}^{\mathtt{B}}(c,[\![\mathtt{l}]\!]_n))\,; \\
&\quad i \sim [\![\texttt{<i> = phi <t> \{[<v>, <l>]\}+}]\!] && \text{(4.7b)} \\
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) &\longrightarrow \mathrm{gep}^{a}_{[\![\mathtt{t}]\!],[\![\mathtt{it}]\!]_1,\ldots,[\![\mathtt{it}]\!]_n}([\![\mathtt{p}]\!],[\![\mathtt{iv}]\!]_1,\ldots,[\![\mathtt{iv}]\!]_n)\,; \\
&\quad i \sim [\![\texttt{<i> = getelementptr <t>, <t>* <p>[, {<it> <iv>}+]}]\!] && \text{(4.7c)}
\end{aligned}$$

$$\begin{aligned}\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow\ & \phi_{[\![\mathtt{t}]\!]}(a_1,b_1,\ldots,a_{|R|},b_{|R|})\,; \\ & c = \mathrm{parent}(c_f) \wedge i = \mathrm{callsite}(c_f) \wedge \\ & i \sim [\![\texttt{<i> = call <t> <fptr>(\{<ty> <arg>\}*)}]\!] \wedge \\ & R = \{(a_i,b_i) : a_i = \varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c_f,[\![\mathtt{v}]\!]) \wedge b_i = \vec{\eta}^{\mathtt{I}}(c_f,j) \wedge \\ & \qquad j \sim [\![\texttt{<j> = ret <t> <v>}]\!]\}\end{aligned} \tag{4.7d}$$

More need of explanation arises for the `call` instruction. For encoding `call` it is necessary to retrieve the value from the corresponding `ret` of the called function. A function may have multiple exit points, though, which makes rule equation (4.7d) considerably more complex. In this rule c_f indicates the called context, i matches the call instruction itself. The set R is generated by matching j with all returns in the context c_f and storing a pair (a_i, b_i) where a_i is the returned value and b_i the execution condition of the `ret` itself. The rule then builds a ϕ from these pairs, selecting a `ret`'s value if that particular instruction was executed. Note that the arguments a_i, b_i of the ϕ are meant to be the same a_i, b_i that make up the pairs in the set R. Functions with return type `void` do not need to be handled here.

Furthermore, note that intentionally none of the rules presented so far cared for the flags `nuw`, `nsw`, and `exact`, which indicate undefined behavior and poison values (see section 3.2.2). These flags are handled in separate rewrite rules presented in section 4.5.

4.2.2 Function Arguments

The function $\varepsilon^{\mathtt{A}}$ represents evaluation of a function argument. For encoding of this, the matching argument of the `call` instruction that called this context is identified and the function accordingly rewritten.

$$\begin{aligned}\varepsilon^{\mathtt{A}}_{[\![\mathtt{t}]\!]_i}(c,a_i) \longrightarrow\ & \varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]_i}(\mathrm{parent}(c),[\![\mathtt{arg}]\!]_i)\,;\ \mathrm{index}(a_i) = i \wedge \\ & \mathrm{callsite}(c) \sim [\![\texttt{call <ty> <fptr>(\{<t> <arg>\}*)}]\!]\end{aligned} \tag{4.8}$$

where $index(a) = i$ indicates that a is the i$^{\text{th}}$ argument.

4.2.3 Constants

Constant integers, which are always unsigned in LLVM-IR, are represented by $\varepsilon^{\mathtt{N}}$ and are rewritten as follows:

$$\varepsilon^{\mathtt{N}}_t(n) \longrightarrow (N)_t \tag{4.9}$$

Where n is an LLVM-IR constant with value N and its type is matching ILR's sort t.

Global variables, which are a form of constant pointers, can be handled similarly:

$$\varepsilon^{\mathtt{G}}(g) \longrightarrow (p^a_g)_t \tag{4.10a}$$

Where p_g^a is the position of global variable g for the architecture a. This can be retrieved by compiling the program to the target architecture a and extracting the position of the symbols from the binary file, or, if this level of precision is not required, unique addresses can be assigned at will, as long as it is ensured that global variables do not overlap with other memory objects.

Note that the addresses of global variables could also be chosen non-deterministically. This can be done for example by using uninterpreted pointer constants and adding appropriate non-overlapping constraints, similar to how this is done in chapter 5 for dynamically allocated memory.

4.3 Control Flow and Execution Conditions

An execution condition is the condition under which an instruction, a basic block, or a function is executed.[4] It implicitly encodes all control flow decisions so far and it is used as an antecedent to all properties to be checked because most properties should only be checked if the associated instruction is actually executed. This section presents a number of rewrite rules related to execution conditions.

Execution conditions are represented by the symbol η. $\overleftarrow{\eta}^{\mathrm{I}}$ and $\overrightarrow{\eta}^{\mathrm{I}}$ stand for the condition before and after execution of an instruction. Similarly, $\overleftarrow{\eta}^{\mathrm{B}}$ and $\overrightarrow{\eta}^{\mathrm{B}}$ represent the execution of a basic block, and $\overleftarrow{\eta}^{\mathrm{F}}$ and $\overrightarrow{\eta}^{\mathrm{F}}$ the execution of a function. Last but not least, η^{J} stands for execution of a control flow edge, i.e. execution of an edge in the control flow graph. The relationship of execution conditions to the source-level objects is also shown in figure 4.1

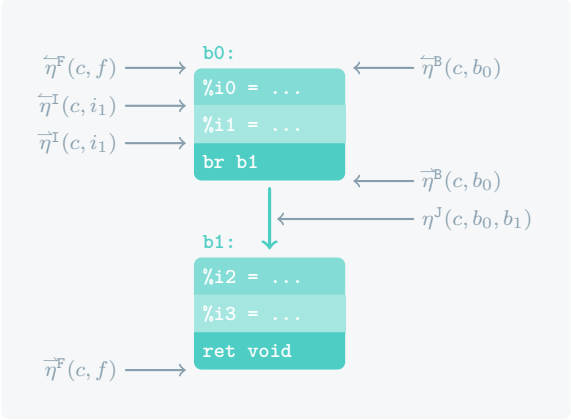


Figure 4.1: Illustration of execution conditions and their relation to LLVM-IR for $(V_f, B_f, r_f) = (\{b_0, b_1\}, \{(b_0, b_1)\}, b_0)$

An instruction is executed when the previous instruction's execution finishes. However, if an instruction is the first in the basic block, it is executed when the basic

[4] This is conceptually similar to path conditions in symbolic execution.

block is executed:

$$\overleftarrow{\eta}^{\mathtt{I}}(c,i) \longrightarrow \overleftarrow{\eta}^{\mathtt{B}}(c,b)) \,;\, i = \mathrm{first}_{\mathtt{B}}(b) \tag{4.11a}$$

$$\overleftarrow{\eta}^{\mathtt{I}}(c,i) \longrightarrow \vec{\eta}^{\mathtt{I}}(c,j) \,;\, \mathrm{succ}_{\mathtt{B}}(j,i) \tag{4.11b}$$

Note that this is only true for sequential programs. In concurrent programs, other threads or processes might modify the program's state in between execution of instructions and might modify the execution condition, e.g. by terminating the program.

Execution of a basic block ends when its last instruction is executed:

$$\vec{\eta}^{\mathtt{B}}(c,b) \longrightarrow \vec{\eta}^{\mathtt{I}}(i) \,;\, i = \mathrm{term}_{\mathtt{B}}(b) \tag{4.12}$$

The function $\eta^{\mathtt{J}}(c, b_1, b_2)$ represents the execution condition of the control flow edge from b_1 to b_2 in context c. This auxiliary function splits rewriting of control flow graphs into two separate rewrite operations, which makes it easier to reason about control flow edges. The instructions `br`, `indirectbr`, and `switch` are responsible for a program's control flow edge (see section 3.2.2). In LLBMC `indirectbr` and `switch` are replaced on the LLVM-IR level by one or more branches. This leaves only conditional and unconditional branches to be encoded:

$$\eta^{\mathtt{J}}(c, b_1, \texttt{<b2>}) \longrightarrow \vec{\eta}^{\mathtt{B}}(c, b_1) \,; \quad term(b_1) \sim [\![\texttt{br label <b2>}]\!] \tag{4.13a}$$

$$\eta^{\mathtt{J}}(c, b_1, \texttt{<b2>}) \longrightarrow \vec{\eta}^{\mathtt{B}}(c, b_1) \wedge \varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v}]\!]) \,; \quad term(b_1) \sim [\![\texttt{br i1 <v>, label <b2>, label <b3>}]\!] \tag{4.13b}$$

$$\eta^{\mathtt{J}}(c, b_1, \texttt{<b2>}) \longrightarrow \vec{\eta}^{\mathtt{B}}(c, b_1) \wedge \neg\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c, [\![\mathtt{v}]\!]) \,; \quad term(b_1) \sim [\![\texttt{br i1 <v>, label <b3>, label <b2>}]\!] \tag{4.13c}$$

A basic block is executed when any of the incoming control flow edges are executed. However, if the basic block is the entry block, it is executed when the function's execution begins:

$$\overleftarrow{\eta}^{\mathtt{B}}(c,b) \longrightarrow \overleftarrow{\eta}^{\mathtt{F}}(c,f) \,;\, b = \mathrm{entry}(c) \tag{4.14a}$$

$$\overleftarrow{\eta}^{\mathtt{B}}(c,b) \longrightarrow \bigvee_{b_1 \,:(b_1,b)\in B_f} \eta^{\mathtt{J}}(c, b_1, b) \,;\, b \neq \mathrm{entry}(c) \tag{4.14b}$$

In equation (4.14b) (V_f, B_f, r_f) is f's bounded control flow graph, as defined in section 3.5. Note that this is only valid because in LLVM-IR the entry block is not allowed to have predecessors.

All encoding so far was intra-procedural, meaning restricted to a single function. However, extending encoding to be inter-procedural is easily possible.

Execution of a function is finished, when any of the exit points of the function (basic blocks with terminator `ret`), are finished:

$$\vec{\eta}^{\mathtt{F}}(c,f) \longrightarrow \bigvee_{b\,:\,\mathrm{exit}(c,b)} \vec{\eta}^{\mathtt{B}}(c,b) \tag{4.15}$$

Furthermore, the execution condition before calling a function is:

$$\overleftarrow{\eta}^{\mathtt{F}}(c, f) \longrightarrow \overleftarrow{\eta}^{\mathtt{I}}(c', i)\ ;\ c' = \mathrm{parent}(c) \wedge i = \mathrm{callsite}(c) \tag{4.16a}$$

$$\overleftarrow{\eta}^{\mathtt{F}}(c, f) \longrightarrow \overleftarrow{\eta}^{\mathtt{P}}\ ;\ \text{otherwise} \tag{4.16b}$$

In LLVM-IR regular control flow is exclusively done using terminator instructions such as `br`. Nonetheless, some non-terminator instructions i might terminate the entire program such as C's `exit()` function. Furthermore, undefined behavior can be assumed to terminate the program. After all, nothing is known about a program after undefined behavior has occurred. This leads to the following set of rules:

$$\begin{aligned}\overrightarrow{\eta}^{\mathtt{I}}(c, i) \longrightarrow\ & \overrightarrow{\eta}^{\mathtt{F}}(c_f, f)\ ;\ i = callsite(c_f) \wedge f = [\![\texttt{fptr}]\!] \wedge \\ & \mathrm{callsite}(c) \sim [\![\texttt{call <ty> <fptr>(\{<t> <arg>\}*)}]\!]\end{aligned} \tag{4.17a}$$

$$\overrightarrow{\eta}^{\mathtt{I}}(c, i) \longrightarrow F\ ;\ i \sim [\![\texttt{call <void> exit()}]\!] \tag{4.17b}$$

$$\overrightarrow{\eta}^{\mathtt{I}}(c, i) \longrightarrow \overleftarrow{\eta}^{\mathtt{I}}(c, i) \wedge \sigma^{\mathtt{I}}(c, i)\ ;\ \text{otherwise} \tag{4.17c}$$

Semantics of the function $\sigma^{\mathtt{I}}(c, i)$ can be chosen as needed. LLBMC's default $\sigma^{\mathtt{I}}$ (introduced in section 4.5) is false if instruction i in context c is unsafe. This could also be handled differently, e.g. terminating on safety only for specific contexts, instructions or types of instructions.

The execution condition at the beginning of the program is true, the execution condition at the end of the program is the same as after the main function:

$$\overleftarrow{\eta}^{\mathtt{P}}(c, p) \longrightarrow T \tag{4.18a}$$

$$\overrightarrow{\eta}^{\mathtt{P}}(c, p) \longrightarrow \overrightarrow{\eta}^{\mathtt{F}}(c, f)\ ;\ f \text{ is main} \tag{4.18b}$$

4.4 Memory

This section describes how memory accesses, pointer arithmetic, stack management, and global variables are encoded.

4.4.1 Memory State

In the following, m indicates the appropriate memory sort for the program's target architecture while s stands for endianness of the program's target architecture.

Some LLVM-IR instructions modify the memory state of the program, others make use of the current memory state. Two separate functions are defined for representing memory states: $\overleftarrow{\mu}^{\mathtt{I}}(c, i)$ represents the memory state before instruction i is executed in context c, and $\overrightarrow{\mu}^{\mathtt{I}}(c, i)$ represents the memory state after i is executed in c.

Most instructions do not change memory, so the memory state after their execution is the same as before. Some instructions modify memory, most notably `store`:

$$\begin{aligned}\vec{\mu}^{\mathtt{I}}_m(c,i) \longrightarrow\ & \mathrm{store}^{s}_{m,[\![\mathtt{t2}]\!]}(\overleftarrow{\mu}^{\mathtt{I}}_m(c,i), \varepsilon^{\mathtt{V}}_{[\![\mathtt{t1}]\!]}(c,[\![\mathtt{v1}]\!]), \varepsilon^{\mathtt{V}}_{[\![\mathtt{t2}]\!]}(c,[\![\mathtt{v2}]\!]))\ ; \\ & i = [\![\texttt{store [volatile] <t1> <v1>, <t2> <v2>}]\!] \end{aligned} \tag{4.19a}$$

$$\begin{aligned}\vec{\mu}^{\mathtt{I}}_m(c,i) \longrightarrow\ & \vec{\mu}^{\mathtt{F}}_m(c_f, f)\ ; \\ & c = \mathrm{parent}(c_f) \wedge i = \mathrm{callsite}(c_f) \wedge \\ & i \sim [\![\texttt{<i> = call <t> <fptr>(\{<ty> <arg>\}*)}]\!] \end{aligned} \tag{4.19b}$$

$$\vec{\mu}^{\mathtt{I}}_m(c,i) \longrightarrow \overleftarrow{\mu}^{\mathtt{I}}_m(c,i)\ ;\ \text{otherwise} \tag{4.19c}$$

The `load` instruction loads from the current memory state:

$$\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow \mathrm{load}\ s_{m,[\![\mathtt{t}]\!]}(\overleftarrow{\mu}^{\mathtt{I}}_m(c,i), v)\ ;\ i \sim [\![\texttt{<i> = load <t> <v>}]\!] \tag{4.20a}$$

If the instruction is the first instruction in the basic block, then the memory state before execution of this instruction is the same as before execution of the basic block as a whole. If the instruction is not the first instruction in a basic block, and assuming that nothing changes the memory state in between execution of the two instructions, then the memory state before execution of the second instruction is the same as the memory state after execution of the first instruction. These two properties lead to the following rewrite rules:

$$\overleftarrow{\mu}^{\mathtt{I}}_m(c,i) \longrightarrow \overleftarrow{\mu}^{\mathtt{B}}_m(c,b))\ ;\ i = \mathrm{first}_{\mathtt{B}}(b) \tag{4.21a}$$

$$\overleftarrow{\mu}^{\mathtt{I}}_m(c,i) \longrightarrow \vec{\mu}^{\mathtt{I}}_m(c,i_1))\ ;\ \mathrm{succ}_{\mathtt{B}}(i_1,i) \tag{4.21b}$$

The memory state after execution of a basic block is the same as the memory state after execution of the last instruction in the basic block:

$$\vec{\mu}^{\mathtt{B}}_m(c,b) \longrightarrow \vec{\mu}^{\mathtt{I}}_m(c,i)\ ;\ i = \mathrm{term}_{\mathtt{B}}(b) \tag{4.22}$$

This again assumes the program is sequential.

The memory state before execution of a basic block depends on which basic block was executed immediately before. This can be handled by using phi instructions, similarly to how LLVM-IR handles values coming from one of multiple predecessor basic blocks:

$$\begin{aligned}\overleftarrow{\mu}^{\mathtt{B}}_m(c,b) \longrightarrow\ & \phi(\vec{\mu}^{\mathtt{B}}_m(c,b_1), \eta^{\mathtt{J}}(c,b_1,b), \ldots, \vec{\mu}^{\mathtt{B}}_m(c,b_{|P|}), \eta^{\mathtt{J}}(c,b_{|P|},b))\ ; \\ & \neg\,\mathrm{entry}(b) \wedge P = \{b_i : \mathrm{succ}_{\mathtt{F}}(b_i,b)\} \end{aligned} \tag{4.23a}$$

$$\overleftarrow{\mu}^{\mathtt{B}}_m(c,b) \longrightarrow \overleftarrow{\mu}^{\mathtt{F}}_m(c,f)\ ;\ \mathrm{entry}(b) \tag{4.23b}$$

Note that it is not sufficient to use $\vec{\eta}^{\mathtt{B}}$ in equation (4.23a), but $\eta^{\mathtt{J}}$ is strictly required. This can be seen in figure 4.2, where $\vec{\eta}^{\mathtt{B}}(c,b_0)$ and $\vec{\eta}^{\mathtt{B}}(c,b_1)$ can be true at the same time, if control flow goes from %b0 via %b1 to %b2. This means execution conditions

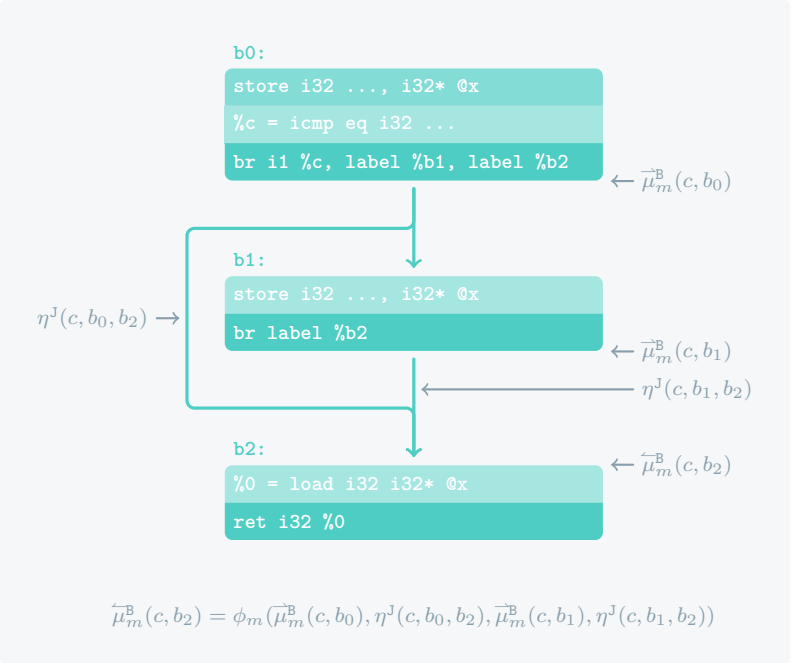


Figure 4.2: Illustration of the memory encoding rule for memory states before execution of basic blocks

at the end of basic blocks are not sufficient to distinguish between these control flow edges, as the η^{B} are not mutually excluse, but the η^{J} are.

A `call` instruction does not modify the memory contents, so the memory state at the beginning of the execution of a function block is the same as before execution of the call instruction itself:

$$\overleftarrow{\mu}^{\mathrm{F}}_m(c, f) \longrightarrow \overleftarrow{\mu}^{\mathrm{I}}_m(c', i)\,;\; c' = \mathrm{parent}(c) \wedge i = \mathrm{callsite}(c) \tag{4.24a}$$

$$\overleftarrow{\mu}^{\mathrm{F}}_m(c, f) \longrightarrow \overleftarrow{\mu}^{\mathrm{P}}_m(c, p)\,;\; \forall c' \left(c' \neq \mathrm{parent}(c)\right) \tag{4.24b}$$

The memory state after execution of a function is the memory state after execution of the `ret` instruction. If there are multiple `ret` instructions in a single function, the right one is selected using a ϕ function. This is similar to how the `phi` instruction handles values in LLVM-IR itself:

$$\begin{aligned}\vec{\mu}^{\mathrm{F}}_m(c, f) \longrightarrow\; & \phi_{[\![\mathtt{t}]\!]}(a_1, b_1, \ldots, a_{|R|}, b_{|R|})\,; \\ & R = \{(a_i, b_i) : a_i = \vec{\mu}^{\mathrm{I}}_m(c, j) \wedge b_i = \vec{\eta}^{\mathrm{I}}(c, j), \wedge \\ & \quad j \sim [\![\texttt{<j> = ret <t1> <v>}]\!]\}\end{aligned} \tag{4.25}$$

Again, like for the `phi` instruction, ϕ's function arguments a_i and b_i are meant to match a_i and b_i in R.

The program state before execution of the program is constructed by writing the values specified in the initialization expressions of the program's global variables to

an otherwise uninterpreted fresh memory constant. The memory state after execution of a program is the memory state after execution of the main instruction:

$$\vec{\mu}^{\mathtt{P}}_m(c,p) = \vec{\mu}^{\mathtt{F}}_m(c,f)\ ;\ \mathsf{f}\ \text{is main} \tag{4.26}$$

4.4.2 Stack

Many stack allocated variables in LLVM-IR are moved to virtual registers by LLVM's `mem2reg` optimization pass. This is not always possible, though, e.g. when a pointer to a stack allocated variable is passed to a function. For these cases LLBMC still needs to keep track of the operating system's stack. LLBMC uses the function symbol τ to represent the state of the stack.

τ objects have the architecture's `i8`-pointer type. For simplicity, we assume the stack grows downwards, like on the x86 architecture. This means the stack's top is, counter-intuitively, the stack's lowest address.

Returning from a function call implicitly restores the state of the stack before the function call. Because of this, calls do not need to be handled specially and no $\vec{\tau}^{\mathtt{F}}$ is actually needed. Furthermore, the stack after execution of the program is always empty, so $\vec{\tau}^{\mathtt{P}}$ is not required.

Stack memory is allocated in LLVM-IR using the `alloca` instruction. The effect of an alloca on the state of the stack is modeled by the following set of rewrite rules:

$$\begin{aligned}
\vec{\tau}^{\mathtt{I}}_s(c,i) \longrightarrow\ & \mathrm{sub}_{s,t}(\overleftarrow{\tau}^{\mathtt{I}}_s(c,i), (\mathrm{allocwidth}^a_{[\![\mathtt{t}]\!]})_t)\ ; \\
& i \sim [\![\texttt{<i> = alloca <t>}]\!] \wedge |t| = |s| && (4.27a)\\
\vec{\tau}^{\mathtt{I}}_s(c,i) \longrightarrow\ & \mathrm{sub}_{s,[\![\mathtt{t2}]\!]}(\overleftarrow{\tau}^{\mathtt{I}}_s(c,i), \mathrm{mul}_{[\![\mathtt{t2}]\!]}([\![\mathtt{n}]\!], (\mathrm{allocwidth}^a_{[\![\mathtt{t1}]\!]})_{[\![\mathtt{t2}]\!]}))\ ; \\
& i \sim [\![\texttt{<i> = alloca <t1>, <t2> <n>}]\!] && (4.27b)\\
\vec{\tau}^{\mathtt{I}}_s(c,i) \longrightarrow\ & \overleftarrow{\tau}^{\mathtt{I}}_s(c,i)\ ;\ \text{otherwise} && (4.27c)
\end{aligned}$$

Because the stack grows downwards, calculating the position of the last allocated stack variable is simple. This is simply the current stack top pointer:

$$\begin{aligned}
\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow\ & \mathrm{bitcast}_{s,[\![\mathtt{t}]\!]}(\vec{\tau}^{\mathtt{I}}_s(c,i))\ ; \\
& i \sim [\![\texttt{<i> = alloca <t> [, <t2> <n>]}]\!] && (4.28)
\end{aligned}$$

Note that we currently do not take alignment into account, but this should easily be possible by masking the required number of least significant bits.

The stack state is propagated through the program and all instructions not related to stack management with the following set of rewrite rules:

$$\begin{aligned}
\overleftarrow{\tau}^{\mathtt{P}}_s(c,p) \longrightarrow\ & (\mathrm{stacktop}^a)_t && (4.29a)\\
\overleftarrow{\tau}^{\mathtt{F}}_s(c,f) \longrightarrow\ & \overleftarrow{\tau}^{\mathtt{I}}_s(c',i)\ ;\ c' = \mathrm{parent}(c) \wedge i = \mathrm{callsite}(c) && (4.29b)\\
\overleftarrow{\tau}^{\mathtt{F}}_s(c,f) \longrightarrow\ & \overleftarrow{\tau}^{\mathtt{P}}_s(c,p)\ ;\ \neg\exists c'\ \big(c' = parent(c)\big) && (4.29c)\\
\overleftarrow{\tau}^{\mathtt{B}}_s(c,b) \longrightarrow\ & \phi(\vec{\tau}^{\mathtt{B}}_s(c,b_1), \eta^{\mathtt{J}}(c,b_1,b), \ldots, \vec{\tau}^{\mathtt{B}}_s(c,b_{|P|}), \eta^{\mathtt{J}}(c,b_{|P|},b))\ ; \\
& \neg\,\mathrm{entry}(b) \wedge P = \{b_i : \mathrm{succ}_{\mathtt{F}}(b_i,b)\} && (4.29d)
\end{aligned}$$

$$\overleftarrow{\tau}^{\mathtt{B}}_s(c,b) \longrightarrow \overleftarrow{\tau}^{\mathtt{F}}(c,f)\,;\,\mathrm{entry}\,b \tag{4.29e}$$
$$\overrightarrow{\tau}^{\mathtt{B}}_s(c,b) \longrightarrow \overrightarrow{\tau}^{\mathtt{I}}_s(c,i)\,;\,i = \mathrm{term}_{\mathtt{B}}(b) \tag{4.29f}$$
$$\overleftarrow{\tau}^{\mathtt{I}}_s(c,i) \longrightarrow \overleftarrow{\tau}^{\mathtt{B}}_s(c,b)\,;\,i = \mathrm{first}_{\mathtt{B}}(b) \tag{4.29g}$$
$$\overleftarrow{\tau}^{\mathtt{I}}_s(c,i) \longrightarrow \overrightarrow{\tau}^{\mathtt{I}}_s(c,i_1)\,;\,\mathrm{succ}_{\mathtt{B}}(i_1,i) \tag{4.29h}$$

4.5 Safety

LLBMC is designed for asserting safety properties, not liveness properties (see section 2.1.5). In temporal logic, safety properties are of the form $\mathsf{G}\,\phi$. In simplified terms, the predicate ϕ describes the safety of a single state, while G indicates that ϕ has to hold for all states. The presented encoding differs from this view, in that the safety property is not attached to a state but to a state transition instead. In LLBMC, such a state transition is the execution of an instruction, or more specifically the execution of an instruction in a specific context.

The function $\sigma^{\mathtt{I}}(c,i)$ encodes safety of the instruction i in the context c. In terms of temporal logic, one could say that if $\sigma^{\mathtt{I}}(c,i)$, then execution of the instruction i in context c leads to some safe state, while if $\neg\sigma^{\mathtt{I}}(c,i)$ said execution leads to an error state.

In LLBMC user-defined safety properties can be expressed by adding calls to the function `assert()` [ISOC99, section 7.2.1.1] to the C code.

Checking these user-defined properties is usually not sufficient though. As explained in section 3.2, LLVM-IR can have undefined behavior. If undefined behavior occurs, nothing may be assumed about the further execution of the program. Because of this, undefined behavior is never safe. LLBMC treats the absence of undefined behavior as a safety property and checks for it by default.

Furthermore, LLBMC only checks a bounded fragment of the program. If configured incorrectly, these bounds might be chosen too low. In this case, other safety properties might not be checked, which means that reaching a bound should never go unnoticed by the user. This leads to LLBMC's other built-in safety property, which encodes the absence of reached bounds.

4.5.1 Custom Assertions and Source-level Properties

The instrumentation of the C code presented in section 3.3 adds calls to various assertion functions to the IR code. These calls are either written by the developer of the software system using `assert()`, or similar calls to assertions are added by the code generator to check for undefined behavior on the C level.

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow \mathrm{ne}_{[\![\mathtt{t}]\!]}(v,0)\,;\,[\![\texttt{call void assert(<t> <v>)}]\!] \tag{4.30}$$

4.5.2 Undefined Behavior and Poison Values

Several LLVM-IR instructions, e.g. `udiv`, can cause undefined behavior. The results of other instructions, e.g. `add`, might be poison values, which in turn can cause undefined behavior in subsequently executed instructions using these values.

The rule set in this section is used to check for LLVM-IR-level undefined behavior and poison values. Note that in this rule set, poison values are not treated as intended in LLVM. Instead, whenever an instruction generates poison values, this is treated as undefined behavior. This is an overapproximation of undefined behavior in LLVM, though experience has shown that this has little effect in practice.

Note that most rules in this section are not necessary, if the approach introduced in section 3.3 is implemented, because in this case the properties to be checked are already explicitly added to the bitcode. However, the rules shown in this section make it possible to use LLBMC to analyze bitcode even for programs where the original source language is not supported by LLBMC.

The `add` instruction can occur with the two flags `nuw` and `nsw`. These flags indicate that no unsigned or respectively no signed behavior is expected to occur. If this happens anyway, the result is a poison value. The following rewrite rules encode all four variants of `add`:

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow T\,;\, i \sim [\![\texttt{<i> = add <t> <v0>, <v1>}]\!] \tag{4.31a}$$

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow \neg\,\mathrm{addo}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = add nuw <t> <v0>, <v1>}]\!] \tag{4.31b}$$

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow \neg\,\mathrm{addo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = add nsw <t> <v0>, <v1>}]\!] \tag{4.31c}$$

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow \neg\,\mathrm{addo}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!])) \wedge \neg\,\mathrm{addo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = add nuw nsw <t> <v0>, <v1>}]\!] \tag{4.31d}$$

Similar to the `add` instruction, the `sub` instruction has four variants for all combinations of the flags `nuw` and `nsw`:

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow T\,;\, i \sim [\![\texttt{<i> = sub <t> <v0>, <v1>}]\!] \tag{4.32a}$$

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow \neg\,\mathrm{subo}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = sub nuw <t> <v0>, <v1>}]\!] \tag{4.32b}$$

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow \neg\,\mathrm{subo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = sub nsw <t> <v0>, <v1>}]\!] \tag{4.32c}$$

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow \neg\,\mathrm{subo}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!])) \wedge \neg\,\mathrm{subo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = sub nuw nsw <t> <v0>, <v1>}]\!] \tag{4.32d}$$

Like add and sub, mul comes in four variants:

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow T\,;\, i \sim [\![\texttt{<i> = mul <t> <v0>, <v1>}]\!] \tag{4.33a}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow \neg\,\mathrm{mulo}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = mul nuw <t> <v0>, <v1>}]\!] \tag{4.33b}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow \neg\,\mathrm{mulo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = mul nsw <t> <v0>, <v1>}]\!] \tag{4.33c}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow \neg\,\mathrm{mulo}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \neg\,\mathrm{mulo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = mul nuw nsw <t> <v0>, <v1>}]\!] \tag{4.33d}$$

Division and remainder are notably more complex concerning their causes of undefined behavior and occurrences of poison values. For all four division related operations, division by zero is undefined behavior. Additionally sdiv, udiv may have the exact flag set which indicates that the division's remainder is expected to be zero. The instructions udiv and urem never overflow. In contrast to this, a single combination of divisor and dividend exists for which sdiv can overflow. This is because two's complement representation of signed integers is asymmetrical. Theoretically, srem cannot overflow, but for consistency only, it is defined to overflow in the same case as sdiv:

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow \neg\,\mathrm{divz}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = udiv <t> <v0>, <v1>}]\!] \tag{4.34a}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow \neg\,\mathrm{divz}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \neg\,\mathrm{divx}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = udiv exact <t> <v0>, <v1>}]\!] \tag{4.34b}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow \neg\,\mathrm{divz}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \neg\,\mathrm{divo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = sdiv <t> <v0>, <v1>}]\!] \tag{4.34c}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow \neg\,\mathrm{divz}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \neg\,\mathrm{divx}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \neg\,\mathrm{divo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = sdiv exact <t> <v0>, <v1>}]\!] \tag{4.34d}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow \neg\,\mathrm{divz}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = urem <t> <v0>, <v1>}]\!] \tag{4.34e}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow \neg\,\mathrm{divz}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!])) \wedge \neg\,\mathrm{divo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,;\; i \sim [\![\texttt{<i> = srem <t> <v0>, <v1>}]\!] \tag{4.34f}$$

Left shifts in LLVM-IR cause undefined behavior if the number by which a value is shifted is larger than the bitwidth of the shifted value. Additionally, a left shift can overflow as an unsigned and as a signed value. Whether this is defined to be a poison value or not depends on the flags set:

$$\begin{aligned}
\sigma^{\mathrm{I}}(c,i) \longrightarrow\ & \neg\,\mathrm{sho}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,; \\
& i \sim [\![\texttt{<i> = shl <t> <v0>, <v1>}]\!] && (4.35a)\\
\sigma^{\mathrm{I}}(c,i) \longrightarrow\ & \neg\,\mathrm{sho}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \\
& \neg\,\mathrm{shlo}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,; \\
& i \sim [\![\texttt{<i> = shl nuw <t> <v0>, <v1>}]\!] && (4.35b)\\
\sigma^{\mathrm{I}}(c,i) \longrightarrow\ & \neg\,\mathrm{sho}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \\
& \neg\,\mathrm{shlo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,; \\
& i \sim [\![\texttt{<i> = shl nsw <t> <v0>, <v1>}]\!] && (4.35c)\\
\sigma^{\mathrm{I}}(c,i) \longrightarrow\ & \neg\,\mathrm{sho}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \\
& \neg\,\mathrm{shlo}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \\
& \neg\,\mathrm{shlo}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,; \\
& i \sim [\![\texttt{<i> = shl nuw nsw <t> <v0>, <v1>}]\!] && (4.35d)
\end{aligned}$$

Similar to the left shift instruction, both variations of the right shift instruction, logic and arithmetic right shift, cause undefined behavior if shifted by a value larger than the bitwidth of the shifted value. Additionally, poison values can be introduced due to the `nuw`, `nsw`, and `exact` flags.

$$\begin{aligned}
\sigma^{\mathrm{I}}(c,i) \longrightarrow\ & \neg\,\mathrm{sho}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,; \\
& i \sim [\![\texttt{<i> = ashr <t> <v0>, <v1>}]\!] && (4.36a)\\
\sigma^{\mathrm{I}}(c,i) \longrightarrow\ & \neg\,\mathrm{sho}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \\
& \neg\,\mathrm{shrx}^{s}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,; \\
& i \sim [\![\texttt{<i> = ashr exact <t> <v0>, <v1>}]\!] && (4.36b)\\
\sigma^{\mathrm{I}}(c,i) \longrightarrow\ & \neg\,\mathrm{sho}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,; \\
& i \sim [\![\texttt{<i> = lshr <t> <v0>, <v1>}]\!] && (4.36c)\\
\sigma^{\mathrm{I}}(c,i) \longrightarrow\ & \neg\,\mathrm{sho}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\wedge \\
& \neg\,\mathrm{shrx}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\,; \\
& i \sim [\![\texttt{<i> = lshr exact <t> <v0>, <v1>}]\!] && (4.36d)
\end{aligned}$$

An `unreachable` instruction is never expected to be executed. The compiler's code generator and optimization passes are supposed to ensure `unreachable` is in fact never reached. If this happens anyway this is undefined behavior and indicates a bug in the compiler:

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow F\,;\ i \sim [\![\texttt{unreachable}]\!] \tag{4.37}$$

Not all possible sources of undefined behavior or poison values are currently supported in LLBMC. One prominent example is undefined behavior introduced by the `inbounds` variant of the `getelementptr` instruction. Another example is undefined behavior introduced by `loads` and `stores` with bitwidths that are not a multiple of 8 and where the type of a `load` does not match the type of the previous `store` to the same location.

