FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Abstraction-Based Machine-Code Program Verification

by

Ing. Jan Onderka

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Doctoral study programme: Informatics
Department of Digital Design

Prague, August 2024

Supervisor:
doc. Dipl.-Ing. Dr. techn. Stefan Ratschan
Department of Digital Design
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Abstract

This dissertation thesis is focused on formal verification of machine-code systems using model checking with abstraction. The background and state of the art of machine-code model checking are presented, and weaknesses of previous approaches are noted. The author's research described in this dissertation thesis and previous conference proceedings articles presents novel solutions to the major problems of previous research: the systems are described in the Rust programming language and are inherently simulable, automatically converted to verification equivalents and verified within a novel framework based on Three-Valued Abstraction Refinement. Special care is taken to allow efficient verification of variables based on bit-vectors. The author has created a formal verification tool implementing the introduced techniques, and its performance is evaluated in the thesis. The tool can be used to verify arbitrary finite-state digital systems, with a special focus on systems with behaviour determined by machine-code programs. To the author's knowledge, the created tool is the first free, open-source, and publicly available tool of its kind.

Keywords:

Machine-code verification, translation of simulable descriptions, three-valued abstraction refinement, bit-vector domain

Abstrakt

Tato disertační práce pojednává o formální verifikaci systémů založených na strojovém kódu pomocí techniky kontroly modelu s použitím abstrakce. Je prezentován současný stav poznání v tomto oboru a je poukázáno na slabá místa předchozích přístupů. Autorův výzkum popsaný v této disertační práci a předchozích článcích v konferenčních sbornících prezentuje nové způsoby řešení problémů předchozího výzkumu: systémy jsou popsány v programovacím jazyce Rust a samy o sobě simulovatelné, jsou automaticky konvertovány do verifikačních ekvivalentů a verifikovány v originální konstrukci založené na zjemňování trojhodnotové abstrakce. Pro účinnou verifikaci je speciálně zacházeno s proměnnými založenými na bitových vektorech. Autor práce vytvořil nástroj pro formální verifikaci, který implementuje představené techniky, a jeho schopnosti jsou v práci vyhodnoceny. Nástroj může být použit pro verifikaci libovolných konečných číslicových systémů, se zaměřením na systémy, kde je chování určeno programy ve strojovém kódu. Pokud je autorovi známo, vytvořený nástroj je první bezplatný a veřejně dostupný nástroj svého druhu s otevřeným zdrojovým kódem.

Klíčová slova:

Verifikace strojového kódu, překlad simulovatelných popisů, zjemňování trojhodnotové abstrakce, doména bitových vektorů

Acknowledgements

I would like to thank my dissertation thesis supervisor Stefan Ratschan for his insight and support during my research. I would also like to thank the other academic and non-academic staff of the Department of Digital Design for their help with the formalities during the course of my studies. Finally, I thank my family for their support.

My research has been partially supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS20/211/OHK3/3T/18 and No. SGS23/208/OHK3/3T/18.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

Commonly Used Abbreviations

BDD	Binary Decision Diagram
CEGAR	Counterexample-guided Abstraction Refinement
CTL	Computation Tree Logic
GA	Generating Automaton
GPIO	General-Purpose Input/Output
KS	Kripke Structure
KMTS	Kripke Modal Transition Structure
LTL	Linear Time Logic
PC	Program Counter
PKS	Partial Kripke Structure
SAT	Boolean Satisfiability Problem
SMT	SAT Modulo Theories
SP	Stack Pointer
SRAM	Static Random-Access Memory
TVAR	Three-valued Abstraction Refinement

Less Common Mathematical Notation

10_2	Number 3 expressed in binary numeral system
`0x1F`	Number 31 expressed in hexadecimal numeral system
$\{0,1\}^n$	n-ary Cartesian power of set $\{0,1\}$
$\stackrel{\text{def}}{=}$	Defined as

Three-valued Abstraction Notation

'0'	Three-valued abstraction value corresponding to "definitely 0"
'1'	Three-valued abstraction value corresponding to "definitely 1"
'X'	Three-valued abstraction value corresponding to "perhaps 0, perhaps 1"
"1X10"	A tuple of three-valued abstraction values

Typesetting

italic	A concept that is being introduced
bold	A tool, an executable, a competition, or a temporal logic operator represented by a letter
`typewriter`	A code identifier or a Rust package

Chapter 1

Introduction

The presence of bugs in programs for computers and embedded systems may have severe consequences for safety, security, reliability, etc. Source-code-level and hardware-level verification has been explored in great detail, resulting in applicable tools for formal verification. Machine code level, especially important for embedded systems with wide use in safety-critical industries such as medical, automotive, and aeronautics, has not enjoyed the study and availability of formal verification tools on such a scale. Formal verification of machine-code systems is problematic due to a unique combination of challenges: large state spaces, the loss of high-level information about the programs, and the diversity of various processor architectures.

Previously, in my diploma thesis [A.4], I noted that there were no publicly available formal verification tools for machine-code systems, and created a tool based on previous research, using the ubiquitously used model checking with abstraction. However, it was not good enough for practical purposes due to the aforementioned challenges. In particular, *abstraction refinement* is typically used in state-of-the-art tools for formal verification of source-code and hardware systems to reduce the state space size, but adding it while allowing for diverse processor architectures was previously not reasonably possible.

I set out to overcome the challenges, at least partially, during my doctoral research. I devised three novel techniques, described them in publications and implemented them in my publicly available, free, and open-source formal verification tool **machine-check**[1], a spiritual successor to my previous tool. In this thesis, I comprehensively present the background of my work, the devised techniques, as well as their experimental evaluation. Using my tool, I was able to verify that various specifications hold in bare-metal programs for the AVR ATmega328 microcontroller. The techniques enable verification in reasonable time and memory without problems of the previous tools, substantially improving the state of the art in formal verification of machine-code systems.

[1]The latest release of **machine-check** is available at `https://crates.io/crates/machine-check`. The current release at the time of writing this thesis is available at `https://crates.io/crates/machine-check/0.3.0`.

1.1 Contribution

I devised, described, and, where applicable, formally proved three novel techniques:

- **Translation of simulable machine-code system descriptions.** Previously, tools for formal verification of machine-code systems were largely tailored to a specific processor, and abstraction was managed manually, making the addition of new processors and architectures highly complicated and time-consuming. I devised a scheme where the processors are described in the programming language Rust and automatically translated to their verification analogues at compile time using meta-programming. I published an overview of the scheme [A.2].

- **Input-based three-valued abstraction refinement framework.** The usual abstraction refinement scheme is Counterexample-guided Abstraction Refinement (CEGAR), which cannot verify some system properties, importantly including whether the system can recover to a specified state from any state. Properties such as recovery can be verified using the stronger Three-valued Abstraction Refinement (TVAR), but the previous abstraction refinement frameworks based on TVAR were problematic. Therefore, I devised a novel framework based on TVAR that resolves the problems, and proved that it can be used for formal verification of arbitrary properties of propositional µ-calculus. A preprint of my work is available [A.3].

- **Three-valued bit-vector arithmetic.** When using abstraction, the abstract domains of system-state variables must be chosen, with a dramatic impact on verification speed and the ability to prove or disprove properties. In machine-code systems, it is typical that only one specific bit of an input port determines whether the property holds or not. This can be abstracted well by three-valued bit-vector abstraction, but it was previously not possible to quickly compute useful results of arithmetic operations using such abstraction. I devised a novel technique for computing useful results in polynomial time, with the best possible results for addition and multiplication computable in linear and quadratic time, respectively. I described, formally proved, and published the technique together with my supervisor [A.1].

I implemented the techniques in my free and open-source tool **machine-check**. It is able to verify properties of machine-code programs as well as any other systems that can be described as finite-state machines. The required time and memory are reasonable for simple programs, and the framework and its implementation are conducive to improvements in abstraction and refinement strategies, paving the way to full formal verification of security- and safety-critical systems, not just their hardware or source-code components.

While my research was focused on the machine-code level, the techniques are general and of interest in other fields of formal verification, especially of source-code and hardware systems. The input-based TVAR framework in particular is fully general and can be used to verify properties not verifiable by current state-of-the-art tools, further supporting the overarching goal of leveraging formal verification for greater security and safety.

1.2 Organisation of This Thesis

The dissertation thesis is organised into chapters as follows:

1. **Introduction**: Describes the contribution of my research.
2. **Background and State of the Art**: Introduces the reader to the necessary theoretical background and surveys the current state of the art.
3. **Translation of Simulable Machine-Code System Descriptions**: Describes the technique of translation of simulable descriptions used in my tool **machine-check**. Contains previously published material [A.2] with further additions.
4. **Input-based Three-valued Abstraction Refinement Framework**: Describes the Three-valued Abstraction Refinement framework I proposed and implemented, elegant yet more powerful than the commonly used Counterexample-guided Abstraction Refinement frameworks. It is proven that its important characteristics, which make it suitable for formal verification, depend only on fairly simple requirements. Contains material available as a preprint [A.3].
5. **Abstract Three-valued Bit-vector Arithmetic**: Describes a three-valued abstraction domain that is useful for digital (especially machine-code) systems, and presents fast algorithms for addition and multiplication within the domain, with proofs that they produce the best possible results. Contains previously published material [A.1].
6. **Created Machine-Code Formal Verification Tool machine-check**: Describes the combination of the techniques from Chapter 3, 4, and 5 and further considerations for implementation of the formal verification tool that I created during work on this thesis. Discusses an experimental evaluation of the tool on machine-code systems.
7. **Conclusion**: Summarises the results of my doctoral research, suggests possible topics of further research, and concludes the dissertation thesis.

Chapter 2

Background and State of the Art

The rise of digital electronic systems to ubiquity in our lives has brought a multitude of challenges. Computing devices are no longer restricted to mainframes and personal computers but are present in all kinds of aspects of our lives including medical devices, home appliances, or toys. Most of us carry a mobile phone, and we rely on transport by cars, trains, ships, and aeroplanes, all of them increasingly dependent on electronic control. All aspects of our lives, including our security and safety, are reliant on the systems behaving correctly. Nevertheless, just during the last two months of writing this thesis,

- the outage caused by the CrowdStrike software glitch paralysed the global economy and resulted in losses estimated in billions of dollars [1],
- the leading processor manufacturer Intel has responded to the instability of its 13th and 14th generation processors, releasing a processor microcode update which is supposed to fix incorrect voltage requests that result in processor degradation [2],
- a vulnerability was found in AMD processors including the current generation [3], undetected for almost 20 years, allowing the planting of nearly undetectable malware once kernel-level access is obtained [4].

While informal testing of systems can reveal some bugs, *formal verification* can definitely prove or disprove that a given specification holds in a given system, preventing bugs that arise from unconsidered corner cases. Unfortunately, it is much more problematic to formally verify systems than to create them. Continual advances in formal verification are necessary to prevent the consequences of bugs such as drastic monetary loss, data theft, loss of privacy, and even human injury.

Formal verification is a wide field of computer science which overlaps with other fields such as graph theory, automata theory, combinatorial optimisation, static program analysis, and testing. As such, in this chapter, I discuss only topics of direct relevance to the subject of this dissertation thesis. Related work that is only relevant to a single chapter will be discussed in the respective chapter.

In this thesis, I focus on formal verification of machine-code programs against specifications. The main processors under consideration are simple embedded microcontrollers, the programs are bare-metal, without any operating system layer. Machine-code verification is especially sensible in this scenario, as source-code verification may not be able to verify properties such as correct initialisation and usage of peripherals. Some parts of the program may also be hand-written in assembly language to achieve maximum performance, precluding source code verification. Lastly, the compiler may contain bugs, resulting in possible issues that are undetectable using source-level verification. The aforementioned concerns make machine-code verification an important and irreplaceable avenue of approach.

For a comprehensive understanding of the task, there are three major areas to explore:

- The digital systems under verification, their levels (hardware, machine code, source code), and commonalities shared by systems of all levels.
- The specifications for which we are trying to decide whether they hold in a given system.
- The techniques for formal verification.

These areas are interrelated: the digital systems are usually formalised as automata, and so the typical specification formalisms are based on states and paths through the automata. The verification techniques are based on the formalised systems and specifications. Adherence of finite-state systems to common temporal specifications can be verified in time and space linear to the state space size, with further improvements possible through the use of advanced techniques.

In this chapter, I will explore systems, specifications, and verification techniques. After that, I will focus on formal verification using model checking with abstraction refinement. Finally, I will examine the state of the art in formal verification of digital systems, noting the lack of usable tools for verification of machine-code systems.

2.1 Digital Systems

As proven by Shannon [5], systems comprised of switching circuits can be used to solve arbitrary problems specified in Boolean algebra by the construction of logic gates. The rise of transistor technology, especially Complementary Metal-Oxide-Semiconductor (CMOS) technology, has allowed us to construct systems with computation capabilities far overshadowing other kinds of systems. The systems are inherently parallel in nature, each logic gate only dependent on the ones producing its inputs.

While electronic hardware systems can be designed to perform fixed computations with great performance and little power consumption, the design and initial manufacturing expenses for such devices are prohibitive for most applications. As such, programmable devices are now commonly used as well. It is possible to group them into two categories, notwithstanding System on a Chip (SoC) combinations:

- Programmable Logic Devices (PLDs) are devices in which reconfigurable digital circuits can be configured to perform specified computations using basic elements such as logic gates and flip-flops, similarly to building the digital circuits themselves. The most complex of these devices are Field-Programmable Gate Arrays (FPGAs).

- Processors are devices that manipulate their state according to machine-code-program instructions. This typically results in less parallelism, with sequential program flow in each processor core. The programmer typically writes a source-code program in a programming language such as C, which is then compiled to machine code that is executed by the processor.

Let us suppose that we want to create a digital system. We are only responsible for designing a small part of the overall system, building on top of underlying components. For example, in a machine-code system, we design the machine code that will be executed on the selected processor and rely on the guarantees by the processor manufacturer that it will perform as described in its accompanying documentation. We devise the machine code based on these guarantees (not the physical device itself): without knowing anything about the processor behaviour, the machine code is just a meaningless sequence of bits.

In this thesis, I will use the noun *design* to refer to the part of the system that is under our control, and the noun *guarantees* to refer to the guarantees about the behaviour of the underlying parts of the system outside of our control. The design and guarantees combine to form the *system*. For verification purposes, the whole system must be considered, as visualised in the block overview in Figure 2.1.

The digital system will ultimately be backed by a physical device that behaves according to the physical reality. As such, there must be fundamental guarantees that the device behaves digitally. During verification, we assume that all guarantees hold, as we are only concerned with detecting problems where we are at fault, not problems arising due to the given guarantees not holding in the actual device.

In case the design is described in a language with formal syntax and semantics, basic guarantees are defined by the semantics of the formal language. However, there might be additional guarantees.

Example 2.1.1. Let us consider that we are writing a source-code program in the C language, and our compiler adheres to the C99 standard [6]. The language semantics defined in the standard give us basic guarantees about how the program will behave once compiled and executed, barring e.g. bugs in our compiler or a defective processor we will be compiling or executing the program on. We can also use libraries, with which we communicate using Application Programming Interfaces (APIs), with additional guarantees of their behaviour. When verifying our program, we take the source code we have written and the guarantees (language semantics and API guarantees) into account.

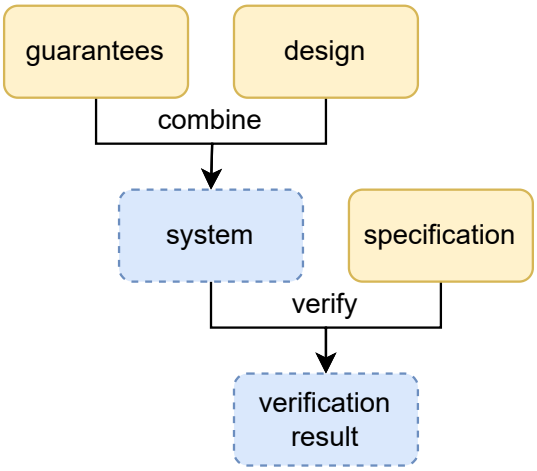

Figure 2.1: A high-level overview of formal verification of digital systems. The solid yellow cells represent verification inputs, while the dashed blue cells represent an automated combination or result. The guarantees and the design are combined together to form the system under verification. It is then determined if the specification holds.

2.1.1 Digital System Levels

In the vast majority of cases, a digital system can be placed into one of three separate levels:

- A *hardware system* is a combination of the digital design described in a design language, the basic guarantees provided by the design language, and possibly additional guarantees provided by e.g. Intellectual Property (IP) blocks. The design language is typically a Hardware Description Language (HDL) such as VHDL or Verilog. For formal verification, the systems are typically first translated to the AIGER format [7, 8] that describes the whole system as a sequence of gates or the Btor2 format [9] that describes the systems by bit-vectors and bit-vector arrays, preserving operations such as addition or multiplication instead of translating them to logic gate combinations. After translation, the guarantees are formed by the AIGER/Btor2 format semantics.

- A *machine-code system* is a combination of the design in the form of machine code, the basic executing processor guarantees, and possibly additional guarantees for e.g. circuits connected to the processor pins. For bare-metal use, the machine code is typically in the Intel HEX format [10]. For usage with operating systems, the machine code is typically bundled with some additional information, e.g. the Executable and Linkable Format (ELF) for Unix-like operating systems and the Portable Executable (PE) format for the Windows operating system. Unfortunately, the processor guarantees are usually not available as a formal specification, the informal documentation being provided in the form of the processor datasheet, the user manual, the instruction set architecture manual, etc.

- A *source-code system* is a combination of the design in the form of source code in a programming language, the basic guarantees provided by the semantics of the programming language, and possibly additional guarantees provided by e.g. the APIs of the used libraries. The programming languages can be standardised, as in the case of the ubiquitous C99 standard [6], but their more difficult semantics are typically described informally.

Some digital systems cannot be placed into a single level, such as source-code programs with inline assembly which combine source-code and machine-code characteristics, but I will not discuss them in this thesis for the sake of conciseness. Between the three levels, there are two special system types that seemingly mix the characteristics but tend to be closer to one level:

- A *bytecode system* is a combination of the design in the form of bytecode for a Virtual Machine (VM), basic guarantees provided by the bytecode specification, and possibly additional guarantees. The bytecode serves as an intermediate stage before interpretation or compilation on the target machine. The typical program bytecode is for the Java Virtual Machine (JVM). LLVM IR (Intermediate Representation) is used as an intermediate compilation stage for the LLVM compiler suite [11]. While the bytecode is a sequence of bits similar to machine code, the system as a whole is much more similar to a source-code system, with device-agnostic guarantees. Bytecode is sometimes used for verification in place of source code due to similar expressivity but simpler constructs.

- A *microcode system* is a combination of the design in the form of microcode and the underlying hardware guarantees, implementing a processor that is supposed to execute machine code with the guarantees given by the processor manufacturer. Structurally, the microcode system can be considered a machine-code system which serves to provide a Virtual Machine for the higher-level machine code.

During the discussion of the state of the art in Section 2.6, I will discuss the bytecode and microcode systems grouped with source-code and machine-code systems, respectively.

Example 2.1.2. Throughout this thesis, I will focus on AVR ATmega328P, a mid-line 8-bit microcontroller which is famously used in the Arduino Uno development boards. The microcontroller integrates an 8-bit AVR processor core with 32 working registers and additional Input/Output (I/O) registers together with 2048 bytes of Static Random Access Memory (SRAM) that is used as data memory and 1024 bytes of Electrically Erasable Programmable Read-Only Memory (EEPROM) that is used as program memory [12].