If code generation is done as described in section 3.3, and the generated LLVM-IR consequently contains checks for undefined behavior on the C level, and if those checks cover all cases of undefined behavior, then it is not necessary to check for undefined behavior on the LLVM-IR level. This is because the compiler is not allowed to introduce undefined behavior itself.[5] Therefore, any occurrence of undefined behavior on the LLVM-IR is a consequence of undefined behavior on the C level. Consequently, any checks for undefined behavior on the LLVM-IR would be redundant in relation to the checks inserted by the approach described in section 3.3.

4.5.3 Bounds Checking

Because LLBMC only analyses a bounded fragment of a program, the analysis might not be complete. Checking for loop iteration bounds and call depth bounds ensures that the user is notified about this situation. This is done by marking the involved `br` and `call` instructions as unsafe. With the bounded call graph of the function f containing the instruction i being given as $G = (V_f, B_f, r_f)$, the following rewrite rules ensure insertion of loop bound checks:

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow T \,;\, [\![\texttt{dst}]\!] \in B_f \wedge i \sim [\![\texttt{br label <dst>}]\!] \tag{4.38a}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow F \,;\, [\![\texttt{dst}]\!] \notin B_f \wedge i \sim [\![\texttt{br label <dst>}]\!] \tag{4.38b}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow T \,;\, [\![\texttt{then}]\!] \in B_f \wedge [\![\texttt{else}]\!] \in B_f \wedge i \sim [\![\texttt{br i1 <cond>, label <then>, label <else>}]\!] \tag{4.38c}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow [\![\texttt{cond}]\!] \,;\, [\![\texttt{then}]\!] \in B_f \wedge [\![\texttt{else}]\!] \notin B_f \wedge i \sim [\![\texttt{br i1 <cond>, label <then>, label <else>}]\!] \tag{4.38d}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow \mathrm{not}([\![\texttt{cond}]\!]) \,;\, [\![\texttt{then}]\!] \notin B_f \wedge [\![\texttt{else}]\!] \in B_f \wedge i \sim [\![\texttt{br i1 <cond>, label <then>, label <else>}]\!] \tag{4.38e}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow F \,;\, [\![\texttt{then}]\!] \notin B_f \wedge [\![\texttt{else}]\!] \notin B_f \wedge i \sim [\![\texttt{br i1 <cond>, label <then>, label <else>}]\!] \tag{4.38f}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow T \,;\, i \sim [\![\texttt{call <t> <fptr>(\{<ty> <arg>\}*)}]\!] \wedge (c,i) \in G \tag{4.38g}$$

$$\sigma^{\mathrm{I}}(c,i) \longrightarrow F \,;\, i \sim [\![\texttt{call <t> <fptr>(\{<ty> <arg>\}*)}]\!] \wedge (c,i) \notin G \tag{4.38h}$$

[5] This obviously assumes the compiler is correct.

4.5.4 Safety of a Whole Program

The function $\sigma^{\mathrm{I}}(c, i)$ is sufficient to encode a predicate ϕ for a single state, but not yet sufficient to encode $\mathsf{G}\,\phi$. To encode this we introduce the function $\sigma^{\mathrm{P}}(c, p)$ which indicates safety of a program as a whole. However, this is restricted to the bounded fragment described by the call graph of which c is the root node. The function σ^{F} is then rewritten recursively over the whole call graph until only σ^{I} remains:

$$\sigma^{\mathrm{P}}(c, p) \longrightarrow \sigma^{\mathrm{F}}(c, f)\ ;\ f = \mathrm{fun}(c) \tag{4.39a}$$

$$\sigma^{\mathrm{F}}(c, f) \longrightarrow \left(\bigwedge_{b \in f} \bigwedge_{i \in b} \widetilde{\eta}^{\mathrm{I}}(c, i) \to \sigma^{\mathrm{I}}(c, i) \right) \wedge \left(\bigwedge_{c' \in \{c_i : c = \mathrm{parent}(c_i)\}} \sigma^{\mathrm{F}}(c', f') \right) \tag{4.39b}$$

where $f' = \mathrm{fun}(c')$. The σ^{I} are rewritten as explained above. Note that this approach assumes that every safety property is attached to exactly one instruction. This is safe to assume because every state transition is caused by an instruction so that for any violated safety property there is an instruction that causes the state transition from a safe to an unsafe state. If an instruction's safety property does not hold the instruction is called *unsafe* (or sometimes *failing*).

4.6 A Term Rewriting System for Encoding LLVM-IR

The rewrite rule schemata presented in equations (4.1) and (4.3) to (4.39) can be instantiated for any LLVM-IR program p to retrieve a concrete term rewriting system $\mathcal{R}^{p}_{enc}$. If rules of this term rewriting system are applied exhaustively, the resulting formula will be nearly reduced to the core ILR language as defined in section 3.7. Any remaining function symbols from the extension introduced in this chapter can safely be replaced by uninterpreted constants. This primarily includes the initial memory state and values read from volatile variables. The resulting formula can then be simplified as in section 6.1 and solved as in section 6.2.

4.6.1 Variations of the Term Rewriting System

A bounded model checker based on the presented term rewriting system can easily be adapted to different use cases by adding, removing, and replacing term rewriting rules as desired.

One area where this kind of configuration is interesting is undefined behavior. In the case of undefined behavior, nothing may be assumed about the further execution of the program, and consequently LLBMC treats this case as program termination. Unfortunately, this might shadow other defects that occur later on during execution because the relevant code is effectively turned into dead code. This gives false confidence in this code, and once the first occurrence of undefined behavior is removed, the previously shadowed defects are shown to the user unexpectedly, as they likely do not understand the interaction between these defects. Hence, while it

is technically correct to treat undefined behavior as program termination, in practice users of such tools expect the tool to continue execution and behave "as expected".

The rule in equation (4.17c) could be changed to

$$\vec{\eta}^{\mathtt{I}}(c,i) \longrightarrow \overleftarrow{\eta}^{\mathtt{I}}(c,i), \tag{4.40}$$

to indicate that undefined behavior does not terminate the program. But consider the case of an `add` with the `nuw`. Because the instruction has the `nuw` flag set, LLVM might use the undefined behavior for compiler optimizations which might cause unexpected behavior. This can be prevented by not running optimizations or by simply removing the `nuw` before optimization. While this is not guaranteed to make the program behave "as expected", it likely is a sensible approximation.

Additionally, one might replace undefined behavior by an undefined value. If overflow occurs, the result of the instruction is replaced by a fresh uninterpreted constant symbol of the appropriate sort. This can be done by substituting the rule in equation (4.3a) by the following rule:

$$\begin{aligned}\varepsilon^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow\ & \mathrm{select}_{[\![\mathtt{t}]\!]}(\\ & \quad \mathrm{addo}^{u}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!])),k, \\ & \quad \mathrm{add}_{[\![\mathtt{t}]\!]}(\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v2}]\!]))\ ; \\ & i \sim [\![\texttt{<i> = add nuw <t> <v1>, <v2>}]\!]\end{aligned} \tag{4.41}$$

where k is a fresh uninterpreted constant of sort $[\![\mathtt{t}]\!]$.

Another example where configuration might be done are `store` operations marked as `volatile`. As `load`s from this location will not actually read from memory, the store might as well be removed entirely:

$$\begin{aligned}\vec{\mu}^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow\ & \mathrm{store}(\overleftarrow{\mu}^{\mathtt{I}}(c,i),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v0}]\!]),\varepsilon^{\mathtt{V}}_{[\![\mathtt{t}]\!]}(c,[\![\mathtt{v1}]\!]))\ ; \\ & i = [\![\texttt{store <t0> <v0>, <t1> <v1>}]\!]\end{aligned} \tag{4.42a}$$

$$\begin{aligned}\vec{\mu}^{\mathtt{I}}_{[\![\mathtt{t}]\!]}(c,i) \longrightarrow\ & \overleftarrow{\mu}^{\mathtt{I}}(c,i)\ ; \\ & i = [\![\texttt{store volatile <t0> <v0>, <t1> <v1>}]\!]\end{aligned} \tag{4.42b}$$

As already mentioned in section 4.5.2, the rewriting rules presented above do not model poison values as intended. This has not yet been of any concern for multiple reasons, not the least of which is the fact that the approach presented above is more strict than necessary, and LLBMC therefore does not risk missing any defects due to this. This is because the introduction of poison values is simply a means to delay the effects of an invalid operation, which would otherwise be undefined behavior, to the point of using the result of the operation. If the result is not used, no undefined behavior is present at all. This means these invalid operations can only result in undefined behavior if poison values occur. Because LLBMC reports the calculation of poison values, the tool is guaranteed to report all occurrences of undefined behavior. Another reason why LLBMCs handling of poison values is a good approach in practice is that no case has occurred in competitions or industrial applications so far where LLBMC was too strict because of this. More importantly though, this becomes irrelevant once properties are annotated as presented in section 3.3, which

makes checking for poison values superfluous. Nonetheless the rewriting system can be modified to handle poison values faithfully. We will not go into details here, but the basic idea is to add a new family of functions π which follows roughly the same pattern as the ε family. Instead of encoding the evaluation of an instruction it encodes meta information about the instruction's poisonousness, which is then checked at the appropriate places.

4.6.2 Implementation of the Term Rewriting System

LLBMC's implementation deviates considerably from the presented representation of the encoding process as a term rewriting system.

First of all, rewrite rule schemata are not instantiated but implemented in its schematic form. Not instantiating the rules reduces memory consumption and execution time of LLBMC.

Instead of implementing the function symbols such as $\varepsilon^{\mathtt{I}}$ or $\mu^{\mathtt{B}}$ as part of a term rewriting system they are implemented as maps. For example, $\varepsilon^{\mathtt{I}}$ is implemented as a map $\{(c, i) \mapsto s\}$, where c and i are the arguments of $\varepsilon^{\mathtt{I}}$ and s is the term that would result from applying the encoding term rewriting system to $\varepsilon^{\mathtt{I}}(c, i)$. LLBMC then directly creates the term s to which $\varepsilon_t^{\mathtt{I}}(c, i)$ would be rewritten and stores it in this map. This is made possible because LLBMC traverses the call and control flow graphs in the direction of execution, which ensures that the arguments of any rewrite rules' right hand side are stored in the map before the term itself is created. LLBMC uses this map as a cache for term sharing during encoding, but more importantly, the map can be used to generate LLVM-IR counterexamples from models of the ILR formula. Corresponding maps exist for all other function symbols, too.

LLBMC's implementation is the combination of tree traversal of the call graph and a linear traversal of a function for each node in the call graph. This algorithm's termination is obvious. A proof of termination of the term rewriting system presented above is omitted here. According to Bündgen [Bün98], a term rewriting system terminates if a term order $\succ$ exists so that for all rewrite rules $l \longrightarrow r$, $l \succ r$ holds. A suitable term order can be based on an order defined for all pairs (c, i) of contexts c and instructions i. This order counts the maximal number of instructions executed since the beginning of the program's execution. Intuitively, the encoding's term rewriting can be seen as walking backwards through the call graph and the code with respect to the direction of program's execution. Eventually, it will always reach the first instruction of the bounded fragment and then terminate.

4.7 Summary and Outlook

LLBMC's encoding is based on a language extension of the core ILR language. The encoding is formalized as a term rewriting system. Both the language extension and the term rewriting system are schemata that need to be instantiated for concrete programs. LLBMC can be easily configured by selecting an appropriate set of term rewriting rules, e.g. by leaving out specific rules or replacing them by alternative rules.

A likely downside of the presented approach is that a single array is used. This places unnecessary burden on the SMT solver to find out which memory objects might align. A possible improvement here would be the use of a (possibly cheap) alias analysis to find groups of related memory objects. These can then be placed in separate arrays. This might result in formulæ which are easier to solve for SMT solvers.

Chapter 5

Dynamic Memory Allocation and Memory Access Safety

As previously described, LLBMC uses the theory of arrays to implement a flat memory model, meaning that a single array is used to model the program's entire address space. A program's *address space* is a range of addresses, each of which can be used to store information at. Storing information in memory is done using the theory of array's store function, while loading values from memory is modeled with the select function[1]. This approach makes use of the performance provided by SMT solvers which are optimized for the theory of arrays. This allows modeling the main memory's contents easily, but it does not provide any means for deciding if a given address range in the address space may actually be accesses at a given point of time. This chapter shows how LLBMC models memory access safety for statically and dynamically allocated memory. While *statically allocated memory*, e.g. memory ranges used for storing global variables, poses little challenge in this regard, *dynamically allocated memory* requires means to track the current state of memory allocation for multiple address ranges during execution of the program. The techniques implemented in LLBMC to do so are the main focus of this chapter.

Section 5.1 of this chapter gives an overview of dynamic memory allocation in C. We then present a theory for dynamic memory allocation as an extension to ILR in section 5.2. In addition, a partial decision procedure for this theory based on term rewriting is presented in section 5.3 which is sufficiently expressive for LLBMC's purposes. Additionally, an extension to the term rewriting system presented in chapter 4 which modifies the safety predicate σ so that it also takes memory access safety into account is shown in section 5.4. Note, that the theory presented in this chapter is related to the decision procedure previously published in [FMS11] but the procedure presented here is supplemented by an axiomatization of the C standard on which the decision procedure is based. Furthermore a proof is provided that shows that the decision procedure is correct with respect to this axiomatization.

The presented theory was designed with careful attention to a separation of concerns. This means the theory strictly focuses on dynamic memory allocation and memory

[1] The naming of the theory of arrays functions follows the convention set by the SMT-LIB standard

access correctness and does neither handle the memory's contents nor loading and storing. This makes it possible to combine this theory with the most suitable approach for modeling the memory contents for any given use case. This also sets the presented work apart from related work such as the work by Cohen et al. [Coh+09b], which provides a combined theory for heap allocation and contents.

Dynamic memory allocation is rarely used in embedded systems. In fact, it is explicitly disallowed by the MISRA-C guidelines [MIS13]. Nonetheless, dynamic memory is important for the analysis of nearly all non-embedded software as well as for a meaningful comparison with competing tools.

LLBMC uses LLVM-IR as its input language and by doing so, LLBMC is language independent. This is not possible for dynamic memory allocation, however, because LLVM-IR has no built-in notion of dynamic memory allocation. LLVM initially did provide a dedicated `malloc` instruction, which was closely modeled after C's `malloc` function. For languages of the C family such an instruction did not provide any benefits over simply calling C's `malloc` function. Other languages supported by LLVM had no use for the instruction at all, e.g. Java, which has a decidedly distinct memory model. For these reasons, the `malloc` instruction was removed from the language specification with release 2.7.

Because of this, LLBMC's support for dynamically allocated memory cannot be done language-independently and is therefore closely tied to the source language. This thesis focuses on C's memory model, as defined in the publicly available draft of the C standard [ISOC99]. This is motivated to a large degree C's prevalence in the embedded market.

The example in listing 5.1 shows an example of the use of `malloc` and `free` in C and the translation thereof to LLVM-IR. Note that `malloc` returns a pointer to an untyped memory range of the requested size (or zero if no allocation was possible). The pointer is only then typecast to the intended type.

```
1  int* foo() {
2    // allocation of an object
3    // large enough for an int
4    int* i = (int*)malloc(sizeof(
         int));
5    bar(i);
6    if (*i == 0) {
7      // either deallocation
8      // of the object
9      free(i);
10     return 0;
11   } else {
12     // or return of the
13     // allocated object
14     return i;
15   }
16 }
```

(a) Original source file

```
1  define i32* @foo() {
2  entry:
3    %0 = call i8* malloc(i32 4)
4    %i = bitcast i8* %0 to i32*
5    call void @bar(i32* %i)
6    %1 = load i32, i32* %i
7    %2 = icmp eq %1, 0
8    br i1 %2, label %then, label↩
         %else
9
10 then:
11   call void @free(i32* %i)
12   ret i32* 0
13
14 else:
15   return i32* %i
16 }
```

(b) Translation to LLVM-IR

Listing 5.1: Example for use of `malloc` in C

5.1 Dynamic Memory Allocation in the C Standard

For the reasons mentioned above the memory allocation model presented in this chapter is modeled after the one presented in the ISO C'99 standard draft [ISOC99]. Subsets of the C language have been previously formalized, e.g. as in the tool Clight by Blazy and Leroy [BL09] and in the Formalin project by Krebbers and Wiedijk [KW11]. Clight aims at a large subset of the C language, while Formalin aims for full support of the entire standard. In contrast to these formalizations, we prefer LLVM for language-independence wherever we can and only concern ourselves with the parts of the C standard related to dynamic memory allocation.

These parts are spread out all over the ISO C'99 standard including sections 3, 6, and 7. Section 3 of the C standard introduces terms, definitions, and symbols used throughout the standard. In this section, the standard defines an *object* as:

> Region of data storage in the execution environment, the contents of which can represent values. [ISOC99, section 3.14]

An object must be composed of contiguous sequences of bytes:

> Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined. [ISOC99, section 6.2.6.1]

Number, order, and encoding are taken care of by our implementation of choice, LLVM, which in turn tries to be compatible either to the GCC compiler or to the MSVC compiler.

Furthermore, the standard introduces the notion of storage duration of objects:

> An object has a *storage duration* that determines its lifetime. There are three storage durations: static, automatic, and allocated. Allocated storage is described in 7.20.3. [ISOC99, section 6.2.4]

In LLVM-IR, static storage duration corresponds to global variables and automatic storage duration refers to virtual registers and stack allocated variables. Both of these are supported by LLBMC as described in chapter 4. This leaves only objects with allocated storage duration, which is the standard's name for dynamically allocated memory, to be supported by the theory.

Section 7 of the standard is concerned with C's standard libraries including the functions `malloc`, `free`, `realloc`, and `calloc` which the standard provides to the programmer for managing dynamically allocated memory:

> The order and contiguity of storage allocated by successive calls to the **`calloc`**, **`malloc`**, and **`realloc`** functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated). The lifetime of an allocated object extends from the allocation until the deallocation. Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If

> the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object. [ISOC99, section 7.20.3]

The most important function related to dynamic memory allocation in the standard is the `malloc` function with signature `void *malloc(size_t size)`.

> The **`malloc`** function allocates space for an object whose size is specified by **`size`** and whose value is indeterminate. [ISOC99, section 7.20.3.3]

Furthermore:

> The **`malloc`** function returns either a null pointer or a pointer to the allocated space. [ISOC99, section 7.20.3.3]

The counterpart of `malloc` is `free` with signature `void free(void *ptr)`. The standard states:

> The **`free`** function causes the space pointed to by **`ptr`** to be deallocated, that is, made available for further allocation. If **`ptr`** is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the **`calloc`**, **`malloc`**, or **`realloc`** function, or if the space has been deallocated by a call to **`free`** or **`realloc`**, the behavior is undefined. [ISOC99, section 7.20.3.2]

We will leave out the `calloc` function as it requires modification of the memory contents which is not in the focus of this chapter. We will also not discuss the function `realloc` as it can be emulated with `malloc` and `free`[2].Apart from this, we aim to describe a theory of heaps that faithfully captures the semantics described above.

The sections of the standard cited so far refer to memory allocation and deallocation leaving out accesses to memory. Two sections of the standard are relevant for this, first:

> The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime. [ISOC99, section 6.2.4]

C refers to addressable objects as *lvalues*:

> An *lvalue* is an expression with an object type or an incomplete type other than **`void`**; if an lvalue does not designate an object when it is evaluated, the behavior is undefined. [ISOC99, section 6.3.2.1]

This is in contrast to *rvalues* which are non-addressable temporary values. The quoted sections of the C standard are sufficient to define a theory of C-style dynamic memory allocation based on them.

[2] This is possible, despite of the fact that this is made more complicated by the fact that `realloc` might return the same pointer.

5.2 Extending ILR

This section extends ILR by an additional sort representing heap allocation states as well as multiple functions operating on this sort.

5.2.1 Sorts

Support for C's heap memory allocation is implemented as an extension to the ILR language as well as an extension to the term rewriting systems for encoding. The theory uses pairs of pointers and integers to encode individual objects but these pairs are neither part of the language extension nor the term rewriting system. Instead, these objects are only used in axioms and proofs and the presented extension introduces only a single new sort which represents the heap memory allocation state as a whole. The new sorts are shown in table 5.1. In this table $\langle N \rangle$ indicates the width of the heap model's pointers.

Sort Family	Pattern	Examples
Heap allocation state	h$\langle N \rangle$	h16, h32, h64

Table 5.1: Heap sort

Apart from the sort for the heap state itself, the language extension also requires sorts for indices and sizes. The theory, as presented here, is not restricted to a single index sort but can be used with any sort as an index sort that provides the functions $+$ and the predicates $<$ and $\leq$, each with the usual meaning. Bitvectors as well as mathematical integers are both conceivable for this, though we will restrict ourselves to ILR's pointer for indices and integers for sizes and offsets, both of which are bitvectors.

As already done in section 3.7 and chapter 4, we use placeholders instead of concrete sorts because the theory is independent of the target architecture's bitwidth. These placeholders are shown in table 5.2. Note that we intentionally reuse the placeholders $\mathcal{P}$ and $\mathcal{I}$ as these are the sorts used in LLBMC.

Sort Placeholder	Instantiatable Sorts
$\mathcal{H}$	Heap allocation state
$\mathcal{P}$	Pointer sort
$\mathcal{I}$	Size and offset sort

Table 5.2: Heap sort placeholders used in the following and their possible instantiations

For consistency with the rewrite rules presented in other parts of this thesis, we will use ILR's notation for the addition and comparison operations, namely $\mathrm{add}_{\mathcal{P},\mathcal{I}}$ and $\mathrm{eq}_{\mathcal{I}}$, $\mathrm{le}^{u}_{\mathcal{I}}$. To improve readability of axioms and proofs of this chapter, a simplified notation is used for these. For example, we will assume a single target architecture with a single bitwidth for pointers and we will assume all integers to be of the

same bitwidth.[3] This allows us to omit the subscripts indicating the functions' sorts. Furthermore, we use the usual mathematical symbols for addition, equality, and integer comparison instead of ILR's function symbols. Finally, we will omit the predicate $\langle\cdot\rangle$ for the boolean sort.

Variable naming conventions in the chapter are shown in table 5.3. As usual, if any of these names are used, the sort is implied to be as given in this table.

Sort Family	Variables
Heap allocation state	h, h_1, h_2
Pointers	p, q, r, o
Sizes and offsets	s, t, u

Table 5.3: Heap variable naming conventions

For the axiomatization of the heap theory, we use pairs of pointers and integers to represent objects, as defined in [ISOC99, section 6.2.6.1]. Given an object (p, s), we will call p its address and s its size. Furthermore, the interval $[p, p + s)$ will be called the object's *memory range*. Equality of pairs is defined by $\forall p, s, q, t \left((p, s) = (q, t) \leftrightarrow (p = q \wedge s = t)\right)$.

According to [ISOC99, section 6.3.2.1], an lvalue may only be referenced if it refers to an object. From a low-level perspective this means a memory range may only be accessed, if it is completely contained in an object's memory range. This motivates the definition of the auxiliary predicate $\mathrm{contains}$:

Definition 5.1 (Contains). *The predicate* $\mathrm{contains}$ *($\mathcal{P} \times \mathcal{I} \times \mathcal{P} \times \mathcal{I}$) expresses that a memory range (p, s) another memory range (q, t) and is defined by the following axiom:*

$$\forall p, s, q, t \left(\mathrm{contains}(p, s, q, t) \leftrightarrow p \leq q \wedge q + t \leq p + s\right).$$

Similarly, [ISOC99, section 7.20.3] requires objects to be mutually disjoint motivating to the following definition of the auxiliary function $\mathrm{disjoint}$:

Definition 5.2 (Disjoint). *The predicate* $\mathrm{disjoint}$ *($\mathcal{P} \times \mathcal{I} \times \mathcal{P} \times \mathcal{I}$) expresses that two memory ranges (p, s) and (q, t) do not overlap and is defined by the following axiom:*

$$\forall p, s, q, t \left(\mathrm{disjoint}(p, s, q, t) \leftrightarrow p + s < q \vee q + t < p\right).$$

Note that the functions $\mathrm{contains}$ and $\mathrm{disjoint}$ are not part of ILR but rule templates as described in section 2.1.6, though in the following, the two functions can be simply assumed to be rewritten by the following rewrite rules:

$$\mathrm{contains}_{\mathcal{P},\mathcal{I}}(p, s, q, t) \longrightarrow \mathrm{and}(\mathrm{le}^u_{\mathcal{P}}(p, q), \mathrm{le}^u_{\mathcal{P}}(\mathrm{add}_{\mathcal{P},\mathcal{I}}(q, t), \mathrm{add}_{\mathcal{P},\mathcal{I}}(p, s))) \quad (5.1a)$$

$$\mathrm{disjoint}_{\mathcal{P},\mathcal{I}}(p, s, q, t) \longrightarrow \mathrm{or}(\mathrm{lt}^u_{\mathcal{P}}(\mathrm{add}_{\mathcal{P},\mathcal{I}}(p, s), q), \mathrm{lt}^u_{\mathcal{P}}(\mathrm{add}_{\mathcal{P},\mathcal{I}}(q, t), p)) \quad (5.1b)$$

[3] While LLBMC allows for objects and heap allocation states of different target architectures in the same formula, we assume that pointers are only used on their own architecture and we therefore will not take differing bitwidths into account.

Given the definition of $\mathrm{disjoint}$, we can now define a heap allocation state as the set of currently allocated objects where each object is identified by a base pointer and a size:

Definition 5.3 (Heap Allocation State). *Given a bitwidth $n \in \mathbb{N}$, a heap allocation state h of sort $h\langle N \rangle$ is a set of pairs $(\mathcal{P} \times \mathcal{I})$ where*

$$\forall (p, s) \in h \left(p \neq 0\right), \text{ and} \tag{5.2}$$

$$\forall (p, s) \in h, (q, t) \in h \left((p, s) \neq (q, t) \rightarrow \mathrm{disjoint}(p, s, q, t)\right). \tag{5.3}$$

Both the requirement for $p \neq 0$ and the one for disjointness of objects follow directly from [ISOC99, section 7.20.3.3]. The additional constraint concerning alignment mentioned in that section is omitted in this thesis.

The terms heap and heap state are usually used to refer to the contents of dynamically allocated memory objects. Because we do not handle memory's contents in this chapter and for the sake of brefity, we will refer to the heap allocation state as heap or heap state exclusively.

5.2.2 Functions

The core functions of the theory of dynamic memory allocation are listed in table 5.4. The function $\mathrm{empty}_{\mathcal{H}}$ creates a heap constant without any allocated objects. The function malloc allocates a memory object given by a pointer to its lowest address and its size, while the function $\mathrm{free}_{\mathcal{H}}$ deallocates a previously allocated memory object. The functions $\mathrm{validaccess}_{\mathcal{H}}$ and $\mathrm{validfree}_{\mathcal{H}}$ check if memory allocation related operations are valid. The predicate $\mathrm{allocatable}_{\mathcal{H}}$ asserts that a memory object given by a pair of a pointer and size can be allocated.

Symbol : Signature	Interpretation
$\mathrm{empty}_{\mathcal{H}} : \rightarrow \mathcal{H}$	Empty heap
$\mathrm{malloc}_{\mathcal{H}} : \mathcal{H} \times \mathcal{P} \times \mathcal{I} \rightarrow \mathcal{H}$	Heap allocation
$\mathrm{free}_{\mathcal{H}} : \mathcal{H} \times \mathcal{P} \rightarrow \mathcal{H}$	Heap deallocation
$\mathrm{validaccess}_{\mathcal{H}} : \mathcal{H} \times \mathcal{P} \times \mathcal{I} \rightarrow \mathrm{bool}$	Access validity
$\mathrm{validfree}_{\mathcal{H}} : \mathcal{H} \times \mathcal{P} \rightarrow \mathrm{bool}$	Deallocation validity
$\mathrm{allocatable}_{\mathcal{H}} : \mathcal{H} \times \mathcal{P} \times \mathcal{I} \rightarrow \mathrm{bool}$	Allocatability

Table 5.4: Functions encoding heap state ($|\mathcal{H}| = |\mathcal{P}| = |\mathcal{I}|$)

An empty heap does not contain any objects:

$$\mathrm{empty}_{\mathcal{H}} = \emptyset \tag{5.4}$$

The function malloc is the counterpart to C's `malloc` function. In contrast to C's function, malloc does not return a pointer to the allocated heap but takes the pointer as an argument instead.

There are two reasons for this, one of which is that malloc already returns the modified heap state. This could be handled in multiple ways, e.g. by returning the modified heap state and the pointer as a pair. Another approach is to return only the modified heap state and provide a separate function for retrieving a pointer to the last allocated object on a heap.

The second and more important reason for choosing the mentioned approach is that we consider `malloc` to have two separate concerns: First it identifies an allocatable memory range, and second it modifies the heap state to contain an object allocated at this address. The presented signature of malloc allows separating these two concerns cleanly. With the proposed solution, malloc itself is only concerned with modifying the heap allocation state by adding an object at a given memory range. We provide the function $\mathrm{allocatable}_{\mathcal{H}}$ for identifying suitable memory ranges, but the use of it is not enforced and it is perfectly fine to identify allocatable memory ranges in other ways. For example, LLBMC can be configured to either use $\mathrm{allocatable}_{\mathcal{H}}$ to identify allocatable memory ranges or to use fixed addresses which do not overlap by construction.

This separating of concerns is not without its disadvantages, however. The approach makes it possible to pass invalid memory ranges to malloc, Namely, those that overlap with ranges already contained in the heap object. To prevent this, the "good" and the "bad" case need to be handled separately:

$$\forall h, p, s \left(p \neq 0 \wedge \forall (r, u) \in h \left(\mathrm{disjoint}(r, u, p, s)\right) \rightarrow \mathrm{malloc}(h, p, s) = h \cup \{(p, s)\}\right) \tag{5.5a}$$

$$\forall h, p, s \left(p = 0 \vee \exists (r, u) \in h \left(\neg\, \mathrm{disjoint}(r, u, p, s)\right) \rightarrow \mathrm{malloc}(h, p, s) = h\right) \tag{5.5b}$$

The function $\mathrm{free}_{\mathcal{H}}$ removes an object from the heap allocation state if it was previously allocated and does nothing if it was not (see [ISOC99, section 7.20.3.2]):

$$\forall h, p, q, t \left(\mathrm{free}_{\mathcal{H}}(h, p) = h \setminus \{(q, t) : q = p\}\right) \tag{5.6}$$

Deallocating a non-allocated object is undefined behavior in C, but we strongly prefer the function $\mathrm{free}_{\mathcal{H}}$ to be total. If `free` is called for a non-existent object, the only sensible thing to do is to ignore the request. Because of this, $\mathrm{free}_{\mathcal{H}}$ is a NOOP if the pointer does not point at a currently allocated object.