The hardware level of the microcontroller is known to the AVR processor manufacturer Microchip (which has acquired the former manufacturer Atmel), but not to the general public. The programs are usually written either in the C language (source code level) and compiled to machine code or in the AVR assembly language that corresponds to the machine-code instructions directly and is assembled to machine code. The instruction set is publicly available [13].

Note 2.1.3. Digital systems can be described using many specific languages, such as modelling languages (UML, SysML), simulation-oriented languages (Matlab-Simulink etc.), or verification-specific languages [14]. However, these languages usually present some general overview of the system, not a fully specified system that can be used in the real world. I will not discuss the specific languages further and will show in Chapter 3 that it is possible to describe digital systems using a general-purpose programming language and still retain the ability to use advanced verification techniques.

2.1.2 Digital System Commonalities

The transition from hardware up to source code is essentially a transition from physical systems to systems that correspond to human (predominantly sequential) reasoning. There are important commonalities between the systems, combining building blocks that are physically efficient and those that are conducive to human reasoning. These commonalities can be found by examining HDL languages, common processor architectures, and imperative programming languages:

- **Binary digits.** While other bases such as ternary and decimal have enjoyed some popularity in the past, the current digital systems are ubiquitously binary.
- **Finite-width bit-vector variables.** Unlike mathematical variables, the variables refer to some over-writable physical memory location. Only finite-width bit-vectors are physically implementable in the binary digital logic. They are used as basic building blocks for describing real-world digital systems.
- **Arrays and array indexing.** Bit-vector arrays are ubiquitously used. In machine-code systems, only a few arrays are exposed through the machine-code instructions, typically including working registers and either the main memory (von Neumann architecture) or the program memory and the data memory (Harvard architecture). In imperative programming languages, only the main memory is exposed, and variables used to index into it are called *pointers* (typically treated differently from other variables to prevent bugs).
- **Fixed-point bit-vector operations.** There are five basic types of almost universally available bit-vector operations: bitwise operations, bit-shift operations, bit length manipulation operations, arithmetic operations, and relational operations. Some operations (such as bit extension or division) are dependent on the interpretation of the bit-vector, which is today almost universally treated as either unsigned or signed in two's complement. The interpretation is chosen either by the variable type (e.g. in typical imperative programming languages or VHDL `numeric_std`) or by a special operation choice (e.g. the processor instruction type).

While bit-vectors do not perfectly correspond to the mathematical notions of numbers, arithmetic operations can be performed using them if the distinctions are observed (e.g. sizing the variables to prevent overflows). While the arithmetic and relational operations

are provided due to the need to perform arithmetic and comparisons in number-based algorithms, the bitwise and bit-shift operations are provided because they are efficiently implementable. The combination allows for a number of "hacks", such as fast multiplication and division by powers of 2 using bit-shifting [15].

Note 2.1.4. Floating-point operations are outside of the scope of this thesis. Common processors and language implementations typically follow the IEEE 754 standard to a certain extent. Floating-point variables can be described as bit-vectors and the operations can be converted into bit-vector operations (*soft floating point*).

Example 2.1.5. In ATmega328P, the working registers, the data memory, and the program memory are the most important bit-vector arrays. I/O addresses can correspond to I/O registers or have special behaviour (e.g. reading digital values of microcontroller pins). Typical instructions perform indexing (e.g. of two registers) and perform some fixed-point operations using the indexed locations (e.g. adding the two registers and writing the result into one of them, writing status flags afterwards). The arithmetic operations mostly operate on 8-bit bit-vectors. Floating-point operations are not supported and must be emulated with soft floating point if necessary. The instructions correspond closely to C operations on 8-bit integers and are efficiently implemented in hardware, with most instructions executing in one clock cycle.

The commonalities can be used to describe the system at another system level or even automatically translate between the levels, adjusting the design to the new guarantees so that the overall system behaviour remains the same. In my approach, the machine-code system guarantees (mainly describing the processor behaviour) are described at the source-code level in the Rust programming language, leveraging its advantages. This will be elaborated upon in Chapter 3.

2.1.3 Formalisation as Moore Machines

Digital systems can be formalised as general automata with outputs. In practice, constructable systems are always finite, and can be formalised by deterministic Moore or Mealy Finite State Machines (FSMs): the system deterministically changes its state based on the values of its inputs and its behaviour is reflected in its outputs. I will discuss the Moore machine formalism as it is simpler.

Definition 2.1.6. A Moore machine M is a tuple $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$ where

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- Σ is the input alphabet, a finite set,
- Ω is the output alphabet, a finite set,

- $\delta : S \times \Sigma \to S$ is the state transition function,
- $\lambda : S \to \Omega$ is the output function.

The behaviour of the system is determined by the outputs of the successive states, starting in the initial state and applying the state transition function with the selected inputs. Dropping the requirements of finite S, Σ, and Ω, the resulting formalisation allows for non-constructable systems as well, such as source-code programs with variables that are unrestricted natural numbers.

The formalisation captures the system behaviour but not the practical considerations. Most notably, it is typically only necessary to consider the states and transitions reachable from the initial state, as the others are irrelevant to system behaviour. The commonalities from Subsection 2.1.2 are hidden in the definitions of $S, \Sigma, \Omega, \delta$, and λ, despite having an important practical role in the speed of simulation and verification of the system.

2.2 Property Specifications

Formalising digital systems as Moore machines allows for proving their predicate calculus properties. However, automatically formally verifying predicate calculus properties w.r.t. the machines is problematic for two reasons:

- **Proving in reasonable time and memory.** Trivially, proving or disproving that finite specifications (of finite length and with finite quantified variables) hold in finite systems can be accomplished in finite time and with finite memory using brute force. However, the amount of reachable states tends to grow exponentially to the input size, and verifying the specification can introduce further slowdowns. Furthermore, checking the properties of infinite paths is even more problematic.
- **Specifications difficult to express in predicate calculus.** In the specifications, we typically are concerned about properties of system states and paths through the system that might not be intuitive to express in predicate calculus.

While the full predicate calculus is too general, properties that only consider a single state of the system are too limited. *Temporal logics*, which describe the behaviour as paths through the state space are taken, are a good compromise and have become the most commonly used specification formalisms in formal verification. A temporal logic forms a useful, well-defined set of formulas with respect to the system under verification, referring not only to individual states but also to paths, which are typically infinite. Formulas of common temporal logics can be directly translated to predicate logic formulas.

The most important temporal logics in formal verification are Computation Tree Logic (CTL), Linear Time Logic (LTL), and CTL*, of which CTL and LTL are subsets. For conciseness, I will define CTL* first and then introduce CTL and LTL using it.

Definition 2.2.1. A CTL* property is a logical formula consisting of either an atomic proposition or a logical operator combining other CTL* properties. There are three kinds of such operators in CTL* [16, p. 7-10]:

- **Propositional logic operators.** Typically, these are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$, $\phi \Leftrightarrow \psi$.
- **Temporal operators.** These operators encode the desired behaviour on an infinite path through the system. There are five such operators:
  - $\mathbf{X}\ \phi$ (ne**x**t). The property ϕ has to hold at the next state of the path.
  - $\mathbf{G}\phi$ (**g**lobally). The property ϕ has to hold in every state on the path.
  - $\mathbf{F}\phi$ (**f**inally). The property ϕ has to hold in some state on the path.
  - $[\phi\mathbf{U}\psi]$ (**u**ntil). The property ϕ has to hold until ψ holds (not including the first state where ψ holds), and ψ must hold in some state on the path.
  - $[\phi\mathbf{R}\psi]$ (**r**elease)[1]. The property ψ has to hold before and during the first state in which ϕ holds, but ϕ does not have to ever hold (in which case, ψ must hold forever). In other words, ϕ *releases* ψ.
- **Path quantifiers.** These operators encode the quantification of paths from a given state.
  - $\mathbf{A}\phi$ (along *a*ll paths, inevitably). The property ϕ must be true in all paths from the given state.
  - $\mathbf{E}\phi$ (there *e*xists a path, possibly). The property ϕ must be true in at least one path from the given state.

For evaluation, the CTL* formula is implicitly enclosed by an along-all-paths quantifier if necessary (i.e. there exists a temporal operator not enclosed by a path quantifier), similarly to implicit universal quantification of free variables in predicate calculus. The resultant formula can be evaluated on arbitrary states of the system under verification. When a system with multiple initial states is considered, the CTL* property must hold in all initial states to hold in the system itself.

Example 2.2.2. Some of the more notable CTL* formula schemes are:

- **Safety.** $\mathbf{AG}[\phi]$, i.e. on all paths, ϕ holds forever.
- **Reachability.** $\mathbf{AF}[\phi]$, i.e. on all paths, a state where ϕ holds is reached.

[1] In some literature, the release operator is not included in CTL* and its subsets, simplifying the definition at the expense of losing operator duality. Alternatively, $[\phi\mathbf{R}\psi]$ can be thought of as an alias for $\neg[(\neg\phi)\mathbf{U}(\neg\psi)]$.

- **Recovery.** $\mathbf{AG}[\mathbf{EF}[\phi]]$, i.e. from every reachable state, there exists a path to a state where ϕ holds in the future. In other words, we can always somehow coerce the system to reach a state where ϕ holds.
- **Invariant lock.** $\mathbf{AG}[\phi \Rightarrow \mathbf{AG}[\phi]]$, i.e. once ϕ holds, it holds forever.
- **Action-reaction.** $\mathbf{AG}[\phi \Rightarrow \mathbf{AF}[\psi]]$, i.e. once ϕ holds, ψ must hold in that state or some successive state.

As stated previously, the two most ubiquitous subsets of CTL* are CTL and LTL:

- In CTL, the path quantifiers and temporal operators can only come in pairs in that order, e.g. $\mathbf{AG}\phi$ or $\mathbf{E}[\phi\mathbf{U}\psi]$. In essence, CTL allows posing statements about the current state and states following it, but not arbitrary paths.
- In LTL, no path quantifiers are permitted in the formula, the only one being the implicit along-all-paths quantifier. For example, the LTL formula $\mathbf{FG}\phi$ could be more clearly expressed in CTL* as $\mathbf{AFG}\phi$ [16, p. 9]. In essence, LTL describes each path through the state space separately, and the result is whether this description holds for all paths.

Each CTL* property can be translated into an equivalent formula of the (stronger) predicate μ-calculus [17, p. 907]. While the proofs in Chapter 4 are general enough for arbitrary μ-calculus properties, I will not discuss μ-calculus in detail as it is less understandable than classic temporal logics and it is not easy to find interesting properties expressible in it but not in CTL*.

2.3 Formal Verification Using Model Checking

Formal verification of systems against specifications can be accomplished in a multitude of ways. The most basic one is through manual proofs. However, that approach is only realistic for very simple systems. It is possible to construct automated proofs, but as the systems become complicated, devising the choices to make in the proofs becomes problematic. Fortunately, for finite-state state-transition systems, it is possible to use the *model-checking* approach to prove or disprove the properties completely automatically. In model-checking, the state space of the finite-state system is constructed and the properties are verified against the state space instead of the original system [16, p. 1-3]. This allows for completely automatic verification of the system in finite time and memory.

Unfortunately, while model checking seems excellent in theory, there are two main practical challenges encountered [16, p. 3-4]:

- **Scalability.** While the time and memory needed for model-checking is finite, in practice, it is infeasible to model-check all but the simplest systems without using advanced techniques. This is mainly due to the exponential explosion during the

construction of the reachable state space (also termed *state space explosion*): in each state, N input bits result in up to 2^N successor states being generated.

- **Modelling.** The classic model-checking formalisms essentially verify whether temporal logic specifications hold in finite-state machines. The modelling challenge is to fully capture many possible systems of interest, including unbounded systems and differently descriptive specifications (i.e. real-time logics).

The scalability and modelling challenges are subjects of ample research [16]. Both are major challenges to machine-code verification, although the modelling challenge is present mainly in the practical rather than in the theoretical sense, as the classic model-checking formalisms capture the nature of digital systems well.

2.3.1 Classic Model-Checking Formalisms

I will now introduce the classic formalisms for formal verification using model checking [16]. The name *model checking* is taken from mathematical logic: we are checking whether the formalism of the system, a Kripke structure K, is a model of the specification ϕ, i.e. whether all sentences in ϕ are true with respect to K. The fact that K is a model of ϕ is usually written as $K \models \phi$, and that fact that it is not is written as $K \not\models \phi$. The model-checking tool, given K and ϕ, ideally outputs either $K \models \phi$ or $K \not\models \phi$ (and perhaps some additional information such as the reasons for that result). In practice, it also may not give us any answer at all, such as when verification time or memory is exceeded.

Definition 2.3.1. A Kripke structure over a set $\mathbb{A}$ of atomic propositions is defined as a tuple $K = (S, S_0, R, L)$ where

- S is the set of states,
- $S_0 \subseteq S$ is the set of initial states,
- $R \subseteq S \times S$ is a transition relation,
- $L : S \times \mathbb{A} \to \{0, 1\}$ is a labelling function, which determines whether each atomic proposition holds in a state or does not.

Note 2.3.2. I use the Kripke structure definition with initial states throughout this thesis for correspondence with real-life digital systems. Some definitions omit the initial states. I use a characteristic labelling function instead of the more common definition $L : S \to 2^{\mathbb{A}}$ for easier formalisation of abstraction in Section 2.5.

The Moore machines from Section 2.1.3 can be easily turned into Kripke structures by replacing the output function with the labelling function (which may expose the internal state of the machine as well as the outputs), and turning the state transition function into a relation by considering all input possibilities. The major insight here is that the actual input values are unnecessary for computing the verification result. However, as discussed

later in Chapter 4, the inputs become relevant again when we are trying to determine what caused the result.

In classic model checking, the Kripke structure is checked against a CTL, LTL, or CTL* property. Although these temporal logics work with infinite paths, there are algorithms that can verify their properties in a reasonable time:

- **CTL.** Running time depends linearly both on the size of K and the length of the CTL formula [16, p. 11].
- **LTL.** Running time depends linearly on the size of K and exponentially on the length of the LTL formula [16, p. 13].
- **CTL*.** The algorithms for CTL and LTL can be simply combined [18, p. 69], resulting in running time depending linearly on the size of K and exponentially on the length of the CTL* formula, the same as for LTL.

As the size of K is the major limiting factor, the ability to verify in time linear to it is crucial, explaining the popularity of CTL, LTL, and CTL*. The state space explosion becomes the main problem.

Example 2.3.3. Let us consider formal verification of ATmega328P using naïve model checking. The I/O registers are reset during the device reset [12, p. 56], but the 32 working registers and 2048 SRAM bytes are not and may contain any value. The number of possibilities after reset is $2^{2048+32} = 2^{2080}$, which results in infeasibly many initial states. Even ignoring the initial possibilities does not save us. The General Purpose Input/Output peripheral allows reading of up to 8 binary pin values during single instruction execution. Reading four times in succession to different working registers produces $(2^8)^4 = 2^{32}$ combinations. Clearly, naïve model checking is not suitable for machine-code verification.

2.4 Advanced Techniques for Model Checking

As the state space explosion precludes verification of complex systems, advanced techniques have been devised to verify the system without constructing and model-checking the whole Kripke structure. Such techniques can be roughly classified in three groups [16, p. 15-18]:

- **Abstraction** is an approach where, instead of model-checking the original Kripke structure K, an abstract structure $\hat{K}$ with less information is model-checked. The result is either the same as for the original structure or is unknown due to the lack of information. As an unknown result is not useful, *abstraction refinement* can be used to keep adding information to $\hat{K}$ where it is necessary for verification until a definite result of model-checking is obtained.

- **Symbolic methods** avoid construction of the Kripke structure by using symbolic logic expressions to represent states and/or transitions, essentially compressing the state space by using a more compact representation. There are two main symbolic method subgroups:

  - Model checking with Binary Decision Diagrams (BDDs). Useful especially for low-level hardware circuits described by Boolean expressions where they can dramatically reduce state space size while retaining the verification complexity, but problematic to use with arithmetic expressions due to the reintroduction of exponential explosion.

  - Model checking based on solving the Boolean satisfiability problem (SAT) and its extensions. The system and the specification are encoded into SAT formulas that are solved by general SAT solvers. This approach allows the separation of describing the systems from the actual verification, which is reduced to a combinatorial problem. LTL or ACTL* formulas can be verified by SAT solvers[2]. For CTL and its supersets, the stronger Quantified Boolean Formula (QBF) solvers are necessary.

- **Structural methods** exploit the structure of the code that defines the system. The structural methods are usually associated with parallel systems or complex reasoning, using symmetries, partial orders, or other higher-level information to avoid storing the whole Kripke structure.

In practice, the groups of approaches tend to be combined. In particular, symbolic methods and abstraction are very conducive to combination since they are cleanly separated: the symbolic methods are applied once the system is abstracted. In fact, the widely-used Counterexample-Guided Abstraction Refinement (CEGAR) methodology was originally described as used with BDDs [19] and later extended for use with SAT solvers [20]. The SAT solvers themselves have evolved to Satisfiability Modulo Theories (SMT) solvers, which support solving formulas with e.g. bit-vector or mathematical integer variables in addition to Boolean variables.

For my core use case of machine-code verification, the use of abstraction is key to reasonable state space sizes. Symbolic methods can be used with abstraction, and they are well-researched [21, 22], but their use is mostly an implementation decision rather than a fundamental concept in a verification tool. I did not feel the need to use structural methods at this point as they are most useful when some kind of parallelism is introduced, and non-parallel machine-code programs are problematic enough as-is. As such, I have focused on abstraction in my work, and will not discuss the other groups of techniques further except where used in state-of-the-art tools discussed in Section 2.6.

[2]For systems with left-total transition relations, as later noted in Section 2.5.1.

2.5 Abstraction and Abstraction Refinement

The ideal process of model-checking can be represented as a function $P_\phi : \mathbb{K} \to \{0, 1\}$ where $\mathbb{K}$ is the set of all Kripke structures containing the atomic propositions present in the specification ϕ. For every $K \in \mathbb{K}$, $P_\phi(K) = 0$ means that the model-checker determined $K \not\models \phi$, and $P_\phi(K) = 1$ means it determined $K \models \phi$. We can extend the process to model-checking Kripke-like structures that can lack some of the original information, representing the incomplete model-checking process by $\hat{P}_\phi : \hat{\mathbb{K}} \to \{0, 1, \bot\}$, where the results $0, 1$ behave the same as previously and $\bot$ (unknown) means nothing was proven or disproven due to the lack of information[3]. The three distinct valuations 0, 1, $\bot$ give rise to *three-valued logic*, which will be discussed in more detail later in Chapters 4 and 5.