A memory access is either a read or a write operation. This can be a `load` or a `store` instruction but also a `memcpy` or a similar function. A memory access to a range of memory is valid if and only if an object exists that contains the range (see [ISOC99, section 6.2.4]):

$$\forall h, p, s \left(\mathrm{validaccess}_{\mathcal{H}}(h, p, s) \leftrightarrow \exists (r, u) \in h \left(\mathrm{contains}(r, u, p, s)\right)\right) \tag{5.7}$$

A deallocation is valid if the pointer to be deallocated was allocated in the heap allocation state (see [ISOC99, section 7.20.3.2]):

$$\forall h, p \left(\mathrm{validfree}_{\mathcal{H}}(h, p) \leftrightarrow \exists (r, u) \in h \left(r = p\right)\right) \tag{5.8}$$

The function $\mathrm{allocatable}_{\mathcal{H}}$ can be used to ensure that a pointer points at a suitable range of memory for allocating an object of a given size. An object is allocatable if

and only if it does not overlap with any previously allocated object and its address is not null (see [ISOC99, section 7.20.3]):

$$\forall h, p, s \left(\mathrm{allocatable}_{\mathcal{H}}(h, p, s) \leftrightarrow (p \neq 0 \wedge \forall (r, u) \in h \left(\mathrm{disjoint}(r, u, p, s)\right))\right) \quad (5.9)$$

5.3 A Partial Decision Procedure Based on Term Rewriting

In this section, we present a partial decision procedure for the theory of heap allocation as presented above. The procedure is based on the reduction of the problem to an equisatisfiable problem restricted to the theories of bitvectors and arrays.

The presented decision procedure is only partial because it does not handle uninterpreted constants of the heap allocation state sort. This is sufficient for the use in LLBMC though, as the tool is designed for whole program analysis and therefore always starts with an empty heap. All other heap states are then derived from this initial heap state by using malloc and $\mathrm{free}_{\mathcal{H}}$. If analysis of a non-main function is desired, an appropriate heap needs to be constructed explicitly.

As a second restriction, the decision procedure requires that the antecedent for equation (5.5b) (the "bad" case) is always false and the one for equation (5.5a) (the "good" case) is always true. This is ensured in LLBMC by construction of the formula. We can therefore use a simplified axiom for malloc:

$$\forall h, p, s \left(\mathrm{malloc}(h, p, s) = h \cup \{(p, s)\}\right) \quad (5.10)$$

Like in the C standard, we use the null pointer as a sentinel value. Whenever an object with specific properties is requested, the sentinel indicates that no such object exists. For this we frequently need to distinguish between the working and the failing case which are mutually exclusive. We will use the symbol $\veebar$ to indicate exclusive or, where $(x \veebar y) \leftrightarrow ((x \wedge \neg y) \vee (\neg x \wedge y))$.

The general idea for the procedure is inspired by the eager decision procedure for the theory of arrays presented in section 2.1.4. The idea is to "move" the predicates $\mathrm{validaccess}_{\mathcal{H}}$, $\mathrm{validfree}_{\mathcal{H}}$, and $\mathrm{allocatable}_{\mathcal{H}}$ over the functions $\mathrm{malloc}_{\mathcal{H}}$, $\mathrm{free}_{\mathcal{H}}$ and $\mathrm{empty}_{\mathcal{H}}$. At each such step, the corresponding part of their definition is expanded. After repeating this step exhaustively, an equisatisfiable formula is generated which only contains terms of sorts bitvector and array.

The decision procedure requires the two auxiliary functions listed in table 5.5 to be part of the term rewriting system. The function $\mathrm{container}_{\mathcal{H}}$ identifies the object containing a given range of memory. This is used for checking memory access validity. The predicate $\mathrm{overlapper}_{\mathcal{H}}$ is a counterpart to $\mathrm{container}_{\mathcal{H}}$ which identifies objects that overlap with a given memory range. Note that a memory range's container is unique if it exists at all, while a memory range might overlap with multiple objects at the same time. Because of this, $\mathrm{container}_{\mathcal{H}}$ is a function, which is easier to handle with term rewriting, while $\mathrm{overlapper}_{\mathcal{H}}$ must be a predicate.

The function $\mathrm{container}_{\mathcal{H}}(h, o, p, s)$ either returns the address of an object which

Symbol : Signature	Interpretation
$\mathrm{container}_{\mathcal{H}} : \mathcal{H} \times \mathcal{P} \times \mathcal{I} \to \mathcal{P}$	An object containing a range
$\mathrm{overlapper}_{\mathcal{H}} : \mathcal{H} \times \mathcal{P} \times \mathcal{P} \times \mathcal{I}$	An object overlapping with a range

Table 5.5: Auxiliary functions of the decision procedure for dynamic memory allocation

contains a given address range or it returns the null pointer if no such object exists.

$$\forall h, o, p, s \left(o = \mathrm{container}_{\mathcal{H}}(h, p, s) \leftrightarrow (o = 0 \veebar \exists (r, u) \in h \left(o = r \wedge \mathrm{contains}(r, u, p, s)\right))\right) \tag{5.11}$$

This function is used to decide whether an access to a memory range is valid or not.

The predicate $\mathrm{overlapper}_{\mathcal{H}}(h, o, p, s)$ is true if q is either the null pointer or q is an object in h which overlaps with the range (p, s). The predicate is false otherwise.

$$\forall h, o, p, s \left(\mathrm{overlapper}_{\mathcal{H}}(h, o, p, s) \leftrightarrow (o = 0 \veebar \exists (r, u) \in h \left(o = r \wedge \neg\, \mathrm{disjoint}(r, u, p, s)\right))\right) \tag{5.12}$$

This auxiliary predicate is used to decide allocatability, as for a range (p, s) the existence of such an overlapping object makes the range not allocatable.

For a sound and complete decision procedure based on term rewriting any possible application of any of the decision procedure's rewrite rules must produce an equisatisfiable formula. Therefore it is necessary to show for each rewrite rule $l \longrightarrow r$ that $l = r$. In the following, we will introduce the rewrite rules as well as prove equality of l and r for each.

5.3.1 Rewriting Validity Checks

We begin the presentation of the term rewriting system of the decision procedure with the six rewrite rules for $\mathrm{validaccess}_{\mathcal{H}}$ and $\mathrm{validfree}_{\mathcal{H}}$.

A memory access to an empty heap is never valid:

$$\mathrm{validaccess}_{\mathcal{H}}(\mathrm{empty}_{\mathcal{H}}, p, s) \longrightarrow F \tag{5.13}$$

Proof. For all p, s:

$$\begin{aligned} & \mathrm{validaccess}_{\mathcal{H}}(\mathrm{empty}_{\mathcal{H}}, p, s) \\ \Longleftrightarrow\ & \exists (r, u) \in \mathrm{empty}_{\mathcal{H}} \left(\mathrm{contains}(r, u, p, s)\right) && [5.7] \\ \Longleftrightarrow\ & \exists (r, u) \in \emptyset \left(\mathrm{contains}(r, u, p, s)\right) \\ \Longleftrightarrow\ & \bot \end{aligned}$$

□

A $\mathrm{validaccess}_{\mathcal{H}}$ can be "moved over" a $\mathrm{malloc}_{\mathcal{H}}$. It is either contained in the newly allocated object or it was already valid before the allocation:

$$\begin{aligned}&\mathrm{validaccess}_{\mathcal{H}}(\mathrm{malloc}_{\mathcal{H}}(h,q,t),p,s) \longrightarrow\\ &\mathrm{or}(\mathrm{contains}_{\mathcal{P},\mathcal{I}}(q,t,p,s),\mathrm{validaccess}_{\mathcal{H}}(h,p,s))\end{aligned} \tag{5.14}$$

Proof. For all h, p, s, q, t:

$$\begin{aligned}&\mathrm{validaccess}_{\mathcal{H}}(\mathrm{malloc}_{\mathcal{H}}(h,q,t),p,s)\\ \iff &\exists(r,u)\in\mathrm{malloc}_{\mathcal{H}}(h,q,t)\left(\mathrm{contains}(r,u,p,s)\right) && [5.7]\\ \iff &\exists(r,u)\in h\cup\{(q,t)\}\left(\mathrm{contains}(r,u,p,s)\right) && [5.10]\\ \iff &\exists(r,u)\in\{(q,t)\}\left(\mathrm{contains}(r,u,p,s)\right)\vee\\ &\exists(r,u)\in h\left(\mathrm{contains}(r,u,p,s)\right)\\ \iff &\mathrm{contains}(q,t,p,s)\vee\exists(r,u)\in h\left(\mathrm{contains}(r,u,p,s)\right)\\ \iff &\mathrm{contains}(q,t,p,s)\vee\mathrm{validaccess}_{\mathcal{H}}(h,p,s) && [5.10]\end{aligned}$$

□

The remaining proofs in this section follow roughly the same pattern as the proof above.

The function $\mathrm{validaccess}_{\mathcal{H}}$ cannot be "moved over" a $\mathrm{free}_{\mathcal{H}}$ as easily as $\mathrm{malloc}_{\mathcal{H}}$ in equation (5.14). This is because the $\mathrm{free}_{\mathcal{H}}$ lacks the size operand to immediately decide if the access is contained in the deallocated object. But the access can be rewritten by identifying the object which contains the access using the $\mathrm{container}_{\mathcal{H}}$ function. Consequently, a heap access is only valid if there is an object containing it:

$$\mathrm{validaccess}_{\mathcal{H}}(\mathrm{free}_{\mathcal{H}}(h,q),p,s)\longrightarrow\mathrm{ne}_{\mathcal{P}}(\mathrm{container}_{\mathcal{H}}(\mathrm{free}_{\mathcal{H}}(h,q),p,s),(0)_{\mathcal{P}}) \tag{5.15}$$

Note that we can prove a slightly more general case, as a $\mathrm{validaccess}_{\mathcal{H}}$ can always be expressed using $\mathrm{container}_{\mathcal{H}}$ and not just in the above case:

Proof. For all h, p, s:

$$\begin{aligned}&\mathrm{validaccess}_{\mathcal{H}}(h,p,s)\\ \iff &\exists(r,u)\in h\left(\mathrm{contains}(r,u,p,s)\right) && [5.7]\\ \iff &\exists(r,u)\in h\left(\exists o\left((o=r)\wedge\mathrm{contains}(r,u,p,s)\right)\right)\\ \iff &\exists o\left(\exists(r,u)\in h\left(o=r\wedge\mathrm{contains}(r,u,p,s)\right)\right)\\ \iff &\exists o\left((r=0\veebar\exists(r,u)\in h\left(o=r\wedge\mathrm{contains}(r,u,p,s)\right))\wedge o\neq 0\right)\\ \iff &\exists o\left(o=\mathrm{container}_{\mathcal{H}}(h,p,s)\wedge o\neq 0\right) && [5.11]\\ \iff &\exists o\left(\mathrm{container}_{\mathcal{H}}(h,p,s)\neq 0\right)\\ \iff &\mathrm{container}_{\mathcal{H}}(h,p,s)\neq 0\end{aligned}$$

□

It is trivial to "move" a $\mathrm{validfree}_{\mathcal{H}}$ over an empty heap because no object exists in an empty heap:

$$\mathrm{validfree}_{\mathcal{H}}(\mathrm{empty}_{\mathcal{H}},p)\longrightarrow F \tag{5.16}$$

Proof. For all p:

$$\begin{aligned}
&\mathrm{validfree}_{\mathcal{H}}(\mathrm{empty}_{\mathcal{H}}, p) \\
\iff &\exists (r,u) \in \mathrm{empty}_{\mathcal{H}} \left(r = p\right) && [5.8] \\
\iff &\exists (r,u) \in \emptyset \left(r = p\right) && [5.4] \\
\iff &\bot
\end{aligned}$$

□

A free is valid if and only if the object was previously allocated:

$$\mathrm{validfree}_{\mathcal{H}}(\mathrm{malloc}_{\mathcal{H}}(h,q,t),p) \longrightarrow \mathrm{or}(\mathrm{validfree}_{\mathcal{H}}(h,p), \mathrm{eq}_{\mathcal{P}}(q,p)) \tag{5.17}$$

Proof. For all h, p, s, q, t:

$$\begin{aligned}
&\mathrm{validfree}_{\mathcal{H}}(\mathrm{malloc}_{\mathcal{H}}(h,q,t),p) \\
\iff &\exists (r,u) \in \mathrm{malloc}_{\mathcal{H}}(h,q,t) \left(r = p\right) && [5.8] \\
\iff &\exists (r,u) \in h \cup \{(q,t)\} \left(r = p\right) && [5.10] \\
\iff &\exists (r,u) \in h \left(r = p\right) \vee \exists (r,u) \in \{(q,t)\} \left(r = q\right) \\
\iff &\exists (r,u) \in h \left(r = p\right) \vee q = p \\
\iff &\mathrm{validfree}_{\mathcal{H}}(h,p) \vee q = p && [5.8]
\end{aligned}$$

□

A free cannot be valid if the pointer was already freed before (this is called a double-free). The following rewrite rule reflects this:

$$\mathrm{validfree}_{\mathcal{H}}(\mathrm{free}_{\mathcal{H}}(h,q),p,s) \longrightarrow \mathrm{and}(\mathrm{validfree}_{\mathcal{H}}(h,p,s), \mathrm{ne}_{\mathcal{P}}(q,p)) \tag{5.18}$$

Proof. For all h, p, s, q:

$$\begin{aligned}
&\mathrm{validfree}_{\mathcal{H}}(\mathrm{free}_{\mathcal{H}}(h,p),q) \\
\iff &\exists (r,u) \in \mathrm{free}_{\mathcal{H}}(h,p) \left(r = q\right) && [5.8] \\
\iff &\exists (r,u) \in h \setminus \{(r',u') : r' = p\} \left(r = q\right) && [5.6] \\
\iff &\exists (r,u) \in h \left(r = q \wedge r \neq p\right) \\
\iff &\exists (r,u) \in h \left(r = q \wedge q \neq p\right) \\
\iff &\exists (r,u) \in h \left(r = q\right) \wedge q \neq p \\
\iff &\mathrm{validfree}_{\mathcal{H}}(h,p,s) \wedge q \neq p && [5.6]
\end{aligned}$$

□

5.3.2 Rewriting the Auxiliary Functions

Like with the previously presented rewrite rules, for the two auxiliary functions $\mathrm{container}_{\mathcal{H}}$ and $\mathrm{overlapper}_{\mathcal{H}}$ a rule is required for moving each of these two functions

over every function of sort heap, so that no such function remains in the rewritten formula.

The function $\mathrm{container}_{\mathcal{H}}$ needs to be rewritten, too. Obviously, an empty heap does not contain any objects:

$$\mathrm{container}_{\mathcal{H}}(\mathrm{empty}_{\mathcal{H}}, p, s) \longrightarrow (0)_{\mathcal{P}} \tag{5.19}$$

Proof. for all o, p, s:

$$\begin{aligned}
& o = \mathrm{container}_{\mathcal{H}}(\mathrm{empty}_{\mathcal{H}}, p, s) \\
\iff & o = 0 \veebar \exists (r,u) \in \mathrm{empty}_{\mathcal{H}} \left(o = r \wedge \mathrm{contains}(r,u,p,s)\right) && [5.11] \\
\iff & o = 0 \veebar \exists (r,u) \in \emptyset \left(o = r \wedge \mathrm{contains}(r,u,p,s)\right) && [5.4] \\
\iff & o = 0 \veebar \bot \\
\iff & o = 0
\end{aligned}$$

□

We defined the function select in section 3.7 in equation (3.9).

Lemma 5.1. *For all d, c, a, b:*

$$d = \mathrm{select}(c,a,b) \iff (c \wedge (d = a)) \veebar (\neg c \wedge (d = b)) \tag{5.20}$$

The proof of this lemma is trivial and therefore omitted. Using this lemma, we can now prove the next rewrite rule for moving $\mathrm{container}_{\mathcal{H}}$ over $\mathrm{malloc}_{\mathcal{H}}$:

$$\begin{aligned}
& \mathrm{container}_{\mathcal{H}}(\mathrm{malloc}_{\mathcal{H}}(h,q,t),p,s) \longrightarrow \\
& \mathrm{select}_{\mathcal{P}}(\mathrm{contains}_{\mathcal{P},\mathcal{I}}(q,t,p,s),p,\mathrm{container}_{\mathcal{H}}(h,p,s))
\end{aligned} \tag{5.21}$$

Note that the proof uses the fact that objects on a heap do ot overlap.

Proof. For all h, p, s, q, t, o:

$$\begin{aligned}
& o = \mathrm{container}_{\mathcal{H}}(\mathrm{malloc}_{\mathcal{H}}(h,q,t),p,s) \\
\iff & o = 0 \veebar \exists (r,u) \in \mathrm{malloc}_{\mathcal{H}}(h,q,t) \left(o = r \wedge \mathrm{contains}(r,u,p,s)\right) && [5.11] \\
\iff & o = 0 \veebar \exists (r,u) \in h \cup \{q,t\} \left(o = r \wedge \mathrm{contains}(r,u,p,s)\right) && [5.10] \\
\iff & o = 0 \veebar \exists (r,u) \in h \left(o = r \wedge \mathrm{contains}(r,u,p,s)\right) \veebar \\
& \quad \exists (r,u) \in \{q,t\} \left(o = r \wedge \mathrm{contains}(r,u,p,s)\right) \\
\iff & o = 0 \veebar \exists (r,u) \in h \left(o = r \wedge \mathrm{contains}(r,u,p,s)\right) \veebar \\
& \quad (o = q \wedge \mathrm{contains}(q,t,p,s)) \\
\iff & o = \mathrm{container}_{\mathcal{H}}(h,p,s) \veebar (o = q \wedge \mathrm{contains}(q,t,p,s)) && [5.11] \\
\iff & (o = \mathrm{container}_{\mathcal{H}}(h,p,s) \wedge \neg\, \mathrm{contains}(q,t,p,s)) \vee \\
& \quad (o = q \wedge \mathrm{contains}(q,t,p,s)) \\
\iff & o = \mathrm{select}(\mathrm{contains}(q,t,p,s),q,\mathrm{container}_{\mathcal{H}}(h,p,s)) && [5.20]
\end{aligned}$$

□

The function $\mathrm{container}_{\mathcal{H}}$ similarly excludes deallocated objects:

$$\begin{aligned}&\mathrm{container}_{\mathcal{H}}(\mathrm{free}_{\mathcal{H}}(h,q),p,s) \longrightarrow\\ &\mathrm{select}_{\mathcal{P}}(q = \mathrm{container}_{\mathcal{H}}(h,p,s),(0)_{\mathcal{P}},\mathrm{container}_{\mathcal{H}}(h,p,s))\end{aligned} \tag{5.22}$$

Proof. For all h, p, s, q, r:

$$\begin{aligned}
& o = \mathrm{container}_{\mathcal{H}}(\mathrm{free}_{\mathcal{H}}(h,q),p,s)\\
\iff & o = 0 \veebar \exists (r,u) \in \mathrm{free}_{\mathcal{H}}(h,q)\,\big(o = r \wedge \mathrm{contains}(r,u,p,s)\big) && [5.11]\\
\iff & o = 0 \veebar \exists (r,u) \in h \setminus \{(r',u') : r' = q\}\,\big(o = r \wedge \mathrm{contains}(r,u,p,s)\big) && [5.6]\\
\iff & o = 0 \veebar (\exists (r,u) \in h\,\big(o = r \wedge \mathrm{contains}(r,u,p,s) \wedge r \neq q\big))\\
\iff & o = 0 \veebar (\exists (r,u) \in h\,\big(o = r \wedge \mathrm{contains}(r,u,p,s) \wedge o \neq q\big))\\
\iff & o = 0 \veebar (\exists (r,u) \in h\,\big(o = r \wedge \mathrm{contains}(r,u,p,s)\big) \wedge o \neq q)\\
\iff & (\mathrm{container}_{\mathcal{H}}(h,p,s) = q \wedge o = 0) \vee\\
& (\mathrm{container}_{\mathcal{H}}(h,p,s) \neq q \wedge o = \mathrm{container}_{\mathcal{H}}(h,p,s))\\
\iff & o = \mathrm{select}(\mathrm{container}_{\mathcal{H}}(h,p,s) = q, 0, \mathrm{container}_{\mathcal{H}}(h,p,s))
\end{aligned}$$

□

An empty heap, which is commonly used to model the heap state at the beginning of a program's execution, can obviously not contain an object that overlaps with a given range:

$$\mathrm{overlapper}_{\mathcal{H}}(\mathrm{empty}_{\mathcal{H}},r,p,s) \longrightarrow \bot \tag{5.23}$$

Proof. For all r, p, s:

$$\begin{aligned}
& \mathrm{overlapper}_{\mathcal{H}}(\mathrm{empty}_{\mathcal{H}},o,p,s)\\
\iff & \exists (r,u \in \mathrm{empty}_{\mathcal{H}})\,\big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s)\big) && [5.12]\\
\iff & \exists (r,u) \in \emptyset\,\big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s)\big) && [5.4]\\
\iff & \bot
\end{aligned}$$

□

An object q overlapping with a range (p,s) in a heap h created by a $\mathrm{malloc}_{\mathcal{H}}$ can only exist if it either was allocated by this $\mathrm{malloc}_{\mathcal{H}}$ or if it was already part of the heap before:

$$\begin{aligned}&\mathrm{overlapper}_{\mathcal{H}}(\mathrm{malloc}_{\mathcal{H}}(h,q,t),o,p,s) \longrightarrow\\ &\mathrm{or}(\mathrm{and}(\mathrm{eq}_{\mathcal{P}}(q,o),\neg\,\mathrm{disjoint}_{\mathcal{P},\mathcal{I}}(q,t,p,s)),\mathrm{overlapper}_{\mathcal{H}}(h,o,p,s))\end{aligned} \tag{5.24}$$

Proof. For all h, q, t, o, p, s:

$$\begin{aligned}
& \mathrm{overlapper}_{\mathcal{H}}(\mathrm{malloc}_{\mathcal{H}}(h,q,t),o,p,s)\\
\iff & \exists (r,u) \in \mathrm{malloc}_{\mathcal{H}}(h,q,t)\,\big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s)\big) && [5.12]\\
\iff & \exists (r,u) \in h \cup \{(q,t)\}\,\big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s)\big) && [5.10]
\end{aligned}$$

$$
\begin{aligned}
&\iff \exists (r,u) \in \{(q,t)\}\ \big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s)\big) \vee \\
&\quad\ \exists (r,u) \in h\ \big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s)\big) \\
&\iff \exists (r,u) \in \{(q,t)\}\ \big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s)\big) \vee \\
&\quad\ \mathrm{overlapper}_{\mathcal{H}}(h,o,p,s) && [5.12] \\
&\iff (q = o \wedge \neg\,\mathrm{disjoint}(q,t,p,s)) \vee \mathrm{overlapper}_{\mathcal{H}}(h,o,p,s)
\end{aligned}
$$

□

An object q in the heap state h can only contain a range (p, s) if the object was not previously deallocated:

$$
\mathrm{overlapper}_{\mathcal{H}}(\mathrm{free}_{\mathcal{H}}(h,q),o,p,s) \longrightarrow \mathrm{and}(\mathrm{overlapper}_{\mathcal{H}}(h,o,p,s), \mathrm{ne}_{\mathcal{P}}(o,q)) \tag{5.25}
$$

Proof. For all h, q, o, p, s:

$$
\begin{aligned}
&\quad \mathrm{overlapper}_{\mathcal{H}}(\mathrm{free}_{\mathcal{H}}(h,q),o,p,s) \\
&\iff \exists (r,u) \in \mathrm{free}_{\mathcal{H}}(h,q)\ \big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s)\big) && [5.12] \\
&\iff \exists (r,u) \in h \setminus \{(r',u') : r' = q\}\ \big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s))\big) && [5.6] \\
&\iff \exists (r,u) \in h\ \big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s) \wedge r \neq q\big) \\
&\iff \exists (r,u) \in h\ \big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s) \wedge o \neq q\big) \\
&\iff \exists (r,u) \in h\ \big(o = r \wedge \neg\,\mathrm{disjoint}(r,u,p,s)\big) \wedge o \neq q \\
&\iff \mathrm{overlapper}_{\mathcal{H}}(h,o,p,s) \wedge o \neq q && [5.12]
\end{aligned}
$$

□

This concludes the theory's rewrite rules and the related proofs. The previously mentioned case of underspecification of the $\mathrm{malloc}_{\mathcal{H}}$ axiom and the related rewrite rule still needs to be handled.

5.3.3 Allocatability

Memory allocation of a range (p, s) via the function $\mathrm{malloc}_{\mathcal{H}}$ can only be successful if $p \neq 0 \wedge \forall (r,u) \in h\,\big(\mathrm{disjoint}(r,u,p,s)\big)$. This is exactly the definition of $\mathrm{allocatable}_{\mathcal{H}}(h,p,s)$, so obviously adding the constraint $\mathrm{allocatable}_{\mathcal{H}}(h,p,s)$ for every $\mathrm{malloc}_{\mathcal{H}}(h,p,s)$ ensures that the axiom equation (5.5a) applies for all $\mathrm{malloc}_{\mathcal{H}}(h,p,s)$ and axiom equation (5.5b) is never needed.

Note that for a given size s, an allocatable object might not exist because no sufficiently large, contiguous range of memory is available. $\mathrm{allocatable}_{\mathcal{H}}(h,p,s)$ is unsatisfiable in this case and, because the negation of this term is added to the following instruction's execution condition, any instruction after this allocation is unreachable. This makes the code after the `malloc` dead code. LLBMC is currently not able to detect when this happens.

Obviously, $\mathrm{allocatable}_{\mathcal{H}}$ needs to be rewritten, too. The rule for this is as follows:

$$
\mathrm{allocatable}_{\mathcal{H}}(h,p,s) \longrightarrow \mathrm{and}(\mathrm{overlapper}_{\mathcal{H}}(h,o_i,p,s), \mathrm{and}(o_i = 0, \mathrm{ne}_{\mathcal{P}}(p,(0)_{\mathcal{P}}))) \tag{5.26}
$$

where o_i is a previously uninterpreted pointer constant. Both sides of the rewrite rules are equivalent, if we assume that the o_i was part of the formula from the beginning. This is possible because the structure of the formula ensures that only a finite number of o_i will be introduced during rewriting.

Proof. For all h, p, s, and o_i:

$$
\begin{aligned}
& \mathrm{allocatable}_{\mathcal{H}}(h, p, s) \\
\iff & p \neq 0 \wedge \forall (r, u) \in h \left(\mathrm{disjoint}(r, u, p, s)\right) && \text{[5.9]} \\
\iff & p \neq 0 \wedge \neg \exists (r, u) \in h \left(\neg\, \mathrm{disjoint}(r, u, p, s)\right) \\
\iff & ((\exists (r, u) \in h \left(q_i = r \wedge \neg\, \mathrm{disjoint}(r, u, p, s)\right) \wedge o_i \neq 0) \vee \\
& (\neg \exists (r, u) \in h \left(q_i = r \wedge \neg\, \mathrm{disjoint}(r, u, p, s)\right) \wedge o_i = 0)) \wedge o_i = 0 \wedge p \neq 0 \\
\iff & (\exists (r, u) \in h \left(q_i = r \wedge \neg\, \mathrm{disjoint}(r, u, p, s)\right) \veebar o_i = 0) \wedge o_i = 0 \wedge p \neq 0 \\
\iff & \mathrm{overlapper}_{\mathcal{H}}(h, o_i, p, s) \wedge o_i = 0 \wedge p \neq 0 && \text{[5.12]}
\end{aligned}
$$

□

5.3.4 Summary

The rewrite system given by equation (5.1) and equations (5.13) to (5.19) and (5.21) to (5.26) is a decision procedure for a fragment of the theory of dynamic memory allocation presented above. Exhaustive application of these rewrite rules results in an equisatisfiable formula which does not contain any of the function symbols $\mathrm{validaccess}_{\mathcal{H}}$, $\mathrm{validfree}_{\mathcal{H}}$, $\mathrm{allocatable}_{\mathcal{H}}$, $\mathrm{malloc}_{\mathcal{H}}$, $\mathrm{free}_{\mathcal{H}}$, and $\mathrm{empty}_{\mathcal{H}}$. Satisfiability of the resulting formula can then be checked by an off-the-shelf SMT solver for the quantifier free logic of bitvectors and arrays. The decision procedure is only partial because it currently does not support uninterpreted constants of the sort heap allocation state.

In the future, extending the decision procedure to support equality of heaps would open up further possibilities for checking heap allocation related function safety properties as well as modifications to LLBMC's core model checking algorithm. In particular the former would greatly benefit if LLBMC had support for uninterpreted heap constants, which currently only exists in prototypical form.

5.4 Encoding Dynamic Memory Allocation

Function symbols related to heap memory management are shown in table 5.6. In this section, the symbol h represents the appropriate heap sort for the program's architecture.

Propagation of the heap allocation state through the program is done nearly identical to that of the memory state in section 4.4:

$$\overleftarrow{\rho}^{\mathsf{P}}_h(c, p) \longrightarrow \mathrm{empty}_t \tag{5.27a}$$

$$\overrightarrow{\rho}^{\mathsf{P}}_h(c, p) \longrightarrow \overrightarrow{\rho}^{\mathsf{F}}(c, f)\ ;\ \mathsf{f\ is\ main} \tag{5.27b}$$

Symbol : Signature	Interpretation
$\overleftarrow{\rho}^{\mathtt{B}}_t : \mathtt{C} \times \mathtt{B} \to t$	Heap state before a basic block
$\overrightarrow{\rho}^{\mathtt{B}}_t : \mathtt{C} \times \mathtt{B} \to t$	Heap state after a basic block
$\overleftarrow{\rho}^{\mathtt{F}}_t : \mathtt{C} \times \mathtt{F} \to t$	Heap state before a function
$\overrightarrow{\rho}^{\mathtt{F}}_t : \mathtt{C} \times \mathtt{F} \to t$	Heap state after a function
$\overleftarrow{\rho}^{\mathtt{I}}_t : \mathtt{C} \times \mathtt{I} \to t$	Heap state before an instruction
$\overrightarrow{\rho}^{\mathtt{I}}_t : \mathtt{C} \times \mathtt{I} \to t$	Heap state after an instruction
$\overleftarrow{\rho}^{\mathtt{P}}_t : \mathtt{C} \times \mathtt{P} \to t$	Heap state before a program
$\overrightarrow{\rho}^{\mathtt{P}}_t : \mathtt{C} \times \mathtt{P} \to t$	Heap state after a program
$\rho^{\mathtt{J}}_t : \mathtt{C} \times \mathtt{B} \times \mathtt{B} \to t$	Heap state at a jump

Table 5.6: Functions encoding heap state

$$\overleftarrow{\rho}^{\mathtt{F}}_h(c,f) \longrightarrow \overleftarrow{\rho}^{\mathtt{I}}_h(c',i)\ ;\ c' = \mathrm{parent}(c) \wedge i = \mathrm{callsite}(c) \tag{5.27c}$$

$$\overleftarrow{\rho}^{\mathtt{F}}_h(c,f) \longrightarrow \overleftarrow{\rho}^{\mathtt{P}}_h(c,p)\ ;\ \exists c'(c' = parent(c)) \tag{5.27d}$$

$$\begin{aligned}\overrightarrow{\rho}^{\mathtt{F}}_m(c,f) \longrightarrow\ & \phi_{[\![\mathtt{t}]\!]}(a_1,b_1,\ldots,a_{|R|},b_{|R|})\ ; \\ & R = \{a_i,b_i : a_i = \overrightarrow{\rho}^{\mathtt{I}}_m(c,j) \wedge b_i = \overrightarrow{\eta}^{\mathtt{I}}(c,j), \wedge \\ & \qquad j \sim [\![\texttt{<j> = ret <t1> <v>}]\!]\}\end{aligned} \tag{5.27e}$$

$$\begin{aligned}\overleftarrow{\rho}^{\mathtt{B}}_h(c,b) \longrightarrow\ & \phi(\overrightarrow{\rho}^{\mathtt{B}}_h(c,b_1),\eta^{\mathtt{J}}(c,b_1,b),\ldots,\overrightarrow{\rho}^{\mathtt{B}}_h(c,b_{|P|}),\eta^{\mathtt{J}}(c,b_{|P|},b))\ ; \\ & \neg\,\mathrm{entry}(b) \wedge P = \{b_i : \mathrm{succ}_{\mathtt{F}}(b_i,b)\}\end{aligned} \tag{5.27f}$$

$$\overleftarrow{\rho}^{\mathtt{B}}_h(c,b) \longrightarrow \overleftarrow{\rho}^{\mathtt{F}}(c,f)\ ;\ \mathrm{entry}(b) \tag{5.27g}$$

$$\overrightarrow{\rho}^{\mathtt{B}}_h(c,b) \longrightarrow \overrightarrow{\rho}^{\mathtt{I}}_h(c,i)\ ;\ i = \mathrm{term}_{\mathtt{B}}(b) \tag{5.27h}$$

$$\overleftarrow{\rho}^{\mathtt{I}}_h(c,i) \longrightarrow \overleftarrow{\rho}^{\mathtt{B}}_h(c,b)\ ;\ i = \mathrm{first}_{\mathtt{B}}(b) \tag{5.27i}$$

$$\overleftarrow{\rho}^{\mathtt{I}}_h(c,i) \longrightarrow \overrightarrow{\rho}^{\mathtt{I}}_h(c,i_1)\ ;\ \mathrm{succ}_{\mathtt{B}}(i_1,i) \tag{5.27j}$$

$$\begin{aligned}\overrightarrow{\rho}^{\mathtt{I}}_h(c,i) \longrightarrow\ & \overrightarrow{\rho}^{\mathtt{F}}_h(c_f,f)\ ; \\ & c = \mathrm{parent}(c_f) \wedge i = \mathrm{callsite}(c_f) \wedge \\ & i \sim [\![\texttt{<i> = call <t> <fptr>({<ty> <arg>}*)}]\!]\end{aligned} \tag{5.27k}$$

Memory allocation is treated as two separate operations that need to be done. These operations are the identification of a suitable pointer for the object which is to be allocated and the respective modification of the heap allocation state. These two effects are implemented by the rewrite rules in equation (5.28). In these rules $p_{c,i}$ is an uninterpreted constant of sort i8*, which is unique to this context-instruction pair (c,i), and represents the newly generated pointer. Equation (5.28b) then asserts that the pointer does not overlap with existing objects and is not null, while equation (5.28a) updates the heap allocation state accordingly.

$$\begin{aligned}\overrightarrow{\rho}^{\mathtt{I}}_h(c,i) \longrightarrow\ & \mathrm{malloc}_h(\overleftarrow{\rho}^{\mathtt{I}}_h(c,i),p_{c,i},\varepsilon_{[\![\mathtt{t1}]\!]}(c,[\![\mathtt{s}]\!]))\ ; \\ & i \sim [\![\texttt{<res> = call <t> malloc(<t1> <s>)}]\!]\end{aligned} \tag{5.28a}$$

$$\begin{aligned}\overrightarrow{\eta}^{\mathtt{I}}(c,i) \longrightarrow\ & \mathrm{and}(\overleftarrow{\eta}^{\mathtt{I}}(c,i),\mathrm{allocatable}_h(\overleftarrow{\rho}^{\mathtt{I}}_h(c,i),p_{c,i},\varepsilon^{\mathtt{V}}_{[\![\mathtt{t1}]\!]}(c,[\![\mathtt{s}]\!])))\ ; \\ & i \sim [\![\texttt{<res> = call <t> malloc(<t1> <s>)}]\!]\end{aligned} \tag{5.28b}$$

Note that additional caution must be taken when placing objects with dynamic object duration. It is necessary to ensure that heap allocated objects do not overlap with automatically allocated objects or statically allocated objects. To ensure that this never happens, objects can be placed in separate segments of the address space separated by their storage duration. This closely follows most real computer architectures, so can likely be considered an acceptable under-approximation. The heap segment can then be defined by its lower bound $\mathrm{heapbot}^a$ and its upper bound $\mathrm{heaptop}^a$:

$$\begin{aligned}\vec{\eta}^{\mathtt{I}}(c,i) \longrightarrow\ & \mathrm{and}(\mathrm{and}(\overleftarrow{\eta}^{\mathtt{I}}(c,i), \mathrm{allocatable}_h(\overleftarrow{\rho}_h^{\mathtt{I}}(c,i), p_{c,i}, \varepsilon^{\mathtt{V}}_{[\![\mathtt{t1}]\!]}(c,[\![\mathtt{s}]\!]))),\\ & \mathrm{and}(\mathrm{le}^u((\mathrm{heapbot}^a)_{\mathcal{P}}, p_{c,i}), \mathrm{le}^u(p_{c,i}, (\mathrm{heaptop}^a)_{\mathcal{P}}))) ;\\ & i \sim [\![\texttt{<res> = call <t> malloc(<t1> <s>)}]\!] \end{aligned} \tag{5.29}$$

Furthermore, in the case of bitvectors, wraparound must be taken into account. For this it is sufficient to ensure that every object and every memory access does not wrap around, e.g. by adding suitable constraints using the addo^u function.