Abstraction in model checking consists of devising an incomplete structure $\hat{K}$ for verification and verifying properties of K using it. The abstraction should be *sound*, never producing wrong results:

$$\hat{P}_\phi(\hat{K}) \neq \bot \Rightarrow \hat{P}_\phi(\hat{K}) = P_\phi(K). \tag{2.1}$$

While it may be infeasible to compute $P_\phi(K)$ in practice, devising $\hat{K}$ and computing $\hat{P}_\phi(\hat{K})$ can be easier as $\hat{K}$ can contain less information and have fewer reachable states. The obvious problem is that if $\hat{P}_\phi(\hat{K}) = \bot$, we have not learned anything useful about K, only that $\hat{K}$ is not a good enough abstraction for our purposes. This problem is resolved by *abstraction refinement*, where, after computing $\hat{P}_\phi(\hat{K}) = \bot$, we refine $\hat{K}$ to contain more information, and continue until $\hat{P}_\phi(\hat{K}) \neq \bot$. This refinement loop forms the core of *abstraction refinement frameworks*, which we design to be sound and, optionally, *complete*, always verifying that the specification either holds or does not in finite time and memory for finite systems and specifications.

Example 2.5.1. Continuing in the example of verification of machine-code programs for AVR ATmega328P, we can represent each bit in a bit-vector by one of three values, '0' (definitely zero), '1' (definitely one), or 'X' (unknown — possibly zero, possibly one), forming *three-valued bit-vector abstraction* (discussed further in Chapter 5). Representing each uninitialised or input bit by 'X', we can start with a single abstract state that has all working registers and SRAM locations unknown, and representing all step inputs as unknown as well, produce a single abstract state in each processor step, ending up with a lasso-shaped state space. Unfortunately, as the step function is required to preserve soundness, the result of verifying properties that are dependent on inputs will be unknown, rendering the abstraction fairly useless. It is necessary to choose the abstract bits that will be turned to '0' and '1' possibilities, increasing the amount of information at the cost of increased state space size. Using abstraction refinement, we can choose the bits of interest deductively, without any outside help.

[3]Not being to able to prove or disprove due to the lack of information in $\hat{K}$ is different than not being able to prove due to e.g. the model-checker being terminated due to exceeding time or memory. The former gives us useful information about $\hat{K}$, while the latter gives us nothing at all, and is thus not formally considered.

2.5.1 Methodologies

Now that the basic notions are in place, we can discuss the abstraction refinement methodologies. While Chapter 4 contains a more comprehensive and formal description of common abstraction frameworks based on the methodologies, I will give a basic overview based on Dams and Grumberg [23] here.

First, I will consider that the commonly used *existential abstraction* is used, where the *abstract states* of $\hat{K}$ are related to the *concrete states* of K by a concretization function $\gamma : \hat{S} \to 2^S$. In essence, an abstract state represents that the concrete system might be in any of the concrete states given by the concretization function. For conciseness, I will write that the abstract state *covers* a concrete state in if it contains it in its concretization. Similarly, I will write that a path of abstract states covers a path of concrete states exactly if, in each position, the abstract state covers the concrete one.

Counterexample-guided Abstraction Refinement (CEGAR). The introduced abstract structure $\hat{K}$ is a Kripke structure but has a very different meaning compared to K. The states of $\hat{K}$ are abstract states. The transitions present in the transition relation $\hat{R}$ of $\hat{K}$ give no useful information as they may or may not correspond to concrete transitions between the concrete states covered by the endpoints. However, it is required that the transitions in the complement of $\hat{R}$ do not correspond to any such concrete transitions, i.e.

$$\forall(\hat{s}_{\text{head}}, \hat{s}_{\text{tail}}) \in (\hat{S} \times \hat{S}) \setminus \hat{R} \,.\, \forall(s_{\text{head}}, s_{\text{tail}}) \in \gamma(\hat{s}_{\text{head}}) \times \gamma(\hat{s}_{\text{tail}}) \,.\, (s_{\text{head}}, s_{\text{tail}}) \notin R. \quad (2.2)$$

Additionally requiring covering each state in S_0 by at least one state in $\hat{S}_0$, the set of paths in K is *overapproximated* by the set of paths in $\hat{K}$. Each path in K is covered by some path in $\hat{K}$ but a path in $\hat{K}$ can cover zero paths in K.

Assuming that R is left-total[4], the temporal properties expressible in LTL or the universal fragments of CTL, CTL*, and propositional µ-calculus (the universal fragment essentially precludes existential quantifiers when the property is expressed in negation normal form) depend only on the set of paths, so we can use $\hat{K} \models \phi$ to conclude $K \models \phi$. However, it is not possible to use $\hat{K} \not\models \phi$ to conclude $K \not\models \phi$, as the counterexample path may not be contained in the set of paths in K (it may be *spurious*). However, we can overcome the problem by using the following refinement loop:

1. Model-check $\hat{K}$ instead of K. If $\hat{K} \models \phi$, conclude that $K \models \phi$: as the aforementioned properties are violated by paths and $\hat{K}$ covers all paths in K, the property must hold in K when it does in $\hat{K}$.

2. We know that $\hat{K} \not\models \phi$. Obtain a path that violates the property in $\hat{K}$ (the *counterexample*) and validate if it violates the property in K as well. If it violates the property in K, conclude $K \not\models \phi$, providing the counterexample.

[4]If the transition relation in K is not left-total, there are some implicit existential characteristics in the universal fragment of CTL* and some of its subsets [23, p. 392-393]. Fortunately, digital systems expressible as automata have left-total transition relations (there is always at least one next state).

3. We know that $\hat{K} \not\models \phi$, but the counterexample for $\hat{K}$ is not a counterexample for K (it is *spurious*). Refine $\hat{K}$ somehow, so the abstract paths ideally cover fewer concrete paths, and go back to Step 1.

The core CEGAR methodology can be implemented in various ways, not dictating the choice of the abstract state space beyond existential abstraction nor the choices of refinement. However, it cannot verify properties such as the recovery properties discussed in Example 2.2.2, because they contain both universal and existential quantifiers in negation normal form.

Three-Valued Abstraction Refinement (TVAR). The TVAR methodology allows verification of full propositional μ-calculus and its fragments such as CTL*, CTL, and LTL. For $\hat{K}$, extensions of Kripke structures are used. A Partial Kripke Structure (PKS) introduces the possibility of state labellings being unknown, which means there is a possibility of an unknown result of model-checking $\hat{K}$, in which case $\hat{K}$ is refined. A Kripke Modal Transition System (KMTS) structure further introduces the possibility of transitions having an unknown presence.

Unfortunately, as TVAR is not limited to specification with path counterexamples, the algorithms for verification tend to be more complicated, and it is not as easy to provide a counterexample when a property is violated. The TVAR frameworks and structures will be explored in detail in Chapter 4.

2.5.2 Abstraction Domains

To properly and effectively leverage abstraction, we need to decide how the system will be abstracted, keeping the number of reachable abstract states low but with enough information needed to verify the properties. In practice, it is also necessary to be able to compute the transition function for the abstract states reasonably fast.

Digital system states are typically composed of separate variables. We can assign *abstract domains* to the variables to form the abstraction. There are two general groups of abstract domains, non-relational and relational. We are mostly interested in bit-vector domains since bit-vectors are commonly used in digital systems, as previously discussed in Subsection 2.1.2.

The abstract domains can be considered using various underlying formalisms. For model checking with abstraction refinement, existential abstraction is sufficient, but it is common to describe domains using abstract interpretation, which extends existential abstraction and allows using additional algorithms.

In non-relational domains, each variable is considered separately. Some examples of domains for bit-vectors are:

- **Constant domain.** The abstract bit-vector either has a constant value or can have any value ($\top$).
- **Sign domain.** Only the signedness of the bit-vector in the two's complement is retained. Zero can be treated as a special value as well, and any possibility (negative,

zero, positive) is represented by $\top$, resulting in the abstract value being represented by $\{-, 0, +, \top\}$. Other variations are also possible, e.g. also considering non-negative and non-positive abstract values.

- **Interval domain.** The abstract bit-vector value is restricted to some interval. This requires interpreting the bit-vector value as a number, differing depending on whether we consider it to be signed or unsigned. Problematically, signedness may vary depending on the machine-code instruction or hardware operation, leading to research into wrap-around arithmetic [24].

- **Three-valued bit-vector domain.** Each bit of the bit-vector is considered as separate, expressed in three-valued logic. This domain is discussed in detail in Chapter 5.

Relational domains allow expressing relationships between variables, such as the octagon abstract domain [25]. The richest abstraction domain is predicate abstraction, where the abstract states retain information about whether some chosen predicate calculus formulas hold in them.

Note 2.5.2. The common formalism for abstract model-checking is that of lack of information, using the symbol $\bot$ for no information. However, the common formalism for abstraction domains, coming from program verification and abstract interpretation, is that of possibilities, using the symbol $\top$ for all possibilities. These formalisms are dual and both correspond to the third value in three-valued logic.

The choice of a suitable domain is heavily dependent on the system and the verified property[5]. While a more descriptive abstraction may reduce state space explosion, its effectiveness may be limited due to slower computation of transitions.

It has been previously observed that a major exponential explosion in formal verification of machine-code programs occurs when reading the General Purpose Input/Output (GPIO) port values [26, 27, 28], an instance of this phenomenon already discussed in Example 2.3.3. The most suitable domain for resolving this problem is the three-valued bit-vector domain, which I decided to use in my diploma thesis [A.4]. However, it was previously not possible to perform arithmetic operations in the three-valued bit-vector domain without exponential explosion within the operation, leading me to devise new algorithms that solve this problem, as described in Chapter 5.

Note 2.5.3. I conducted some preliminary research into interval abstraction for machine-code verification, but it is not integrated into my tool **machine-check** yet. While I think that it is worthwhile to use other abstraction domains in addition to the three-valued bit-vector domain, they are outside of the scope of this thesis.

[5] If we had an oracle that could always choose the domain resulting in the quickest verification with a non-$\bot$ result, verification would be quick and there would be no need for abstraction refinement.

2.6 State of the Art in Digital System Verification

By comparing the state-of-the-art formal verification tools for hardware, source-code, and machine-code systems, we can discover how the differences in the system levels result in differences in approaches taken to describe and verify the systems. I will discuss source-code and hardware verification first as they are used in practice with a variety of competing tools [29]. I will restrict the discussion to freely available verification tools and scientific research. Even though the basic techniques used in commercial tools may be similar to the ones in freely available tools, the details are not well-known.

2.6.1 Source-Code Systems

The main source for determining the state of the art in formal verification of source-code systems is the **SV-COMP** competition, organised yearly from 2012 onwards. In the latest competition [30], there were 59 verification tools participating, showing that source-code verification is highly established in the formal verification community. The main programming language in SV-COMP and most competing tools is the C language, widely used e.g. in operating system kernels and drivers where programming bugs can severely impact the security or safety of the affected computers. In the latest **SV-COMP** competition, there were 30300 C verification tasks [30, p. 300], showing the maturity of work on benchmarking of source-code verification. Nevertheless, good results in **SV-COMP** do not necessarily mean that the tools are applicable to industrial use [31].

Notably, all specifications in **SV-COMP** are simple LTL formulas in the form $\mathbf{G}\phi$, where ϕ is an atomic property, or $\mathbf{F}\phi$ for termination [32]. As such, it is possible to verify the specifications using simpler reachability-based algorithms rather than the algorithms for checking e.g. LTL, CTL, or CTL* specifications. Simple path-based counterexamples can be generated where $\mathbf{G}\phi$ is violated. I would argue that the focus on degenerate specifications may be detrimental to the diversity of research: there is no quantitative motivation for verifying more complex specifications that correspond to less trivial violations. Instead, the verification tools are incentivised to present quantitative improvement for the degenerate specifications.

While there are many participating tools in **SV-COMP**, two tools stand out in particular, frequently placing in top three or winning many categories: **CPAchecker**[6] [33] and **Ultimate Automizer**, part of the **Ultimate** program analysis framework [34]. Notably, both of the tools use CEGAR and encode the program and the property into Satisfiability Modulo Theories (SMT) formulas which are checked by underlying SMT solver tools [30, 35]. Further techniques are used to extend this basic concept or introduce additional improvements, but they are beyond the scope of this thesis.

[6] As a part of fulfilment of requirements for submitting my doctoral thesis, I went on a month-long study stay at the Software and Computational Systems Lab of the Ludwig Maximilian University of Munich, which develops **CPAchecker**. I contributed to its predicate abstraction component, allowing verification of programs that use the standard C library memory-manipulation functions `memset`, `memcpy`, and `memmove`, and improved the treatment of quantifiers.

Note 2.6.1. Bytecode was also used for verification. In the main part of the competition, the **DIVINE** model checker, which uses LLVM IR bytecode [36], was entered outside of the competition (hors-concours). The **SV-COMP** competition also features a track on verification of Java programs. The Java track is decidedly less popular, with only 9 tools participating, 4 of them hors-concours. The top three tools **MLB** [37], **JBMC** [38], and **GDart** [39] all use JVM bytecode. All of the hors-concours tools used the JPF framework [30, p. 308-310], which works with JVM bytecode as well [40].

2.6.2 Hardware Systems

The main hardware system formal verification competition is the Hardware Model Checking Competition (**HWMCC**). Despite hardware verification being widespread in industrial practice [29], the latest **HWMCC** at the time of writing this thesis was in 2020 [41], with only 11 verification tools participating, 6 of them competitively, showing distinctly lower popularity than for source-code verification. Out of these tools, the model checkers **AVR** (Abstractly Verifying Reachability) and **ABC** performed the best. **AVR** is based on the IC3 algorithm used with SMT solvers [42]. The IC3 algorithm tries to refine sets of states in steps reachable from the initial states, the sets of states determined by invariants that hold [43]. **ABC** is much more hardware-specific, based on single-bit handling via And-Inverter Graphs, using multiple algorithms to handle verification, one of them being abstraction refinement [44, p. 36].

Recently, the **Btor2C** tool has been introduced, allowing verification of hardware systems using software tools using translation from hardware to C source code [45]. The software tools typically under-performed the hardware ones except for a few of the tasks in the benchmark, an expected result as the tools are tailored to the specific system level. Notably, the translation exploited the commonalities of bit-vectors in digital systems discussed in Section 2.1.

2.6.3 Machine-Code Systems

At the time of writing, I am not aware of any publicly available formal verification tools for machine-code systems other than my tool **machine-check**. There has been some research in the field, but the created tools either never have been publicly available or are not available currently.

A formal verification tool for embedded systems **HOIST** was described by Regehr and Reid [46], and used for stack size estimation [47]. The tool essentially builds abstract Binary Decision Diagrams from the results of instructions of an embedded processor or its simulator. Unfortunately, this results in high time and memory requirements, making the technique costly for 8-bit and infeasible for 16-bit or 32-bit processors. Abstraction refinement was not needed for the authors' use-case of stack size estimation but presumably would have been necessary for verifying arbitrary properties.

The **Estes** model checker was introduced by Mercer and Jones [48]. **Estes** used the **gdb** debugger to step through processor states and thus theoretically could support multiple

processor models as long as **gdb** supported them. However, in practice, extensive changes were necessary to adapt the debugger to model checking on the Motorola 68hc11 processor.

The **StEAM** model checker was introduced by Mehler [49]. It did not perform verification for specific hardware but compiled a C/C++ program under verification to the Internet Virtual Machine (IVM). It could be also considered a bytecode verification tool but was considered to be a machine-code verification tool by Mehler. The approach did not become popular in practice, presumably due to the fact that it reduces the amount of high-level information available in the source code, making verification harder.

The model checker that inspired **machine-check** the most was **Arcade.µC** (previously **[mc]square**), developed at the RWTH Aachen University. It was introduced by Schlich and Kowalewski [26][7], and built the state space directly using a custom simulator written for a specific processor, checking CTL formulas, with special handling of nondeterminism to prevent state space explosion. **Arcade.µC** was developed to use abstraction techniques using three-valued bit-vectors [27], including delayed instantiation of the variables after masking of inputs by logical instructions so that state-space explosion is mitigated [50]. Subsequent versions of **Arcade.µC** introduced static analysis techniques, enabling more efficient verification at the cost of further need for custom tailoring of the verifier to the processor [28, 51]. Interval abstraction was also added, working in concert with three-valued bit-vector abstraction to provide a further reduction of the abstract state space [52].

To reduce the difficulty of adding new microcontroller types and architectures to **Arcade.µC**, synthesis of state space generators was developed [53]. This approach was not entirely successful as the author was unable to implement abstraction in such a way that would not require the description writer to tailor the description to it [53, p. 121]. In the end, the work on verification of microcontroller machine code using **Arcade.µC** was abandoned in favour of verification of Programmable Logic Controller (PLC) programs, which are typically much simpler. To my knowledge, **Arcade.µC** never supported abstraction refinement, though its successor **Arcade.PLC** implemented CEGAR [54]. Furthermore, while **Arcade.µC** used three-valued bit-vectors for abstraction [27], it was not possible to compute arithmetic operations in the abstraction, forcing exponential explosion.

Note 2.6.2. Little has been published on formal verification of microcode but fairly comprehensive summaries of related work have been presented by Davis et al. [55] and Goel et al. [56]. This is expected as microcode is the core intellectual property of processor design companies. Due to its similarity to normal machine code, machine-code verification tools could potentially be used for microcode verification. In addition, some processors contain machine code in Read-only Memory (ROM) programmed by the manufacturer, providing features such as Secure Boot. Bugs in such machine code may be unfixable on already manufactured devices. For example, a buffer overflow vulnerability in manufacturer-provided ROM machine code allowed exploits on a range of NXP devices, and it was only fixed on newly manufactured devices, leaving many devices vulnerable [57].

[7]The paper [26] also mentions a previous machine-code model-checker MCESS. The model-checker seems to be developed in a single diploma thesis, the text of which I was unable to procure. It seems MCESS was not developed further.

2.6.4 Comparison of System Levels

Formal verification of source-code systems is the most popular, with hardware systems less popular but still actively researched, with open-source tools available. Formal verification of machine-code systems, on the other hand, has been under-researched, with a few scarce and later-aborted attempts. This seems to be due to the need to combine easy writing of processor descriptions and management of the abstraction so that the state space size is not infeasibly large but the verification is still useful.

Unfortunately, the inability to formally verify machine-code systems makes it difficult to verify processor-specific details such as peripheral manipulation or hand-written assembly code. Furthermore, it is difficult to formally verify that the machine code generated by the compiler is truly correct. These verification blind spots are dangerous, especially to safety- and security-critical systems.

As for the techniques used, state-of-the-art verifiers have converged to the nearly ubiquitous use of abstraction refinement, using CEGAR or similar techniques (such as IC3) that iteratively take more information into consideration until it is possible to determine whether the specification holds or not. Symbolic techniques, especially when combined with SMT solvers, are widespread, although not universal.

Translation is typically performed by compilers, not verification tools themselves, but translation from hardware systems to source-code systems performed specially for the purpose of verification has appeared in the **Btor2C** tool.

2.7 Summary

In this chapter, I discussed digital systems with a focus on verification and introduced three system levels: hardware, machine-code, and source-code. I noted that there are commonalities between system levels, especially the use of bit-vector and bit-vector array variables with their respective operations, motivated by bringing together physical efficiency and usefulness to humans. I discussed how digital systems of all levels can be formalised as Moore machines (or their non-finite equivalents).

I discussed why standard predicate logic is not typically used for formal verification, noting the problems with feasibility and difficulty of expression of temporal properties in predicate logic. I then introduced the common temporal logics CTL*, CTL, and LTL.

Having introduced the systems and the specifications they are verified against, I introduced *model checking*, a commonly used approach to formal verification of digital systems, and defined the Kripke structure formalism.

After introducing model checking, I introduced the three major groups of advanced model-checking techniques, focusing on the abstraction refinement and noting that there are two methodologies for it, Counterexample-guided Abstraction Refinement (CEGAR) and Three-valued Abstraction Refinement (TVAR). CEGAR can verify LTL properties, but not all CTL or CTL* properties. TVAR can verify all CTL*, CTL, and LTL properties.

I discussed abstraction domains, and how their choice is of crucial importance for efficient verification.

Finally, I discussed the state of the art in formal verification of digital systems. Source-code verification is the most popular and is heavily focused on LTL-style properties. Suitable tools are available for hardware verification as well. However, machine-code verification is currently not practically usable due to the necessity of tailoring the tools to specific architectures or devices and the lack of support for abstraction refinement that is utilised in verification at the source-code and hardware levels.

The inadequacies that I encountered led me to devise novel techniques that will be discussed in the rest of this thesis. In Chapter 3, I focus on enabling machine-code verification without tailoring to a specific architecture using translation of simulable processor descriptions. In Chapter 4, I introduce a novel TVAR framework, so that arbitrary µ-calculus properties (including CTL*, CTL, and LTL properties) can be verified with abstraction refinement. In Chapter 5, I describe the technique of fast computation of arithmetic operation results in three-valued bit-vector abstraction, which was previously not possible and severely limited machine-code verification. Finally, in Chapter 6, I will discuss how I combined the techniques in my tool, and provide experimental results that show its usability for machine-code verification.

Chapter 3

Machine-Code Verification Using Translation of Simulable Descriptions

An important problem for the verification of machine-code systems is that the guarantees for the underlying processors are usually only given informally in the accompanying documentation. While the machine code itself is a well-defined bit sequence, it is necessary to formalise the guarantees before the system can be verified. While there are specially created description languages for verification [14], I strove to instead use a general-purpose programming language because they are popular, well-developed, and provide various conveniences such as syntax highlighting, linting, and library management. I succeeded by devising a novel translation technique.

To formalise the guarantees given for the processor, we can write its *simulable description*, which I define as code in a general-purpose programming language that describes the processor behaviour as a finite-state machine (FSM). The FSM is parameterised by the machine code that will be executed on the processor. By instantiating the simulable description with the machine code as a parameter, the machine-code system is formed, and it can be simulated by stepping the FSM.

Unfortunately, without additional reasoning, the simulable descriptions are only verifiable explicitly, precluding abstraction refinement and making verification of reasonably complex systems infeasible. However, it is typically hard to reason over constructs of general-purpose programming languages as they are written with expressivity in mind.

To ensure that abstraction refinement can be used in conjunction with simulable descriptions, I devised a technique of translating the simulable descriptions to their *verification analogues*, using meta-programming (automatically rewriting code to other code).

A verification analogue is code added to the simulable description (not changing its own behaviour) that is written in the same language and behaves analogously to the description code, but using a different interpretation of the language constructs than the usual. Specifically, in the *abstract analogue*, the data types are changed to abstract types (e.g. bit-vectors to three-valued bit-vectors), and the functions are adjusted accordingly. The *refinement analogue* is used to find the reason for an unknown verification result,

the data types and algorithms are transformed so that the finite-state machine is stepped backwards, deductively finding possible causes for the unknown result. The verification analogues and the translation process will be discussed in more detail in Chapter 6.

I implemented the technique in my formal verification tool **machine-check**, written in the Rust language. The simulable descriptions are written in a subset of the Rust language, and they are translated to their verification analogues using a *macro*, a special Rust language construct that allows meta-programming during compilation. Since the verification analogues themselves are subject to compilation, they can be optimised by the compiler, improving verification performance.

In this chapter, I will introduce the high-level process of verification from the point of view of writing a processor description for verification of machine-code systems using **machine-check**, without considering the internals. I will show the description of a very simplified Reduced Instruction Set Computer (RISC) processor, construct a machine-code system using a hard-coded machine-code program, and discuss some properties that can be verified to hold. After that, I will discuss the subset of Rust in which the descriptions can be written, noting that arbitrary digital systems can be described, not just machine-code systems.

Note 3.0.1. Sections 3.1 and 3.2 in this chapter, describing the point of view of a processor description writer, are based on the contents of my paper [A.2], reworked for inclusion in this thesis.

3.1 Verification of Machine-Code Systems

A machine-code system is composed of the machine code itself and the processor which executes it. This means that both the machine code and the processor description are necessary for formal verification of the system against a specification, as shown in Figure 3.1. While the machine code is some well-defined bit sequence (or multiple sequences in non-consecutive locations), stored e.g. in the Intel HEX format, the processor descriptions are typically only given in the human-readable form of datasheets and user manuals. Sometimes, processor simulators are available, either from the manufacturer or some third party. Unfortunately, the descriptions of the processors in simulators are not usable for formal verification using model checking with abstraction refinement, as that requires the ability to manipulate the description to work with the abstraction of the system rather than the system itself.

In my formal verification tool **machine-check**, I use translation of simulable processor descriptions to verification analogues to support effective verification of machine-code programs. The high-level overview of machine-code verification via **machine-check** is visualised in Figure 3.2. The simulable processor description, written in Rust code, is translated to verification analogues, which are compiled together with algorithms that control the verification process. The machine code and specification are provided as arguments to the resultant executable. As such, the verification is faster and uses less memory than if the system was interpreted, yet allows for flexible, iterative development of the machine code

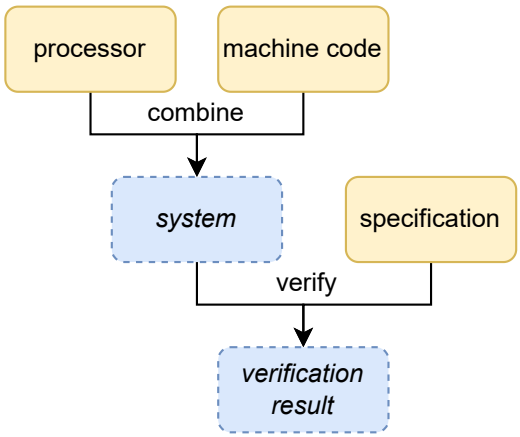

Figure 3.1: A high-level overview of formal verification of machine-code systems. The solid yellow cells represent inputs, while the dashed blue cells represent automated results. The processor and machine code are combined to form the system under verification. It is then determined if the specification holds or does not hold in the system. This figure is a specialisation of Figure 2.1 for machine-code systems.

and specification. The verifier executable can also be used on a dedicated server without installing the Rust language ecosystem. Currently, verification against Computation Tree Logic (CTL) [58] specifications is supported.

The verification result is a yes-no answer of whether the specification holds for the system. The final abstract state space, which serves as a witness to the CTL verification result, is printed out if requested via a command-line parameter. By design, **machine-check** is complete, producing the yes-no answer in finite time (although the needed computation time and memory may be impractical for some combinations of system and specification).

3.2 Processor Descriptions

The simulable descriptions in **machine-check** are designed to make describing processor-based systems simple. Even so, real architectures are still time-consuming to implement due to the size of the instruction set. For example, I have described the AVR ATmega328P microcontroller in approximately 3000 lines, with simple peripheral support only. Fortunately, once coded, the vast majority of the description can be reused for other similar microcontrollers with the same architecture.

A simulable description of a very simplified RISC microcontroller[1] is shown in Figure 3.3. The description is written in a subset of valid Rust code (which will be described later in Section 3.3), using specially provided **machine-check** types for simple transcription of behaviour from datasheets. The machine-code system described in Figure 3.3 can be immediately simulated in Rust by instantiating the `System` structure, with the machine

[1]The whole description is available at `https://docs.rs/crate/machine-check/0.3.0/source/examples/simple_risc.rs`.

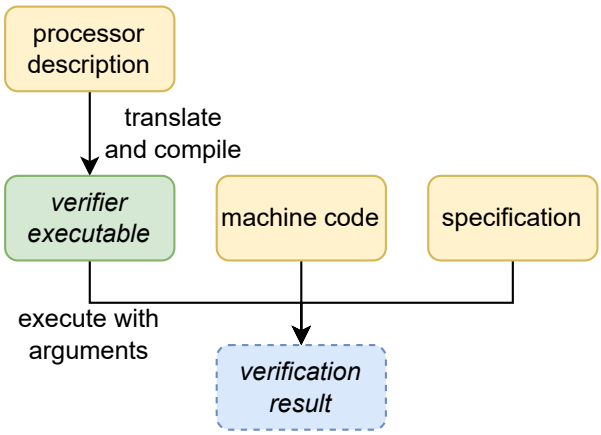

Figure 3.2: A high-level overview of **machine-check** machine-code system verification process. The processor description is translated to verification analogues, then compiled together with verification control algorithms to form a verifier executable for the given processor, visualised in a solid green cell. The verifier is executed with the machine code and specification given as arguments, performing formal verification as in Figure 3.1. The compilation step ensures a speed gain over interpretation. Additional guarantees beyond the processor descriptions are not considered in this chapter for simplicity.

code under simulation contained in field `progmem`, and using the `init` and `next` functions to generate successive states using a given sequence of inputs.

While simulation is performed with a single input sequence, all input sequences must be considered for formal verification. Since each successive state only depends on the previous state and the input, it would be possible to generate the reachable state space that completely captures the system behaviour. However, this is infeasible in practice due to the exponential explosion problem. As such, the `machine_description` macro provided by **machine-check**, applied to the description on line 1 of Figure 3.3, automatically generates verification analogues of the machine, allowing the use of advanced abstraction-refinement techniques. In case the description code does not conform to the subset of Rust processable by **machine-check** translation, a compilation error is issued so the problem can be fixed.

In the description in Figure 3.3, the input, state, and system structures are defined on lines 3–17. Power-of-two array sizes and bit-vector lengths are determined by generic constants, so e.g. the register array `reg` contains $2^2 = 4$ registers, each 8 bits wide. On lines 18-59, the finite-state-machine behaviour is described by the functions `init` and `next`. In Rust, if the last statement in a function is not terminated by a semicolon, it is the return value. As such, both functions return new states. The `init` function returns a state with the program counter set to zero and other fields uninitialized (having arbitrary values). The function `next` reads the current instruction from read-only program memory, increments the program counter, and decides on the action to perform depending on the instruction value. The `bitmask_switch` macro is designed to have the same format as conventional instruction set descriptions, filtering on zeros and ones and creating new variables for

```
#[machine_check::machine_description]
mod machine_module {
    pub struct Input {
        gpio_read: BitvectorArray<4, 8>,
        uninit_reg: BitvectorArray<2, 8>,
        uninit_data: BitvectorArray<8, 8>,
    }
    impl ::machine_check::Input for Input {}
    pub struct State {
        pc: Bitvector<7>,
        reg: BitvectorArray<2, 8>,
        data: BitvectorArray<8, 8>,
    }
    impl ::machine_check::State for State {}
    pub struct System {
        pub progmem: BitvectorArray<7, 12>,
    }
    impl ::machine_check::Machine for System {
        type Input = Input;
        type State = State;
        fn init(&self, input: &Input) -> State {
            State {
                pc: Bitvector::<7>::new(0),
                reg: Clone::clone(&input.uninit_reg),
                data: Clone::clone(&input.uninit_data),
            }
        }
        fn next(&self, state: &State, input: &Input)
            -> State {
            let instruction = self.progmem[state.pc];
            let mut pc = state.pc + Bitvector::<7>::new(1);
            let mut reg = Clone::clone(&state.reg);
            let mut data = Clone::clone(&state.data);
            ::machine_check::bitmask_switch!(instruction {
                "00dd_00--_aabb" => { // add
                    reg[d] = reg[a] + reg[b];
                }
                "00dd_01--_gggg" => { // read input
                    reg[d] = input.gpio_read[g];
                }
                "00rr_1kkk_kkkk" => { // jump if bit 0 is set
                    if reg[r] & Bitvector::<8>::new(1)
                        == Bitvector::<8>::new(1) {
                        pc = k;
                    };
                }
                "01dd_kkkk_kkkk" => { // load immediate
                    reg[d] = k;
                }
                "10dd_nnnn_nnnn" => { // load direct
                    reg[d] = data[n];
                }
                "11ss_nnnn_nnnn" => { // store direct
                    data[n] = reg[s];
                }
            });
            State { pc, reg, data }
        }
    }
}
```

Figure 3.3: Example description of a simplified Harvard-architecture RISC microcontroller as a finite-state machine. Less important code details are omitted for conciseness and readability.

letters.

Each system has specific parameters. For example, classic finite-state machines are constructed without any parameters, while machine-code systems must be provided with the machine code, with varying specifics such as instruction length and the number of instructions. As such, in **machine-check**, constructing the system is the responsibility of the description writer. For machine-code systems, the intended approach is to read the machine code from a file given as an argument to the verifier. However, for conciseness, in Figure 3.4, the example system from Figure 3.3 is constructed with a hard-coded toy machine-code program. The constructed system is handed off to the main routine of **machine-check** afterwards, which verifies a specification obtained from arguments to the executable. As such, properties of the system obtained by compiling the code from Figures 3.3 and 3.4 can be formally verified. For example:

- Register 1 is set to 1 before the main loop is reached: $\mathbf{AF}[\texttt{reg}[1] = 1 \wedge \texttt{PC} < 3]$.
- It is always possible to reach program location 9: $\mathbf{AG}[\mathbf{EF}[\texttt{PC} = 9]]$.
- Program locations above 9 are never reached: $\mathbf{AG}[\texttt{PC} \leq 9]$.

The properties are verified nearly instantaneously, below one second of computation time, and with insignificant memory usage. In comparison, naïve model-checking without abstraction would require constructing more than $2^{2^8} = 2^{256}$ states, which is completely infeasible.

3.3 Subset of the Rust Language Usable in Descriptions

Having shown an example of a simulable description in Figure 3.3, I will discuss what subset of the Rust language that can be used in descriptions in the current versions of **machine-check**. This only affects the simulation description code inside the macro `machine_description`, not the related code such as the main function in Figure 3.4.

Note 3.3.1. In the rest of this section, I will write the Rust language constructs underlined. Informal but authoritative information about the constructs is provided in the Rust Reference[2]. In the electronic version of this thesis, the underlined constructs link to the appropriate parts of the Rust Reference.

The basic principle is that the macro `machine_description` emits the code that was originally written, augmented with the verification analogues that are only usable by **machine-check** and opaque to the user. As such, the descriptions can be directly used outside **machine-check** as they would be without the macro, so it is possible to e.g. simulate the described systems by directly stepping the instance of the `Machine` with given inputs.

[2]The latest version of the Rust Reference is available at `https://doc.rust-lang.org/stable/reference/`. In **machine-check** 0.3.0, the current version at the time of writing of this thesis, the minimum supported Rust version is 1.75.0, with the corresponding version of the Rust Reference at `https://doc.rust-lang.org/1.75.0/reference/`.

```
1  fn main() {
2      let toy_program = [
3          // (0) set r0 to zero
4          Bitvector::new(0b0100_0000_0000),
5          // (1) set r1 to one
6          Bitvector::new(0b0101_0000_0001),
7          // (2) set r0 to zero
8          Bitvector::new(0b0110_0000_0000),
9          // --- main loop ---
10         // (3) store r0 content to data location 0
11         Bitvector::new(0b1100_0000_0000),
12         // (4) store r0 content to data location 1
13         Bitvector::new(0b1100_0000_0001),
14         // (5) read input location 0 to r3
15         Bitvector::new(0b0011_0100_0000),
16         // (6) jump to (3) if r3 bit 0 is set
17         Bitvector::new(0b0011_1000_0011),
18         // (7) increment r2
19         Bitvector::new(0b0010_0000_1001),
20         // (8) store r2 content to data location 1
21         Bitvector::new(0b1110_0000_0001),
22         // (9) jump to (3)
23         Bitvector::new(0b0001_1000_0011),
24     ];
25     let mut progmem = BitvectorArray::new_filled(
26         Bitvector::new(0));
27     for (index, instruction) in toy_program
28         .into_iter().enumerate() {
29         progmem[Bitvector::new(index as u64)] = instruction;
30     }
31     let system = machine_module::System { progmem };
32     machine_check::run(system);
33 }
```

Figure 3.4: Example of code for verification of a machine-code system based on the simplified RISC processor from Figure 3.3. On lines 1–25, the first ten instructions are hard-coded, and on lines 26–31, they are assigned into the program memory, pre-filled with zeros. The `System` structure is instantiated on line 32, combining the processor description with the provided program memory, and the verification is run with the provided system. Finally, the system is handed off to **machine-check** on line 33, which verifies properties determined by command-line arguments.
Considering data locations 0 and 1 to be memory-mapped peripherals (e.g. general-purpose outputs), the output behaviour of the program is that the locations are set to zero on initialisation, after which the data location 1 is varied between zero and the content of register 2, which is incremented each time the bit 0 of input location 0 is read as set.

The published version of the paper [A.2] contains slightly wrong comments to instructions (3) and (4) and it is not considered in the figure caption that the data location 1 is periodically set to 0 in the main loop. The main text is not affected.

The only exception to the principle is when the macro is not able to translate to the verification analogues for some reason, in which case it emits an error. In that case, care is taken so that the error is descriptive and localised to the problematic code span so that it can be fixed. It is also possible that the generated code will cause compilation to fail at a later stage. Errors have no impact on soundness as verification cannot proceed if they are emitted.

The macro `machine_description` must be applied to a module introduced by the Rust keyword **`mod`** that forms a separate lexical scope and contains items. I will now describe the basic supported items and the constructs inside them non-exhaustively. Since **machine-check** is not yet stable, the details may still change.

Use declarations. Since the translation occurs without access to the outer scope of the macro, it is necessary to either qualify each item from outside of the module with a full path, such as `::machine_check::Machine`, referring to item `Machine` provided by **machine-check**, or add a use declaration as

```
#[machine_check::machine_description]
mod machine_module {
    ...
    use ::machine_check::Machine;
    ...
}
```

After that, it is then possible to only refer to `Machine` in the scope.

Note 3.3.2. In Figure 3.3 and Figure 3.4, the use declarations that are needed for types `::machine_check::Bitvector` and `::machine_check::BitvectorArray` were skipped for conciseness and readability.

Structs. In Rust, data types can be combined in a struct type. The struct is typically defined by a keyword **struct** followed by named field declarations in braces. In the `machine_description` macro, the permitted field types are the other struct types defined inside the macro in addition to the four types provided by `machine-check`, which are

- `Unsigned`, an unsigned integer type with finite bit-width,
- `Signed`, a two's complement signed integer type with finite bit-width,
- `Bitvector`, a type with finite bit-width where signedness is not specified (and only operations where signedness does not matter are supported),
- `BitvectorArray`, a finite power-of-2 array of `Bitvector` elements that is indexable by `Bitvector` or `Unsigned` of the appropriate bit-width.

Implementations. The defined structs can be provided with implementations, which define items directly related to the struct, especially functions. Specially, a struct can implement a trait, which describes an interface usable by other code without dependence on the actual type. This is the key to describing a finite-state machine that can be verified by **machine-check**. In Figure 3.3, the necessary implementations of the traits for the finite-state machine are seen:

- `::machine_check::Input` marks the structure as usable as a machine input.
- `::machine_check::State` marks the structure as usable as a machine state.
- `::machine_check::Machine` describes the behaviour of the finite-state machine via the `init` and `next` functions. The associated types `Input` and `State` are set to the locally defined struct types `Input` and `State`, deciding the appropriate signatures of the `init` and `next` functions. In addition to the instances of the input and state structures, these functions can access the instance of the implementing type, which provides the system parameters.