Note that equation (5.28b) is not necessary if other means ascertain that objects do not overlap. This could be done by placing each object at a specific address, e.g. by assigning appropriate constants to each context-instruction pair (c,i) if the instruction i is a call to the `malloc` function and its size is known. Care must be taken to ensure that these objects do not overlap with objects that were not placed at fixed addresses, either by placing them in separate regions of the address space or by adding appropriate constraints.

A memory deallocation is handled rather straightforwardly:

$$\begin{aligned}\vec{\rho}_h^{\mathtt{I}}(c,i) \longrightarrow\ & \mathrm{free}_h(\overleftarrow{\rho}_h^{\mathtt{I}}(c,i), \varepsilon_{[\![\mathtt{t1}]\!]}([\![\mathtt{p}]\!])) ;\\ & i \sim [\![\texttt{<res> = call <t> free(<t1> <p>)}]\!] \end{aligned} \tag{5.30a}$$

5.5 Encoding Memory Access Safety

Memory accesses are correct if they either go to a global variable, a stack allocated variable, or a heap allocated variable. This leads to the following extension of the safety property:

We first define the helper functions $\sigma^{\mathtt{I}}_{stack}$, $\sigma^{\mathtt{I}}_{global}$, and $\sigma^{\mathtt{I}}_{heap}$ which are not part of ILR but immediately rewritten according to the following rewrite rules:

$$\begin{aligned}\sigma^{\mathtt{I}}_{stack}(c,i) \longrightarrow\ & \mathrm{contains}(\overleftarrow{\tau}^{\mathtt{I}}_s(c,i), \mathrm{sub}_{\mathcal{I}}((\mathrm{stacktop}^a)_{\mathcal{P}}, \overleftarrow{\tau}^{\mathtt{I}}_s(c,i)), [\![\mathtt{p}]\!], (|[\![\mathtt{t}]\!]|)_{\mathcal{I}}) ;\\ & i \sim [\![\texttt{<i> = load <t>, <t>* <p>}]\!] \end{aligned} \tag{5.31}$$

$$\begin{aligned}\sigma^{\mathtt{I}}_{stack}(c,i) \longrightarrow\ & \mathrm{contains}(\overleftarrow{\tau}^{\mathtt{I}}_s(c,i), \mathrm{sub}_{\mathcal{I}}((\mathrm{stacktop}^a)_{\mathcal{P}}, \overleftarrow{\tau}^{\mathtt{I}}_s(c,i)), [\![\mathtt{p}]\!], (|[\![\mathtt{t}]\!]|)_{\mathcal{I}}) ;\\ & i \sim [\![\texttt{store <t> <v>, <t>* <p>}]\!] \end{aligned} \tag{5.32}$$

$$\begin{aligned}\sigma^{\mathtt{I}}_{global}(c,i) \longrightarrow\ & \bigvee_{g \in G_p} \mathrm{contains}(\varepsilon^{\mathtt{G}}(g), (|g|)_{\mathcal{I}}, [\![\mathtt{p}]\!], (|[\![\mathtt{t}]\!]|)_{\mathcal{I}}) ;\\ & i \sim [\![\texttt{<i> = load <t>, <t>* <p>}]\!] \end{aligned} \tag{5.33}$$

$$\sigma^{\mathtt{I}}_{global}(c,i) \longrightarrow \bigvee_{g \in G_p} \mathrm{contains}(\varepsilon^{\mathtt{G}}(g), (|g|)_{\mathcal{I}}, [\![\mathtt{p}]\!], (|[\![\mathtt{t}]\!]|)_{\mathcal{I}}) \,; \quad i \sim [\![\texttt{store <t> <v>, <t>* <p>}]\!] \tag{5.34}$$

$$\sigma^{\mathtt{I}}_{heap}(c,i) \longrightarrow \mathrm{validaccess}_{\mathcal{H}}(\overleftarrow{\rho}^{\mathtt{I}}(c,i), [\![\mathtt{p}]\!], (|[\![\mathtt{t}]\!]|)_{\mathcal{I}}) \,; \quad i \sim [\![\texttt{<i> = load <t>, <t>* <p>}]\!] \tag{5.35}$$

$$\sigma^{\mathtt{I}}_{heap}(c,i) \longrightarrow \mathrm{validaccess}_{\mathcal{H}}(\overleftarrow{\rho}^{\mathtt{I}}(c,i), [\![\mathtt{p}]\!], (|[\![\mathtt{t}]\!]|)_{\mathcal{I}}) \,; \quad i \sim [\![\texttt{store <t> <v>, <t>* <p>}]\!] \tag{5.36}$$

with $|I| = |P|$.

Note that this modeling is simplified, as it does not model the storing of a call's return address on the stack and therefore cannot detect accidental (or malicious) modification of this address.

This leads to the following adaptation of $\sigma^{\mathtt{I}}$:

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow \mathrm{or}(\mathrm{or}(\sigma^{\mathtt{I}}_{stack}(c,i), \sigma^{\mathtt{I}}_{global}(c,i)), \sigma^{\mathtt{I}}_{heap}(c,i)) \,; \quad i \sim [\![\texttt{<i> = load <t>, <t>* <p>}]\!] \tag{5.37}$$

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow \mathrm{or}(\mathrm{or}(\sigma^{\mathtt{I}}_{stack}(c,i), \sigma^{\mathtt{I}}_{global}(c,i)), \sigma^{\mathtt{I}}_{heap}(c,i)) \,; \quad i \sim [\![\texttt{store <t> <v>, <t>* <p>}]\!] \tag{5.38}$$

A call to the function `free` can fail if the passed-in pointer was not previously allocated and the pointer is not null. The first part is handled by the function $\mathrm{validfree}_{\mathcal{H}}$ presented above:

$$\sigma^{\mathtt{I}}(c,i) \longrightarrow \mathrm{or}(\mathrm{validfree}_h(\overleftarrow{\rho}^{\mathtt{I}}_h(c,i), \varepsilon_{[\![\mathtt{t1}]\!]}([\![\mathtt{p}]\!])), \mathrm{eq}_{[\![\mathtt{t1}]\!]}(\varepsilon_{[\![\mathtt{t1}]\!]}[\![\mathtt{p}]\!], (0)_{\mathcal{P}})) \,; \quad i \sim [\![\texttt{<res> = call <t> free(<t1> <p>)}]\!] \tag{5.39}$$

Note that it is not necessary to modify $\sigma^{\mathtt{I}}$ for a call to `malloc`, as this function cannot cause undefined behavior. According to the ISO C'99 standard's draft [ISOC99, section 7.20.3], a null pointer is returned if the space cannot be allocated. If the requested size is zero, behavior is implementation defined. Either a null pointer is returned or behavior is as if the requested size was non-zero, but the returned pointer shall not be used to access an object. Nonetheless, if the case of malloc returning a null pointer is not explicitly handled, an error might occur later on during execution. To model this, the return value of `malloc` could be set non-deterministically to null or to a pointer to a newly allocated object.

5.6 Summary and Outlook

In this chapter, we introduced the theory of heap memory allocation as well as a partial decision procedure for it. The theory's strength is its closeness to the C standard's definition. For many cases, however, in particular when type safety is not violated, a higher-level model could encode most relevant properties more efficiently, though not as precisely. For example, using type information can improve the tool's performance, as was observed in [Coh+09b].

Furthermore, C not only defines memory accesses as undefined behavior, but in certain situations also pointer arithmetics. For example, it is valid to calculate a pointer which points at an element one past the end of an array but it is not allowed to dereference it. In contrast, it is undefined behavior to even calculate a pointer more than one past the end of an array. For a complete safety analysis of memory accesses in C, these potential sources of undefined behavior need to be supported, too. However, at the time of writing this is still future work. Furthermore future work is the support for heap equality and uninterpreted heap constants, which are a requirement for many modularization approaches.

Chapter 6

Simplification, Satisfiability Solving, and Evaluation

This chapter discusses three separate but related topics, formula simplification, satisfiability solving, and a brief evaluation of LLBMC's performance.

Formula simplification in LLBMC serves two purposes. Most simplification rules are designed to improve the SMT solver's performance, e.g. by replacing terms with equivalent terms which are less costly for the SMT solver. A second, smaller set of simplification rules is designed to reduce the complexity of translating ILR formulæ to SMT formulæ. This is done by replacing ILR functions that cannot be translated straightforwardly to SMT by equivalent ones that can. Reduction by simplification is important to keep the translation to SMT itself simple.

The second topic is satisfiability solving of ILR formulæ using off-the-shelf SMT solvers. This part is primarily concerned with the translation of ILR formulæ to SMT-LIB formulæ. This requires simplifications for reducing ILR functions which are not supported by SMT-LIB and translation of the remaining functions to SMT-LIB counterparts. Furthermore, this part shortly discusses how the SMT solver can be driven to detect violations of multiple safety properties with a single SMT solver process by using an SMT solvers incremental solving feature.

Finally, this part also gives a short evaluation of LLBMC's performance by discussing its result in the International Software Verification Competitions as well as showing the results of running LLBMC on a small example.

6.1 Simplifications

Formula simplification reduces the complexity of a formula by replacing subterms (resp. subformulæ) with equivalent terms (resp. formulæ), which are known to be cheaper for the SMT solver. Simplification, and in particular constant propagation, is particularly effective for software bounded model checking because many values that are hard to track in the general case are often constants in each individual context. This opens up many opportunities for constant propagation throughout

the formula. While most state-of-the-art SMT solvers already apply simplifications themselves, simplification on the ILR level is still important to reduce the size of the formula as early as possible during the different stages in LLBMC to avoid an intermediate blow-up of the formula size. A subset of the simplifications presented in this chapter have been presented by Sinha [Sin08].

Because of the large number of simplification rules provided in LLBMC, these rules are listed in appendix A. Rules of particular interest are shown in this section, too, but with the rule's number set in square brackets to indicate that this is not the rule's definition but merely a reference to a rule defined somewhere else, e.g.

$$\mathrm{and}(T, T) \longrightarrow T \qquad \text{[A.1a]}$$

refers to equation (A.1a) defined in appendix A.

6.1.1 Constant Propagation Rules

Constant propagation, similar to the compiler optimization of the same name, replaces functions with constant arguments by the result of evaluation, which is again a constant.[1] Formulæ generated by LLBMC's encoding typically contain many opportunities for constant propagation and making use of these opportunities can greatly reduce the size of the formula passed to the SMT solver.

Consider the example in listing 6.1. The code contains a loop which is executed four times and reads a value from an array each time. Unrolling of the loop and subsequent optimization of the code results in four copies of the loop's body, as is shown in listing 6.2. Note how optimization causes the length of each copy of the loop's body to be reduced from eight instructions to three instructions. In particular, everything related to the loop counter variable `r` was optimized out as well as the `icmp` instructions. This shows how constant propagation is particularly effective in reducing code size after unrolling loops. For the same reasons, this is also true for function inlining, which is closely related to loop unrolling. However, LLBMC's use of an explicit call graph makes the compiler unable to take advantage of these optimization opportunities in some cases because part of the necessary information is only present in the call graph's data structures, which are not available to the compiler's optimization passes. Because of this, LLBMC implements a similar set of optimizations on the ILR formula level to re-enable these optimizations.

The first versions of LLBMC performed inlining on the LLVM-IR level. This made it possible to use LLVM's inlining functionality and subsequently its constant propagation. During development, LLBMC's architecture was changed to perform inlining as part of the encoding which reduces the effectiveness of performing constant propagation on the LLVM-IR level. For compensation, constant propagation for ILR formulæ was added to LLBMC's rewrite system. The rewrite rules for constant propagation are listed extensively in appendix A.1. A typical constant propagation rule is the following rule for addition:

$$\mathrm{add}_{\mathcal{I}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow (n + m)_{\mathcal{I}} \qquad \text{[A.2a]}$$

[1] Note that this does not apply to uninterpreted constants, but only to fully interpreted constants.

```
define i32 @contains(i32* %a) {
entry:
  br label %for.body

for.body:
  %i = phi i32 [ 0, %entry ], [ %inc, %for.body ]
  %r = phi i32 [ 0, %entry ], [ %add, %for.body ]
  %arrayidx = getelementptr i32* %a, i32 %i
  %0 = load i32* %arrayidx, align 4
  %add = add i32 %0, %r
  %inc = add i32 %i, 1
  %cmp = icmp eq i32 %inc, 4
  br i1 %cmp, label %return, label %for.body

return:
  ret i32 %add
}
```

Listing 6.1: Example for constant propagation before unrolling

```
define i32 @contains(i32* %a) {
entry:
  %0 = load i32* %a, align 4
  %arrayidx.1 = getelementptr i32* %a, i32 1
  %1 = load i32* %arrayidx.1, align 4
  %add.1 = add nsw i32 %1, %0
  %arrayidx.2 = getelementptr i32* %a, i32 2
  %2 = load i32* %arrayidx.2, align 4
  %add.2 = add nsw i32 %2, %add.1
  %arrayidx.3 = getelementptr i32* %a, i32 3
  %3 = load i32* %arrayidx.3, align 4
  %add.3 = add i32 %3, %add.2
  ret i32 %add.3
}
```

Listing 6.2: Example for constant propagation after unrolling

The rule can be applied if both arguments of the addition are constants (in this case called $(n)_{\mathcal{I}}$ and $(m)_{\mathcal{I}}$). The addition can be replaced by the constant one receives from evaluating the expression $(n+m)_{\mathcal{I}}$. This is possible because of the way we defined integer and pointer constants in section 3.7.

Note that $(\cdot)_{\mathcal{I}}$ is defined to perform a modulo calculation with $2^{|\mathcal{I}|}$, so there is no need to calculate $(n+m) \mod 2^{|\mathcal{I}|}$ explicitly. We furthermore assume that for any function operating on unsigned (resp. signed) $\mathcal{I}$, n and m are representable as a unsigned (resp. signed) $\mathcal{I}$. Because of this, there is no need to calculate $n \bmod 2^{|\mathcal{I}|}$ (or its signed integer counterpart) explicitly. This allows succinct expression of such rewrite rules, e.g. for unsigned and signed division:

$$\mathrm{div}^u_{\mathcal{I}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow (n/m)_{\mathcal{I}} \quad \text{[A.2d]}$$
$$\mathrm{div}^s_{\mathcal{I}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow (n/m)_{\mathcal{I}} \quad \text{[A.2e]}$$

6.1.2 Non-Constant Simplification Rules

Clearly, not all simplification rules are constant propagation. In particular, boolean operations and arithmetic operations provide further optimization opportunities. Some of these are related to a compiler optimization technique called constant folding, others do not have a direct counterpart in compiler optimizations.

Simplification rules for the functions and, or, eq, and not of the sort bool are listed in appendix A.2. These rules are mostly straightforward.

Rewrite rules for the arithmetic functions $\mathrm{add}_{\mathcal{I}}$, $\mathrm{sub}_{\mathcal{I}}$, $\mathrm{mul}_{\mathcal{I}}$, $\mathrm{div}^u_{\mathcal{I}}$, $\mathrm{div}^s_{\mathcal{I}}$, $\mathrm{rem}^u_{\mathcal{I}}$, and $\mathrm{rem}^s_{\mathcal{I}}$ are listed in appendix A.3. A number of these rules are closely related to compiler optimizations, e.g. unsigned division can be replaced by a simple shift if the division is by a power of two. Similar operations are possible for calculation of a product or a remainder:

$$\mathrm{mul}_{\mathcal{I}}(x, (2^m)_{\mathcal{I}}) \longrightarrow \mathrm{shl}_{\mathcal{I}}(x, (m)_{\mathcal{I}} \quad \text{[A.14e]}$$
$$\mathrm{div}^u_{\mathcal{I}}(x, (2^m)_{\mathcal{I}}) \longrightarrow \mathrm{shr}^u_{\mathcal{I}}(x, (m)_{\mathcal{I}} \quad \text{[A.15b]}$$
$$\mathrm{rem}^u_{\mathcal{I}}(x, (m)_{\mathcal{I}}) \longrightarrow \mathrm{and}_{\mathcal{I}}(x, (m-1)_{\mathcal{I}}) \; ; \; \exists n \in \mathbb{N}(2^n = m) \quad \text{[A.16b]}$$

Calculation of the signed remainder for the division by a power of two number is slightly more complicated as both the positive and negative case need to be taken into account:

$$\begin{aligned} \mathrm{rem}^s_{\mathcal{I}}(x, (m)_{\mathcal{I}}) \longrightarrow \; & \mathrm{select}_{\mathcal{I}}(\\ & \quad \mathrm{ge}^s_{\mathcal{I}}(x, (0)_{\mathcal{I}}), \\ & \quad \mathrm{and}_{\mathcal{I}}(x, (m-1)_{\mathcal{I}}), \\ & \quad \mathrm{sub}_{\mathcal{I}}(0, \mathrm{and}_{\mathcal{I}}(\mathrm{sub}_{\mathcal{I}}((0)_{\mathcal{I}} - x), (m-1)_{\mathcal{I}})))) \; ; \\ & \exists n \in \mathbb{N}(2^n = m) \end{aligned} \quad \text{[A.16g]}$$

Rewrite rules for safety checking functions are listed in appendix A.4. Bitwise operations are handled in appendix A.5 and shifts in appendix A.6. Rules related to integer and pointer comparison are listed in appendix A.7, while other rules, mostly type conversion, are shown in appendix A.8.

6.1.3 Reduction Rules

ILR provides features that are closely related to LLVM-IR but do not have a counterpart in SMT, e.g. the gep function and multi-byte loading and storing. To simplify translation of ILR formulæ to SMT, LLBMC contains a set of rewrite rules for reducing such ILR functions to ones that match SMT featues more closely. These rules are listed in appendix A.9.

The rules in this particular set of rules often follow the original definition of the function closely, e.g. a multi-byte load ($\mathrm{load}^b_{\mathcal{I}}$) is defined by the axiom

$$\forall m, p \left(|\mathcal{I}| > 8 \rightarrow \mathrm{load}^b_{\mathcal{I}}(m,p) = \mathrm{concat}_{\mathsf{i8},\mathcal{I}}(\mathrm{load}_{\mathsf{i8}}(m,p), \mathrm{load}^b_{\mathcal{I}_1}(m,p+1)) \right) \tag{3.15}$$

and can be rewritten by the rewrite rule

$$\begin{aligned} \mathrm{load}^b_{\mathcal{I}}(m,p) \;\longrightarrow\; & \mathrm{concat}_{\mathsf{i8},\mathcal{I}}(\mathrm{load}_{\mathsf{i8}}(m,p), \mathrm{load}^b_{\mathcal{I}_1}(m,p+1)) \;; \\ & |\mathcal{I}| > 8 \wedge |\mathcal{I}_1| = |\mathcal{I}| - 8, \end{aligned} \tag{A.25a}$$

which is directly derived from the above axiom. In this example, the antecedent of the axiom is turned into the rewrite rule's condition, while the rewrite rule itself results from replacing the equality symbol with the rewrite symbol oriented from left to right.

6.1.4 Running Simplifications

The function SIMPLIFY introduced in listing 3.1 and listed in listing 6.3 is responsible for applying the simplification rules listed above on an ILR formula. TRS_{simpl} is a term rewriting system containing all rules listed in appendices A.1 to A.8. TRS_{heap} contains the rules listed in chapter 5 and TRS_{red} contains the rules listed in appendix A.9.

```
1 function SIMPLIFY(φ)
2     φ ← INNERMOSTFIRST(TRS_simpl, φ)
3     φ ← INNERMOSTFIRST(TRS_simpl ∪ TRS_heap, φ)
4     φ ← INNERMOSTFIRST(TRS_simpl ∪ TRS_heap ∪ TRS_red, φ)
5     return φ
6 end function
```

Listing 6.3: Algorithm for formula simplification

As apparent from the function's listing, simplification in LLBMC is done in multiple passes. The reason for this is that reduction rules, in particular heap rules, tend to increase the size of the formula. Therefore, all other simplification rules are applied to the formula to reduce the formula's size as far as possible, first, then the rules introduced as part of the decision procedure for dynamic memory allocation presented in section 5.3 are applied to the formula together with the simplification rules. Finally, the full set of simplifying rewrite rules is used. While the result is the same as applying all rules at once, this approach reduces the maximal size of the

formula during rewriting. We will call the subset of ILR which remains after running the full set of rewrite rules described so far the *reduced ILR*.

Simplification in LLBMC is done *innermost-first*. This means the term rewriting system tries to rewrite subterms before their respective parent terms. This provides better performance for LLBMC than outermost-first because constant propagation, the most important kind of simplifications in LLBMC, works best with an innermost-first approach.

For specific use cases, additional simplification passes can at times cause performance improvements of rewriting. Nonetheless, the setup described above is a good all-round configuration of LLBMC, as was confirmed in multiple international software verification competitions (see section 6.3).

6.2 Solving ILR Formulæ

In LLBMC, an ILR formula is solved by translating it to an equisatisfiable SMT formula, which in turn is solved by an off-the-shelf SMT solver for the quantifier-free logic of arrays and bitvectors. However, this translation is implemented in LLBMC only for a subset of ILR. All terms of ILR that are not in this subset are replaced by equivalent terms of this subset using the reduction rules presented in the previous section. The sorts remaining after simplification are mapped to matching SMT sorts, e.g. integers and pointers to bitvectors and memory states to arrays. All terms remaining after simplification are then trivial to translate to SMT because they either have a direct counterpart in SMT, as is the case for arithmetic and bitwise functions, for example, or the functions are NOOPs in SMT. This is for example the case for all conversion functions that do not change the value's bitwidth such as a $\mathrm{bitcast}$. Note that a subset of the functions defined in chapter 4 are neither removed during encoding nor during simplification or reduction. Among others, this is the case for the initial memory state and for all arguments of the root context. During translation to SMT, these functions are replaced by uninterpreted constants of the appropriate sort. The translation and the subsequent execution of the SMT solver on the generated SMT formula is implemented in LLBMC's SOLVE function (see listing 3.1).

Due to the preceding simplifications, the translation process is straightforward. Furthermore, because the SMT solver is treated as a black box in LLBMC, the solving of SMT formulæ is of no further concern in the thesis. LLBMC's work is not complete with solving, though: while solving an ILR formula allows deciding if a program is safe or not, it is not sufficient to identify which instruction in the program is unsafe and under which circumstances safety properties are violated. This can be approached by making use of the fact that most SMT solvers make it easy to retrieve interpretations for arbitrary terms of satisfiable SMT formulæ. These interpretations can be mapped back to terms of ILR formulæ and finally to an LLVM-IR program's state during execution.

As mentioned in section 2.1.1, given a model m of a formula ϕ we use $[\![s]\!]^m$ for the interpretation of an arbitrary subterm s of ϕ. Recall now that the first step of the term rewriting system for encoding is implemented as a set of maps (see section 4.6.2). For example, $\varepsilon_t^{\mathrm{I}}(c, i)$ is not a term in a term rewriting system but

an entry in a map $\varepsilon^{\mathtt{I}} : \{(c, i) \mapsto s\}$ that maps the pair (c, i) to the term s which would result from rewriting $\varepsilon^{\mathtt{I}}_t(c, i)$. While this means that the term $\varepsilon^{\mathtt{I}}(c, i)$ itself is not part of the formula passed to the SMT solver anymore, the map can be used to retrieve the corresponding term s which is part of the solved formula. Given a context c and an instruction i which is not `void`, $[\![\varepsilon(c, i)]\!]^m$ results in the value stored in i's virtual register during execution of context c in the model m. Note that we can similarly retrieve execution conditions, the memory state, and all other information related to executed instructions from corresponding maps.

A simple use case for this is the identification of the failing instruction associated with a model. The failing instruction associated with a model m is the instruction i for which a context c exists so that $[\![\neg\, \sigma(c, i)]\!]^m$. This is only possible because the encoding presented in chapter 4 ensures that any model always has exactly one failing instruction. Modifications to the encoding introduced later in this section will relax this property, which makes it necessary at times to retrace the program's execution in order to identify the first failing instruction.

While it is important to the tool's user to be able to identify the failing instruction, LLBMC can provide additional benefits to the user in the form of a counterexample which illustrates how execution of the program leads to the instruction's failure. Counterexample generation is implemented in LLBMC's function COUNTEREXAMPLE. The concept of counterexamples is based on traces:

Definition 6.1 (Trace). *A trace $t = ((c_1, i_1), \ldots, (c_n, i_n))$ is a list of pairs (c, i) of contexts c and instructions i where an assignment to the program's input variables exists so that all $(c, i) \in t$ are executed exactly in this order.*

Note that a trace does not have to begin with the main context's first instruction and it does not have to end at the main context's return instruction. Furthermore, with this definition, a trace does not contain the actual values stored in virtual registers or memory locations.

A counterexample is a trace through a program which leads to a failing instruction and which is enriched by all necessary information about the values stored in virtual registers and in memory at any point of time:

Definition 6.2 (Counterexample). *Given a program p, a call graph (V_p, E_p, c_p) of p, a trace t through p, and a model m of the formula generated by rewriting $\neg\sigma^{\mathtt{P}}(c, p)$, we call (t, m) a counterexample if and only if for all (c, i) in t, $[\![\eta^{\mathtt{I}}(c, i)]\!]^m$ is true and for all (c, i) not in t, $[\![\neg\eta^{\mathtt{I}}(c, i)]\!]^m$ holds.*

Note that a counterexample (t, m) always begins at the main context's first instruction and ends in the context c at instruction i for which $[\![\neg\sigma^{\mathtt{I}}(c, i)]\!]^m$.

Counterexamples are easily created by starting with the first executed instruction of the main context, walking along the sequence of executed instruction, and evaluating each one along the trace. Branch conditions need to be evaluated to identify the next instruction to be executed at the end of basic blocks. LLBMC prints out counterexamples on the command line, with one line per instruction. However, for better readability, LLBMC indents counterexamples according to the call depth of an instruction's context. In the case of memory modifying instructions, the modified memory range can be shown on demand. If available, LLBMC uses LLVM's debug information to map the contents of virtual registers and memory locations to variables

of the source language.[2]

6.2.1 Working with Multiple Counterexamples

The model checking algorithm presented in listing 3.1 can be used to check a large number of safety properties with a single run of the algorithm. However, it always returns at most a single counter example. This is sufficient to decide whether the program as a whole is safe or to show that at least one safety property is violated, but not if multiple safety properties are violated. This is because, if a counterexample is found for one of the safety properties, the algorithm terminates and nothing is known about other instructions and safety properties. In contrast to this, if it can be guaranteed that at least one for each violated safety property a counterexample is generated, then all safety properties for which no counterexample is generated are proven safe within the bounds.

A simplistic algorithm for generating a counterexample per violated safety property is to run the core model checking algorithm (see listing 3.1) separately for each property. However, when checking multiple properties of the same program, large parts of the generated formulæ are nearly identical. This is because these parts are primarily related to the program's code and mostly independent of the checked properties. While the SMT solver is running, it accumulates a considerable amount of information about the program, e.g. aliasing information. However, separate runs of the core algorithm for each property lose all information gained by the SMT solver about the formula during previous runs. Because of this, it is highly desirable to reduce the number of restarts of the core algorithm as much as possible.

A simple way to do this is to first check all properties in a single run of the core algorithm. Then, if a counterexample is found, the violated safety property can be removed from the list of properties to check. The core algorithm is then restarted with the reduced list of properties. This process is repeated until no more counterexamples are generated. All properties for which no counterexample was generated are thereby proven safe. While this does not eliminate restarting of the core algorithm completely, the number of restarts is reduced from the number of checked properties to the number of violated properties. Because the number of violated properties is usually significantly smaller than the number of checked properties, this approach can result in considerable performance improvements over the previously presented approach.

However, the number of restarts can be reduced further if the SMT solver supports incremental solving. With *incremental solving* the SMT solver retains its internal state after it finishes satisfiability checking. Additional constraints can then be added to the formula and execution of the solver can be continued. The added constraints are taken into account when searching for a new model.

In software verification, incremental solving is often used for abstraction refinement. For this, a simpler, more abstract model of the original program is encoded in an SMT formula. Satisfiability of the formula is then checked with an SMT solver which supports incremental solving. If the solver reports satisfiability, the counterexample generated from the model is examined. If the counterexample is spurious, it is used to

[2] An example for LLBMC's counterexample output, which is based on the program in listing 6.7, is shown in listing 6.9.

refine the formula to more closely match the real problem to be solved. Execution of the solver is then continued until the SMT solver returns a model which corresponds to a non-spurious counterexample or until the formula becomes unsatisfiable. In contrast to this, LLBMC uses incremental solving not for abstraction refinement but for the generation of multiple counterexamples.

The core model checking algorithm shown in listing 3.1 can be adapted to generate multiple counterexamples. This adaptation is shown in listing 6.4. The main difference to the core algorithm is the loop and the CONSTRAINT function. This function adds additional constraints to the formula φ which ensure that a different model is found the next time SOLVE is called. Constraints can be used to ensure the same safety property is not checked again.

```
1  function MULTICHECK(p, e, b_l, b_c)
2      C ← ∅
3      p ← OPTIMIZE(p)
4      p ← UNROLL(p, b_l)
5      g ← CALLGRAPH(p, e, b_c)
6      φ ← ENCODE(p, g)
7      φ ← SIMPLIFY(φ)
8      r, m ← SOLVE(φ)
9      while ¬r do
10         c ← COUNTEREXAMPLE(p, φ, m)
11         C ← C ∪ {c}
12         φ ← φ ∧ CONSTRAINT(p, g, c)
13         r, m ← SOLVE(φ)
14     end while
15     return C
16 end function
```

Listing 6.4: Adapted software bounded model checking algorithm for multiple counterexamples

If a counterexample with the trace $t = (c_1, i_1, \ldots, c_n, i_n)$ was found, the constraint

$$\neg \left(\bigwedge_{c_f : \mathrm{fun}(c_f) = \mathrm{fun}(c)} \sigma^{\mathrm{I}}(c_f, i) \right)$$

can be added to the solver to ensure no further counterexamples ending at instruction i are found. Because every safety property is associated with exactly one instruction, this ensures that for each violated safety property exactly one counterexample is generated.