The functions inside the implementations can have only the types permitted by the macro in their signatures. They contain a block expression that determines their behaviour. The block expression is introduced by curly braces and contains a sequence of statements that can be followed by an expression determining the result value. Each statement is separated with a semicolon, and three kinds of statements are supported in the `machine_description` macro:

- Let statements that introduce an optionally-initialised variable, such as the statement `let a = Bitvector::<8>::new(255);`.
- Expression statements that compute an expression, such as the assignment expression `reg[d] = reg[a] + reg[b];`, and discard its return value.
- Macro invocation statements that execute a macro in statement position, such as `::machine_check::bitmask_switch!(...);`. The supported macros are the bitmask switch macro provided by the `machine-check` library package and standard library macros **`panic!`**, **`unimplemented!`**, and **`todo!`**, which allow the program to terminate due to some unexpected cause (e.g. a situation that can only occur due to a previous bug). When verified by **machine-check**, the inherent lack of panics can either be verified on its own or before verification of another property (in which case the verification returns an error if the inherent lack of panics is violated).

Out of the many expression types in Rust, only some are supported in the macro:

- Literal expressions, e.g. `255` or the string literal `"00rr_1kkk_kkkk"`.
- Path expressions, e.g. `k` or `::machine_check::bitmask_switch`, which denote a local variable or an item.
- Block expressions, e.g. `{}`.
- Operator expressions with a supported operator. Standard binary arithmetic, logical, bit-shift, and comparison operators `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, `<=` are supported[3], as well as unary arithmetic negation (`-`) and bitwise NOT (`!`). Special

[3]Division and remainder are currently not available, as I have not yet decided on the behaviour for division by zero and signed division overflow, but they are technically supported by the translation.

supported operators are assignment ($pc = k$) and reference ($\&k$), which takes the reference of the variable instead of the variable itself, useful especially for calling functions that should not take ownership of the variable, only access its value.

- Parenthesised expressions, to determine the precedence order of expressions.
- Array indexing expressions, e.g. `reg[a]`.
- Struct expressions that instantiate a struct given the field names and values, supporting a shorthand when the field names are the same as the corresponding variable names, e.g. `State { pc, reg, data }`.
- Call expressions that call a function, e.g. `Bitvector::<8>::new(1)`.
- Field access expressions that access a struct field, e.g. `state.pc`.
- If expressions that branch the execution conditionally, such as the if expression `if a < b { c = a; } else { c = b; }`.

The currently supported subset of Rust does not contain loop expressions, so it is impossible to inadvertently preclude verification due to an infinite loop. It is possible to introduce infinite recursion, which will typically result in stack overflow during verification, although it is also possible that the computation will continue indefinitely due to removal of recursion through optimisation. As avoiding infinite recursion in system descriptions is not very problematic, in the current version of **machine-check**, it is assumed that it is avoided. In the future, loop expressions may be added and checking for the absence of infinite loops and recursion may be introduced, so that the expressiveness is enhanced while assuring finite computation time for the `init` and `next` functions.

3.4 Further Notes

This chapter was a light introduction to the simulable descriptions of processors from the view of a description writer. The machine-code system use-case is ideal for the translation technique, as the processor description can be compiled only once per device, after which many programs and specifications can be verified using the resulting executable. A source-code or hardware system description can also be produced by translation from the original formal language to the subset of Rust permitted for descriptions, but these approaches require recompilation each time the system is changed.

In Chapter 6, the implementation of the translation in the macro `machine_description` that produces the verification analogues will be described in more detail, once the interacting techniques from Chapters 4 and 5 are also introduced.

Chapter 4

Input-based Three-valued Abstraction Refinement Framework

The conventional methodology for abstraction refinement, Counterexample-guided Abstraction Refinement (CEGAR) [19], is not strong enough to verify all properties of the CTL* logic. The shortcoming is resolved by the Three-Valued Abstraction Refinement (TVAR) methodology. However, despite the ubiquity of CEGAR-based verification tools, there were no publicly available TVAR-based formal verification tools. I suspect that this is due to the previously introduced abstraction frameworks based on TVAR [59, 60, 61, 62] being too complex and unwieldy for intuitive understanding and implementation.

In this chapter, I describe a novel abstraction framework based on Three-Valued Abstraction Refinement (TVAR) that I devised during my doctoral study and formally prove that it can be used for model checking with abstraction refinement. It performs refinement on system inputs rather than system states as previous TVAR-based abstraction frameworks have done, resulting in simpler formalisms and implementation. The framework retains the advantages of TVAR while being simpler than the previous frameworks, and it is implemented in my free and open-source verification tool **machine-check**.

Note 4.0.1. This chapter is based on the contents of an available preprint [A.3], reworked for inclusion in this thesis.

4.1 The Need for a New Three-valued Abstraction Refinement Framework

CEGAR is limited to a specific set of temporal properties, which notably includes Linear Time Logic (LTL) properties but not all Computation Tree Logic (CTL) properties. Notably, *recovery properties*, which state that the system under verification can always return to a well-known state using some sequence of inputs, cannot be verified using CEGAR. The recovery properties are crucial especially for hardware systems, so that they can be reset

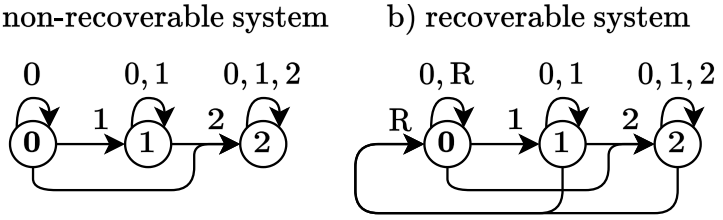

Figure 4.1: Systems that compute the maximum running value of the input, with initial value 0. In a), the possible input values are 0, 1, and 2. The recovery property **AG**(**EF**(`start`)) is violated. In b), a return-to-zero input value is added, and the property is fulfilled. As for similar properties without alternating quantifiers, **AG**(**AF**(`start`)) is violated in both systems and **EG**(**EF**(`start`)) is fulfilled in both.

without a hard power cycle, and they can also preclude software systems from "freezing". They can be expressed in CTL as **AG**(**EF**(`start`)) [18, p. 63].

As CEGAR-based tools are ubiquitous, especially in software verification [30], blind spots occur: a crucial property not verifiable by CEGAR may not be checked because there is no suitable verifier to check it. The property may not even be considered during design, since it does not fit into the LTL-centric view.

The blind-spot problem can be resolved by TVAR. However, the previous works on TVAR frameworks [59, 60, 61, 62] do not contain information about any implemented tools. Even after further research, I have not found any free and open-source verification tools that implement TVAR[1]. I have tried to implement the previous frameworks into my free and open-source model-checking tool **machine-check**, but was unable to due to the complicated formal structures used, doubts about some proofs, under-specification, and undesirable characteristics of the previous frameworks. This led me to devise a novel TVAR framework and implement an instance of it in **machine-check**. Using the implementation, I was able to verify recovery properties on systems similar to those in Figure 4.1 while preventing exponential explosion due to irrelevant inputs and computations, as will be discussed later in Section 4.6.

4.2 Previous Work on Three-valued Abstraction

In this section, I list previous work on three-valued abstraction and model checking relevant to this chapter, in roughly chronological order. Additional information about previous work on three-valued model checking can be found in [23, 62].

Three-valued logic. The typical underlying logic used for three-valued abstraction is Kleene's strong three-valued logic [64]. While the logic was introduced in the discussion of results of partial recursive functions, it can be used to represent uncertainty as well,

[1] I am aware of a non-free implementation of verification with three-valued abstraction refinement of strategic abilities in multi-agent systems [63].

in which case the three distinct values are typically named "false" (0), "true" (1), and "unknown" ($\perp$). Operations featuring only "false" and "true" follow classical logic.

Partial Kripke Structures (PKS). Verification on *partial state spaces* was introduced by Bruns and Godefroid [65], motivated by the idea of disregarding some system information to produce a smaller state space (i.e. model checking with abstraction). The introduced formalism of PKS enriches standard KS by making labellings of states possibly unknown.

Definition 4.2.1. A partial Kripke structure (PKS) is a tuple (S, S_0, R, L) where

- S is the set of states,
- $S_0 \subseteq S$ is the set of initial states,
- $R \subseteq S \times S$ is a transition relation,
- $L : S \times \mathbb{A} \to \{0, 1, \perp\}$ is a labelling function mapping each atomic proposition $a \in \mathbb{A}$ to 0 ($\neg a$ holds in state), 1 (a holds in state), or $\perp$ (it is unknown which one holds).

Note 4.2.2. Partial Kripke structures were introduced without initial states [65]. For correspondence with real systems, initial states are used. Using Definition 4.2.1, a Kripke structure (KS) is a PKS where the labelling function L is restricted to $S \times \mathbb{A} \to \{0, 1\}$.

Kripke Modal Transition Structures (KMTS). KMTS were introduced by Huth et al. [66], building on previous work on PKS and on Modal Transition Systems (MTS) [67]. The conventional form of KMTS differs from PKS only in the presence of two transition functions instead of one. The *must*-transitions underapproximate the system, while the *may*-transitions overapproximate it.

Definition 4.2.3. A Kripke Modal Transition Structure (KMTS) is defined as a tuple $(S, S_0, R^{\text{may}}, R^{\text{must}}, L)$ where

- S, S_0, and L follow Definition 4.2.1,
- $R^{\text{may}} \subseteq S \times S$ is the set of transitions which may be present,
- $R^{\text{must}} \subseteq R^{\text{may}}$ is the set of transitions which are definitely present.

Computation of properties. Interpretation of µ-calculus properties on PKS and KMTS can be computed by converting the structure to two KS, applying standard model-checking algorithms, and combining the results [68, 69]. As a consequence, the complexity is the same as of standard model checking.

Expressivity. While PKS can only convey abstraction by states with unknown labellings, KMTS also can convey it by transitions in $R^{\text{may}} \setminus R^{\text{must}}$, which have unknown presence. It has been shown that PKS and KMTS are equally expressive [70], i.e. the KMTS formalism is unnecessarily redundant.

4.2.1 Previous Frameworks and Their Problems

I consider three characteristics necessary for a TVAR framework[2] to be well-usable:

- Refinement monotonicity, i.e. a verifiable property never becomes unverifiable after refinement.
- Ability to use standard, well-researched model-checking algorithms.
- No risk of unnecessary exponential explosion introduced by the formalisms.

All of these characteristics are fulfilled by standard CEGAR. I will examine the three previously introduced TVAR frameworks and show that each one violates some characteristic.

Program-verification TVAR (Godefroid & Jagadeesan). A framework for TVAR was proposed previously [69], combining the concept of abstraction refinement with avoidance of unnecessary refinement for formulas such as $\neg p \vee p$. As this avoidance leads to high time complexity, I will not discuss it further. The abstract state space is formalised by KMTS and constructed within each iteration of the refinement loop. The may- and must-transitions are constructed exactly if they are permissible according to the concrete state space. The refinement strategy is not discussed in detail, and is not monotone, as it uses KMTS and refinement on states, so must-transitions can be lost [71].

Game-based TVAR (Shoham & Grumberg). Introduced for CTL specifications [59] and later extended to μ-calculus [60]. KTMS are used, and a game is played by a *prover* and a *refuter*. If neither wins, the abstraction is refined and KMTS is recomputed. Later, it was proven that two standard model-checking games can be used instead of one three-valued game [61]. Unfortunately, the need for specially devised algorithms based on game theory makes the framework difficult to use.

BDD-based TVAR (Shoham & Grumberg). Introduced for CTL specifications [71] and later generalised to alternation-free μ-calculus [72]. Abstract states are stored using Binary Decision Diagrams (BDDs), and BDD operations are used similarly to standard BDD-based symbolic model checking. To achieve monotonicity, Generalized KMTS (GKMTS) are used instead of KMTS, at the expense of possible exponential explosion introduced by hyper-transitions in the GKMTS.

4.3 State-based and Input-based Refinement

My framework is based on the concept of performing *input-based refinement* instead of *state-based refinement*, fulfilling all characteristics from Subsection 4.2.1 and allowing use of the simple Partial Kripke Structures. In this section, I will explain the concept using an example of performing one refinement.

I use the classic existential abstraction of the state space, i.e. there is a *concretization function* $\gamma : \hat{S} \to 2^S \setminus \{\emptyset\}$ that maps each abstract state $\hat{s} \in \hat{S}$ to a non-empty set

[2] I use the term *framework* in the sense of abstraction frameworks in [23]. From the user's point of view, a three-valued abstraction framework maps each combination of system and property to 0, 1, or $\perp$.

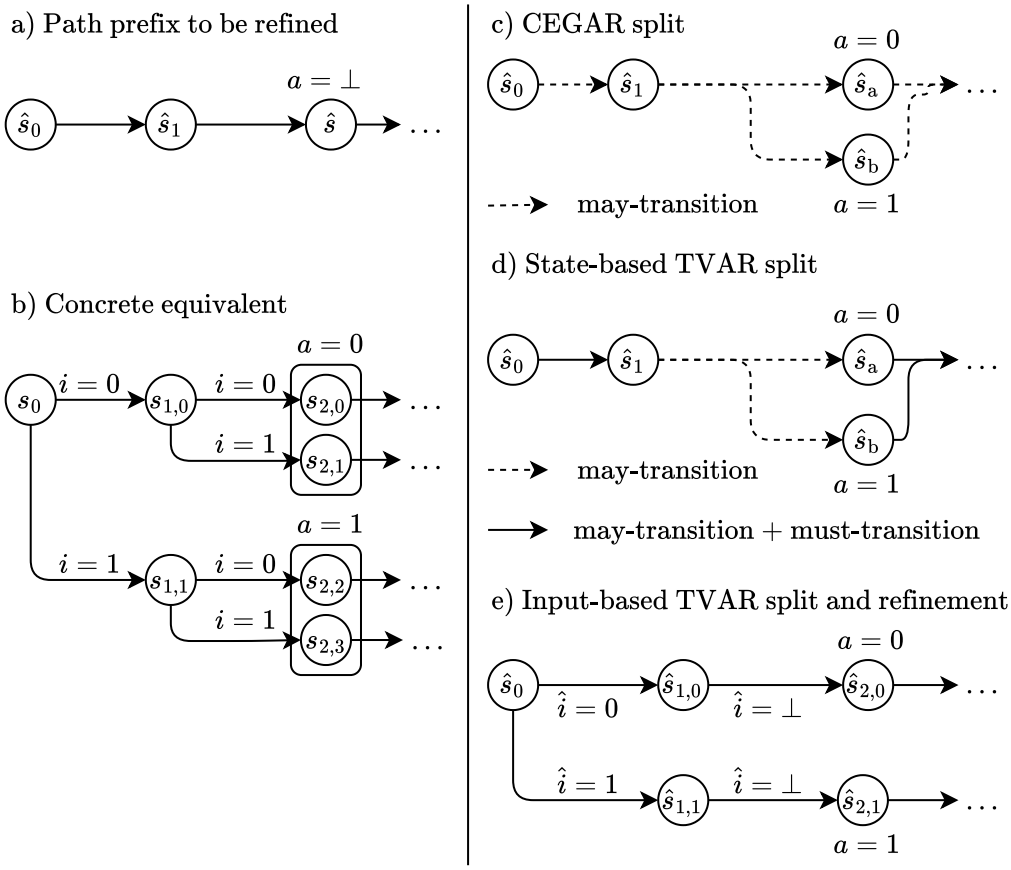

Figure 4.2: Strategies for splitting of path prefix where we are interested in property a in state $\hat{s}$. For CEGAR and state-based TVAR, the actual refinement is performed by eliminating spurious paths after splitting.

of concrete states it represents. Let us consider abstraction refinement in a situation where we generated a state space expressible within PKS and three-valued model-checking returned $\perp$. We determined the reason for the unknown result to be an abstract path prefix shown in Figure 4.2(a). In $\hat{s}$, the last state of the prefix, the atomic proposition a is unknown even though it affects the model-checking result, so the state $\hat{s}$ should be split. The concrete equivalent of the path prefix in Figure 4.2(b) shows that the value of the proposition depends only on the value of input i in the transition from s_0.

CEGAR can be formalised using KMTS with no must-transitions [23, p. 412]. The state $\hat{s}$ can be safely split to $\hat{s}_a$ and $\hat{s}_b$ such that $\gamma(\hat{s}) \subseteq \gamma(\hat{s}_\mathrm{a}) \cup \gamma(\hat{s}_\mathrm{b})$ as in Figure 4.2(c), as there is no requirement for $\hat{s}_a$ and $\hat{s}_b$ to actually correspond to system states since they are only reachable by may-transitions. However, this splitting does not actually refine the system (decreasing the number of properties where the model-checking result is $\perp$). The actual refinement is done when CEGAR eliminates paths that violate the specification but are spurious.

State-based refinement used in previous TVAR frameworks would first split $\hat{s}$ into

two abstract states $\hat{s}_\mathrm{b}$, $\hat{s}_\mathrm{b}$ such that $\gamma(\hat{s}) \subseteq \gamma(\hat{s}_\mathrm{a}) \cup \gamma(\hat{s}_\mathrm{b})$, similarly to CEGAR. However, the must-transitions from $\hat{s}_1$ must be eliminated, since it is unclear if either of the transitions actually exists, as seen in Figure 4.2(d). This means that some properties are no longer provable in the new system (e.g. that $\mathbf{AF}\phi$ holds given that ϕ holds in $\hat{s}$ but not $\hat{s}_0$ and $\hat{s}_1$), so the refinement is not monotonic. This scenario was the reason for introducing GKMTS in [71]. After splitting, it is necessary to test for spuriousness after splitting to actually refine the system, similarly to CEGAR.

Input-based refinement. By deducing which inputs could have caused the unknown result, we can discover that the value of a in $\hat{s}$ can depend on the value of $\hat{i}$ in the step from $\hat{s}_0$, but not on the value of $\hat{i}$ in the step from $\hat{s}_1$. By abstracting the inputs as well as states, we can refine the inputs that are suspected of causing the unknown result, as visualised in Figure 4.2(e).

Unlike state-based refinement where we are not sure if the split states are actually present, we know that each concrete input must be present since we are considering all possible inputs during verification. Therefore, each abstract input can be split (or combined) at will, provided that all concrete inputs are preserved. As there is no need to use may-transitions, PKS can be used for abstract state space formalisation. As seen in Figure 4.2(e), the amount of newly generated states can be higher than for state-based splitting, but there is no further need to test for spuriousness.

4.4 Input-based Abstraction Framework

To formalise refinement on inputs, I first define *generating automata* that include inputs in their formalisation and generate Partial Kripke Structures, so that standard model-checking algorithms can be used on the generated PKS. I then describe the main refinement loop of the proposed abstraction framework.

4.4.1 Generating Automata

I chose the formalism for the generating automata so it is largely consistent with the generated PKS. Generating automata are cleanly separated from the model-checking algorithms by PKS, and are completely distinct from nondeterministic automata used for actual model checking, e.g. Büchi automata for LTL checking [73] or tree automata for three-valued checking [74].