6.2.2 Shadowing of Checks

As already mentioned in section 3.3, undefined behavior in C is a challenge for any static code analysis tool. This is also true for undefined behavior in LLVM-IR, as it is closely modeled after the former. Because nothing can be known about a program's

behavior if undefined behavior occurs, there are no sensible assumptions to be made about a program's state in this case. Consequently, LLBMC's encoding essentially cuts off all traces through a program as soon as undefined behavior occurs. At times, this results in unexpected behavior of LLBMC for the user.

Consider the example in listing 6.5. The two add instructions fail for exactly the same values of %x and %y. However, if the first instruction causes undefined behavior, LLBMC treats the program as if execution terminates immediately (see equation (4.17c)). This means execution never reaches the second instruction for those cases for which undefined behavior would occur. LLBMC will therefore report the second instruction as safe. Experience shows that many users argue that the second instruction should be unsafe, after all it is identical to the first instruction. And, to some degree, they rightfully do: if anything can happen with undefined behavior, execution of the program can also continue with the second add instruction and %x and %y unchanged.

```
1 define void @foo(i32 %x, i32 %y) {
2 entry:
3   %0 = add nsw i32 %x, %y
4   %1 = add nsw i32 %x, %y
5   ret void
6 }
```

Listing 6.5: Example for shading of checks

This means, neither is it useful to allow any behavior after undefined behavior occurs, nor is it ideal to cut off the program's execution on undefined behavior entirely. How to handle this, needs to be defined for each instruction and for each cause of undefined behavior individually.

LLBMC can be easily modified to prevent shadowing of checks. In its simplest form, LLBMC would then assume execution continues with the successor instruction after undefined behavior. This non-shadowing mode can be realized through two changes to the system.

First, the rewrite rule in equation (4.17c) needs to be changed to

$$\vec{\eta}^{\mathrm{I}}(c,i) \longrightarrow \overleftarrow{\eta}^{\mathrm{I}}(c,i) \tag{6.1}$$

to allow continuation of the program's execution even if an instruction is unsafe. However, as a minor consequence of this, a model does not have a unique failing instruction anymore. If two instructions fail, the counterexample needs to be traced to identify which one failed first.

If LLBMC uses the previously presented algorithm to generate multiple counterexamples, modification of the rewrite rule is not yet sufficient to prevent shadowing. This is because after a counterexample for the first instruction is found, the algorithm adds a constraint to the formula which excludes any further counterexamples for this instruction to force the SMT solver to search for a counterexample for a different instruction. If the second instruction only fails if the first instruction does, this implicitly excludes any counterexamples for the second instruction, too.

The solution to this problem requires adding one uninterpreted constant t_i of sort bool per checked instruction i, where each t_i can be used to turn off checks for instruction i. For this, the additional function symbol σ_t^{I} must be introduced along with the rewrite rule

$$\sigma_t^{\mathrm{I}}(c, i) \longrightarrow \mathrm{and}(\sigma^{\mathrm{I}}(c, i), t_i). \tag{6.2}$$

Equation (4.39b) then needs to be adapted to use σ_t^{I} in place of σ^{I}. All counterexamples for instruction i can then be disabled by adding the constraint $\neg\langle t_i \rangle$ to the formula.

6.3 Evaluation

This section is dedicated to the evaluation of LLBMC in comparison with other tools. The evaluation of software verification tools is no easy task for a number of reasons.

Even though LLBMC in theory supports most languages that are supported by LLVM, in practice, development of LLBMC was focused primarily on embedded software written in C or C++. Furthermore, LLBMC is designed for checking safety properties, or more specifically undefined behavior. LLBMC can therefore not be compared to tools that target other kinds of software systems, such as concurrent systems, other languages, such as Java, or other types of properties, such as liveness properties. LLBMC can only be compared meaningfully to other static analysis tools which are designed for checking safety properties of sequential C or C++ programs.

However, even when tools focus on the same general use case, the set of supported features often varies greatly and different tools are optimized for different properties. This is in particular true for tools which check for undefined behavior in C and C++ as no tool supports all causes and kinds of undefined behavior listed in the standard.

Furthermore, all tools make some assumptions about the code in question. Some are more obvious, such as the bounds used in bounded model checking or the use of mathematical integers instead of bitvector arithmetics. Other assumptions are less obvious, such as the restrictions implied by a tool's memory model or assumptions about the aliasing of objects. This is particularly complex given C programmer's tendency to perform low-level operations that subvert C's type system.

A fair and unbiased evaluation of software verification tools which compares tools by running them on a representative set of benchmarks is a challenge of its own. A good way to guarantee an unbiased selection of benchmarks and tools is an independently organized and executed tool competition. In the case of LLBMC, the best-fitting competition is the International Software Verification Competition. LLBMC took part in the first competition [Bey12] in 2012, in the second competition [BW13] in 2013 and in the third competition [Bey14] in 2014. Therefore, the following evaluation of LLBMC is based on these competitions.

The *International Software Verification Competition* (*SV-COMP*) is an international competition comparing academic software verification tools. The competition is chaired by Dirk Beyer and took place for the first time in 2012. In the competition, the participating tools are executed on all benchmarks from the competition's benchmark set. The benchmarks consist of safe and unsafe programs and the tools get scored based on whether they provided the correct answer for both kinds of benchmarks.

The precise number of points given varies between the years, but in general, safe benchmarks are higher-rated than unsafe benchmarks, because proving safety is considered the harder challenge. Furthermore, an incorrect result leads to a negative score which is always bigger than the corresponding positive score. This ensures that a randomly guessing tool's score has a negative expected value. The rules were modified over the course of the years and improved upon each year by the competition's jury, which consists of one member of each participating team. Benchmarks are mostly reduced to reachability of error labels, so tools do not need to handle different kinds of runtime errors explicitly. This considerably lowers the barrier of entry for tools and research prototypes. It also means the competition is more aimed at comparing a tool's algorithm than a tool's feature set.

In total 14 tools competed in the three years in which LLBMC participated. The tools implement a wide variety of different approaches. *Symbiotic* combines code instrumentation, program slicing, and symbolic execution. *UFO* combines interpolants and abstract interpretation in an abstraction refinement loop. *CPAchecker* is a whole software verification framework supporting many different approaches ranging from predicate abstraction to interpolation-based refinement. Like CPAChecker, *Ultimate* is a framework supporting different approaches for software verification. Both CPAChecker and Ultimate participated with multiple tools implementing different approaches based on the respective framework. The tool *BLAST* is based on counterexample-guided abstraction refinement. *Frankenbit* combines bitvector decision procedures with invariant generalization of linear arithmetics. *ESBMC*, like LLBMC, does software bounded model checking, as is already discussed in section 2.2.5. The tools *CSeq*, in all its variations, and *Threader* are designed for concurrent programs. Internally, both tools use bounded model checking to some degree. *Predator* is based on separation logic, though it uses a graph-based heap representation internally. The tool *FShell* is designed for test case generation and uses bounded model checking with a CBMC-based implementation. *SATabs* is a bit-precise verifier based on predicate abstraction. *Wolverine* uses Craig interpolation to derive program invariants. Finally, *QARMC-HSF* is based on horn clauses.

6.3.1 International Software Verification Competition 2012

In 2012, LLBMC participated in six out of seven categories and won one gold and one silver medal.

LLBMC won a silver medal in the category Heap Manipulation, being only second to Predator. These two tools were the only tools that did not give incorrect results in this category. Predator performed noticeably better than LLBMC both in the number of solved instances and with respect to runtime, though this was expected considering that Predator is explicitly designed for checking properties related to heap manipulation.

LLBMC performed well in the category DeviceDrivers, winning the gold medal in this category, but performed badly in the category DeviceDrivers64. Because LLBMC uses bitvectors, a performance hit was expected from using 64 bits, though not in this magnitude. Analysis of the results showed that LLBMC's inferior performance is mostly caused by a number of benchmarks in this category which rely on a simplified memory model which LLBMC does not support well.

In contrast to other competitions, such as the ones for SAT and SMT solvers, in SV-COMP tools are not disqualified if they return an incorrect result for a benchmark. This is mostly because otherwise it would result in the disqualification of nearly all tools as nearly all tools return incorrect results for at least one benchmark. However, this also means SV-COMP is open for falsification tools. Even though LLBMC can do verification if a sufficiently large bound is set, for SV-COMP LLBMC was configured for falsification by using small bounds and deactivating bounds checks. This setup is based on the small scope hypothesis which assumes that most bugs can be found by checking in a relatively small scope and in the case of bounded model checking, this is applied to the bounds. LLBMC used the same bounds settings per category as ESBMC did which ranged between 2 and 8 loop iterations. This provided a large runtime improvement and often resulted in good results whereas a setting tuned for verification resulted in timeouts.

6.3.2 International Software Verification Competition 2013

LLBMC took part in the International Software Verification Competition 2013 and won two gold and four silver medals.

Like in the previous year, LLBMC was configured with an extremely low loop iteration count. Despite of this, LLBMC was the only tool in that year's competition that did not generated incorrect results. This can be interpreted as strong evidence for the small scope hypothesis of software bounded model checking.

LLBMC had a perfect score in the category bitvectors, solving all benchmarks correctly, as well as a perfect score and the best runtime in the subcategory ControlFlow-Integer-MemPrecise. Nonetheless, LLBMC was not scored in the category Control-FlowInteger overall because it did not participate in the subcategory ControlFlow-Integer-MemSimple. The category is based on a simplified memory model which expects specific behavior from the verification tool where the C standard has undefined behavior. LLBMC does not support this simplified memory model and therefore did not take part in this category.

Learning from last years experience with the category DeviceDrivers64. LLBMC explicitly opted out of the category DeviceDrivers64 due to the simplified memory model, which LLBMC does not support.

LLBMC showed good average performance for the category FeatureChecks, even being comparable to the category's winner Predator, though the results strongly impacted by three benchmarks which LLBMC faired particularly bad at. Interestingly in this category LLBMC timed out for exactly those cases which made Predator crash, so while the tools have markedly different technology, they seem to have similar strong and weak points. This also becomes apparent when comparing the overall performance of these two tools.

LLBMC did not participate in the category Overall because the results for this category were calculated from all categories' results including the ones a tool opted-out from. A tool which simply detects functions of the `phreads` library in the source code therefore had considerably better scores than a tool that opted out for the concurrency category. The effect of this scoring can be seen in UFO's scores for the overall category, which are negative even though the tool itself won several gold

medals and performed exceptionally well in all categories which it did not opt out from.

6.3.3 International Software Verification Competition 2014

LLBMC took part in SV-COMP 2014, won one gold, one silver, and four bronze medals. Furthermore, later that year, the LLBMC team won a Gödel medal during the FLoC Olympic Games for the tool's performance in the past three years.

For the year 2014's competition, LLBMC was modified to detect the use of unsupported features, e.g. the use of `pthreads`, and immediately return `unknown` in this case. This modification resulted in a more realistic scoring of LLBMC in the overall category. Because of this LLBMC took part in the Overall category and won a bronze medal.

6.3.4 Detailed Example

This section is dedicated to show a full example of how LLBMC operates, from the initial C source code to the generated counterexamples.

```
1  int isintmax(int n) {
2    return n + 1 < n;
3  }
4
5  int *allocate(int n, int v) {
6    int *p = 0;
7    if (!isintmax(n)) {
8      p = (int*) malloc(sizeof(int));
9    }
10   *p = v;
11   return p;
12 }
```

Listing 6.6: C source of full example

Appendix C.1.1 shows the LLVM-IR code generated for the example by clang. This code is already valid input for LLBMC, though the excessive memory accesses would result in excessive strain on the SMT solver's decision procedure for the theory of arrays. These memory accesses therefore are an obvious target for optimization.

Running compiler optimizations on this code results in the LLVM-IR code shown in appendix C.1.2. In particular `mem2reg` and `early-cde` are effective for this example. LLVM's default optimization settings causes the function `isintmax` to be inlined. However, this makes it impossible to demonstrate the use of multiple contexts in this example. Because of this, LLVM's `inline` pass was disabled. The code resulting from optimization contains significantly fewer memory accesses but it also does not contain the addition anymore. The latter is troubling, as this means undefined behavior got lost during optimization and cannot be checked anymore.

The instrumentation presented in section 3.3 fixes this by producing the LLVM-IR code shown in appendix C.1.3. The instrumentation functions called in this

example are listed in appendix C.1.4. Internally, these function call LLBMC's built-in functions which map directly to the corresponding ILR function. Note in particular that after instrumentation, LLVM's optimizations do not hide the undefined behavior in `isintmax` anymore, as can be seen in listing 6.7.

```
define i32 @isintmax(i32 %n) {
entry:
  call void @check.saddo.i32(i32 %n, i32 1)
  ret i32 0
}

define i32* @allocate(i32 %n, i32 %v) {
entry:
  %call = call i32 @isintmax(i32 %n)
  %tobool = icmp ne i32 %call, 0
  br i1 %tobool, label %if.end, label %if.then

if.then:
  %call1 = call i8* @malloc(i32 4)
  %0 = bitcast i8* %call1 to i32*
  br label %if.end

if.end:
  %p.0 = phi i32* [ null, %entry ], [ %0, %if.then ]
  %t = bitcast i32* %p.0 to i8*
  call void @check.access(i8* %t, i32 4)
  store i32 %v, i32* %p.0, align 4
  ret i32* %p.0
}
```

Listing 6.7: Full example optimized instrumented LLVM-IR code

For this example, LLBMC is configured to check only for the assertions inserted during instrumentation and not for cases of undefined behavior on the LLVM-IR level. Furthermore the call graph shown in appendix C.2.1 was generated and used for this example. In addition, two call graphs are generated, one in appendix C.2.2 and appendix C.2.3.

After instrumentation and optimization, encoding can now take place. Because not all instructions have a named virtual register, we will refer to instructions by i suffixed with the instruction's line number. For example, i_3 refers to the call to `check.saddo.i8` in `isintmax` in listing 6.7. In the following equations, each line represents a different state during the encoding:

$$\sigma^{\mathrm{P}}(c_1, p)$$

$$\sigma^{\mathrm{F}}(c_1, \mathrm{allocate}) \tag{6.3}$$

$$\mathrm{and}(\eta^{\mathrm{I}}(c_1, i_3), \sigma^{\mathrm{I}}(c_1, i_3), \sigma^{\mathrm{F}}(c_2, \mathrm{isintmax})) \tag{6.4}$$

$$\mathrm{and}(\mathrm{or}(\mathrm{not}(\eta^{\mathrm{I}}(c_1, i_3)), \sigma^{\mathrm{I}}(c_1, i_3)), \mathrm{or}(\mathrm{not}(\eta^{\mathrm{I}}(c_2, i_{21}), \sigma^{\mathrm{I}}(c_2, i_{21}))) \tag{6.5}$$

As can be seen above, the formula's size grows rapidly during encoding. The same is true for the nesting depth of terms and at the same time, term sharing gets more frequent. For readability, we will use the notation introduced in section 3.7.7 to

represent the terms in a flattened form:

$$
\begin{aligned}
t_5 &:= \overleftarrow{\eta}^{\mathtt{I}}(c_2, i_3) \\
t_4 &:= \sigma^{\mathtt{I}}(c_2, i_3) \\
t_3 &:= \overleftarrow{\eta}^{\mathtt{I}}(c_1, i_{20}) \\
t_2 &:= \sigma^{\mathtt{I}}(c_1, i_{20}) \\
t_1 &:= \mathrm{and}(\mathrm{or}(\mathrm{not}(t_5), t_4), \mathrm{or}(\mathrm{not}(t_3) t_2,)) \\
&\quad\ t_1
\end{aligned}
$$

Note that the last line is not in definitorial form but contains the top-level constraint, which is to be checked for satisfiability later on. Applying term rewriting to each of the named terms above results in the following formula:

$$
\begin{aligned}
t_5 &:= \overleftarrow{\eta}^{\mathtt{F}}(c_2, \mathtt{isintmax}) \\
t_4 &:= \mathrm{not}(\mathrm{addo}^{s}_{\mathsf{i32}}(\varepsilon^{\mathtt{A}}_{\mathsf{i32}}(c_2, n))) \\
t_3 &:= \overrightarrow{\eta}^{\mathtt{I}}(c_1, i_{19}) \\
t_2 &:= \mathrm{validaccess}_{\mathsf{h32}}(\overleftarrow{\rho}^{\mathtt{I}}_{\mathsf{h32}}(c_1, i_{20}), \varepsilon^{\mathtt{I}}_{\mathsf{i32}*}(c_1, i_{19}), (4)_{\mathsf{i32}}) \\
t_1 &:= \mathrm{and}(\mathrm{or}(\mathrm{not}(t_5), t_4), \mathrm{or}(\mathrm{not}(t_3) t_2,)) \\
&\quad\ t_1
\end{aligned}
$$

As can be seen in appendix C.3, which contains the state of the formula after three additional steps of parallel term rewriting, the formula's size continues to grow quickly. The formula resulting from encoding can be seen in appendix C.3.4. Note that for technical reasons and for reasons of readability, the naming of terms after encoding uses different letters to indicate the terms' sort, e.g. b for booleans, i for integers, and p for pointers. In additional, the encoding used for generating this formula deviates slightly from the encoding presented in this thesis due to differences between LLBMC's implementation and the encoding TRS presented in this thesis.

Simplifications considerably reduce the formula's complexity, as can be seen by the following result of simplification:

$$
\begin{aligned}
i_0 &:= n \\
b_0 &:= \mathrm{addo}^{s}_{\mathsf{i32}}(i_0, (1)_{\mathsf{i32}}) \\
b_1 &:= \mathrm{not}(b_0) \\
&\quad\ b_1
\end{aligned}
$$

Finally, the formula simplified in this way is reduced to the set of functions supported by SMT, which in this case means addo^{s} is expanded to an equivalent term and converted to the SMT-LIB formula shown in listing 6.8.

Solving this formula with an SMT solver and mapping the model back to a LLVM-IR level counterexample results in the counterexample shown in listing 6.9.

```
(set-logic QF_AUFBV)
(set-info :smt-lib-version 2.0)
(set-info :status unknown)
(set-info :category "industrial")
(declare-fun n_0 () (_ BitVec 32))
(assert
(let ((?x1 (_ bv1 32)))
(let ((?x2 ((_ extract 31 31) n_0)))
(let ((?x3 ((_ extract 31 31) ?x1)))
(let ((?x4 (bvadd n_0 ?x1)))
(let ((?x5 ((_ extract 31 31) ?x4)))
(let ((?x6 (bvnot ?x5)))
(let ((?x7 (bvand ?x2 ?x3)))
(let ((?x8 (bvor ?x2 ?x3)))
(let ((?x9 (bvnot ?x8)))
(let ((?x10 (bvand ?x7 ?x6)))
(let ((?x11 (bvand ?x9 ?x5)))
(let ((?x12 (bvor ?x10 ?x11)))
(let (($x13 (= ?x12 (_ bv1 1))))
(let (($x14 (not $x13)))
(let (($x15 (not $x14)))
$x15
))))))))))))))
)
(check-sat)
(exit)
```

Listing 6.8: SMT-LIB Formula for the detailed example

```
initial memory content:
default = 00

define i32* @allocate(i32 %n, i32 %v) {
  ; i32 %n = 2147483647
  ; i32 %v = 0

entry:                                                ; executed
  %call = call i32 @isintmax(i32 %n)                  ; 0

  define i32 @isintmax(i32 %n) {
    ; i32 %n = 2147483647

  entry:                                                ; executed
    call void @check.saddo.i32(i32 %n, i32 1)

    define void @check.saddo.i32(i32 %x, i32 %y) {
      ; i32 %x = 2147483647
      ; i32 %y = 1

    entry:                                                ; executed
      %0 = call i1 @llbmc.saddo.i32(i32 %x, i32 %y)     ; unknown
      call void @llbmc.assert(i1 %0)                    ; error
```

Listing 6.9: Counterexample for the detailed example

6.4 Summary and Outlook

LLBMC uses a set of formula simplifications, which are implemented in a term rewriting system, to reduce the formulæ's size and complexity. The same term rewriting system is used to reduce any ILR formula to a simple, SMT-like subset. Finally, the simpler bounded model checking approach presented in listing 3.1 was extended to return multiple counterexamples for different violated safety properties.

A limitation of the simplification approach presented in this chapter is its strict separation from LLBMC's encoding component, which is presented in chapter 4. Both components are formalized as a term rewriting system and combining these systems in a single system would reduce the size of intermediate formulæ because the encoding's negative effect on the formulas size is balanced to some degree by the simplifications' positive impact. Combining these systems more closely would also open up opportunities to achieve a closer approximation of the program's real call graph because simplifications are often sufficient to decide if a call is executed at all.

Chapter 7

Conclusion

In this thesis we identified precision, trustworthiness, scalability, and extensive language support as core challenges concerning the use of software bounded model checking for the verification of runtime errors in embedded software systems. We also presented a number of scientific and technical contributions aiming at the method's practical application. In addition, the thesis provided an in-depth insight into the award-winning, state-of-the-art low-level software bounded model checker LLBMC.

In particular, we showed how LLBMC leverages the compiler framework LLVM, its intermediate representation LLVM-IR, and compiler optimizations for use in software bounded model checking. However, using a compiler intermediate representation is not without disadvantages, e.g. the required adaptations to the compiler's front-end and optimizations in order to support this markedly different use case, and the thesis therefore discussed suitable measures to counteract these.

LLBMC's memory model was also discussed in this thesis: It uses a flat memory model, which leverages an SMT solver's performance, primarily by having the SMT solver's highly-optimized decision procedure for the theory of arrays decide pointer aliasing. Formula simplification and memory objects with fixed addresses are used to counteract the approach's negative impacts on the tool's performance. We participated with LLBMC in multiple competitions which showed that LLBMC's memory model provides a good compromise between precision and performance. The thesis introduced a highly precise model for C-style dynamic memory allocation, which is an advancement on previously published research.

The thesis introduced LLBMC's intermediate logic representation ILR, which is specifically designed as a logic counterpart to LLVM-IR, as well as a term rewriting-based formalization of LLBMC's logical encoding, which clearly showed the assumptions made by the tool about the code under verification. In addition, the thesis described how term rewriting can be used for simplification of ILR formulæ, for the reduction of dynamic memory accesses to pure bitvector logic, and finally for the translation from ILR to a quantifier-free formula for the logic of bitvectors and arrays.

The combination of the contributions listed above comprise a first step towards the application of software bounded model checking for the static analysis of runtime errors in real-life embedded C and C++ programs.

Appendix A

Simplification Rules

A.1 Constant Propagation

A.1.1 Boolean

$$\mathrm{and}(T,T) \longrightarrow T \tag{A.1a}$$
$$\mathrm{and}(T,F) \longrightarrow F \tag{A.1b}$$
$$\mathrm{and}(F,T) \longrightarrow F \tag{A.1c}$$
$$\mathrm{and}(F,F) \longrightarrow F \tag{A.1d}$$
$$\mathrm{or}(T,T) \longrightarrow T \tag{A.1e}$$
$$\mathrm{or}(T,F) \longrightarrow T \tag{A.1f}$$
$$\mathrm{or}(F,T) \longrightarrow T \tag{A.1g}$$
$$\mathrm{or}(F,F) \longrightarrow F \tag{A.1h}$$
$$\mathrm{eq}(F,F) \longrightarrow T \tag{A.1i}$$
$$\mathrm{eq}(T,F) \longrightarrow F \tag{A.1j}$$
$$\mathrm{eq}(F,T) \longrightarrow F \tag{A.1k}$$
$$\mathrm{eq}(T,T) \longrightarrow F \tag{A.1l}$$
$$\mathrm{not}(F) \longrightarrow T \tag{A.1m}$$
$$\mathrm{not}(T) \longrightarrow F \tag{A.1n}$$

A.1.2 Arithmetics

$$\mathrm{add}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (n+m)_{\mathcal{I}} \tag{A.2a}$$
$$\mathrm{sub}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (n-m)_{\mathcal{I}} \tag{A.2b}$$
$$\mathrm{mul}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (n\cdot m)_{\mathcal{I}} \tag{A.2c}$$
$$\mathrm{div}^{u}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow ({}^{n}/_{m})_{\mathcal{I}} \tag{A.2d}$$
$$\mathrm{div}^{s}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow ({}^{n}/_{m})_{\mathcal{I}} \tag{A.2e}$$

$$\mathrm{rem}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (n \operatorname{rem} m)_{\mathcal{I}} \quad \text{(A.2f)}$$
$$\mathrm{rem}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (n \operatorname{rem} m)_{\mathcal{I}} \quad \text{(A.2g)}$$

A.1.3 Overflows

$$\mathrm{addo}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n+m > \mathrm{intmax}^u_{\mathcal{I}} \quad \text{(A.3a)}$$
$$\mathrm{addo}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n+m \leq \mathrm{intmax}^u_{\mathcal{I}} \quad \text{(A.3b)}$$
$$\mathrm{addo}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n+m > \mathrm{intmax}^s_{\mathcal{I}} \vee n+m < \mathrm{intmin}^u_{\mathcal{I}} \quad \text{(A.3c)}$$
$$\mathrm{addo}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n+m \leq \mathrm{intmax}^s_{\mathcal{I}} \wedge n+m \geq \mathrm{intmin}^u_{\mathcal{I}} \quad \text{(A.3d)}$$
$$\mathrm{subo}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n-m < 0 \quad \text{(A.3e)}$$
$$\mathrm{subo}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n-m \geq 0 \quad \text{(A.3f)}$$
$$\mathrm{subo}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n-m > \mathrm{intmax}^s_{\mathcal{I}} \vee n-m < \mathrm{intmin}^u_{\mathcal{I}} \quad \text{(A.3g)}$$
$$\mathrm{subo}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n-m \leq \mathrm{intmax}^s_{\mathcal{I}} \wedge n-m \geq \mathrm{intmin}^u_{\mathcal{I}} \quad \text{(A.3h)}$$
$$\mathrm{mulo}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n \cdot m > \mathrm{intmax}^u_{\mathcal{I}} \quad \text{(A.3i)}$$
$$\mathrm{mulo}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n \cdot m \leq \mathrm{intmax}^u_{\mathcal{I}} \quad \text{(A.3j)}$$
$$\mathrm{mulo}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n \cdot m > \mathrm{intmax}^s_{\mathcal{I}} \vee n \cdot m < \mathrm{intmin}^u_{\mathcal{I}} \quad \text{(A.3k)}$$
$$\mathrm{mulo}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n \cdot m \leq \mathrm{intmax}^s_{\mathcal{I}} \wedge n \cdot m \geq \mathrm{intmin}^u_{\mathcal{I}} \quad \text{(A.3l)}$$
$$\mathrm{divo}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n = \mathrm{intmin}^s_{\mathcal{I}} \wedge m = -1 \quad \text{(A.3m)}$$
$$\mathrm{divo}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n \neq \mathrm{intmin}^s_{\mathcal{I}} \vee m \neq -1 \quad \text{(A.3n)}$$
$$\mathrm{divz}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, m = 0 \quad \text{(A.3o)}$$
$$\mathrm{divz}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, m \neq 0 \quad \text{(A.3p)}$$
$$\mathrm{divx}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n \operatorname{rem} m \neq 0 \quad \text{(A.3q)}$$
$$\mathrm{divx}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n \operatorname{rem} m = 0 \quad \text{(A.3r)}$$
$$\mathrm{divx}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n \operatorname{rem} m \neq 0 \quad \text{(A.3s)}$$
$$\mathrm{divx}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n \operatorname{rem} m = 0 \quad \text{(A.3t)}$$
$$\mathrm{sho}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, m \geq |I| \quad \text{(A.3u)}$$
$$\mathrm{sho}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, m < |I| \quad \text{(A.3v)}$$
$$\mathrm{shrx}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n \operatorname{rem} 2^m \neq 0 \quad \text{(A.3w)}$$
$$\mathrm{shrx}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n \operatorname{rem} 2^m = 0 \quad \text{(A.3x)}$$
$$\mathrm{shrx}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow T\,;\, n \operatorname{rem} 2^m \neq 0 \quad \text{(A.3y)}$$
$$\mathrm{shrx}^u_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow F\,;\, n \operatorname{rem} 2^m = 0 \quad \text{(A.3z)}$$

A.1.4 Bitwise and Shift

$$\mathrm{and}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (l)_{\mathcal{I}}\,;\, \forall i\,\big(0 \leq i < |m| \rightarrow l[i] = m[i] \wedge n[i]\big) \quad \text{(A.4a)}$$
$$\mathrm{or}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (l)_{\mathcal{I}}\,;\, \forall i\,\big(0 \leq i < |m| \rightarrow l[i] = m[i] \vee n[i]\big) \quad \text{(A.4b)}$$
$$\mathrm{xor}_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (l)_{\mathcal{I}}\,;\, \forall i\,\big(0 \leq i < |m| \rightarrow l[i] = \neg(m[i] \leftrightarrow n[i])\big) \quad \text{(A.4c)}$$
$$\mathrm{not}_I((n)_{\mathcal{I}}) \longrightarrow (l)_{\mathcal{I}}\,;\, \forall i\,\big(0 \leq i < |n| \rightarrow l[i] = \neg n[i]\big) \quad \text{(A.4d)}$$
$$\mathrm{shr}^s_{\mathcal{I}}((n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (n/2^m)_{\mathcal{I}} \quad \text{(A.4e)}$$

$$\mathrm{shr}^u_{\mathcal{I}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow ({}^{n}\!/_{2^m})_{\mathcal{I}} \tag{A.4f}$$
$$\mathrm{shl}_{\mathcal{I}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow (2^m n)_{\mathcal{I}} \tag{A.4g}$$

A.1.5 Memory

$$\mathrm{load}_{\mathsf{i8}}(\mathrm{store}_{\mathsf{i8}}(m, (p)_{\mathsf{i8}*}, (n)_{\mathsf{i8}}), (q)_{\mathsf{i8}*}) \longrightarrow (n)_{\mathsf{i8}} \; ; \; p = q \tag{A.5a}$$
$$\mathrm{load}_{\mathsf{i8}}(\mathrm{select}_{\mathsf{i8}}(c, m_1, m_2), p) \longrightarrow \mathrm{load}_{\mathsf{i8}}(m_1, p) \; ; \; c \tag{A.5b}$$
$$\mathrm{load}_{\mathsf{i8}}(\mathrm{select}_{\mathsf{i8}}(c, m_1, m_2)) \longrightarrow \mathrm{load}_{\mathsf{i8}}(m_2, p) \; ; \; \neg c \tag{A.5c}$$
$$\mathrm{add}_{\mathcal{P},\mathcal{V}}((n)_{\mathcal{P}}, (m)_{\mathcal{I}}) \longrightarrow (n+m)_{\mathcal{P}} \tag{A.5d}$$
$$\mathrm{sub}_{\mathcal{P},\mathcal{V}}((n)_{\mathcal{P}}, (m)_{\mathcal{I}}) \longrightarrow (n-m)_{\mathcal{P}} \tag{A.5e}$$
(A.5f)

A.1.6 Comparison

$$\mathrm{eq}_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow T \; ; \; (n)_{\mathcal{I}} = (m)_{\mathcal{I}} \tag{A.6a}$$
$$\mathrm{eq}_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow F \; ; \; (n)_{\mathcal{I}} \neq (m)_{\mathcal{I}} \tag{A.6b}$$
$$\mathrm{ne}_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow T \; ; \; (n)_{\mathcal{I}} \neq (m)_{\mathcal{I}} \tag{A.6c}$$
$$\mathrm{ne}_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow F \; ; \; (n)_{\mathcal{I}} = (m)_{\mathcal{I}} \tag{A.6d}$$
$$\mathrm{ge}^u_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow T \; ; \; (n)_{\mathcal{I}} \geq (m)_{\mathcal{I}} \tag{A.6e}$$
$$\mathrm{ge}^u_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow F \; ; \; (n)_{\mathcal{I}} \ngeq (m)_{\mathcal{I}} \tag{A.6f}$$
$$\mathrm{le}^u_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow T \; ; \; (n)_{\mathcal{I}} \leq (m)_{\mathcal{I}} \tag{A.6g}$$
$$\mathrm{le}^u_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow F \; ; \; (n)_{\mathcal{I}} \nleq (m)_{\mathcal{I}} \tag{A.6h}$$
$$\mathrm{gt}^u_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow T \; ; \; (n)_{\mathcal{I}} > (m)_{\mathcal{I}} \tag{A.6i}$$
$$\mathrm{gt}^u_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow F \; ; \; (n)_{\mathcal{I}} \ngtr (m)_{\mathcal{I}} \tag{A.6j}$$
$$\mathrm{lt}^u_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow T \; ; \; (n)_{\mathcal{I}} < (m)_{\mathcal{I}} \tag{A.6k}$$
$$\mathrm{lt}^u_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow F \; ; \; (n)_{\mathcal{I}} \nless (m)_{\mathcal{I}} \tag{A.6l}$$
$$\mathrm{ge}^s_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow T \; ; \; (n)_{\mathcal{I}} \geq (m)_{\mathcal{I}} \tag{A.6m}$$
$$\mathrm{ge}^s_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow F \; ; \; (n)_{\mathcal{I}} \ngeq (m)_{\mathcal{I}} \tag{A.6n}$$
$$\mathrm{le}^s_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow T \; ; \; (n)_{\mathcal{I}} \leq (m)_{\mathcal{I}} \tag{A.6o}$$
$$\mathrm{le}^s_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow F \; ; \; (n)_{\mathcal{I}} \nleq (m)_{\mathcal{I}} \tag{A.6p}$$
$$\mathrm{gt}^s_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow T \; ; \; (n)_{\mathcal{I}} > (m)_{\mathcal{I}} \tag{A.6q}$$
$$\mathrm{gt}^s_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow F \; ; \; (n)_{\mathcal{I}} \ngtr (m)_{\mathcal{I}} \tag{A.6r}$$
$$\mathrm{lt}^s_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow T \; ; \; (n)_{\mathcal{I}} < (m)_{\mathcal{I}} \tag{A.6s}$$
$$\mathrm{lt}^s_{\mathcal{V}}((n)_{\mathcal{I}}, (m)_{\mathcal{I}}) \longrightarrow F \; ; \; (n)_{\mathcal{I}} \nless (m)_{\mathcal{I}} \tag{A.6t}$$