Definition 4.4.1. A generating automaton (GA) is a tuple $G = (S, s_0, I, q, f, L)$ where

- S is the set of automaton states,
- $s_0 \in S$ is the initial state,
- I is the set of all possible step inputs,
- $q : S \to 2^I \setminus \{\emptyset\}$ is the input qualification function,

- $f : S \times I \to S$ is the step function, mapping the combination of the current state and step input to the next state,
- $L : S \times \mathbb{A} \to \{0, 1, \bot\}$ is a labelling function.

It generates the PKS $K = (S, \{s_0\}, R, L)$ as

$$R = \{(s, f(s, i)) \mid s \in S \wedge i \in q(s)\}. \tag{4.1}$$

Generating automata are similar to classic deterministic automata, but include an input qualification function q that selects the inputs that will be used to generate transitions from each state. This is not especially useful for automata representing concrete systems but will allow deciding which abstract inputs will be used in each state, allowing input-based refinement.

Example 4.4.2. Abstracting over a system with input set $\{0, 1\}$, we can use $\{0, 1, \bot\}$ as the abstract input set. In each state, we can decide whether to split the input, either setting $q(s) = \{0, 1\}$ so that we can verify properties based on the value or setting $q(s) = \{\bot\}$ to mitigate state space explosion. This was previously done in Figure 4.2(e).

Finite GA with $q = \{(s, I) \mid s \in S\}$ are a special case of Moore machines with output alphabet $\mathbb{A} \to \{0, 1, \bot\}$. For simplicity, a single initial state is used. Multiple initial states can be supported by treating s_0 as a dummy state whose direct successors are the actual initial states. As it is customary to consider the property ϕ to hold exactly when it holds in all initial states, the property $\mathbf{AX}[\phi]$ on the generated PKS with the dummy initial state s_0 can be model-checked instead of the property ϕ on the PKS with multiple initial states.

4.4.2 High-level View of the Input-based Framework

To visualise the algorithmic actions performed during verification in the input-based framework, I will start with a simple setup using a generating automaton but no abstraction yet, shown in Figure 4.3. The original system under verification may not be described in the exact form we need (e.g. is expressed as a program instead of a finite-state machine or does not contain labellings), so it is first translated to the *concrete generating automaton* (CGA) $G = (S, s_0, I, \{(s, I) \mid s \in S\}, f, L)$ that represents it. The CGA generates a PKS that can be model-checked by classic algorithms. There may be unknown labellings in L, but if there are none as the system is fully specified, the generated PKS is a KS.

Example 4.4.3. Let us consider the system from Figure 4.1a) that always remembers the maximum value of its input, ranging from 0 to 2, and starts in 0. We will also use a single labelling a expressing that the state is equal to 0. The resulting CGA $G_{\text{ex}} = (S_{\text{ex}}, s_{0,\text{ex}}, I_{\text{ex}}, \{(s, I_{\text{ex}}) | s \in S_{\text{ex}}\}, f_{\text{ex}}, L_{\text{ex}})$ can be expressed as

$$\begin{aligned} &S_{\text{ex}} \stackrel{\text{def}}{=} \{0, 1, 2\},\ s_{0,\text{ex}} \stackrel{\text{def}}{=} 0,\ I_{\text{ex}} \stackrel{\text{def}}{=} S_{\text{ex}}, \\ &f_{\text{ex}}(s, i) \stackrel{\text{def}}{=} \max(s, i),\ L_{\text{ex}}(s, a) \stackrel{\text{def}}{=} 1 - \operatorname{sign} s. \end{aligned} \tag{4.2}$$

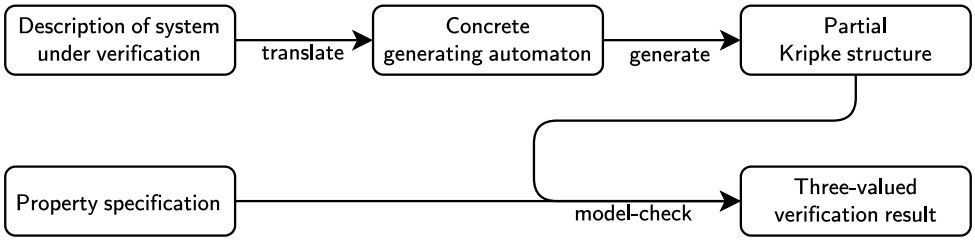

Figure 4.3: Block overview of concrete verification using concrete generating automaton. Blocks represent structures. Arrows represent algorithm execution, and the algorithms are assumed to terminate in finite time given finite structures.

Next, I define the input concretization function $\zeta : \hat{I} \to 2^I \setminus \{\emptyset\}$ in addition to the state concretization function $\gamma : \hat{S} \to 2^S \setminus \{\emptyset\}$. I then introduce the *abstract generating automaton* (AGA) $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$. Specially, I set $\gamma(\hat{s}_0) = \{s_0\}$. The final block overview of the input-based framework is shown in Figure 4.4. The CGA no longer generates a PKS directly but is abstracted to an AGA, which is then continually refined within a three-valued refinement loop.

Example 4.4.4. The choice of abstraction domain determines the AGA. Continuing with the system from Example 4.4.3, we could choose e.g. interval abstraction or bit-vector abstraction [A.1]. Using interval abstraction, we could construct an AGA $\hat{G} = (\hat{S}_{\text{ex}}, \hat{s}_{0,\text{ex}}, \hat{I}_{\text{ex}}, \hat{q}_{\text{ex}}, \hat{f}_{\text{ex}}, \hat{L}_{\text{ex}})$ with concretization functions $\gamma_{\text{ex}}, \zeta_{\text{ex}}$ as

$$\begin{aligned}
&\gamma_{\text{ex}}([\hat{s}_{\text{lower}}, \hat{s}_{\text{upper}}]) \stackrel{\text{def}}{=} \{s \mid \hat{s}_{\text{lower}} \leq s \leq \hat{s}_{\text{upper}}\},\ \zeta_{\text{ex}} \stackrel{\text{def}}{=} \gamma_{\text{ex}},\\
&\hat{S}_{\text{ex}} \stackrel{\text{def}}{=} \{[0,0], [1,1], [2,2], [0,1], [1,2], [0,2]\},\\
&\hat{f}_{\text{ex}}(\hat{s}, \hat{i}) \stackrel{\text{def}}{=} [\max(\hat{s}_{\text{lower}}, \hat{i}_{\text{lower}}), \max(\hat{s}_{\text{upper}}, \hat{i}_{\text{upper}})],\\
&\hat{q}_{\text{ex}} \stackrel{\text{def}}{=} \{[0,0], [1,2]\},\\
&\hat{L}_{\text{ex}}(\hat{s}, a) \stackrel{\text{def}}{=} \begin{cases} b & b \in \{0,1\},\ \forall s \in \gamma_S(\hat{s})\ .\ L_{\text{ex}}(s,a) = b,\\ \bot & \text{otherwise.} \end{cases}
\end{aligned} \tag{4.3}$$

$\hat{G}_{\text{ex}}$ generates a PKS with fewer reachable states and transitions than the PKS generated by G_{ex}, but it is still possible to verify that $\mathbf{AG}[\mathbf{EF}[a]]$ does not hold.

The framework allows using standard model-checking algorithms and there is no risk of unnecessary exponential explosion due to the formalisms, which are two of the three characteristics of a well-usable TVAR framework from Subsection 4.2.1. Of course, functional characteristics of the input-based framework are also important: whether we can guarantee *soundness* (that it never returns a wrong result), *monotonicity* (that properties that are known will not become unknown after refinement), and *completeness* (that it always returns). While soundness is necessary for an abstraction framework, monotonicity is the third characteristic of a well-usable TVAR framework, and completeness further increases the usefulness of the framework.

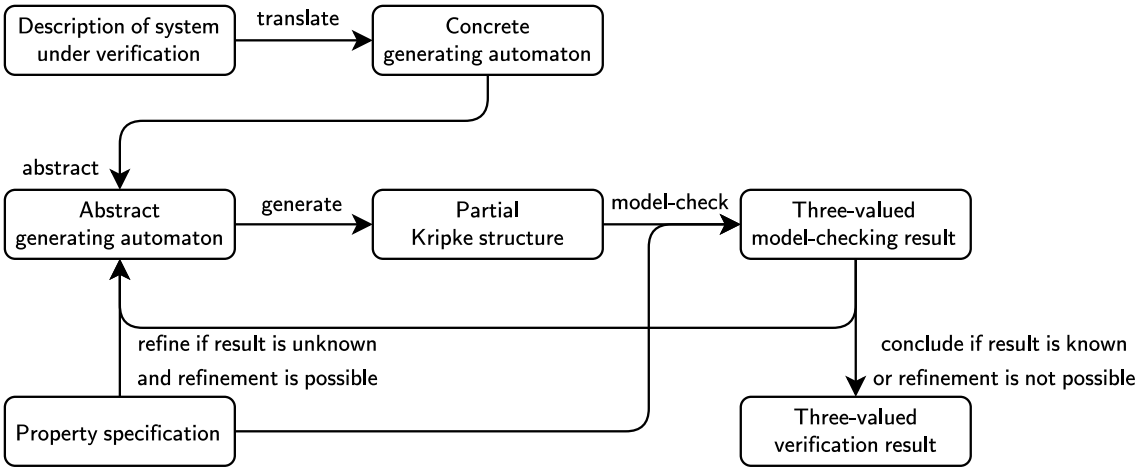

Figure 4.4: Block overview of the proposed input-based Three-valued Abstraction Refinement framework, which extends Figure 4.3. In addition to describing the formalism, this high-level overview can be directly used as a blueprint for verification tool development. Formally, the abstract generating automaton is replaced by a new one after each iteration of the refinement loop.

4.5 Soundness, Monotonicity, and Completeness

In this section, I will prove that the proposed input-based framework is sound, monotone, and complete. Of course, whether these characteristics hold in an instance of the framework depends on the chosen abstract generating automata. I will formulate the theorems that underpin soundness, monotonicity, and completeness first, explaining the requirements informally to provide an intuitive understanding, before proving the theorems at the end of the section. For conciseness, I write that an abstract state $\hat{s}$ or input $\hat{i}$ *covers* a concrete $s \in S$ or $i \in I$ when $s \in \gamma(\hat{s})$ or $i \in \zeta(\hat{i})$, respectively, and that it *covers* another abstract state $\hat{s}^* \in \hat{S}$ or input $\hat{i}^* \in \hat{I}$ when $\gamma(\hat{s}^*) \subseteq \gamma(\hat{s})$ or $\zeta(\hat{i}^*) \subseteq \zeta(\hat{i})$, respectively.

Definition 4.5.1. Soundness of Kripke structure K^1 is preserved in Kripke structure K^2 if, for every property ϕ of μ-calculus over the set of atomic propositions $\mathbb{A}$, denoting its 3-valued interpretation on PKS K by $[\![\phi]\!](K)$, it holds that

$$[\![\phi]\!](K^2) \neq \bot \Rightarrow [\![\phi]\!](K^1) = [\![\phi]\!](K^2). \tag{4.4}$$

The soundness of generating automata follows their generated Kripke structures.

Informally, all interpretations of properties that are not unknown in the "coarse" K^2 are the same in the "fine" K^1. However, an interpretation that is not unknown in the fine K^1 can still be unknown in the coarse K^2.

Theorem 4.5.2 (Soundness)**.** Soundness of a concrete generating automaton G is preserved in an abstract generating automaton $\hat{G}$ if $\hat{G}$ fulfils

$$\begin{aligned}
&\forall \hat{s} \in \hat{S} \,.\, \forall s \in \gamma(\hat{s}) \,.\, \forall a \in \mathcal{A} \,.\, (\hat{L}(\hat{s}, a) \neq \bot \Rightarrow \hat{L}(\hat{s}, a) = L(s, a)),\\
&\forall (\hat{s}, i) \in \hat{S} \times I \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, i \in \zeta(\hat{i}),\\
&\forall (\hat{s}, \hat{i}) \in \hat{S} \times \hat{I} \,.\, \forall (s, i) \in \gamma(\hat{s}) \times \zeta(\hat{i}) \,.\, f(s, i) \in \gamma(\hat{f}(\hat{s}, \hat{i})).
\end{aligned} \tag{4.5}$$

Informally, there are three requirements:

1. **Labelling soundness.** Each abstract state labelling must either correspond to the labelling of all concrete states it covers or be unknown.
2. **Full input coverage.** In every abstract state, each concrete input must be covered by some qualified abstract input.
3. **Step soundness.** Each result of the abstract step function must cover all results of the concrete step function where its arguments are covered by the abstract step function arguments.

Example 4.5.3. In Example 4.4.4, soundness of G_{ex} is preserved in $\hat{G}_{\text{ex}}$.

Definition 4.5.4. Abstract generating automaton $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ is monotone with respect to concretization if

$$\begin{aligned}
&\forall (\hat{s}_1, \hat{s}_2, a) \in \hat{S} \times \hat{S} \times \mathcal{A} \,.\\
&\qquad ((\gamma(\hat{s}_1) \subseteq \gamma(\hat{s}_2) \wedge \hat{L}(\hat{s}_2, a) \neq \bot) \Rightarrow \hat{L}(\hat{s}_2, a) = L(\hat{s}_1, a)),\\
&\forall (\hat{s}_1, \hat{s}_2, \hat{i}_1, \hat{i}_2) \in \hat{S} \times \hat{S} \times \hat{I} \times \hat{I} \,.\\
&\qquad ((\gamma(\hat{s}_1) \times \zeta(\hat{i}_1) \subseteq \gamma(\hat{s}_2) \times \zeta(\hat{i}_2)) \Rightarrow \gamma(\hat{f}(\hat{s}_1, \hat{i}_1)) \subseteq \gamma(\hat{f}(\hat{s}_2, \hat{i}_2)).
\end{aligned} \tag{4.6}$$

Informally, Equation 4.6 requires the labellings to have an inclination to being unknown monotonically with coverage, and for the coverage of the results of the step function to tend to increase monotonically with the coverage of its arguments.

Theorem 4.5.5 (Monotonicity)**.** Soundness of an abstract generating automaton $\hat{G}^1 = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}^1, \hat{f}^1, \hat{L})$ is preserved in an abstract generating automaton $\hat{G}^2 = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}^2, \hat{f}^2, \hat{L})$ if $\hat{G}^2$ is monotone with respect to concretization and

$$\begin{aligned}
&\forall \hat{s} \in \hat{S} \,.\, \forall \hat{i}^1 \in \hat{q}^1(\hat{s}) \,.\, \exists \hat{i}^2 \in \hat{q}^2(\hat{s}) \,.\, \zeta(\hat{i}^1) \subseteq \zeta(\hat{i}^2),\\
&\forall \hat{s} \in \hat{S} \,.\, \forall \hat{i}^2 \in \hat{q}^2(\hat{s}) \,.\, \exists \hat{i}^1 \in \hat{q}^1(\hat{s}) \,.\, \zeta(\hat{i}^1) \subseteq \zeta(\hat{i}^2),\\
&\forall (\hat{s}, \hat{i}) \in \hat{S} \times \hat{I} \,.\, \gamma(\hat{f}^1(\hat{s}, \hat{i})) \subseteq \gamma(\hat{f}^2(\hat{s}, \hat{i})).
\end{aligned} \tag{4.7}$$

Informally, refining from the coarse $\hat{G}^2$ to the fine $\hat{G}^1$, we do not lose properties provable in $\hat{G}^2$ by guaranteeing the soundness of $\hat{G}^1$ is preserved in $\hat{G}^2$. For that, we require that $\hat{G}^2$ is monotone w.r.t. concretization and also:

1. **Fine qualified inputs are not spurious.** Each fine qualified input is covered by at least one coarse qualified input.

2. **Coarse qualified inputs are not lost.** Each coarse qualified input covers at least one fine qualified input.

3. **Step function coverage.** The result of the fine step function is always covered by the result of the coarse step function.

The need for both quantifier combinations in the first two requirements in Equation 4.7 may be surprising. Their violations correspond to transition addition and removal, respectively, both of which can change verifiable properties.

Theorem 4.5.6 (Completeness)**.** Assume an AGA $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$, corresponding to $G = (S, s_0, I, \{s, I \mid s \in S\}, f, L)$, that fulfils Equation 4.5 and is monotone with respect to concretization. Its every finite μ-calculus property can be verified in finite time by abstraction refinement if $S, I, \hat{S}, \hat{I}$ are finite and

$$\forall s \in S \,.\, \exists \hat{s} \in \hat{S} \,.\, \gamma(\hat{s}) = \{s\}, \;\; \forall i \in I \,.\, \exists \hat{i} \in \hat{I} \,.\, \zeta(\hat{i}) = \{i\}. \tag{4.8}$$

As completeness can be achieved by monotonically refining to an equivalent to the CGA, it suffices for concrete states and inputs to have abstract equivalents.

The framework. Considering the whole framework in Figure 4.4, translation is an implementation problem, PKS generation is trivial, and algorithms for model-checking μ-calculus or its fragments on PKS are well-known [68, 62], so only the relationships between CGA and AGA when abstracting and refining have to be considered, as follows:

- **Soundness**. Each AGA preserves soundness of the CGA as per Theorem 4.5.2.

- **Monotonicity**. Each AGA preserves soundness of the AGA it is refined to as per Theorem 4.5.5.

- **Completeness**. Each AGA follows Theorem 4.5.6, and an AGA that produces the same results as the CGA is used in finite time (it exists as per Subsection 4.5.4).

It remains to prove the theorems.

4.5.1 Soundness Preservation through Modal Simulation

To prove the theorems efficiently, without even directly defining μ-calculus, I use the previously proven relationship of soundness preservation and modal simulation [23, p. 408-410], which I simplified for use with PKS.

Definition 4.5.7. Let $K^1 = (S^1, S_0^1, R^1, L^1)$ and $K^2 = (S^2, S_0^2, R^2, L^2)$ be PKS over the set of atomic propositions $\mathcal{A}$. Let $H \subseteq S^1 \times S^2$ be a relation. H is a modal simulation from K^1 to K^2 if and only if all of the following hold,

$$\begin{aligned}
&\forall (s^1, s^2) \in H \,.\, \forall a \in \mathcal{A} \,.\, (L^2(s^2, a) \neq \bot \Rightarrow L^1(s^1, a) = L^2(s^2, a)),\\
&\forall (s^1, s^2) \in H \,.\, \forall s'^1 \in S^1 \,.\, (R^1(s^1, s'^1) \Rightarrow \exists s'^2 \in S^2 \,.\, (R^2(s^2, s'^2) \wedge H(s'^1, s'^2))),\\
&\forall (s^1, s^2) \in H \,.\, \forall s'^2 \in S^2 \,.\, (R^2(s^2, s'^2) \Rightarrow \exists s'^1 \in S^1 \,.\, (R^1(s^1, s'^1) \wedge H(s'^1, s'^2))).
\end{aligned} \tag{4.9}$$

Informally, H relates the states of K^1 and K^2 so that the states in the fine K^1 preserve all known labellings of their related states from the coarse K^2, and transitions are respected: for every pair of related states in H, each transition from one element of the pair must correspond to at least one transition from the other element with the endpoints related by H.

Definition 4.5.8. K^1 is modal-simulated by K^2, denoted $K^1 \preceq K^2$, if H is a modal simulation from K^1 to K^2 and furthermore,

$$\forall s_0^1 \in S_0^1 \,.\, \exists s_0^2 \in S_0^2 \,.\, H(s_0^1, s_0^2). \tag{4.10}$$

Property 4.5.9. Soundness of K^1 is preserved in K^2 if $K_1 \preceq K_2$ [23, p. 410].