A.1.7 Miscellaneous

$$\mathrm{ext}^u_{\mathcal{I}_1,\mathcal{I}_2}((n)_{\mathcal{I}_1}) \longrightarrow (n)_{\mathcal{I}_2} \tag{A.7a}$$

$$\text{ext}^s_{\mathcal{I}_1,\mathcal{I}_2}((n)_{\mathcal{I}_1}) \longrightarrow (n)_{\mathcal{I}_2} \tag{A.7b}$$
$$\text{trunc}_{\mathcal{I}_1,\mathcal{I}_2}((n)_{\mathcal{I}_1}) \longrightarrow (n)_{\mathcal{I}_2} \tag{A.7c}$$
$$\text{inttoptr}_{\mathcal{I},\mathcal{P}}((n)_{\mathcal{I}}) \longrightarrow (n)_{\mathcal{P}} \tag{A.7d}$$
$$\text{ptrtoint}_{\mathcal{P},\mathcal{I}}((n)_{\mathcal{P}}) \longrightarrow (n)_{\mathcal{I}} \tag{A.7e}$$
$$\text{bitcast}_{\mathcal{P}_1,\mathcal{P}_2}((n)_{\mathcal{P}_1}) \longrightarrow (n)_{\mathcal{P}_2} \tag{A.7f}$$
$$\text{select}_{\mathcal{I}}(T,(n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (n)_{\mathcal{I}} \tag{A.7g}$$
$$\text{select}_{\mathcal{I}}(F,(n)_{\mathcal{I}},(m)_{\mathcal{I}}) \longrightarrow (m)_{\mathcal{I}} \tag{A.7h}$$
$$\text{select}_{\mathcal{P}}(T,(n)_{\mathcal{P}},(m)_{\mathcal{P}}) \longrightarrow (n)_{\mathcal{P}} \tag{A.7i}$$
$$\text{select}_{\mathcal{P}}(F,(n)_{\mathcal{P}},(m)_{\mathcal{P}}) \longrightarrow (m)_{\mathcal{P}} \tag{A.7j}$$
$$\phi_{\mathcal{U}}((m_1)_{\mathcal{I}},b_1,\ldots,(m_n)_{\mathcal{I}},b_n) \longrightarrow (m_i)_{\mathcal{I}}\ ;\ b_i \tag{A.7k}$$
$$\text{concat}_{\mathcal{I}_1,\mathcal{I}_2}((n)_{\mathcal{I}_1},(m)_{\mathcal{I}_2}) \longrightarrow (2^{|\mathcal{I}_2|}n+m)_{\mathcal{I}}\ ;\ |\mathcal{I}| = |\mathcal{I}_1| + |\mathcal{I}_2| \tag{A.7l}$$
$$\text{extract}_{\mathcal{I}_1,i,j}((n)_{\mathcal{I}_1}) \longrightarrow ({}^{n}\!/_{2^j} \bmod 2^{j-i+1})_{\mathcal{I}_2}\ ;\ |\mathcal{I}_2| = j-i+1 \tag{A.7m}$$

A.2 Boolean

A.2.1 Logical Conjunction

$$\text{and}(T,b) \longrightarrow b \tag{A.8a}$$
$$\text{and}(F,b) \longrightarrow F \tag{A.8b}$$
$$\text{and}(b,T) \longrightarrow b \tag{A.8c}$$
$$\text{and}(b,F) \longrightarrow F \tag{A.8d}$$
$$\text{and}(b,b) \longrightarrow b \tag{A.8e}$$
$$\text{and}(b,\text{not}(b)) \longrightarrow F \tag{A.8f}$$
$$\text{and}(\text{not}(b),b) \longrightarrow F \tag{A.8g}$$
$$\text{and}(\text{or}(b_0,b_1),\text{or}(b_0,\text{not}(b_1))) \longrightarrow b_1 \tag{A.8h}$$
$$\text{and}(\text{or}(b_0,b_1),\text{or}(\text{not}(b_0),b_1)) \longrightarrow b_1 \tag{A.8i}$$
$$\text{and}(\text{or}(b_0,b_1),\text{or}(b_1,\text{not}(b_0))) \longrightarrow b_1 \tag{A.8j}$$
$$\text{and}(\text{or}(b_0,b_1),\text{or}(\text{not}(b_1),b_0)) \longrightarrow b_0 \tag{A.8k}$$
$$\text{and}(b_0,\text{or}(b_0,b_1)) \longrightarrow b_0 \tag{A.8l}$$
$$\text{and}(b_0,\text{or}(b_1,b_0)) \longrightarrow b_0 \tag{A.8m}$$
$$\text{and}(\text{or}(b_0,b_1),b_1) \longrightarrow b_1 \tag{A.8n}$$
$$\text{and}(\text{or}(b_0,b_1),b_1) \longrightarrow b_1 \tag{A.8o}$$
$$\text{and}(b_0,\text{not}(\text{or}(b_0,b_1))) \longrightarrow F \tag{A.8p}$$
$$\text{and}(b_0,\text{not}(\text{or}(b_1,b_0))) \longrightarrow F \tag{A.8q}$$
$$\text{and}(\text{not}(\text{or}(b_0,b_1)),b_1) \longrightarrow F \tag{A.8r}$$
$$\text{and}(\text{not}(\text{or}(b_1,b_0)),b_1) \longrightarrow F \tag{A.8s}$$

A.2.2 Logical Disjunction

$$\begin{aligned}
\mathrm{or}(F, b) &\longrightarrow b && \text{(A.9a)}\\
\mathrm{or}(T, b) &\longrightarrow T && \text{(A.9b)}\\
\mathrm{or}(b, F) &\longrightarrow b && \text{(A.9c)}\\
\mathrm{or}(b, T) &\longrightarrow T && \text{(A.9d)}\\
\mathrm{or}(b, b) &\longrightarrow b && \text{(A.9e)}\\
\mathrm{or}(b, \mathrm{not}(b)) &\longrightarrow T && \text{(A.9f)}\\
\mathrm{or}(\mathrm{not}(b), b) &\longrightarrow T && \text{(A.9g)}\\
\mathrm{or}(\mathrm{and}(b_0, b_1), \mathrm{and}(b_0, \mathrm{not}(b_1))) &\longrightarrow b_0 && \text{(A.9h)}\\
\mathrm{or}(\mathrm{and}(b_0, b_1), \mathrm{and}(\mathrm{not}(b_1), b_0)) &\longrightarrow b_0 && \text{(A.9i)}\\
\mathrm{or}(\mathrm{and}(b_1, b_0), \mathrm{and}(b_0, \mathrm{not}(b_1))) &\longrightarrow b_0 && \text{(A.9j)}\\
\mathrm{or}(\mathrm{and}(b_1, b_0), \mathrm{and}(\mathrm{not}(b_1), b_0)) &\longrightarrow b_0 && \text{(A.9k)}\\
\mathrm{or}(\mathrm{not}(\mathrm{and}(b_0, b_1)), b_1) &\longrightarrow T && \text{(A.9l)}\\
\mathrm{or}(\mathrm{not}(\mathrm{and}(b_1, b_0)), b_1) &\longrightarrow T && \text{(A.9m)}\\
\mathrm{or}(b_1, \mathrm{not}(\mathrm{and}(b_0, b_1))) &\longrightarrow T && \text{(A.9n)}\\
\mathrm{or}(b_1, \mathrm{not}(\mathrm{and}(b_1, b_0))) &\longrightarrow T && \text{(A.9o)}\\
\mathrm{or}(b_0, \mathrm{and}(b_0, b_1)) &\longrightarrow b_0 && \text{(A.9p)}\\
\mathrm{or}(b_0, \mathrm{and}(b_1, b_0)) &\longrightarrow b_0 && \text{(A.9q)}\\
\mathrm{or}(\mathrm{and}(b_0, b_1), b_0) &\longrightarrow b_0 && \text{(A.9r)}\\
\mathrm{or}(\mathrm{and}(b_1, b_0), b_0) &\longrightarrow b_0 && \text{(A.9s)}
\end{aligned}$$

A.2.3 Logical Equivalence

$$\begin{aligned}
\mathrm{eq}(b, T) &\longrightarrow b && \text{(A.10a)}\\
\mathrm{eq}(T, b) &\longrightarrow b && \text{(A.10b)}\\
\mathrm{eq}(b, F) &\longrightarrow \mathrm{not}(b) && \text{(A.10c)}\\
\mathrm{eq}(F, b) &\longrightarrow \mathrm{not}(b) && \text{(A.10d)}\\
\mathrm{eq}(b, b) &\longrightarrow T && \text{(A.10e)}\\
\mathrm{eq}(b, \mathrm{not}(b)) &\longrightarrow F && \text{(A.10f)}\\
\mathrm{eq}(\mathrm{not}(b), b) &\longrightarrow F && \text{(A.10g)}
\end{aligned}$$

A.2.4 Negation

$$\mathrm{not}(\mathrm{not}(b)) \longrightarrow b \qquad \text{(A.11a)}$$

A.3 Arithmetic

A.3.1 Addition

$$\mathrm{add}_{\mathcal{I}}(x, 0) \longrightarrow x \tag{A.12a}$$
$$\mathrm{add}_{\mathcal{I}}(0, x) \longrightarrow x \tag{A.12b}$$
$$\mathrm{add}_{\mathcal{I}}(x, \mathrm{sub}_{\mathcal{I}}(y, x)) \longrightarrow y \tag{A.12c}$$
$$\mathrm{add}_{\mathcal{I}}(\mathrm{sub}_{\mathcal{I}}(y, x), x)) \longrightarrow y \tag{A.12d}$$

A.3.2 Subtraction

$$\mathrm{sub}_{\mathcal{I}}(x, 0) \longrightarrow x \tag{A.13a}$$
$$\mathrm{sub}_{\mathcal{I}}(x, x) \longrightarrow 0 \tag{A.13b}$$
$$\mathrm{sub}_{\mathcal{I}}(\mathrm{add}_{\mathcal{I}}(x, y), x) \longrightarrow y \tag{A.13c}$$
$$\mathrm{sub}_{\mathcal{I}}(\mathrm{add}_{\mathcal{I}}(x, y), y) \longrightarrow x \tag{A.13d}$$
$$\mathrm{sub}_{\mathcal{I}}(x, \mathrm{add}_{\mathcal{I}}(x, y)) \longrightarrow \mathrm{sub}_{\mathcal{I}}((0)_{\mathcal{I}}, y) \tag{A.13e}$$
$$\mathrm{sub}_{\mathcal{I}}(y, \mathrm{add}_{\mathcal{I}}(x, y)) \longrightarrow \mathrm{sub}_{\mathcal{I}}((0)_{\mathcal{I}}, x) \tag{A.13f}$$

A.3.3 Multiplication

$$\mathrm{mul}_{\mathcal{I}}(1, x) \longrightarrow x \tag{A.14a}$$
$$\mathrm{mul}_{\mathcal{I}}(x, 1) \longrightarrow x \tag{A.14b}$$
$$\mathrm{mul}_{\mathcal{I}}(0, x) \longrightarrow 0 \tag{A.14c}$$
$$\mathrm{mul}_{\mathcal{I}}(x, 0) \longrightarrow 0 \tag{A.14d}$$
$$\mathrm{mul}_{\mathcal{I}}(x, (2^m)_{\mathcal{I}}) \longrightarrow \mathrm{shl}_{\mathcal{I}}(x, (m)_{\mathcal{I}} \tag{A.14e}$$

A.3.4 Division

$$\mathrm{div}^u_{\mathcal{I}}(x, 1) \longrightarrow x \tag{A.15a}$$
$$\mathrm{div}^u_{\mathcal{I}}(x, (2^m)_{\mathcal{I}}) \longrightarrow \mathrm{shr}^u_{\mathcal{I}}(x, (m)_{\mathcal{I}} \tag{A.15b}$$
$$\mathrm{div}^u_{\mathcal{I}}(x, x) \longrightarrow (1)_{\mathcal{I}} \tag{A.15c}$$
$$\mathrm{div}^s_{\mathcal{I}}(x, 1) \longrightarrow x \tag{A.15d}$$
$$\mathrm{div}^s_{\mathcal{I}}(x, (-1)_{\mathcal{I}}) \longrightarrow \mathrm{sub}_{\mathcal{I}}((0)_{\mathcal{I}}, x) \tag{A.15e}$$
$$\mathrm{div}^s_{\mathcal{I}}(x, x) \longrightarrow (1)_{\mathcal{I}} \tag{A.15f}$$
$$\mathrm{div}^s_{\mathcal{I}}(x, (\mathrm{intmin}^u_{\mathcal{I}})_{\mathcal{I}}) \longrightarrow \mathrm{select}_{\mathcal{I}}(\mathrm{eq}_{\mathcal{I}}(x, (\mathrm{intmin}^u_{\mathcal{I}})_{\mathcal{I}}), (1)_{\mathcal{I}}, (0)_{\mathcal{I}}) \tag{A.15g}$$

A.3.5 Remainder

$$\begin{aligned}
\mathrm{rem}^u_{\mathcal{I}}(x,(1)_{\mathcal{I}}) &\longrightarrow (0)_{\mathcal{I}} && \text{(A.16a)}\\
\mathrm{rem}^u_{\mathcal{I}}(x,(m)_{\mathcal{I}}) &\longrightarrow \mathrm{and}_{\mathcal{I}}(x,(m-1)_{\mathcal{I}})\,;\,\exists n\in\mathbb{N}(2^n=m) && \text{(A.16b)}\\
\mathrm{rem}^u_{\mathcal{I}}(x,x) &\longrightarrow (0)_{\mathcal{I}} && \text{(A.16c)}\\
\mathrm{rem}^s_{\mathcal{I}}(x,(1)_{\mathcal{I}}) &\longrightarrow (0)_{\mathcal{I}} && \text{(A.16d)}\\
\mathrm{rem}^s_{\mathcal{I}}(x,(-1)_{\mathcal{I}}) &\longrightarrow (0)_{\mathcal{I}} && \text{(A.16e)}\\
\mathrm{rem}^s_{\mathcal{I}}(x,x) &\longrightarrow (0)_{\mathcal{I}} && \text{(A.16f)}\\
\mathrm{rem}^s_{\mathcal{I}}(x,(m)_{\mathcal{I}}) &\longrightarrow \mathrm{select}_{\mathcal{I}}(\\
&\qquad \mathrm{ge}^s_{\mathcal{I}}(x,(0)_{\mathcal{I}}),\\
&\qquad \mathrm{and}_{\mathcal{I}}(x,(m-1)_{\mathcal{I}}),\\
&\qquad \mathrm{sub}_{\mathcal{I}}(0,\mathrm{and}_{\mathcal{I}}(\mathrm{sub}_{\mathcal{I}}((0)_{\mathcal{I}}-x),(m-1)_{\mathcal{I}})))\,;\\
&\quad \exists n\in\mathbb{N}(2^n=m) && \text{(A.16g)}
\end{aligned}$$

A.4 Safety

$$\begin{aligned}
\mathrm{addo}^u_{\mathcal{I}}(x,(0)_{\mathcal{I}}) &\longrightarrow F && \text{(A.17a)}\\
\mathrm{addo}^u_{\mathcal{I}}((0)_{\mathcal{I}},y) &\longrightarrow F && \text{(A.17b)}\\
\mathrm{addo}^s_{\mathcal{I}}(x,(0)_{\mathcal{I}}) &\longrightarrow F && \text{(A.17c)}\\
\mathrm{addo}^s_{\mathcal{I}}((0)_{\mathcal{I}},y) &\longrightarrow F && \text{(A.17d)}\\
\mathrm{subo}^s_{\mathcal{I}}(x,(0)_{\mathcal{I}}) &\longrightarrow F && \text{(A.17e)}\\
\mathrm{mulo}^s_{\mathcal{I}}((0)_{\mathcal{I}},y) &\longrightarrow F && \text{(A.17f)}\\
\mathrm{mulo}^s_{\mathcal{I}}((1)_{\mathcal{I}},y) &\longrightarrow F && \text{(A.17g)}\\
\mathrm{mulo}^s_{\mathcal{I}}(x,(0)_{\mathcal{I}}) &\longrightarrow F && \text{(A.17h)}\\
\mathrm{mulo}^s_{\mathcal{I}}(x,(1)_{\mathcal{I}}) &\longrightarrow F && \text{(A.17i)}\\
\mathrm{divo}^s_{\mathcal{I}}(x,y) &\longrightarrow \mathrm{eq}_{\mathcal{I}}(x,\mathrm{intmin}^s_{\mathcal{I}})\wedge\mathrm{eq}_{\mathcal{I}}(y,(-1)_{\mathcal{I}}) && \text{(A.17j)}\\
\mathrm{addo}^u_{\mathcal{I}}(\mathrm{ext}^s_{\mathcal{I}_1,\mathcal{I}}(x),\mathrm{ext}^s_{\mathcal{I}_2,\mathcal{I}}(y)) &\longrightarrow F\,;\,|\mathcal{I}_1|<|I|\wedge|\mathcal{I}_2|<|I| && \text{(A.17k)}\\
\mathrm{addo}^s_{\mathcal{I}}(\mathrm{ext}^s_{\mathcal{I}_1,\mathcal{I}}(x),\mathrm{ext}^s_{\mathcal{I}_2,\mathcal{I}}(y)) &\longrightarrow F\,;\,|\mathcal{I}_1|<|I|\wedge|\mathcal{I}_2|<|I| && \text{(A.17l)}
\end{aligned}$$

A.5 Bitwise Operations

A.5.1 Bitwise Conjunction

$$\begin{aligned}
\mathrm{and}_{\mathcal{I}}((1)_{\mathcal{I}},x) &\longrightarrow x && \text{(A.18a)}\\
\mathrm{and}_{\mathcal{I}}((0)_{\mathcal{I}},x) &\longrightarrow (0)_{\mathcal{I}} && \text{(A.18b)}\\
\mathrm{and}_{\mathcal{I}}(x,(1)_{\mathcal{I}}) &\longrightarrow x && \text{(A.18c)}\\
\mathrm{and}_{\mathcal{I}}(x,(0)_{\mathcal{I}}) &\longrightarrow (0)_{\mathcal{I}} && \text{(A.18d)}\\
\mathrm{and}_{\mathcal{I}}(x,x) &\longrightarrow x && \text{(A.18e)}
\end{aligned}$$

$$\text{and}_{\mathcal{I}}(x, \text{not}_{\mathcal{I}}(x)) \longrightarrow (0)_{\mathcal{I}} \quad \text{(A.18f)}$$
$$\text{and}_{\mathcal{I}}(\text{not}_{\mathcal{I}}(x), x) \longrightarrow (0)_{\mathcal{I}} \quad \text{(A.18g)}$$
$$\text{and}_{\mathcal{I}}(\text{or}_{\mathcal{I}}(x, y), \text{or}_{\mathcal{I}}(x, \text{not}_{\mathcal{I}}(y))) \longrightarrow x \quad \text{(A.18h)}$$
$$\text{and}_{\mathcal{I}}(\text{or}_{\mathcal{I}}(x, y), \text{or}_{\mathcal{I}}(\text{not}_{\mathcal{I}}(x), y)) \longrightarrow y \quad \text{(A.18i)}$$
$$\text{and}_{\mathcal{I}}(\text{or}_{\mathcal{I}}(x, y), \text{or}_{\mathcal{I}}(y, \text{not}_{\mathcal{I}}(x)) \longrightarrow y \quad \text{(A.18j)}$$
$$\text{and}_{\mathcal{I}}(\text{or}_{\mathcal{I}}(x, y), \text{or}_{\mathcal{I}}(\text{not}_{\mathcal{I}}(y), x)) \longrightarrow x \quad \text{(A.18k)}$$
$$\text{and}_{\mathcal{I}}(x, \text{or}_{\mathcal{I}}(x, y)) \longrightarrow x \quad \text{(A.18l)}$$
$$\text{and}_{\mathcal{I}}(x, \text{or}_{\mathcal{I}}(y, x)) \longrightarrow x \quad \text{(A.18m)}$$
$$\text{and}_{\mathcal{I}}(\text{or}_{\mathcal{I}}(x, y), y)) \longrightarrow y \quad \text{(A.18n)}$$
$$\text{and}_{\mathcal{I}}(\text{or}_{\mathcal{I}}(x, y), y)) \longrightarrow y \quad \text{(A.18o)}$$
$$\text{and}_{\mathcal{I}}(x, \text{not}_{\mathcal{I}}(\text{or}_{\mathcal{I}}(x, y))) \longrightarrow (0)_{\mathcal{I}} \quad \text{(A.18p)}$$
$$\text{and}_{\mathcal{I}}(x, \text{not}_{\mathcal{I}}(\text{or}_{\mathcal{I}}(y, x))) \longrightarrow (0)_{\mathcal{I}} \quad \text{(A.18q)}$$
$$\text{and}_{\mathcal{I}}(\text{not}_{\mathcal{I}}(\text{or}_{\mathcal{I}}(x, y)), y) \longrightarrow (0)_{\mathcal{I}} \quad \text{(A.18r)}$$
$$\text{and}_{\mathcal{I}}(\text{not}_{\mathcal{I}}(\text{or}_{\mathcal{I}}(y, x)), y) \longrightarrow (0)_{\mathcal{I}} \quad \text{(A.18s)}$$

A.5.2 Bitwise Disjunction

$$\text{or}_{\mathcal{I}}((0)_{\mathcal{I}}, x) \longrightarrow x \quad \text{(A.19a)}$$
$$\text{or}_{\mathcal{I}}((-1)_{\mathcal{I}}, x) \longrightarrow (-1)_{\mathcal{I}} \quad \text{(A.19b)}$$
$$\text{or}_{\mathcal{I}}(x, (0)_{\mathcal{I}}) \longrightarrow x \quad \text{(A.19c)}$$
$$\text{or}_{\mathcal{I}}(x, (-1)_{\mathcal{I}}) \longrightarrow (-1)_{\mathcal{I}} \quad \text{(A.19d)}$$
$$\text{or}_{\mathcal{I}}(x, x) \longrightarrow x \quad \text{(A.19e)}$$
$$\text{or}_{\mathcal{I}}(x, \text{not}_{\mathcal{I}}(x)) \longrightarrow (-1)_{\mathcal{I}} \quad \text{(A.19f)}$$
$$\text{or}_{\mathcal{I}}(\text{not}_{\mathcal{I}}(x), x) \longrightarrow (-1)_{\mathcal{I}} \quad \text{(A.19g)}$$
$$\text{or}_{\mathcal{I}}(\text{and}_{\mathcal{I}}(x, y), \text{and}_{\mathcal{I}}(x, \text{not}_{\mathcal{I}}(y))) \longrightarrow x \quad \text{(A.19h)}$$
$$\text{or}_{\mathcal{I}}(\text{and}_{\mathcal{I}}(x, y), \text{and}_{\mathcal{I}}(\text{not}_{\mathcal{I}}(y), x)) \longrightarrow x \quad \text{(A.19i)}$$
$$\text{or}_{\mathcal{I}}(\text{and}_{\mathcal{I}}(y, x), \text{and}_{\mathcal{I}}(x, \text{not}_{\mathcal{I}}(y))) \longrightarrow x \quad \text{(A.19j)}$$
$$\text{or}_{\mathcal{I}}(\text{and}_{\mathcal{I}}(y, x), \text{and}_{\mathcal{I}}(\text{not}_{\mathcal{I}}(y), x)) \longrightarrow x \quad \text{(A.19k)}$$
$$\text{or}_{\mathcal{I}}(\text{not}_{\mathcal{I}}(\text{and}_{\mathcal{I}}(x, y)), y) \longrightarrow (-1)_{\mathcal{I}} \quad \text{(A.19l)}$$
$$\text{or}_{\mathcal{I}}(\text{not}_{\mathcal{I}}(\text{and}_{\mathcal{I}}(y, x)), y) \longrightarrow (-1)_{\mathcal{I}} \quad \text{(A.19m)}$$
$$\text{or}_{\mathcal{I}}(y, \text{not}_{\mathcal{I}}(\text{and}_{\mathcal{I}}(x, y))) \longrightarrow (-1)_{\mathcal{I}} \quad \text{(A.19n)}$$
$$\text{or}_{\mathcal{I}}(y, \text{not}_{\mathcal{I}}(\text{and}_{\mathcal{I}}(y, x))) \longrightarrow (-1)_{\mathcal{I}} \quad \text{(A.19o)}$$
$$\text{or}_{\mathcal{I}}(x, \text{and}_{\mathcal{I}}(x, y)) \longrightarrow x \quad \text{(A.19p)}$$
$$\text{or}_{\mathcal{I}}(x, \text{and}_{\mathcal{I}}(y, x)) \longrightarrow x \quad \text{(A.19q)}$$
$$\text{or}_{\mathcal{I}}(\text{and}_{\mathcal{I}}(x, y), x) \longrightarrow x \quad \text{(A.19r)}$$
$$\text{or}_{\mathcal{I}}(\text{and}_{\mathcal{I}}(y, x), x) \longrightarrow x \quad \text{(A.19s)}$$

A.5.3 Bitwise Exclusive Or

$$\mathrm{xor}_{\mathcal{I}}(x, (0)_{\mathcal{I}}) \longrightarrow x \tag{A.20a}$$
$$\mathrm{xor}_{\mathcal{I}}(x, (-1)_{\mathcal{I}}) \longrightarrow \mathrm{not}_{\mathcal{I}}(x) \tag{A.20b}$$
$$\mathrm{xor}_{\mathcal{I}}((0)_{\mathcal{I}}, y) \longrightarrow y \tag{A.20c}$$
$$\mathrm{xor}_{\mathcal{I}}((-1)_{\mathcal{I}}, y) \longrightarrow \mathrm{not}_{\mathcal{I}}(y) \tag{A.20d}$$
$$\mathrm{xor}_{\mathcal{I}}(x, x) \longrightarrow (0)_{\mathcal{I}} \tag{A.20e}$$
$$\mathrm{xor}_{\mathcal{I}}(x, \mathrm{not}_{\mathcal{I}}(x)) \longrightarrow (-1)_{\mathcal{I}} \tag{A.20f}$$
$$\mathrm{xor}_{\mathcal{I}}(\mathrm{not}_{\mathcal{I}}(x), x) \longrightarrow (-1)_{\mathcal{I}} \tag{A.20g}$$
$$\mathrm{xor}_{\mathcal{I}}(\mathrm{not}_{\mathcal{I}}(x), \mathrm{not}_{\mathcal{I}}(y)) \longrightarrow \mathrm{xor}_{\mathcal{I}}(x, y) \tag{A.20h}$$

A.5.4 Bitwise Negation

$$\mathrm{not}_I(\mathrm{not}_I(x)) \longrightarrow x \tag{A.21a}$$
$$\mathrm{not}_I(\mathrm{not}_I(x)) \longrightarrow x \tag{A.21b}$$
$$\mathrm{not}_I(\mathrm{eq}_I(x, y)) \longrightarrow \mathrm{ne}_I(x, y) \tag{A.21c}$$
$$\mathrm{not}_I(\mathrm{ne}_I(x, y)) \longrightarrow \mathrm{eq}_I(x, y) \tag{A.21d}$$
$$\mathrm{not}_I(\mathrm{lt}^u_I(x, y)) \longrightarrow \mathrm{ge}^u_I(x, y) \tag{A.21e}$$
$$\mathrm{not}_I(\mathrm{le}^u_I(x, y)) \longrightarrow \mathrm{gt}^u_I(x, y) \tag{A.21f}$$
$$\mathrm{not}_I(\mathrm{gt}^u_I(x, y)) \longrightarrow \mathrm{le}^u_I(x, y) \tag{A.21g}$$
$$\mathrm{not}_I(\mathrm{ge}^u_I(x, y)) \longrightarrow \mathrm{ge}^u_I(x, y) \tag{A.21h}$$
$$\mathrm{not}_I(\mathrm{lt}^s_I(x, y)) \longrightarrow \mathrm{ge}^s_I(x, y) \tag{A.21i}$$
$$\mathrm{not}_I(\mathrm{le}^s_I(x, y)) \longrightarrow \mathrm{gt}^s_I(x, y) \tag{A.21j}$$
$$\mathrm{not}_I(\mathrm{gt}^s_I(x, y)) \longrightarrow \mathrm{le}^s_I(x, y) \tag{A.21k}$$
$$\mathrm{not}_I(\mathrm{ge}^s_I(x, y)) \longrightarrow \mathrm{ge}^s_I(x, y) \tag{A.21l}$$

A.6 Shifts

$$\mathrm{shr}^s_{\mathcal{I}}(x, (0)_{\mathcal{I}}) \longrightarrow x \tag{A.22a}$$
$$\mathrm{shr}^s_{\mathcal{I}}((0)_{\mathcal{I}}, y) \longrightarrow (0)_{\mathcal{I}} \tag{A.22b}$$
$$\mathrm{shr}^s_{\mathcal{I}}((-1)_{\mathcal{I}}, y) \longrightarrow (-1)_{\mathcal{I}} \tag{A.22c}$$
$$\mathrm{shr}^u_{\mathcal{I}}(x, (0)_{\mathcal{I}}) \longrightarrow (0)_{\mathcal{I}} \tag{A.22d}$$
$$\mathrm{shr}^u_{\mathcal{I}}((0)_{\mathcal{I}}, y) \longrightarrow (0)_{\mathcal{I}} \tag{A.22e}$$
$$\mathrm{shl}_{\mathcal{I}}(x, (0)_{\mathcal{I}}) \longrightarrow (0)_{\mathcal{I}} \tag{A.22f}$$
$$\mathrm{shl}_{\mathcal{I}}((0)_{\mathcal{I}}, y) \longrightarrow (0)_{\mathcal{I}} \tag{A.22g}$$

A.7 Comparison

$$\mathrm{eq}_{\mathcal{V}}(x,x) \longrightarrow T \tag{A.23a}$$
$$\mathrm{ne}_{\mathcal{V}}(x,x) \longrightarrow F \tag{A.23b}$$
$$\mathrm{ge}^{u}_{\mathcal{V}}(x,x) \longrightarrow T \tag{A.23c}$$
$$\mathrm{le}^{u}_{\mathcal{V}}(x,x) \longrightarrow T \tag{A.23d}$$
$$\mathrm{gt}^{u}_{\mathcal{V}}(x,x) \longrightarrow F \tag{A.23e}$$
$$\mathrm{lt}^{u}_{\mathcal{V}}(x,x) \longrightarrow F \tag{A.23f}$$
$$\mathrm{ge}^{s}_{\mathcal{V}}(x,x) \longrightarrow T \tag{A.23g}$$
$$\mathrm{le}^{s}_{\mathcal{V}}(x,x) \longrightarrow T \tag{A.23h}$$
$$\mathrm{lt}^{s}_{\mathcal{V}}(x,x) \longrightarrow F \tag{A.23i}$$
$$\mathrm{gt}^{s}_{\mathcal{V}}(x,x) \longrightarrow F \tag{A.23j}$$

A.8 Miscellaneous

$$\mathrm{ext}^{u}_{\mathcal{I}_1,\mathcal{I}_2}(\mathrm{ext}^{u}_{\mathcal{I},\mathcal{I}_1}(x)) \longrightarrow \mathrm{ext}^{u}_{\mathcal{I},\mathcal{I}_2}(x) \tag{A.24a}$$
$$\mathrm{ext}^{s}_{\mathcal{I}_1,\mathcal{I}_2}(\mathrm{ext}^{s}_{\mathcal{I},\mathcal{I}_1}(x)) \longrightarrow \mathrm{ext}^{s}_{\mathcal{I},\mathcal{I}_2}(x) \tag{A.24b}$$
$$\mathrm{trunc}_{\mathcal{I}_1,\mathcal{I}_2}(\mathrm{trunc}_{\mathcal{I},\mathcal{I}_1}(x)) \longrightarrow \mathrm{trunc}_{\mathcal{I},\mathcal{I}_2}(x) \tag{A.24c}$$
$$\mathrm{inttoptr}_{\mathcal{I},\mathcal{P}}(\mathrm{ptrtoint}_{\mathcal{P}_1,\mathcal{I}}(p)) \longrightarrow \mathrm{bitcast}_{\mathcal{P}_1,\mathcal{P}}(x) \tag{A.24d}$$
$$\mathrm{ptrtoint}_{\mathcal{P},\mathcal{I}}(\mathrm{inttoptr}_{\mathcal{I},\mathcal{P}}(i)) \longrightarrow i \tag{A.24e}$$
$$\mathrm{ptrtoint}_{\mathcal{P},\mathcal{I}}(\mathrm{inttoptr}_{\mathcal{I}_1,\mathcal{P}}(i)) \longrightarrow \mathrm{ext}^{u}_{\mathcal{I}_1,\mathcal{I}}(i)\ ;\ |\mathcal{I}_1| < |\mathcal{I}| \tag{A.24f}$$
$$\mathrm{ptrtoint}_{\mathcal{P},\mathcal{I}}(\mathrm{inttoptr}_{\mathcal{I}_1,\mathcal{P}}(i)) \longrightarrow \mathrm{trunc}_{\mathcal{I}_1,\mathcal{I}}(i)\ ;\ |\mathcal{I}_1| > |\mathcal{I}| \tag{A.24g}$$
$$\mathrm{bitcast}_{\mathcal{P},\mathcal{P}}(p) \longrightarrow p \tag{A.24h}$$
$$\mathrm{bitcast}_{\mathcal{P}_1,\mathcal{P}_2}(\mathrm{bitcast}_{\mathcal{P},\mathcal{P}_1}(p) \longrightarrow \mathrm{bitcast}_{\mathcal{P},\mathcal{P}_2}(p) \tag{A.24i}$$
$$\mathrm{select}_{\mathcal{U}}(b,u,u) \longrightarrow u \tag{A.24j}$$
$$\mathrm{select}_{\mathcal{U}}(T,u_1,u_2) \longrightarrow u_1 \tag{A.24k}$$
$$\mathrm{select}_{\mathcal{U}}(F,u_1,u_2) \longrightarrow u_2 \tag{A.24l}$$

A.9 Reduction

A.9.1 Memory

$$\mathrm{load}^{b}_{\mathcal{I}}(m,p) \longrightarrow \mathrm{concat}_{\mathsf{i8},\mathcal{I}}(\mathrm{load}_{\mathsf{i8}}(m,p), \mathrm{load}^{b}_{\mathcal{I}_1}(m,p+1))\ ;\ |\mathcal{I}| > 8 \wedge |\mathcal{I}_1| = |\mathcal{I}| - 8 \tag{A.25a}$$
$$\mathrm{load}^{l}_{\mathcal{I}}(m,p) \longrightarrow \mathrm{concat}_{\mathcal{I},\mathsf{i8}}(\mathrm{load}^{l}_{\mathcal{I}_1}(m,p+1), \mathrm{load}_{\mathsf{i8}}(m,p))\ ;\ |\mathcal{I}| > 8 \wedge |\mathcal{I}_1| = |\mathcal{I}| - 8 \tag{A.25b}$$
$$\mathrm{load}_{\mathcal{I}}(m,p) \longrightarrow \mathrm{trunc}_{\mathsf{i8},\mathcal{I}}(\mathrm{load}_{\mathsf{i8}}(m,p))\ ;\ |\mathcal{I}| < 8 \tag{A.25c}$$

$$\begin{aligned}
\mathrm{store}^{b}_{\mathcal{I}}(m,p,x) &\longrightarrow \mathrm{store}^{b}_{\mathcal{I}_2}(\mathrm{store}_{\mathsf{i8}}(m,p,\mathrm{extract}_{\mathcal{I},|\mathcal{I}|-8,|\mathcal{I}|-1}(x)),\\
&\qquad p+1,\mathrm{extract}_{\mathcal{I},0,|\mathcal{I}|-9}(x)) && \text{(A.25d)}\\
\mathrm{store}^{l}_{\mathcal{I}}(m,p,x) &\longrightarrow \mathrm{store}^{l}_{\mathcal{I}_2}(\mathrm{store}_{\mathsf{i8}}(m,p,\mathrm{extract}_{\mathcal{I},0,7}(x)),\\
&\qquad p+1,\mathrm{extract}_{\mathcal{I},8,|\mathcal{I}|-1}(x)) && \text{(A.25e)}\\
\mathrm{add}_{\mathcal{P},\mathcal{I}}(p,x) &\longrightarrow \mathrm{inttoptr}_{\mathcal{I},\mathcal{P}}(\mathrm{add}_{\mathcal{I}}(\mathrm{ptrtoint}_{\mathcal{P},\mathcal{I}}(p),x)) && \text{(A.25f)}\\
\mathrm{sub}_{\mathcal{P},\mathcal{I}}(p,x) &\longrightarrow \mathrm{inttoptr}_{\mathcal{I},\mathcal{P}}(\mathrm{sub}_{\mathcal{I}}(\mathrm{ptrtoint}_{\mathcal{P},\mathcal{I}}(p),x)) && \text{(A.25g)}
\end{aligned}$$

A.9.2 Miscellaneous

$$\begin{aligned}
\phi_{\mathcal{U}}(u_1,b_1,\ldots,u_{i-1},b_{i-1},u_i,b_i,u_{i+1},b_{i+1},\ldots,u_n,b_n) &\longrightarrow\\
\phi_{\mathcal{U}}(u_1,b_1,\ldots,u_{i-1},b_{i-1},u_{i+1},b_{i+1},\ldots,u_n,b_n)\ ;\ \neg b_i && \text{(A.26a)}\\
\phi_{\mathcal{U}}(u_1,b_1,\ldots u_n,b_n) &\longrightarrow u_i\ ;\ b_i && \text{(A.26b)}
\end{aligned}$$

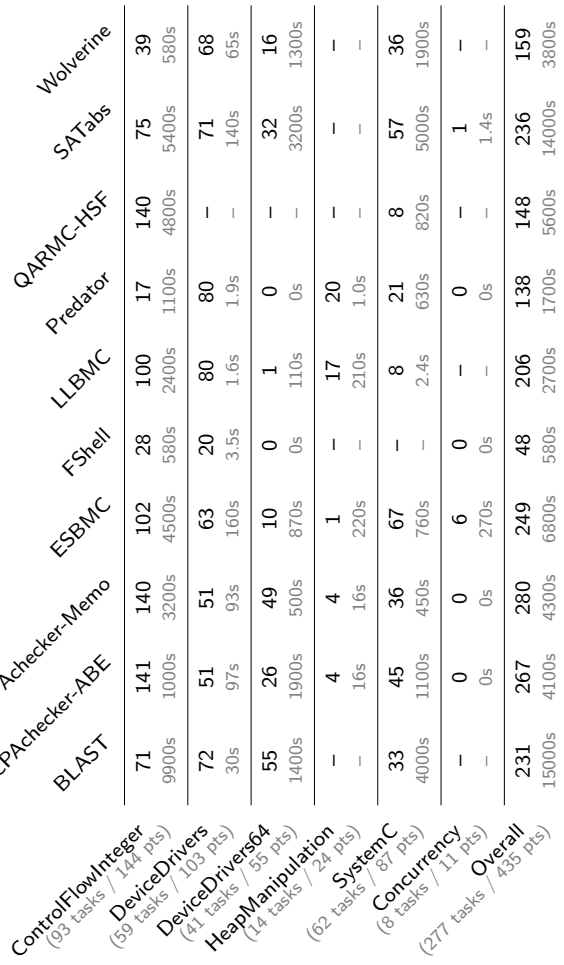
Appendix B

Evaluation Results

B.1 Participants of SV-COMP 2012

Tool	Jury Member	Affiliation
Predator 2011-10-11	Tomas Vojnar	Brno University of Technology, Czech Rep.
BLAST 2.7	Vadim Mutilin	Russian Academy of Sciences, Russia
CPAchecker-Memorizing	Daniel Wonisch	University of Paderborn, Germany
ESBMC 1.17	Bernd Fischer	University of Southampton, UK
LLBMC 0.9	Carsten Sinz	Karlsruhe Institute of Technology, Germany
Wolverine 0.5c	Georg Weissenbacher	Princeton University, USA
CPAchecker-ABE 1.0.10	Philipp Wendler	University of Passau, Germany
SATabs 3.0	Michael Tautschnig	Oxford University, UK
FShell 1.3	Helmut Veith	TU Vienna, Austria
QARMC-HSF	Andrey Rybalchenko	TU Munich, Germany

B.2 Results of SV-COMP 2012[1]



	BLAST	CPAchecker-ABE	CPAchecker-Memo	ESBMC	FShell	LLBMC	Predator	QARMC-HSF	SATabs	Wolverine
ControlFlowInteger (93 tasks / 144 pts)	71 9900s	141 1000s	140 3200s	102 4500s	28 580s	100 2400s	17 1100s	140 4800s	75 5400s	39 580s
DeviceDrivers (59 tasks / 103 pts)	72 30s	51 97s	51 93s	63 160s	20 3.5s	80 1.6s	80 1.9s	– –	71 140s	68 65s
DeviceDrivers64 (41 tasks / 55 pts)	55 1400s	26 1900s	49 500s	10 870s	0 0s	1 110s	0 0s	– –	32 3200s	16 1300s
HeapManipulation (14 tasks / 24 pts)	– –	4 16s	4 16s	1 220s	– –	17 210s	20 1.0s	– –	– –	– –
SystemC (62 tasks / 87 pts)	33 4000s	45 1100s	36 450s	67 760s	– –	8 2.4s	21 630s	8 820s	57 5000s	36 1900s
Concurrency (8 tasks / 11 pts)	– –	0 0s	0 0s	6 270s	0 0s	– –	0 0s	– –	1 1.4s	– –
Overall (277 tasks / 435 pts)	231 15000s	267 4100s	280 4300s	249 6800s	48 580s	206 2700s	138 1700s	148 5600s	236 14000s	159 3800s

[1] Results from http://sv-comp.sosy-lab.org/2012

B.3 Participants of SV-COMP 2013

Tool	Jury Member	Affiliation
BLAST 2.7.1	Vadim Mutilin	Russian Academy of Sciences, Russia
CPAchecker-Explicit 1.1.10	Stefan Löwe	University of Passau, Germany
CPAchecker-SeqCom 1.1.10	Philipp Wendler	University of Passau, Germany
CSeq 2012-10-22	Bernd Fischer	University of Southampton, UK
ESBMC 1.20	Lucas Cordeiro	University of Southampton, UK / UFAM, Brazil
LLBMC 2012-10-23	Carsten Sinz	Karlsruhe Institute of Technology, Germany
Predator 2012-10-20	Tomas Vojnar	Brno University of Technology, Czech Republic
Symbiotic 2012-10-21	Jiri Slaby	Masaryk University at Brno, Czech Republic
Threader 0.92	Andrey Rybalchenko	TU Munich, Germany
UFO 2012-10-22	Arie Gurfinkel	University of Toronto, Canada / SEI, USA
Ultimate 2012-10-25	Matthias Heizmann	University of Freiburg, Germany

B.4 Results of SV-COMP 2013[2]

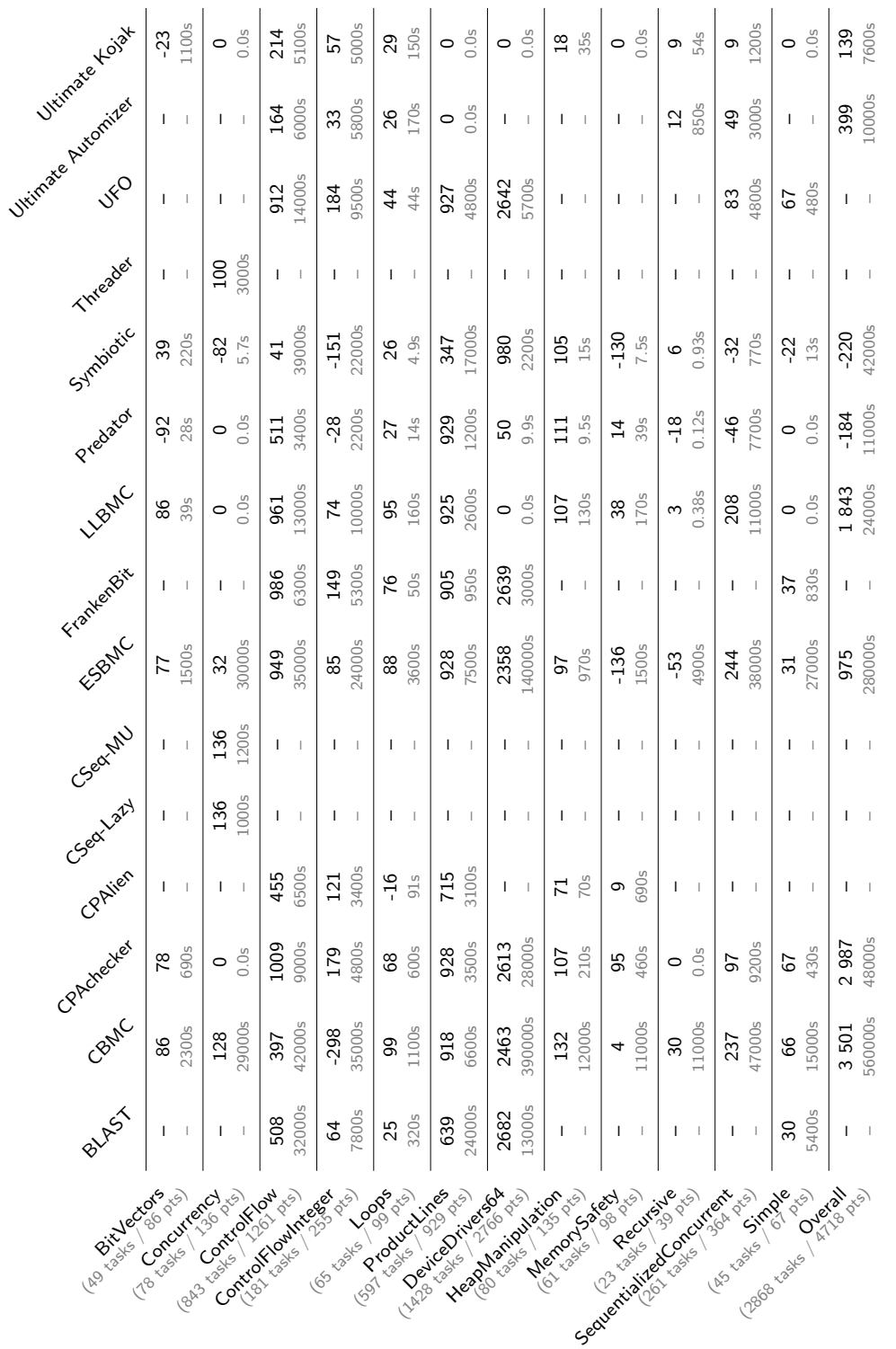
	BLAST	CPAchecker-Explicit	CPAchecker-SeqCom	CSeq	ESBMC	LLBMC	Predator	Symbiotic	Threader	UFO	Ultimate
BitVectors (32 tasks / 60 pts)	–	16	17	–	24	60	-75	–	–	–	–
	–	86s	190s	–	480s	36s	95s	–	–	–	–
Concurrency (32 tasks / 49 pts)	–	0	0	17	15	–	0	–	43	–	–
	–	0s	0s	270s	1 400s	–	0s	–	570s	–	–
ControlFlowInteger (94 tasks / 146 pts)	93	143	141	–	90	–	-27	–	–	146	–
	7100s	1200s	3400s	–	17000s	–	650s	–	–	450s	–
ControlFlowInteger-MemPrecise (48 tasks / 78 pts)	52	78	78	–	69	78	-28	28	–	78	63
	2500s	260s	1300s	–	10000s	70s	650s	34s	–	170s	540s
ControlFlowInteger-MemSimple (46 tasks / 68 pts)	41	65	63	–	22	–	0	–	–	68	–
	4600s	920s	2100s	–	6300s	–	0s	–	–	280s	–
DeviceDrivers64 (1237 tasks / 2419 pts)	2338	2340	2186	–	2233	–	0	870	–	2408	–
	2400s	9700s	30000s	–	46000s	–	0s	230s	–	2500s	–
FeatureChecks (118 tasks / 206 pts)	130	159	159	–	132	166	166	23	–	74	–
	42s	180s	160s	–	86s	250s	6.0s	11s	–	46s	–
HeapManipulation (28 tasks / 48 pts)	–	22	22	–	–	32	40	–	–	–	–
	–	30s	29s	–	–	310s	2.3s	–	–	–	–
Loops (79 tasks / 122 pts)	35	51	50	–	94	112	36	–	–	54	–
	550s	370s	1400s	–	5000s	540s	17s	–	–	750s	–
MemorySafety (36 tasks / 54 pts)	–	0	0	–	3	24	52	–	–	–	–
	–	0s	0s	–	1300s	38s	61s	–	–	–	–
ProductLines (597 tasks / 929 pts)	652	655	915	–	914	926	865	–	–	929	–
	16000s	7300s	3100s	–	1200s	3600s	7500s	–	–	5000s	–
SystemC (62 tasks / 87 pts)	34	61	58	–	57	49	-6	0	–	65	45
	2600s	3500s	1800s	–	8500s	1900s	1400s	0s	–	3000s	4800s
Overall (2315 tasks / 3791 pts)	80	2030	2090	–	1919	–	799	–	–	-208	–
	30000s	22000s	41000s	–	81000s	–	9700s	–	–	12000s	–

[2]Results from http://sv-comp.sosy-lab.org/2013

B.5 Participants of SV-COMP 2014

Tool	Jury Member	Affiliation
BLAST 2.7.2	Vadim Mutilin	Russian Academy of Sciences, Russia
CBMC	Michael Tautschnig	University of Oxford, UK
CPAchecker	Stefan Löwe	University of Passau, Germany
CPAlien	Petr Muller	Brno University of Technology, Czech Republic
CSeq-Lazy	Bernd Fischer	University of Southampton, UK
CSeq-MU	Gennaro Parlato	University of Southampton, UK
ESBMC 1.22	Lucas Cordeiro	University of Southampton, UK / Fed. Univ. of Amazonas, Brazil
FrankenBit	Arie Gurfinkel	SEI, USA / University College Dublin, Ireland
LLBMC	Stephan Falke	Karlsruhe Institute of Technology, Germany
Predator	Tomas Vojnar	Brno University of Technology, Czech Republic
Symbiotic 2	Jiri Slaby	Masaryk University at Brno, Czech Republic
Threader	Corneliu Popeea	TU Munich, Germany
UFO	Aws Albarghouthi	University of Toronto, Canada / SEI, USA
Ultimate Automizer	Matthias Heizmann	University of Freiburg, Germany
Ultimate Kojak	Alexander Nutz	University of Freiburg, Germany

B.6 Results of SV-COMP 2014[3]



	BLAST	CBMC	CPAchecker	CPAlien	CSeq-Lazy	CSeq-MU	ESBMC	FrankenBit	LLBMC	Predator	Symbiotic	Threader	UFO	Ultimate Automizer	Ultimate Kojak
BitVectors (49 tasks / 86 pts)	–	86	78	–	–	–	77	–	86	-92	39	–	–	–	-23
	–	2300s	690s	–	–	–	1500s	–	39s	28s	220s	–	–	–	1100s
Concurrency (78 tasks / 136 pts)	–	128	0	–	136	136	32	–	0	0	-82	100	–	–	0
	–	29000s	0.0s	–	1000s	1200s	30000s	–	0.0s	0.0s	5.7s	3000s	–	–	0.0s
ControlFlow (843 tasks / 1261 pts)	508	397	1009	455	–	–	949	986	961	511	41	–	912	164	214
	32000s	42000s	9000s	6500s	–	–	35000s	6300s	13000s	3400s	39000s	–	14000s	6000s	5100s
ControlFlowInteger (181 tasks / 253 pts)	64	-298	179	121	–	–	85	149	74	-28	-151	–	184	33	57
	7800s	35000s	4800s	3400s	–	–	24000s	5300s	10000s	2200s	22000s	–	9500s	5800s	5000s
Loops (65 tasks / 99 pts)	25	99	68	-16	–	–	88	76	95	27	26	–	44	26	29
	320s	1100s	600s	91s	–	–	3600s	50s	160s	14s	4.9s	–	44s	170s	150s
ProductLines (597 tasks / 929 pts)	639	918	928	715	–	–	928	905	925	929	347	–	927	0	0
	24000s	6600s	3500s	3100s	–	–	7500s	950s	2600s	1200s	17000s	–	4800s	0.0s	0.0s
DeviceDrivers64 (1428 tasks / 2766 pts)	2682	2463	2613	–	–	–	2358	2639	0	50	980	–	2642	–	0
	13000s	390000s	28000s	–	–	–	140000s	3000s	0.0s	9.9s	2200s	–	5700s	–	0.0s
HeapManipulation (80 tasks / 135 pts)	–	132	107	71	–	–	97	–	107	111	105	–	–	–	18
	–	12000s	210s	70s	–	–	970s	–	130s	9.5s	15s	–	–	–	35s
MemorySafety (61 tasks / 98 pts)	–	4	95	9	–	–	-136	–	38	14	-130	–	–	–	0
	–	11000s	460s	690s	–	–	1500s	–	170s	39s	7.5s	–	–	–	0.0s
Recursive (23 tasks / 39 pts)	–	30	0	–	–	–	-53	–	3	-18	6	–	–	12	9
	–	11000s	0.0s	–	–	–	4900s	–	0.38s	0.12s	0.93s	–	–	850s	54s
SequentializedConcurrent (261 tasks / 364 pts)	–	237	97	–	–	–	244	–	208	-46	-32	–	83	49	9
	–	47000s	9200s	–	–	–	38000s	–	11000s	7700s	770s	–	4800s	3000s	1200s
Simple (45 tasks / 67 pts)	30	66	67	–	–	–	31	37	0	0	-22	–	67	–	0
	5400s	15000s	430s	–	–	–	27000s	830s	0.0s	0.0s	13s	–	480s	–	0.0s
Overall (2868 tasks / 4718 pts)	–	3 501	2 987	–	–	–	975	–	1 843	-184	-220	–	–	399	139
	–	560000s	48000s	–	–	–	280000s	–	24000s	11000s	42000s	–	–	10000s	7600s

[3]Results from http://sv-comp.sosy-lab.org/2014

Appendix C

Detailed Example

C.1 LLVM-IR Code

C.1.1 Initial LLVM-IR Code

```
define i32 @isintmax(i32 %n) {
entry:
  %n.addr = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  %0 = load i32* %n.addr, align 4
  %add = add nsw i32 %0, 1
  %1 = load i32* %n.addr, align 4
  %cmp = icmp slt i32 %add, %1
  %conv = zext i1 %cmp to i32
  ret i32 %conv
}

define i32* @allocate(i32 %n, i32 %v) {
entry:
  %v.addr = alloca i32, align 4
  %n.addr = alloca i32, align 4
  %p = alloca i32*, align 4
  store i32 %v, i32* %v.addr, align 4
  store i32 %n, i32* %n.addr, align 4
  store i32* null, i32** %p, align 4
  %0 = load i32* %n.addr, align 4
  %call = call i32 @isintmax(i32 %0)
  %tobool = icmp ne i32 %call, 0
  br i1 %tobool, label %if.end, label %if.then

if.then:
  %call1 = call i8* @malloc(i32 4)
  %1 = bitcast i8* %call1 to i32*
  store i32* %1, i32** %p, align 4
  br label %if.end

if.end:
  %2 = load i32* %v.addr, align 4
  %3 = load i32** %p, align 4
  store i32 %2, i32* %3, align 4
  %4 = load i32** %p, align 4
  ret i32* %4
}
```

C.1.2 Optimized LLVM-IR Code (-O3

```
define i32 @isintmax(i32 %n) {
entry:
  ret i32 0
}

define i32* @allocate(i32 %n, i32 %v) {
entry:
  %call = call i32 @isintmax(i32 %n)
  %tobool = icmp ne i32 %call, 0
  br i1 %tobool, label %if.end, label %if.then

if.then:
  %call1 = call i8* @malloc(i32 4)
  %0 = bitcast i8* %call1 to i32*
  br label %if.end

if.end:
  %p.0 = phi i32* [ null, %entry ], [ %0, %if.then ]
  store i32 %v, i32* %p.0, align 4
  ret i32* %p.0
}
```

C.1.3 Instrumented, non-optimized LLVM-IR Code

```
define i32 @isintmax(i32 %n) {
entry:
  %n.addr = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  %0 = load i32* %n.addr, align 4
  call void @check.saddo.i32(i32 %0, i32 1)
  %add = add nsw i32 %0, 1
  %1 = load i32* %n.addr, align 4
  %cmp = icmp slt i32 %add, %1
  %conv = zext i1 %cmp to i32
  ret i32 %conv
}

define i32* @allocate(i32 %n, i32 %v) {
entry:
  %v.addr = alloca i32, align 4
  %n.addr = alloca i32, align 4
  %p = alloca i32*, align 4
  store i32 %v, i32* %v.addr, align 4
  store i32 %n, i32* %n.addr, align 4
  store i32* null, i32** %p, align 4
  %0 = load i32* %n.addr, align 4
  %call = call i32 @isintmax(i32 %0)
  %tobool = icmp ne i32 %call, 0
  br i1 %tobool, label %if.end, label %if.then

if.then:
  %call1 = call i8* @malloc(i32 4)
  %1 = bitcast i8* %call1 to i32*
  store i32* %1, i32** %p, align 4
  br label %if.end

if.end:
  %2 = load i32* %v.addr, align 4
  %3 = load i32** %p, align 4
  %t = bitcast i32* %3 to i8*
  call void @check.access(i8* %t, i32 4)
  store i32 %2, i32* %3, align 4
  %4 = load i32** %p, align 4
  ret i32* %4
```

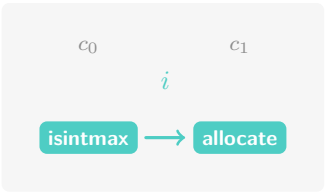
```
41 }
```

C.1.4 Instrumentation

```
1  define void @check.access(i8* %p, i32 %s) {
2  entry:
3    %0 = call i1 @llbmc.valid.access(i8* %p, i32 %s)
4    call void @llbmc.assert(i1 %0)
5    ret void
6  }
7
8  define void @check.saddo.i32(i32 %x, i32 %y) {
9  entry:
10   %0 = call i1@llbmc.saddo.i32(i32 %x, i32 %y)
11   call void @llbmc.assert(i1 %0)
12   ret void
13 }
```

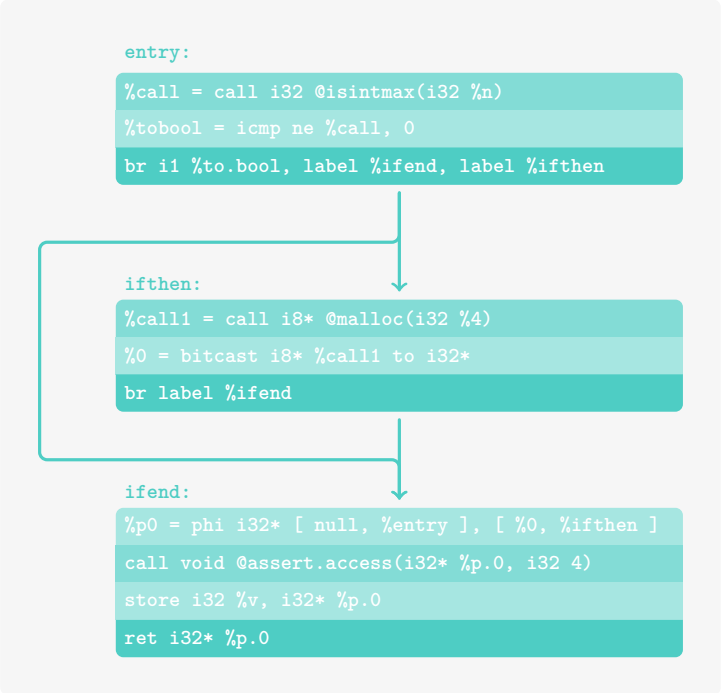
C.2 Call and Control Flow Graphs

C.2.1 Call Graph



C.2.2 Control Flow Graph for `@isintmax`

C.2.3 Control Flow Graph for `@allocate`



C.3 Encoding

C.3.1 Step 1

$$
\begin{aligned}
t_8 &:= \varepsilon^{\mathtt{I}}_{\mathsf{i32*}}(c_1, i_{19}) \\
t_7 &:= \overleftarrow{\rho}^{\mathtt{I}}_{\mathsf{h32}}(c_1, i_{20}) \\
t_6 &:= \varepsilon^{\mathtt{A}}_{\mathsf{i32}}(c_2, n) \\
t_5 &:= \overleftarrow{\eta}^{\mathtt{F}}(c_2, \mathtt{isintmax}) \\
t_4 &:= \mathrm{not}(\mathrm{addo}^{s}_{\mathsf{i32}}(t_6)) \\
t_3 &:= \overrightarrow{\eta}^{\mathtt{I}}(c_1, i_{19}) \\
t_2 &:= \mathrm{validaccess}_{\mathsf{h32}}(t_7, t_8, (4)_{\mathsf{i32}}) \\
t_1 &:= \mathrm{and}(\mathrm{or}(\mathrm{not}(t_5), t_4), \mathrm{or}(\mathrm{not}(t_3), t_2))
\end{aligned}
$$

C.3.2 Step 2

$$
\begin{aligned}
t_8 &:= \phi_{\mathsf{i32}*}\left((0)_{\mathsf{i32}*}, \eta^{\mathsf{J}}(c_1, \texttt{entry}, \texttt{ifend}), \varepsilon^{\mathtt{I}}_{\mathsf{i32}*}(c_1, i_{15}), \eta^{\mathsf{J}}(c_1, \texttt{ifthen}, \texttt{ifend})\right)\\
t_7 &:= \overleftarrow{\rho}^{\mathtt{B}}_{\mathsf{h32}}(c_1, \texttt{ifend})\\
t_6 &:= \varepsilon^{\mathtt{A}}_{\mathsf{i32}}(c_1, n)\\
t_5 &:= \overleftarrow{\eta}^{\mathtt{I}}(c_1, i_9)\\
t_4 &:= \mathrm{not}(\mathrm{addo}^{s}_{\mathsf{i32}}(t_6))\\
t_3 &:= \overleftarrow{\eta}^{\mathtt{B}}(c_1, \texttt{ifend})\\
t_2 &:= \mathrm{validaccess}_{\mathsf{h32}}(t_7, t_8, (4)_{\mathsf{i32}})\\
t_1 &:= \mathrm{and}(\mathrm{or}(\mathrm{not}(t_5), t_4), \mathrm{or}(\mathrm{not}(t_3), t_2))
\end{aligned}
$$

C.3.3 Step 3

$$
\begin{aligned}
t_{11} &:= \eta^{\mathsf{J}}(c_1, \texttt{entry}, \texttt{ifend})\\
t_{10} &:= \varepsilon^{\mathtt{I}}_{\mathsf{i32}*}(c_1, i_{15})\\
t_9 &:= \eta^{\mathsf{J}}(c_1, \texttt{ifthen}, \texttt{ifend})\\
t_8 &:= \phi_{\mathsf{i32}*}\left((0)_{\mathsf{i32}*}, t_1 1, t_1 0, t_9\right)\\
t_7 &:= \overleftarrow{\rho}^{\mathtt{B}}_{\mathsf{h32}}(c_1, \texttt{ifend})\\
t_6 &:= \varepsilon^{\mathtt{A}}_{\mathsf{i32}}(c_1, n)\\
t_5 &:= \overleftarrow{\eta}^{\mathtt{I}}(c_1, i_9)\\
t_4 &:= \mathrm{not}(\mathrm{addo}^{s}_{\mathsf{i32}}(t_6))\\
t_3 &:= \overleftarrow{\eta}^{\mathtt{B}}(c_1, \texttt{ifend})\\
t_1 &:= \mathrm{and}(\mathrm{or}(\mathrm{not}(t_5), t_4), \mathrm{or}(\mathrm{not}(t_3) t_2,))
\end{aligned}
$$

C.3.4 After Encoding

$$
\begin{aligned}
i_0 &:= n \\
b_0 &:= \mathrm{not}(T)) \\
b_1 &:= \mathrm{addo}^{s}_{\mathsf{i32}}(i_0, (1)_{\mathsf{i32}}) \\
b_2 &:= \mathrm{not}(b_1)) \\
b_3 &:= \mathrm{or}(b_0, b_2)) \\
b_4 &:= \mathrm{and}(T, b_2)) \\
i_1 &:= \mathrm{ne}_{\mathsf{i32}}((0)_{\mathsf{i32}}, (0)_{\mathsf{i32}}) \\
b_5 &:= \mathrm{eq}_{\mathsf{i1}}(i_1, (0)_{\mathsf{i1}}) \\
b_6 &:= \mathrm{not}(b_5)) \\
b_7 &:= \mathrm{and}(b_4, b_6)) \\
b_8 &:= \mathrm{and}(b_7, T)) \\
b_9 &:= \mathrm{and}(b_8, T)) \\
p_0 &:= \mathrm{bitcast}_{\mathsf{i8*},\mathsf{i32*}}((2147483644)_{\mathsf{i8*}}) \\
b_{10} &:= \mathrm{eq}_{\mathsf{i1}}(i_1, (0)_{\mathsf{i1}}) \\
b_{11} &:= \mathrm{and}(b_4, b_{10})) \\
b_{12} &:= \mathrm{or}(b_9, b_{11})) \\
p_1 &:= \mathrm{select}_{\mathsf{i32*}}(b_9, p_0, (0)_{\mathsf{i32*}}) \\
p_2 &:= \mathrm{bitcast}_{\mathsf{i32*},\mathsf{i8*}}(p_1) \\
b_{13} &:= \mathrm{le}^{u}_{\mathsf{i8*}}((2147483644)_{\mathsf{i8*}}, p_2) \\
i_2 &:= \mathrm{ptrtoint}_{\mathsf{i8*},\mathsf{i32}}(p_2) \\
i_3 &:= \mathrm{add}_{\mathsf{i32}}(i_2, (4)_{\mathsf{i32}}) \\
p_3 &:= \mathrm{inttoptr}_{\mathsf{i32},\mathsf{i8*}}(i_3) \\
b_{14} &:= \mathrm{le}^{u}_{\mathsf{i8*}}(p_3, (2147483648)_{\mathsf{i8*}}) \\
b_{15} &:= \mathrm{and}(b_{13}, b_{14})) \\
b_{16} &:= \mathrm{and}(b_7, b_{15})) \\
b_{17} &:= \mathrm{addo}^{u}_{\mathsf{i32}}(i_2, (4)_{\mathsf{i32}}) \\
b_{18} &:= \mathrm{not}(b_{17})) \\
b_{19} &:= \mathrm{and}(b_{16}, b_{18})) \\
i_4 &:= \mathrm{select}_{\mathsf{i1}}(b_{19}, (1)_{\mathsf{i1}}, (0)_{\mathsf{i1}}) \\
b_{20} &:= \mathrm{eq}_{\mathsf{i1}}(i_4, (0)_{\mathsf{i1}}) \\
b_{21} &:= \mathrm{not}(b_{12})) \\
b_{22} &:= \mathrm{or}(b_{21}, b_{20})) \\
& b_3 \\
& b_{22}
\end{aligned}
$$

Bibliography

[ÁH14] Erika Ábrahám and Klaus Havelund, eds. *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014.