Lemma 4.5.10. Soundness of a generating automaton $G^1 = (S^1, s_0^1, q^1, f^1, L^1)$ is preserved in a generating automaton $G^2 = (S^2, s_0^2, q^2, f^2, L^2)$ if there exists a relation $H \subseteq S^1 \times S^2$ where

$$\begin{aligned}
&(s_0^1, s_0^2) \in H,\\
&\forall (s^1, s^2) \in H \,.\, \forall a \in \mathcal{A} \,.\, (L^2(s^2, a) \neq \bot \Rightarrow L^1(s^1, a) = L^2(s^2, a)),\\
&\forall (s^1, s^2) \in H \,.\, \forall i^1 \in q^1(s^1) \,.\, \exists i^2 \in q^2(s^2) \,.\, H(f^1(s^1, i^1), f^2(s^2, i^2)),\\
&\forall (s^1, s^2) \in H \,.\, \forall i^2 \in q^2(s^2) \,.\, \exists i^1 \in q^1(s^1) \,.\, H(f^1(s^1, i^1), f^2(s^2, i^2)).
\end{aligned} \tag{4.11}$$

Proof. G^1 generates the PKS $K^1 = (S^1, S_0^1, R^1, L^1)$ and G^2 generates the PKS $K^2 = (S^2, S_0^2, R^2, L^2)$. We will prove that assuming Equation 4.11, soundness of K^1 is preserved in K^2. As $(s_0^1, s_0^2) \in H$, Equation 4.10 holds, and it remains to prove that H is a modal simulation from K^1 to K^2. The first part of Equation 4.9 directly follows from Equation 4.11, and we can rewrite the other two parts, respectively inserting the definition of R^1 and R^2 from Equation 4.1,

$$\begin{aligned}
&\forall (s^1, s^2) \in H \,.\, \forall s'^1 \in S^1 \,.\\
&\quad ((\exists i^1 \in q^1(s^1) \,.\, s'^1 = f^1(s^1, i^1)) \Rightarrow \exists s'^2 \in S^2 \,.\, (R^2(s^2, s'^2) \wedge H(s'^1, s'^2))),\\
&\forall (s^1, s^2) \in H \,.\, \forall s'^2 \in S^2 \,.\\
&\quad ((\exists i^2 \in q^2(s^2) \,.\, s'^2 = f^2(s^2, i^2)) \Rightarrow \exists s'^1 \in S^1 \,.\, (R^1(s^1, s'^1) \wedge H(s'^1, s'^2))).
\end{aligned} \tag{4.12}$$

We move the i^1, i^2 quantifiers out of the implication, negating them as we are moving out of antecedent. The s'^1 and s'^2 variables, respectively, are now uniquely defined in the

antecedent, so we replace their occurrences by the definition and eliminate the quantifiers, obtaining

$$\begin{aligned}&\forall(s^1,s^2)\in H\,.\,\forall i^1\in q^1(s^1)\,.\,\exists s'^2\in S^2\,.\,(R^2(s^2,s'^2)\wedge H(f^1(s^1,i^1),s'^2)),\\&\forall(s^1,s^2)\in H\,.\,\forall i^2\in q^2(s^2)\,.\,\exists s'^1\in S^1\,.\,(R^1(s^1,s'^1)\wedge H(s'^1,f^2(s^2,i^2))).\end{aligned}\tag{4.13}$$

We then respectively insert the definition of R^2 and R^1, pull the i^2, i^1 quantifiers outside, and eliminate the s'^2, s'^1 variables, obtaining

$$\begin{aligned}&\forall(s^1,s^2)\in H\,.\,\forall i^1\in q^1(s^1)\,.\,\exists i^2\in q^2(s^2)\,.\,H(f^1(s^1,i^1),f^2(s^2,i^2)),\\&\forall(s^1,s^2)\in H\,.\,\forall i^2\in q^2(s^2)\,.\,\exists i^1\in q^1(s^1)\,.\,H(f^1(s^1,i^1),f^2(s^2,i^2)),\end{aligned}\tag{4.14}$$

which corresponds to the last two parts of the assumed Equation 4.11. □

4.5.2 Proof of Soundness

Proof (Theorem 4.5.2). To prove that soundness of G is preserved in $\hat{G}$, we define

$$H=\{(s,\hat{s})\in S\times\hat{S}\mid s\in\gamma(\hat{s})\}.\tag{4.15}$$

We assume Equation 4.5 and will prove that Equation 4.11 holds. As $\gamma(\hat{s}_0)=\{s_0\}$, the first part of Equation 4.11 holds. Its second part holds due to the first part of Equation 4.5. We rewrite the last two parts of Equation 4.11 as

$$\begin{aligned}&\forall\hat{s}\in\hat{S}\,.\,\forall s\in\gamma(\hat{s})\,.\,\forall i\in I\,.\,\exists\hat{i}\in\hat{q}(\hat{s})\,.\,f(s,i)\in\gamma(\hat{f}(\hat{s},\hat{i})),\\&\forall\hat{s}\in\hat{S}\,.\,\forall s\in\gamma(\hat{s})\,.\,\forall\hat{i}\in\hat{q}(\hat{s})\,.\,\exists i\in I\,.\,f(s,i)\in\gamma(\hat{f}(\hat{s},\hat{i})).\end{aligned}\tag{4.16}$$

Informally, from each concrete state covered by an abstract state, the first part of Equation 4.16 requires that each concrete step result is covered by some abstract step result, while the second part requires that each abstract step result covers some concrete step result.

First part. We assume $s\in S,\hat{s}\in\gamma(\hat{s}),i\in I$ to be arbitrary but fixed. From the second assumption in Equation 4.5, we know that we can choose some $\hat{i}\in\hat{q}(\hat{s})$ for which $i\in\zeta(\hat{i})$. From the third assumption, $f(s,i)\in\gamma(\hat{f}(\hat{s},\hat{i}))$.

Second part. We assume $s\in S,\hat{s}\in\gamma(\hat{s}),\hat{i}\in\hat{q}(\hat{s})$ to be arbitrary but fixed. As $\zeta:\hat{I}\to 2^I\setminus\{\emptyset\}$, we can always choose some $i\in\zeta(\hat{i})$. From the third assumption in Equation 4.5, $f(s,i)\in\gamma(\hat{f}(\hat{s},\hat{i}))$.

We conclude that the soundness of G is preserved in $\hat{G}$ due to Lemma 4.5.10. □

4.5.3 Proof of Monotonicity

Proof (Theorem 4.5.5). We are to prove that soundness of $\hat{G}^1=(\hat{S},\hat{s}_0,\hat{I},\hat{q}^1,\hat{f}^1,\hat{L})$ is preserved in $\hat{G}^2=(\hat{S},\hat{s}_0,\hat{I},\hat{q}^2,\hat{f}^2,\hat{L})$. We define

$$H=\{(\hat{s}^1,\hat{s}^2)\in\hat{S}\times\hat{S}\mid\gamma(\hat{s}^1)\subseteq\gamma(\hat{s}^2)\}.\tag{4.17}$$

We assume Equation 4.7 as well as Equation 4.6 for $\hat{G}^2$, and will prove that Equation 4.11 holds. The first part of Equation 4.11 holds trivially. Its second part holds due to the first part of Equation 4.6. Assuming that $\hat{s}^1 \in \hat{S}, \hat{s}^2 \in \hat{S}$ are arbitrary but fixed and $\gamma(\hat{s}^1) \subseteq \gamma(\hat{s}^2)$ holds, we rewrite the last two parts of Equation 4.11 as

$$\begin{aligned} &\forall \hat{i}^1 \in \hat{q}^1(\hat{s}^1) \,.\, \exists \hat{i}^2 \in \hat{q}^2(\hat{s}^2) \,.\, \gamma(\hat{f}^1(\hat{s}^1, \hat{i}^1)) \subseteq \gamma(\hat{f}^2(\hat{s}^2, \hat{i}^2)), \\ &\forall \hat{i}^2 \in \hat{q}^2(\hat{s}^2) \,.\, \exists \hat{i}^1 \in \hat{q}^1(\hat{s}^1) \,.\, \gamma(\hat{f}^1(\hat{s}^1, \hat{i}^1)) \subseteq \gamma(\hat{f}^2(\hat{s}^2, \hat{i}^2)). \end{aligned} \tag{4.18}$$

First part. We assume $\hat{i}^1 \in \hat{q}^1(\hat{s}^1)$ arbitrary but fixed and choose an $\hat{i}^2 \in \hat{q}^2(\hat{s}^2)$ for which $\zeta(\hat{i}^1) \subseteq \zeta(\hat{i}^2)$, which exists due to the first part of Equation 4.7.

Second part. We assume $\hat{i}^2 \in \hat{q}^2(\hat{s}^2)$ arbitrary but fixed and choose an $\hat{i}^1 \in \hat{q}^1(\hat{s}^1)$ for which $\zeta(\hat{i}^1) \subseteq \zeta(\hat{i}^2)$, which exists due to the second part of Equation 4.7.

It remains to prove $\gamma(\hat{f}^1(\hat{s}^1, \hat{i}^1)) \subseteq \gamma(\hat{f}^2(\hat{s}^2, \hat{i}^2))$. From the third part of assumed Equation 4.7, we strengthen to $\gamma(\hat{f}^2(\hat{s}^1, \hat{i}^1)) \subseteq \gamma(\hat{f}^2(\hat{s}^2, \hat{i}^2))$, which directly follows from our assumptions: $\gamma(\hat{s}^1) \subseteq \gamma(\hat{s}^2), \zeta(\hat{i}^1) \subseteq \zeta(\hat{i}^2)$, and Equation 4.6 assumed for $\hat{G}^2$. We conclude that the soundness of $\hat{G}^1$ is preserved in $\hat{G}^2$ due to Lemma 4.5.10. □

4.5.4 Proof of Completeness

Proof (Theorem 4.5.6). As $S, I, \hat{S}, \hat{I}$ are finite, both model-checking and monotonic refinement of $\hat{q}, \hat{f}$ can be performed in finite time. It remains to prove that $\hat{G}$ can be monotonically refined to some $\hat{G}^*$ where soundness of G is preserved in $\hat{G}^*$ and soundness of $\hat{G}^*$ is preserved by G, so both have the same interpretations of μ-calculus properties.

Due to Equation 4.8, there exists a function $Z : S \to \hat{S}$ that maps each state s to some state $\hat{s}$ so that $\gamma(\hat{s}) = \{s\}$. We define an abstract generating automaton $\hat{G}^* = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}^*, \hat{f}^*, \hat{L}^*)$ as

$$\begin{aligned} \hat{q}^*(\hat{s}) &= \{\hat{i} \mid \zeta(\hat{i}) = 1\}, \\ \hat{f}^*(\hat{s}, \hat{i}) &= \begin{cases} Z(f(s,i)) & \{s\} = \gamma(\hat{s}), \{i\} = \zeta(\hat{i}), \\ \hat{f}(\hat{s}, \hat{i}) & \text{otherwise}, \end{cases} \\ \hat{L}^*(\hat{s}, a) &= \begin{cases} b & b \in \{0,1\}, \forall s \in \gamma(\hat{s}) \,.\, L(s,a) = b, \\ \bot & \text{otherwise}. \end{cases} \end{aligned} \tag{4.19}$$

Clearly, $\hat{G}^*$ is monotone w.r.t. concretization as per Equation 4.6. As we assume Equation 4.8 holds and Equation 4.5 holds for $\hat{G}$, it can be shown from Equation 4.19 that Equation 4.5 holds for $\hat{G}^*$ as well. Therefore, the soundness of G is preserved in $\hat{G}^*$. Furthermore, as $\hat{G}$ is monotone w.r.t. concretization, and Equation 4.7 between $\hat{G}^1 = \hat{G}^*$ and $\hat{G}^2 = \hat{G}$ holds due to Equation 4.19 and soundness of G is preserved in $\hat{G}$, $\hat{G}$ can be monotonically refined to $\hat{G}^*$. It remains to prove that the soundness of G is preserved in $\hat{G}^*$. We define

$$H = \{(\hat{s}, s) \in \hat{S} \times S \mid \gamma(\hat{s}) = \{s\}\}. \tag{4.20}$$

As $\gamma(\hat{s}_0) = \{s_0\}$, the first part of Equation 4.11 holds. Its second part holds as only the states of $\hat{G}^*$ with exactly one concretization are present in H, and they have the same labelling as their concretization due to Equation 4.19. Rewriting the last two parts of Equation 4.11 results in the same formulas as in Equation 4.16 (only with the order of the parts swapped). The same argument as in the proof of soundness can therefore be used to show Lemma 4.5.10 holds. As such, the soundness of $\hat{G}^*$ is preserved by G, completing the proof. □

4.6 Implementation and Experimental Evaluation

I have implemented an instance of my proposed framework in **machine-check**. I implemented explicit-state CTL model-checking of KS as per Clarke et al. [75], extending to PKS as per Bruns and Godefroid [68, p. 173-174]. As I am primarily interested in verifying machine-code systems, I used three-valued bit-vector abstraction [A.1] in my implementation. To find the path prefix to be refined, as in Figure 4.2, we deduce it from the computed labellings.

In addition to the choice of abstraction, the choice of the initial AGA and its refinements is crucial. I will experimentally evaluate three basic strategies currently supported in **machine-check**:

- **Naïve.** In the initial AGA, only single-concretization inputs are qualified and the abstract step function exactly covers the concrete step functions for single-concretization arguments, similarly to G^* from Subsection 4.5.4. No refinement is performed, and the state space is equivalent to the one generated by the CGA.
- **Input splitting.** In the initial AGA, the qualified inputs have all bits unknown and are refined as necessary.
- **Input splitting with decay.** In addition to input splitting, the abstract step function initially decays step results to states with all bits unknown. The decay is refined as necessary.

In the current implementation, I refine one bit at a time, update the state space after refinement (retaining the parts of the state space that did not change, with infrequent mark-and-sweep cleanup), and recompute all labellings afterwards. Notwithstanding possible bugs, the implementation should be sound, monotone, and complete. The framework was easy to implement, and the main challenges were in sub-problems: abstracting, model checking, and choosing the refinement.

Evaluation. To demonstrate the usability of my framework, I verified recovery properties on systems similar to those in Figure 4.1, but with added parameters. The system input is a tuple $i = (n, w, r)$, where r determines whether the system should be reset. Each state is a tuple $s = (v, u, c)$, where the variable v represents the maximum running value of input n as in Figure 4.1, u is loaded from input w in every step but otherwise unused and irrelevant, and c is an irrelevant free-running counter. The systems are parameterised

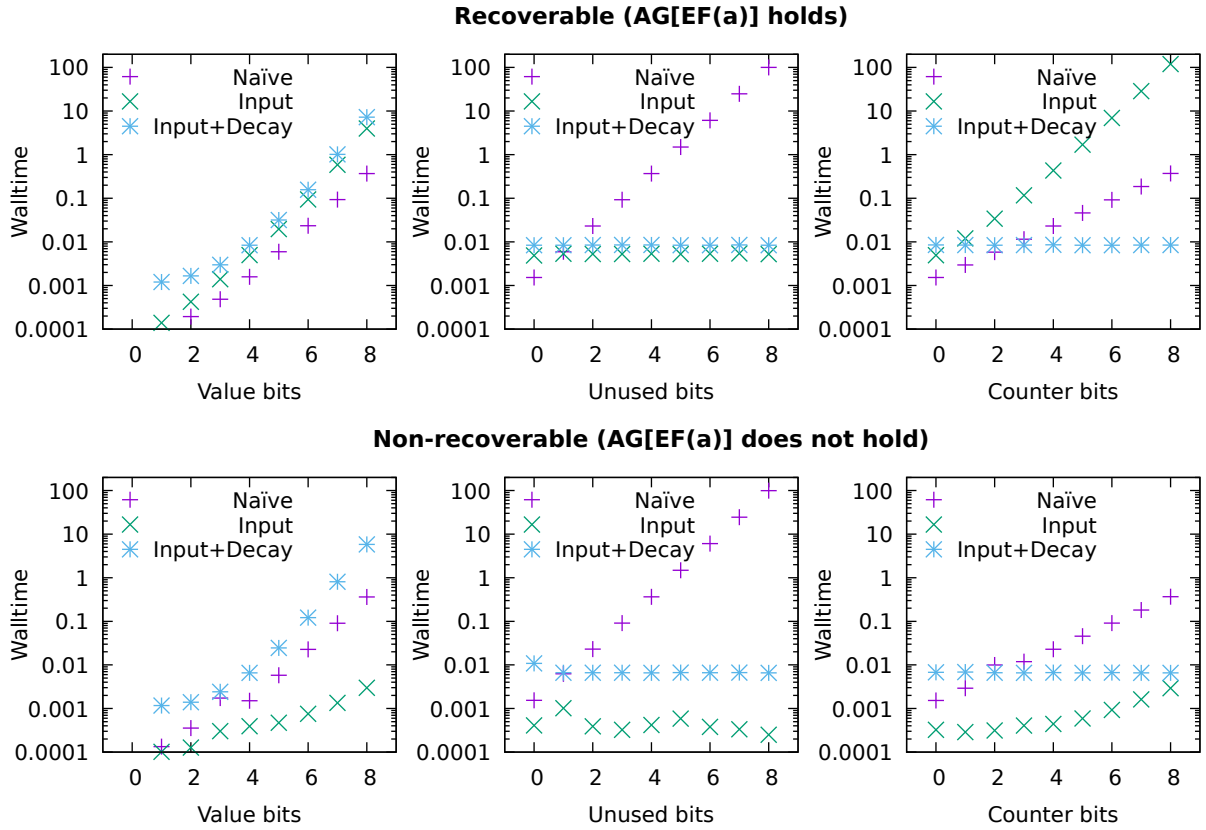

Figure 4.5: Wall-time elapsed during verification of the recovery property. In the left column, V is manipulated, $U = C = 0$. In the centre column, U is manipulated, $V = 4$, $C = 0$. In the right column, C is manipulated, $V = 4$, $U = 0$. Rising-line trends correspond to exponential explosion. Horizontal-line trends show complete insensitivity to the independent variable, with no exponential explosion. The unusually fast verification of the non-recoverable system with the input splitting strategy is caused by the quick generation of a reachable state where $\mathbf{EF}[a]$ does not hold.

by naturals V, U, and C, determining the number of used bits in v, u, and c, respectively. The initial state is $(0, 0, 0)$ and the step functions are

$$\begin{aligned} f_{\text{non-recoverable}}((v,u,c),(n,w,r)) &\stackrel{\text{def}}{=} (\max(v,n), w, c+1 \bmod 2^C), \\ f_{\text{recoverable}}((v,u,c),(n,w,r)) &\stackrel{\text{def}}{=} ((1-r)\max(v,n), w, c+1 \bmod 2^C), \end{aligned} \tag{4.21}$$

Using the implementation in **machine-check**[3], I verified the CTL property $AG[EF[a]]$ (a is true when $v = 0$) for various parameter combinations, and visualised the elapsed wall-time in Figure 4.5.

As we are evaluating the framework, not the tool, I will only discuss the trends shown by the strategies.

[3]The results of the evaluation together with the source code and scripts for reproduction are available as an artefact at `https://doi.org/10.5281/zenodo.13375827`.