[AIB96] Ariane 501 Inquiry Board. *Ariane 5 – Flight 501 Failure*. Tech. rep. European Space Agency, 1996.

[Alb+12] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. "Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 672–678.

[All70] Frances E. Allen. "Control Flow Analysis". In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19.

[AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. "Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers". In: *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*. Ed. by Antti Valmari. Vol. 3925. Lecture Notes in Computer Science. Springer, 2006, pp. 146–162.

[Bar+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387.

[BAS02] Armin Biere, Cyrille Artho, and Viktor Schuppan. "Liveness Checking as Safety Checking". In: *Electr. Notes Theor. Comput. Sci.* 66.2 (2002), pp. 160–177.

[BB09] Robert Brummayer and Armin Biere. "Lemmas on Demand for the Extensional Theory of Arrays". In: *JSAT* 6.1-3 (2009), pp. 165–201.

[Bec+11] Bernhard Beckert, Thorsten Bormer, Florian Merz, and Carsten Sinz. "Integration of Bounded Model Checking and Deductive Verification". In: *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*. Ed. by Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov. Vol. 7421. Lecture Notes in Computer Science. Springer, 2011, pp. 86–104.

[Bey12] Dirk Beyer. "Competition on Software Verification - (SV-COMP)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by Cormac Flanagan and Barbara König. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 504–524.

[Bey14] Dirk Beyer. "Status Report on Software Verification - (Competition Summary SV-COMP 2014)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 373–388.

[BFT15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.5*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2015.

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking". In: *Advances in Computers* 58 (2003), pp. 117–148.

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*. Ed. by Rance Cleaveland. Vol. 1579. Lecture Notes in Computer Science. Springer, 1999, pp. 193–207.

[BL09] Sandrine Blazy and Xavier Leroy. "Mechanized Semantics for the Clight Subset of the C Language". In: *Computing Research Repository* abs/0901.3619 (2009).

[Bru10] Robert Brummayer. "Efficient SMT Solving for Bit Vectors and the Extensional Theory of Arrays". PhD thesis. Johannes Kepler University of Linz, 2010.

[Bry86] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691.

[Bün98] Reinhard Bündgen. *Termersetzungssysteme – Theorie, Implementierung, Anwendung*. Vieweg, 1998.

[Bur+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. "Symbolic Model Checking: 10^20 States and Beyond". In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*. IEEE Computer Society, 1990, pp. 428–439.

[BW13] Dirk Beyer and Philipp Wendler. "Reuse of Verification Results - Conditional Model Checking, Precision Reuse, and Verification Witnesses". In: *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings*. Ed. by Ezio Bartocci and C. R. Ramakrishnan. Vol. 7976. Lecture Notes in Computer Science. Springer, 2013, pp. 1–17.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[CE81] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*. Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Springer, 1981, pp. 52–71.

[CFM09] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. "SMT-Based Bounded Model Checking for Embedded ANSI-C Software". In: *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 2009, pp. 137–148.

[Cim+99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. "NUSMV: A New Symbolic Model Verifier". In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*. Ed. by Nicolas Halbwachs and Doron Peled. Vol. 1633. Lecture Notes in Computer Science. Springer, 1999, pp. 495–499.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: a platform for in-vivo multi-path analysis of software systems". In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. Ed. by Rajiv Gupta and Todd C. Mowry. ACM, 2011, pp. 265–278.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176.

[CKY03] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking". In: *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*. ACM, 2003, pp. 368–371.

[CKY12] Pascal Cuoq, Florent Kirchner, and Boris Yakobowski. "Benchmarking Static Analyzers". In: *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012*. Ed. by Vladimir Klebanov, Bernhard Beckert, Armin Biere, and Geoff Sutcliffe. Vol. 873. CEUR Workshop Proceedings. CEUR-WS.org, 2012, pp. 32–35.

[Cla08] Edmund M. Clarke. "The Birth of Model Checking". In: *25 Years of Model Checking - History, Achievements, Perspectives*. Ed. by Orna Grumberg and Helmut Veith. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008, pp. 1–26.

[Coh+09a] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. "VCC: A Practical System for Verifying Concurrent C". In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 23–42.

[Coh+09b] Ernie Cohen, Michal Moskal, Stephan Tobies, and Wolfram Schulte. "A Precise Yet Efficient Memory Model For C". In: *Electr. Notes Theor. Comput. Sci.* 254 (2009), pp. 85–103.

[DBZ13] Ewen Denney, Tevfik Bultan, and Andreas Zeller, eds. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. IEEE, 2013.

[DKR11] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. "Automatic analysis of DMA races using model checking and *k*-induction". In: *Formal Methods in System Design* 39.1 (2011), pp. 83–113.

[Don+11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. "Software Verification Using k-Induction". In: *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*. Ed. by Eran Yahav. Vol. 6887. Lecture Notes in Computer Science. Springer, 2011, pp. 351–368.

[Dun13] Michael Dunn. *Toyota's Killer Firmware: Bad Design and Its Consequences*. Oct. 2013. URL: http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences.

[ES03] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518.

[ESA15] European Space Agency. *Ariane 5*. Sept. 2015. URL: http://www.esa.int/Our_Activities/Launchers/Launch_vehicles/Ariane_5.

[FIP13] Bernd Fischer, Omar Inverso, and Gennaro Parlato. "CSeq: A concurrency pre-processor for sequential C verification tools". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. Ed. by Ewen Denney, Tevfik Bultan, and Andreas Zeller. IEEE, 2013, pp. 710–713.

[Fix08] Limor Fix. "Fifteen Years of Formal Property Verification in Intel". In: *25 Years of Model Checking - History, Achievements, Perspectives*. Ed. by Orna Grumberg and Helmut Veith. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008, pp. 139–144.

[Flo67] Robert W. Floyd. "Assigning Meanings to Programs". In: *Proceedings of Symposia in Applied Mathematics Vol. 19*. 1967, pp. 19–32.

[FMS11] Stephan Falke, Florian Merz, and Carsten Sinz. "A Theory of C-Style Memory Allocation". In: *9th International Workshop on Satisfiability Modulo Theories (SMT 2011), 14-15 July 2011, Snowbird, UT, USA*. 2011, pp. 71–80.

[FMS13a] Stephan Falke, Florian Merz, and Carsten Sinz. "Extending the Theory of Arrays: memset, memcpy, and Beyond". In: *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*. Ed. by Ernie Cohen and Andrey Rybalchenko. Vol. 8164. Lecture Notes in Computer Science. Springer, 2013, pp. 108–128.

[FMS13b] Stephan Falke, Florian Merz, and Carsten Sinz. "LLBMC: Improved Bounded Model Checking of C Programs Using LLVM - (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Nir Piterman and Scott A. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 623–626.

[FMS13c] Stephan Falke, Florian Merz, and Carsten Sinz. "The bounded model checker LLBMC". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. Ed. by Ewen Denney, Tevfik Bultan, and Andreas Zeller. IEEE, 2013, pp. 706–709.

[FR14] Federal Register. *Airworthiness Directives; The Boeing Company Airplanes*. Oct. 2014. URL: https://www.federalregister.gov/articles/2015/05/01/2015-10066/airworthiness-directives-the-boeing-company-airplanes.

[FSM12] Stephan Falke, Carsten Sinz, and Florian Merz. "A Theory of Arrays with set and copy Operations". In: *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*. Ed. by Pascal Fontaine and Amit Goel. Vol. 20. EPiC Series. EasyChair, 2012, pp. 98–108.

[Gal15] Sean Gallagher. *Airbus Confirms Software Configuration Error Caused Plane Crash*. June 2015. URL: http://arstechnica.com/information-technology/2015/06/airbus-confirms-software-configuration-error-caused-plane-crash.

[Gan14] Jack Ganssle. *Toyota's Expensive Software*. Mar. 2014. URL: http://www.embedded.com/electronics-blogs/break-points/4429601/Toyota-s-Expensive-Software.

[Gat11] Dominic Gates. *Boeing Celebrates 787 Delivery as Program's Costs Top $32 Billion*. Sept. 2011. URL: http://www.seattletimes.com/business/boeing-celebbrates-787-delivery-as-programs-costs-top-32-billion.

[GB14] Arie Gurfinkel and Anton Belov. "FrankenBit: Bit-Precise Verification with Many Bits - (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 408–411.

[GG06] Malay K. Ganai and Aarti Gupta. "Accelerating high-level bounded model checking". In: *2006 International Conference on Computer-Aided Design, ICCAD 2006, San Jose, CA, USA, November 5-9, 2006*. Ed. by Soha Hassoun. ACM, 2006, pp. 794–801.

[GQ11] Ganesh Gopalakrishnan and Shaz Qadeer, eds. *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011.

[Gro+97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. "Call Graph Construction in Object-Oriented Languages". In: *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97), Atlanta, Georgia, October 5-9, 1997*. Ed. by Mary E. S. Loomis, Toby Bloom, and A. Michael Berman. ACM, 1997, pp. 108–124.

[GV08] Orna Grumberg and Helmut Veith, eds. *25 Years of Model Checking - History, Achievements, Perspectives*. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008.

[HN09] Yuusuke Hashimoto and Shin Nakajima. "Modular Checking of C Programs Using SAT-Based Bounded Model Checker". In: *16th Asia-Pacific Software Engineering Conference, APSEC 2009, 1-3 December 2009, Batu Ferringhi, Penang, Malaysia*. Ed. by Shahida Sulaiman and Noor Maizura Mohamad Noor. IEEE Computer Society, 2009, pp. 515–522.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580.

[Hol00] Gerard J. Holzmann. "Logic Verification of ANSI-C Code with SPIN". In: *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*. Ed. by Klaus Havelund, John Penix, and Willem Visser. Vol. 1885. Lecture Notes in Computer Science. Springer, 2000, pp. 131–147.

[Hol97] Gerard J. Holzmann. "The Model Checker SPIN". In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295.

[Inv+14] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. "Lazy-CSeq: A Lazy Sequentialization Tool for C - (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 398–401.

[ISO26262] International Standards Organisation. *ISO 26262-1 - Road Vehicles – Functional Safety*. Tech. rep. International Organization for Standardization, 2009.

[ISOC99] International Standards Organisation. *ISO C Standard 1999*. Tech. rep. International Organization for Standardization, 1999.

[JMP12] Rajeev Joshi, Peter Müller, and Andreas Podelski, eds. *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*. Vol. 7152. Lecture Notes in Computer Science. Springer, 2012.

[Kle+12] Vladimir Klebanov, Bernhard Beckert, Armin Biere, and Geoff Sutcliffe, eds. *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012*. Vol. 873. CEUR Workshop Proceedings. CEUR-WS.org, 2012.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

[KRW09] Daniel Kröning, Philipp Rümmer, and Georg Weissenbacher. "A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard". In: *7th International Workshop on Satisfiability Modulo Theories (SMT 2009), 2-3 August 2009, Montreal, Canada*. 2009.

[KW11] Robbert Krebbers and Freek Wiedijk. "A Formalization of the C99 Standard in HOL, Isabelle and Coq". In: *Intelligent Computer Mathematics - 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011, Bertinoro, Italy, July 18-23, 2011. Proceedings*. Ed. by James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe. Vol. 6824. Lecture Notes in Computer Science. Springer, 2011, pp. 301–303.

[LA04] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88.

[Lan97] Gérard Le Lann. "An Analysis of the Ariane 5 Flight 501 Failure – A System Engineering Perspective". In: *1997 Workshop on Engineering of Computer-Based Systems (ECBS '97), March 24-28, 1997, Monterey, CA, USA*. IEEE Computer Society, 1997, pp. 339–246.

[LGR11] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 609–615.

[LQ15] Akash Lal and Shaz Qadeer. "DAG inlining: a decision procedure for reachability-modulo-theories in hierarchical programs". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 280–290.

[LT93] Nancy G. Leveson and Clark Savage Turner. "Investigation of the Therac-25 Accidents". In: *IEEE Computer* 26.7 (1993), pp. 18–41.

[Mau04] Laurent Mauborgne. "Astrée: verification of absence of run-time error". In: *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*. Ed. by René Jacquart. Vol. 156. IFIP. Kluwer/Springer, 2004, pp. 385–392.

[MC85] Bhubaneswaru Mishra and Edmund M. Clarke. "Hierarchical Verification of Asynchronous Circuits Using Temporal Logic". In: *Theor. Comput. Sci.* 38 (1985), pp. 269–291.

[McC04] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. 2nd edition. Microsoft Press, 2004.

[McC62] John McCarthy. "Towards a Mathematical Science of Computation". In: *IFIP Congress*. 1962, pp. 21–28.

[McM10] Kenneth L. McMillan. "Lazy Annotation for Program Testing and Verification". In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 104–118.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[Mer+10] Florian Merz, Carsten Sinz, Hendrik Post, Thomas Gorges, and Thomas Kropf. "Abstract Testing: Connecting Source Code Verification with Requirements". In: *Quality of Information and Communications Technology, 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010, Porto, Portugal, 29 September - 2 October, 2010, Proceedings*. Ed. by Fernando Brito e Abreu, João Pascoal Faria, and Ricardo Jorge Machado. IEEE Computer Society, 2010, pp. 89–96.

[Mer+15] Florian Merz, Carsten Sinz, Hendrik Post, Thomas Gorges, and Thomas Kropf. "Bridging the gap between test cases and requirements by abstract testing". In: *ISSE* 11.4 (2015), pp. 233–242.

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR". In: *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*. Ed. by Rajeev Joshi, Peter Müller, and Andreas Podelski. Vol. 7152. Lecture Notes in Computer Science. Springer, 2012, pp. 146–161.

[MIS13] Motor Industry Software Reliability Association and Motor Industry Software Reliability Association Staff. *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*. Tech. rep. Motor Industry Research Association, 2013.

[Mor+11] Jeremy Morse, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. "Context-Bounded Model Checking of LTL Properties for ANSI-C Software". In: *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. Ed. by Gilles Barthe, Alberto Pardo, and Gerardo Schneider. Vol. 7041. Lecture Notes in Computer Science. Springer, 2011, pp. 302–317.

[MSF12] Florian Merz, Carsten Sinz, and Stephan Falke. "Challenges in Comparing Software Verification Tools for C". In: *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012*. Ed. by Vladimir Klebanov, Bernhard Beckert, Armin Biere, and Geoff Sutcliffe. Vol. 873. CEUR Workshop Proceedings. CEUR-WS.org, 2012, pp. 60–65.

[Nau66] Peter Naur. "Proof of Algorithms by General Snapshots". English. In: *BIT Numerical Mathematics* 6.4 (1966), pp. 310–316.

[Par97] Berhooz Parhami. "Defect, Fault, Error,..., or Failure?" In: *IEEE Transactions on Reliability* 46.4 (Dec. 1997), pp. 450–451.

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[Pos+08] Hendrik Post, Carsten Sinz, Alexander Kaiser, and Thomas Gorges. "Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking". In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*. IEEE Computer Society, 2008, pp. 188–197.

[Pos+09] Hendrik Post, Carsten Sinz, Florian Merz, Thomas Gorges, and Thomas Kropf. "Linking Functional Requirements and Software Verification". In: *RE 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31 - September 4, 2009*. IEEE Computer Society, 2009, pp. 295–302.

[Pos09] Hendrik Post. "Verifikation von systemnaher Software mittels Bounded Model Checking". PhD thesis. Karlsruhe Institute of Technology, 2009.

[QS82] Jean-Pierre Queille and Joseph Sifakis. "Specification and Verification of Concurrent Systems in CESAR". In: *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Vol. 137. Lecture Notes in Computer Science. Springer, 1982, pp. 337–351.

[Ram+13] Mikhail Ramalho, Mauro Freitas, Felipe Sousa, Hendrio Marques, Lucas C. Cordeiro, and Bernd Fischer. "SMT-Based Bounded Model Checking of C++ Programs". In: *20th IEEE International Conference and Workshops on Engineering of Computer Based Systems, ECBS 2013, Scottsdale, AZ, USA, April 22-24, 2013*. Ed. by Jerzy W. Rozenblit. IEEE Computer Society, 2013, pp. 147–156.

[RE11] David A. Ramos and Dawson R. Engler. "Practical, Low-Effort Equivalence Verification of Real Code". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 669–685.

[RF12] Heinz Riener and Görschwin Fey. "FAuST: A Framework for Formal Verification, Automated Debugging, and Software Test Generation". In: *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*. Ed. by Alastair F. Donaldson and David Parker. Vol. 7385. Lecture Notes in Computer Science. Springer, 2012, pp. 234–240.

[RG05] Ishai Rabinovitz and Orna Grumberg. "Bounded Model Checking of Concurrent Programs". In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 82–97.

[Roc+15] Herbert Rocha, Hussama Ismail, Lucas C. Cordeiro, and Raimundo S. Barreto. "Model Checking C Programs with Loops via k-Induction and Invariants". In: *CoRR* abs/1502.02327 (2015).

[RT03] Silvio Ranise and Cesare Tinelli. "The SMT-LIB Format: An Initial Proposal". In: *Pragmatics of Decision Procedures in Automated Reasoning , PDPAR 2003, Miami, USA, July 29 2003*. 2003.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Global Value Numbers and Redundant Computations". In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. Ed. by Jeanne Ferrante and P. Mager. ACM Press, 1988, pp. 12–27.

[Sch+00] Michael J. Schultey, Mustafa Goky, Pablo I. Balzolay, and Robert W. Brocatoz. "Combined Unsigned and Two's Complement Saturating Multipliers". In: (2000), pp. 185–196.

[SFM10] Carsten Sinz, Stephan Falke, and Florian Merz. "A Precise Memory Model for Low-Level Bounded Model Checking". In: *5th International Workshop on Systems Software Verification, SSV'10, Vancouver, BC, Canada, October 6-7, 2010*. Ed. by Ralf Huuck, Gerwin Klein, and Bastian Schlich. USENIX Association, 2010.

[Shi90] Olin Shivers. "Data-flow Analysis and Type Recovery in Scheme". In: (1990).

[Sin08] Nishant Sinha. "Symbolic Program Analysis Using Term Rewriting and Generalization". In: *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*. Ed. by Alessandro Cimatti and Robert B. Jones. IEEE, 2008, pp. 1–9.

[SRS13] Safety Research and Strategy Inc. *Toyota Unintended Acceleration and the Big Bowl of "Spaghetti" Code*. Nov. 2013. URL: http://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-%C3%A2%C2%80%C2%9Cspaghetti%C3%A2%C2%80%C2%9D-code.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*. Ed. by Warren A. Hunt Jr. and Steven D. Johnson. Vol. 1954. Lecture Notes in Computer Science. Springer, 2000, pp. 108–125.

[Tur49] Alan M. Turing. "Checking a Large Routine". In: *Report on a Conference on High Speed Automatic Computation, June 1949*. Ed. by Anonymous. Inaugural conference of the EDSAC computer at the Mathematical Laboratory, Cambridge, UK. Cambridge, UK: University Mathematical Laboratory, Cambridge University, 1949, pp. 67–69.

[VK12] Milena Vujosevic-Janicic and Viktor Kuncak. "Development and Evaluation of LAV: An SMT-Based Error Finding Platform - System Description". In: *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*. Ed. by Rajeev Joshi, Peter Müller,

and Andreas Podelski. Vol. 7152. Lecture Notes in Computer Science. Springer, 2012, pp. 98–113.

[War02] Henry S. Warren. *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[WF10] Frank Werner and David Faragó. "Correctness of Sensor Network Applications by Software Bounded Model Checking". In: *Formal Methods for Industrial Critical Systems - 15th International Workshop, FMICS 2010, Antwerp, Belgium, September 20-21, 2010. Proceedings*. Ed. by Stefan Kowalewski and Marco Roveri. Vol. 6371. Lecture Notes in Computer Science. Springer, 2010, pp. 115–131.

List of Figures

List of Tables

List of Listings

Symbols

Encoding

$\rightharpoonup$ after. 86

`A` function argument. 86

`B` Basic block. 86

$\leftharpoondown$ before. 86

`C` execution context. 86

ε evaluation of values. 86

`F` function. 86

`G` global variable. 86

η execution condition. 86

`I` instructions. 86

μ memory state. 86

`N` integer constant. 86

`P` program. 86

σ safety. 86

τ stack state. 86

First-Order Logic

arity function's or predicate's arity. 12

$\wedge$ logical conjunction. 11

$\vee$ logical disjunction. 11

$\leftrightarrow$ logical equality. 11

$\bot$ logical false. 11

$\rightarrow$ logical implication. 11

$[\![\cdot]\!]^{\mathcal{D}}$ interpretation. 12

$\neg$ logical negation. 11

Σ signature. 11

$\mathcal{T}$ theory. 12

$\top$ logical true. 11

LLVM-IR

$\mathrm{allocwidth}^a$ the width required to allocate an object on an architecture. 82

$|^a|$ a sort's bitwidth on an architecture. 79

`define` function definition. 33

`exact` exactness flag. 39

exit exit block of a function. 45

$\mathrm{first}_\mathrm{B}$ first instruction in a basic block. 45

`{` function begin indicator. 33

`}` function end indicator. 33

`@` sigil for globally defined values. 36

`%` sigil for locally defined values. 36

`nsw` no-signed-wrap flag. 39

`nuw` no-unsigned-wrap flag. 39

offset^a the offset of an element in a structuren on an architecture. 82

$\mathrm{pointerwidth}^a$ the bitwidth of pointers on an architecture. 79

succ_B successor relation of instructions in a basic block. 45

term_B basic block terminator. 43

`undef` undefined value. 38

`void` void type. 35

Instruction Patterns

`...|...` alternatives. 37

$\cdot \sim [\![\cdot]\!]$ matching operator. 38

`{...}+` one or more repetitions. 37

`[...]` optional element. 37

`<...>` value or type placeholder. 37

`{...}*` zero or more repetitions. 37

Term Rewriting

$\longrightarrow$ term rewriting. 16

SMT-LIB

ite if-then-else. 13

select read from array. 15

store write to array. 15

Temporal Logic

F finally. 15

G globally. 15

R release. 15

U until. 15

X next. 15

Index

Publications[1]

2015

[1] Florian Merz, Carsten Sinz, Hendrik Post, Thomas Gorges, Thomas Kropf: *Bridging the Gap between Test Cases and Requirements by Abstract Testing*. In Innovations in System and Software Engineering, 11(4). Springer, Heidelberg, 2015.

abstract. In this article we propose a technique, called Abstract Testing, which replaces traditional test cases by abstract test cases. By doing so, fewer test cases are needed, and they are linked more closely to the requirements. Abstract tests can be considered as verification scenarios on the source code level which are derived from the requirements. Checking verification scenarios against the source code is done automatically using a software model checker. We also suggest a migration path from traditional tests to abstract test cases, which provides a smooth transition towards this new technique. Finally we demonstrate feasibility of Abstract Testing by a case study from the automotive systems domain.

2014

[2] David Faragó, Florian Merz, Carsten Sinz: *Automatic Heavy-weight Static Analysis Tools for Finding Bugs in Safety-critical Embedded C/C++ Code*. In Softwaretechnik-Trends, 34(3). Springer, Heidelberg, 2014.

abstract. This paper motivates the use of automatic heavy-weight static analysis tools to find bugs in C (and C++) code for safety-critical embedded systems. By heavy-weight we mean tools that employ powerful analysis to cover all cases. The paper introduces two automatic and relatively heavy-weight tools that are currently employed in the automotive industry, and depicts their underlying techniques, advantages, and disadvantages. Since their results are often imprecise (false positives or false negatives), we advocate the use of alternative techniques such as software bounded model checking (SBMC), which can achieve bit-precise results. Finally, the tool LLBMC is described as an example of a tool implementing SBMC, which makes use of satisfiability modulo theories (SMT) decision procedures as well as the LLVM compiler framework.

[1] All publications have been peer reviewed.

2013

[3] Stephan Falke, Florian Merz, Carsten Sinz: *The Bounded Model Checker LLBMC*. In Proceedings of the 28th International Conference on Automated Software Engineering (ASE '13), pages 706–709. Silicon Valley, USA, 2013.

abstract. This paper presents LLBMC, a tool for finding bugs and runtime errors in sequential C/C++ programs. LLBMC employs bounded model checking using an SMT-solver for the theory of bitvectors and arrays and thus achieves precision down to the level of single bits. The two main features of LLBMC that distinguish it from other bounded model checking tools for C/C++ are (i) its bit-precise memory model, which makes it possible to support arbitrary type conversions via stores and loads; and (ii) that it operates on a compiler intermediate representation and not directly on the source code.

[4] Stephan Falke, Florian Merz, Carsten Sinz: *Extending the Theory of Arrays: memset, memcpy, and Beyond*. In Proceedings of the 5th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE '13), pages 108–128. Atherton, USA, 2013.

abstract. The theory of arrays is widely used in program analysis, (deductive) software verification, bounded model checking, and symbolic execution to model arrays in programs or the computer's main memory. Nonetheless, the theory as introduced by McCarthy is not expressive enough in many cases since it only supports array updates at single locations. In programs, memory is often modified at multiple locations at once using functions such as memset or memcpy. Furthermore, initialization loops that store loop-counter-dependent values in an array are commonly used. This paper presents an extension of the theory of arrays with λ-terms which makes it possible to reason about such cases. We also discuss how loops can be automatically summarized using such λ-terms.

[5] Stephan Falke, Florian Merz, Carsten Sinz: *LLBMC: Improved Bounded Model Checking of C Programs Using LLVM (Competition Contribution)*. In Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13), pages 623–626. Rome, Italy, 2013.

abstract. LLBMC is a tool for detecting bugs and runtime errors in C and C++ programs. It is based on bounded model checking using an SMT solver and thus achieves bit-accurate precision. A distinguishing feature of LLBMC in contrast to other bounded model checking tools for C programs is that it operates on a compiler intermediate representation and not directly on the source code.

2012

[6] Stephan Falke, Carsten Sinz, and Florian Merz: *A Theory of Arrays with Set and Copy Operations (Extended Abstract)*. In Proceedings of the 10th International Workshop on Satisfiability Modulo Theories (SMT '12), pages 97–106. Manchester, UK, 2012.

abstract. The theory of arrays is widely used in order to model main memory in program analysis, software verification, bounded model checking, symbolic execution, etc. Nonetheless, the basic theory as introduced by McCarthy is not expressive enough for important practical cases, since it only supports array updates at single locations. In programs, memory is often modified using functions such as memset or memcpy/memmove, which modify a user-specified range of locations whose size might not be known statically. In this paper we present an extension of the theory of arrays with set and copy operations which make it possible to reason about such functions. We also discuss further applications of the theory.

[7] Florian Merz, Carsten Sinz, Stephan Falke: *Challenges in Comparing Software Verification Tools for C*. In Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE '12), pages 60–65. Manchester, UK, 2012.

abstract. Comparing different software verification or bug-finding tools for C programs can be a difficult task. Problems arise from different kinds of properties that different tools can check, restrictions on the input programs accepted, lack of a standardized specification language for program properties, or different interpretations of the programming language semantics. In this discussion paper we describe problem areas and discuss possible solutions. The paper also reflects some lessons we have learned from participating with our tool LLBMC in the TACAS 2012 Competition on Software Verification (SV-COMP 2012).

[8] Carsten Sinz, Florian Merz, and Stephan Falke: *LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation (Competition Contribution)*. In Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '12), pages 542–544. Tallinn, Estonia, 2012.

abstract. We present LLBMC, a bounded model checker for C programs. LLBMC uses the LLVM compiler framework in order to translate C programs into LLVM's intermediate representation (IR). The resulting code is then converted into a logical representation and simplified using rewrite rules. The simplified formula is finally passed to an SMT solver. In contrast to many other tools, LLBMC uses a flat, bit-precise memory model. It can thus precisely model, e.g., memory-based re-interpret casts.

[9] Florian Merz, Stephan Falke, and Carsten Sinz: *LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR*. In Proceedings of the 4th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE '12), pages 146–161. Philadelphia, USA, 2012.

abstract. Bounded model checking (BMC) of C and C++ programs is challenging due to the complex and intricate syntax and semantics of these programming languages. The BMC tool LLBMC presented in this paper thus uses the LLVM compiler framework in order to translate C and C++ programs into LLVM's intermediate representation. The resulting code is then converted into a logical representation and simplified using rewrite rules. The simplified formula is finally passed to an SMT solver. In contrast to many other tools, LLBMC uses a flat, bit-precise memory model. It can thus precisely model, e.g., memory-based re-interpret casts as used in C and static/dynamic casts as used in C++. An empirical evaluation shows that LLBMC compares favorable to the related BMC tools CBMC and ESBMC.

2011

[10] Bernhard Beckert, Thorsten Bormer, Florian Merz, and Carsten Sinz: *Integration of Bounded Model Checking and Deductive Verification*. In Proceedings of the 2nd International Conference on Formal Verification of Object-Oriented Software (FoVeOOS '11), pages 86–104. Torino, Italy, 2011.

abstract. Modular deductive verification of software systems is a complex task: the user has to put a lot of effort in writing module specifications that fit together when verifying the system as a whole. In this paper, we propose a combination of deductive verification and software bounded model checking (SBMC), where SBMC is used to support the user in the specification and verification process, while deductive verification provides the final correctness proof. SMBC provides early – as well as precise – feedback to the user. Unlike modular deductive verification, the SBMC approach is able to check annotations beyond the boundaries of a single module – even if other relevant modules are not annotated (yet). This allows to test whether the different module specifications in the system match the implementation at every step of the specification process.

[11] Stephan Falke, Florian Merz, and Carsten Sinz: *A Theory of C-Style Memory Allocations*. In Proceedings of the 9th International Workshop on Satisfiability Modulo Theories (SMT '11), pages 98–108. Snowbird, USA, 2011.

abstract. This paper introduces the theory for reasoning about the correctness of memory access operations in the context of a C-style heap memory. The proposed approach makes a clear distinction between reasoning about the values stored in memory and checking whether access to a specific memory location is allowed. The theory provides support for malloc and free and is presented in the form of axioms that can be converted into conditional rewrite rules. It is also shown how the theory can be used in a bounded model checker for C programs.

2010

[12] Carsten Sinz, Stephan Falke, and Florian Merz: *A Precise Memory Model for Low-Level Bounded Model Checking*. In Proceedings of the 5th International Workshop on Systems Software Verification (SSV '10). Vancouver, Canada, 2010.

abstract. Formalizing the semantics of programming languages like C or C++ for bounded model checking can be cumbersome if complete coverage of all language features is to be achieved. On the other hand, low-level languages that occur during translation (compilation) have a much simpler semantics since they are closer to the machine level. It thus makes sense to use these low-level languages for bounded model checking. In this paper we present a highly precise memory model suitable for bounded model checking of such low-level languages. Our method also takes memory protection (malloc/free) into account.

[13] Florian Merz, Carsten Sinz, Hendrik Post, Thomas Gorges, and Thomas Kropf: *Abstract Testing: Connecting Source Code Verification with Requirements*. In Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC '10), pages 89–96. Porto, Portugal, 2010.

abstract. Traditionally, test cases are used to check whether a system conforms to its requirements. However, to achieve good quality and coverage, large amounts of test cases are needed, and thus huge efforts have to be put into test generation and maintenance. We propose a methodology, called Abstract Testing, in which test cases are replaced by verification scenarios. Such verification scenarios are more abstract than test cases, thus fewer of them are needed and they are easier to create and maintain. Checking verification scenarios against the source code is done automatically using a software model checker. In this paper we describe the general idea of Abstract Testing, and demonstrate its feasibility by a case study from the automotive systems domain.

2009

[14] Hendrik Post, Carsten Sinz, Florian Merz, Thomas Gorges, and Thomas Kropf: *Linking Functional Requirements and Software Verification*. In Proceedings of the 17th IEEE International Requirements Engineering Conference (RE '09), pages 295–302. Atlanta, USA, 2009.

abstract. Synchronization between component requirements and implementation centric tests remains a challenge that is usually addressed by requirements reviews with testers and traceability policies. The claim of this work is that linking requirements, their scenario-based formalizations, and software verification provides a promising extension to this approach. Formalized scenarios, for example in the form of low-level assume/assert statements in C, are easier to trace to requirements than traditional test sets. For a verification engineer, they offer an opportunity to better participate in requirements changes. Changes in requirements can be more easily propagated because adapting formalized scenarios is often easier than deriving and updating a large set of test cases. The proposed idea is evaluated in a case study encompassing over 50 functional requirements of an automotive software developed at Robert Bosch GmbH. Results indicate that requirement formalization together with formal verification leads to the discovery of implementation problems missed in a traditional testing process.