- **Naïve.** Susceptible to exponential explosion in all shown cases.
- **Input splitting.** Not susceptible to exponential explosion due to irrelevant unused input assignments (middle column of Figure 4.5).
- **Input splitting with decay.** In addition, not susceptible to exponential explosion due to irrelevant computations (right column of Figure 4.5).

My framework is clearly capable of mitigating exponential explosion. The constant complexity factors are determined by the abstraction and refinement strategy choices, which should be chosen to suit the systems under verification.

4.7 Further Notes

The presented input-based Three-Valued Abstraction Refinement (TVAR) framework allows verification of μ-calculus properties on digital systems using abstraction refinement. I have implemented the input-based framework I devised in my formal verification tool **machine-check**, which, as far as I know, is the first free and open-source implementation of TVAR. While the implementation in **machine-check** is currently only able to check CTL properties, this is dependent only on the implementation of the underlying classic algorithms for standard two-valued model checking. The abstraction domain used is also not restricted by anything else than the requirements for soundness (and optionally monotonicity and completeness), providing a high amount of flexibility. The abstraction domain currently implemented in **machine-check** is that of three-valued bitvectors, which supports all common bitvector operations (bitwise, arithmetic etc.) thanks to the techniques shown in Chapter 5. The combination of the techniques in **machine-check** will be discussed and evaluated in Chapter 6.

Chapter 5

Abstract Three-valued Bit-vector Arithmetic

As discussed in Chapter 2.1, one of the major commonalities of digital systems are operations on bit-vectors. Especially important are bitwise logical operations and fixed-point wrap-around arithmetic. To formally verify such systems using model checking with abstraction, the bit-vectors must be abstracted somehow and manipulated with analogues of the operations used on concrete bit-vectors.

For machine-code systems, it is advantageous to use the three-valued bit-vector abstraction, where each abstract bit can have value "zero", "one", or "perhaps one, perhaps zero" (unknown). Using this abstraction, bit and bit-vector movement operations may be performed directly on abstract bits.

Each movement operation produces a single abstract result, avoiding state space explosion. The caveat is that overapproximation is incurred as relationships between unknown values are lost. Bitwise logic operations (AND, OR, NOT...) can be performed in three-valued logic, producing a single abstract result without exponential explosion [27, 50].

When implementing the predecessor to **machine-check**, a verification tool introduced in [A.4], I found that while it was effective to use three-valued bit-vector abstraction, arithmetic operations still required instantiation of the unknown bits to enumerate all concrete input possibilities, treating each arising output possibility as distinct. This would lead not only to output computation time increasing exponentially based on the number of unknown bits but also to the potential creation of multiple new states and the possibility of severe state space explosion. For example, an operation with two 32-bit inputs and a 32-bit output could require up to 2^{64} concrete operation computations and could produce up to 2^{32} new states. This prompted me to research how to quickly compute useful results of arithmetic operations while using three-valued abstraction, with no possibility of exponential explosion due to instantiation.

Note 5.0.1. This chapter is based on the contents of the paper [A.1] that I published together with my supervisor, reworked for inclusion in this thesis. As the paper was a joint work, I have retained the plural first-person pronouns (*we*) in the rest of this chapter.

I was the main contributor, while my supervisor contributed mainly to the fast abstract multiplication proofs in Section 5.6.

In this chapter, we formulate the *forward operation problem*, where an arbitrary operation performed on three-valued abstract bit-vector inputs results in a single three-valued abstract bit-vector output which preserves the soundness of model checking. While the best possible output can always be found in worst-case time exponential in the number of three-valued input bits, this is slow for 8-bit binary operations and infeasible for higher powers of two.

To aid with the construction of polynomial-time worst-case algorithms, we devise a novel *modular extreme-search* technique. Using this technique, we find a linear-time algorithm for abstract addition and a worst-case quadratic-time algorithm for abstract multiplication.

Our results allow model checkers that use the three-valued abstraction technique to compute the state space faster and to manage its size by only performing instantiation when necessary, reducing the risk of state space explosion.

5.1 Related Work

Many-valued logics have been extensively studied on their own, including Kleene logic [64] used for three-valued model checking [27]. Previously, three-valued logic was used for static program analysis of 8-bit microcontroller programs[46]. Binary decision diagrams (BDDs) were used to compress input-output relationships for arbitrary abstract operations. This resulted in high generation times and storage usage, making the technique infeasible to use with 16-bit or 32-bit operands. These restrictions are not present in our approach where we produce the abstract operation results purely algorithmically, but precomputation may still be useful for abstract operations with no known worst-case polynomial-time algorithms.

In addition to machine-code analysis and verification, multi-valued logics are also widely used for register-transfer level digital logic simulation. The IEEE 1164 standard [76] introduces nine logic values, out of which '0' (zero), '1' (one), and 'X' (unknown) directly correspond to three-valued abstraction. For easy differentiation between concrete values and abstract values, we will use the IEEE 1164 notation in this paper, using single quotes to represent an abstract bit as well as double quotes to represent an abstract bit-vector (tuple of abstract bits). While we primarily consider microprocessor machine-code model checking as our use case, we note that the presented algorithms also might be useful for simulation, automated test pattern generation, and formal verification of digital circuits containing adders and multipliers.

Yamane et al. [77] proposed that instantiation may be performed based only on interesting variables. For example, if a status flag "zero" is of interest, a tuple of values "XX" from which the flag is computed should be replaced by the possibilities {"00", "1X", "X1"}. This leads to lesser state space explosion compared to naïve instantiation but is not relevant for our discussion as we discuss avoiding instantiation entirely during operation resolution.

In the paper, we define certain pseudo-Boolean functions and search for their global extremes. This is also called pseudo-Boolean optimisation [78]. Problems in this field are often NP-hard. However, pseudo-Boolean functions for addition and multiplication that we will use in this paper have special forms that will allow us to resolve the corresponding problems in polynomial time without having to resort to advanced pseudo-Boolean optimisation techniques.

5.2 Basic Definitions

Let us consider a binary concrete operation which produces a single M-bit output for each combination of two N-bit operands, i.e. $r : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{B}^M$. We define the *forward operation problem* as the problem of producing a single abstract bit-vector output given supplied abstract inputs, preserving soundness. The output is not pre-restricted (the operation computation moves only *forward*). To preserve soundness, the abstract output must contain all possible concrete outputs that would be generated by first performing instantiation, receiving a set of concrete possibilities, and then performing the operation on each possibility.

To easily formalise this requirement, we first formalise three-valued abstraction using sets. Each three-valued abstract bit value ('0','1','X') identifies all possible values the corresponding concrete bit can take. We define the abstract bit as a subset of $\mathbb{B} = \{0, 1\}$ and the abstract bit values as

$$\text{`0'} \stackrel{\text{def}}{=} \{0\}, \text{`1'} \stackrel{\text{def}}{=} \{1\}, \text{`X'} \stackrel{\text{def}}{=} \{0, 1\}. \tag{5.1}$$

This formalisation corresponds exactly to the meaning of 'X' as "possibly 0, possibly 1". Even though $\emptyset$ is also a subset of $\mathbb{B}$, it is never assigned to any abstract bit as there is always at least a single output possibility.

If an abstract bit is either '0' or '1', we consider it *known*; if it is 'X', we consider it *unknown*. For ease of representation in equations, we also introduce an alternative math-style notation $\hat{X} \stackrel{\text{def}}{=} \{0, 1\}$.

Next, we define abstract bit-vectors as tuples of abstract bits. For clarity, we use hat symbols to denote abstract bit-vectors and abstract operations. We use zero-based indexing for simplicity of representation and correspondence to typical implementations, i.e. $\hat{a}_0$ means the lowest bit of abstract bit-vector $\hat{a}$. We denote slices of the bit-vectors by indexing via two dots between endpoints, i.e. $\hat{a}_{0..2}$ means the three lowest bits of abstract bit-vector $\hat{a}$. In case the slice reaches higher than the most significant bit of an abstract bit-vector, we assume it to be padded with '0', consistent with an interpretation as an unsigned number.

5.2.1 Abstract Bit Encodings

In implementations of algorithms, a single abstract bit may be represented by various *encodings*. First, we formalise a *zeros-ones* encoding of abstract bit $\hat{a}_i$ using concrete bits

$a_i^0 \in \mathbb{B}$, $a_i^1 \in \mathbb{B}$ via

$$a_i^0 = 1 \iff 0 \in \hat{a}_i,\ a_i^1 = 1 \iff 1 \in \hat{a}_i, \tag{5.2}$$

which straightforwardly extends to bit-vectors a^0, a^1. Assuming $\hat{a}$ has $A \in \mathbb{N}_0$ bits, $\hat{a} \in (2^{\mathbb{B}})^A$, while $a^0 \in \mathbb{B}^A$, $a^1 \in \mathbb{B}^A$, i.e. they are concrete bit-vectors.

We also formalise a *mask-value* encoding: the mask bit $a_i^{\mathrm{m}} = 1$ exactly when the abstract bit is unknown. When the abstract bit is known, the value bit a_i^{v} corresponds to the abstract value (0 for '0', 1 for '1'), as previously used in [27]. For simplicity, we further require $a_i^{\mathrm{v}} = 0$ if $a_i^{\mathrm{m}} = 1$. We formalise the encoding of abstract bit $\hat{a}_i$ using concrete bits $a_i^{\mathrm{m}} \in \mathbb{B}$, $a_i^{\mathrm{v}} \in \mathbb{B}$ via

$$a_i^{\mathrm{m}} = 1 \iff 0 \in \hat{a}_i \wedge 1 \in \hat{a}_i,\ a_i^{\mathrm{v}} = 1 \iff 0 \notin \hat{a}_i \wedge 1 \in \hat{a}_i, \tag{5.3}$$

which, again, straightforwardly extends to bit-vectors $a^{\mathrm{m}} \in \mathbb{B}^A$ and $a^{\mathrm{v}} \in \mathbb{B}^A$. We note that the encodings can be quickly converted via

$$\begin{aligned} a_i^0 = 1 &\iff a_i^{\mathrm{m}} = 1 \vee a_i^{\mathrm{v}} = 0,\ a_i^1 = 1 \iff a_i^{\mathrm{m}} = 1 \vee a_i^{\mathrm{v}} = 1, \\ a_i^{\mathrm{m}} = 1 &\iff a_i^0 = 1 \wedge a_i^1 = 1,\ a_i^{\mathrm{v}} = 1 \iff a_i^0 = 0 \wedge a_i^1 = 1. \end{aligned} \tag{5.4}$$

We note that when interpreting each concrete possibility in abstract bit-vector $\hat{a}$ as an unsigned binary number, a^{v} corresponds to the minimum, while a^1 corresponds to the maximum. For conciseness and intuitiveness, we will not explicitly note the conversions in the presented algorithms. Furthermore, where usage of arbitrary encoding is possible, we will write the hat-notated abstract bit-vector, e.g. $\hat{a}$.

5.2.2 Abstract Transformers

We borrow the notions defined in this subsection from abstract interpretation [79, 80], adapting them for the purposes of this paper.

The set of concrete bit-vector possibilities given by a tuple containing A abstract bits, $\hat{a} \in (2^{\mathbb{B}})^A$, is given by a *concretization function* $\gamma : (2^{\mathbb{B}})^A \to 2^{(\mathbb{B}^A)}$,

$$\gamma(\hat{a}) \stackrel{\text{def}}{=} \{a \in \mathbb{B}^A \mid \forall i \in \{0, \ldots, A-1\} \,.\, a_i \in \hat{a}_i\}. \tag{5.5}$$

Conversely, the transformation of a set of bit-vector possibilities $C \in 2^{(\mathbb{B}^A)}$ to a single abstract bit-vector $\hat{a} \in (2^{\mathbb{B}})^A$ is determined by an *abstraction function* $\alpha : 2^{(\mathbb{B}^A)} \to (2^{\mathbb{B}})^A$ which, to prevent underapproximation and to ensure soundness of model checking, must fulfil $C \subseteq \gamma(\alpha(C))$.

An abstract operation $\hat{r} : (2^{\mathbb{B}})^N \times (2^{\mathbb{B}})^N \to (2^{\mathbb{B}})^M$ corresponding to concrete operation $r : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{B}^M$ is an *approximate abstract transformer* if it overapproximates r, that is,

$$\forall \hat{a} \in (2^{\mathbb{B}})^N, \hat{b} \in (2^{\mathbb{B}})^N \,.\, \{r(a,b) \mid a \in \gamma(\hat{a}), b \in \gamma(\hat{b})\} \subseteq \gamma(\hat{r}(\hat{a}, \hat{b})). \tag{5.6}$$

The number of concrete possibilities $|\gamma(\alpha(C))|$ should be minimised to prevent unnecessary overapproximation. For three-valued bit-vectors, the best abstraction function α^{best} is uniquely given by

$$\forall i \in \{0, \ldots, A-1\} \,.\, (\alpha^{\text{best}}(C))_i \stackrel{\text{def}}{=} \{c_i \in \mathbb{B} \mid c \in C\}. \tag{5.7}$$

By using α^{best} to perform the abstraction on the minimal set of concrete results from Equation 5.6, we obtain the *best abstract transformer* for arbitrary concrete operation r, i.e. an approximate abstract transformer resulting in the least amount of overapproximation, uniquely given as

$$\hat{r}_k^{\text{best}}(\hat{a}, \hat{b}) = \alpha^{\text{best}}(\{r_k(a, b) \mid a \in \gamma(\hat{a}), b \in \gamma(\hat{b})\}). \tag{5.8}$$

We note that when no input abstract bit is $\emptyset$, there is at least one concrete result $r(a, b)$ and no output abstract bit can be $\emptyset$. Thus, three-valued representation is truly sufficient.

5.2.3 Algorithm Complexity Considerations

We will assume that the presented algorithms are implemented on a general-purpose processor that operates on binary machine words and can compute bitwise operations, bit shifts, addition and subtraction in constant time. Every bit-vector used fits in a machine word. This is a reasonable assumption, as it is likely that the processor used for verification will have the machine word size equal to or greater than the processor that runs the program under consideration.

We also assume that the ratio of M to N is bounded, allowing us to express the presented algorithm time complexities using only N. Memory complexity is not an issue as the presented algorithms use only a fixed amount of temporary variables in addition to the inputs and outputs.

5.2.4 Naïve Universal Abstract Algorithm

Equation 5.8 immediately suggests a naïve algorithm for computing $\hat{r}^{\text{best}}$ for any given $\hat{a}, \hat{b}$: enumerating all $a, b \in 2^{(\mathbb{B}^N)}$, filtering out the ones that do not satisfy $a \in \gamma(\hat{a}) \ \wedge \ b \in \gamma(\hat{b})$, and marking the results of $r(a, b)$, which is easily done in the zeros-ones encoding. This naïve algorithm has a running time of $\Theta(2^{2N})$.

Average-case computation time can be improved by only enumerating unknown input bits, but worst-case time is still exponential. Even for 8-bit binary operations, the worst-case input combination (all bits unknown) would require 2^{16} concrete operation computations. For 32-bit binary operations, it would require 2^{64} computations, which is infeasible. Finding worst-case polynomial-time algorithms for common operations is therefore of significant interest.

5.3 Formal Problem Statement

Theorem 5.3.1. The best abstract transformer of abstract bit-vector addition is computable in linear time.

Theorem 5.3.2. The best abstract transformer of abstract bit-vector multiplication is computable in worst-case quadratic time.

In Section 5.4, we will introduce a novel *modular extreme-finding technique* which will use a basis for finding fast best abstract transformer algorithms. Using this technique, we will prove Theorems 5.3.1 and 5.3.2 by constructing corresponding algorithms in Sections 5.5 and 5.6, respectively. We will experimentally evaluate the presented algorithms to demonstrate their practical efficiency in Section 5.7.

5.4 Modular Extreme-Finding Technique

The concrete operation function r may be replaced by a pseudo-Boolean function $h : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{N}_0$ where the output of r is the output of h written in base 2. Surely, that fulfils

$$\begin{aligned} &\forall a \in \mathbb{B}^N, b \in \mathbb{B}^N, \forall k \in \{0, \ldots, M-1\} \; . \\ &\qquad r_k(a,b) = 1 \iff (h(a,b) \bmod 2^{k+1}) \geq 2^k. \end{aligned} \tag{5.9}$$

The best abstract transformer definition in Equation 5.8 is then equivalent to

$$\begin{aligned} &\forall k \in \{0, \ldots, M-1\} \; . \\ &\qquad (0 \in \hat{r}_k^{\text{best}} \iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \; . \; (h(a,b) \bmod 2^{k+1}) < 2^k) \wedge \\ &\qquad (1 \in \hat{r}_k^{\text{best}} \iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \; . \; (h(a,b) \bmod 2^{k+1}) \geq 2^k). \end{aligned} \tag{5.10}$$

The forward operation problem is therefore transformed into a problem of solving certain modular inequalities, which is possible in polynomial time for certain operations. We will later show that these include addition and multiplication.

If the inequalities were not modular, it would suffice to find the global minimum and maximum (extremes) of h. Furthermore, the modular inequalities in Equation 5.10 can be thought of as alternating intervals of length 2^k. Intuitively, if it was possible to move from the global minimum to the global maximum in steps of at most 2^k by using different values of $a \in \hat{a}, b \in \hat{b}$ in $h(a,b)$, it would suffice to find the global extremes and determine whether they are in the same 2^k interval. If they were, only one of the modular inequalities would be satisfied, resulting in known r_k (either '0' or '1'). If they were not, each modular inequality would be satisfied by some a, b, resulting in $r_k = \hat{X}$.

We will now formally prove that our reasoning for this *modular extreme-finding method* is indeed correct.

Lemma 5.4.1. Consider a sequence of integers $t = (t_0, t_1, \ldots, t_{T-1})$ that fulfils

$$\forall n \in [0, T-2] \,.\, |t_{n+1} - t_n| \leq 2^k. \tag{5.11}$$

Then,

$$\begin{aligned} &\exists v \in [\min t, \max t] \,.\, (v \bmod 2^{k+1}) < 2^k \iff \\ &\exists n \in [0, T-1] \,.\, (t_n \bmod 2^{k+1}) < 2^k. \end{aligned} \tag{5.12}$$

Proof. As the sequence t is a subset of range $[\min t, \max t]$, the backward direction is trivial. The forward direction trivially holds if v is contained in t. If it is not, it is definitely contained in some range (v^-, v^+), where v^-, v^+ are successive values in the sequence t. Considering successive values of x in the closed range $[v^-, v^+]$, the valuation of $(x \bmod 2^{k+1}) < 2^k$ changes at most once, since $|v^+ - v^-| \leq 2^k$. Therefore, the valuation for the existing v must be the same as the valuation for v^+, v^-, or both. As both v^+ and v^- are in the sequence t, this completes the proof. □

Theorem 5.4.2. Consider a pseudo-Boolean function $f : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{Z}$, two inputs $\hat{a}, \hat{b} \in (2^{\mathbb{B}})^N$, and a sequence $p = (p_0, p_1, \ldots, p_{P-1})$ where each element is a pair $(a, b) \in (\gamma(\hat{a}), \gamma(\hat{b}))$, that fulfil

$$\begin{aligned} &\forall n \in [0, P-2] \,.\, |f(p_{n+1}) - f(p_n)| \leq 2^k, \\ &f(p_0) = \min_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} f(a, b), \\ &f(p_{P-1}) = \max_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} f(a, b). \end{aligned} \tag{5.13}$$

Then,

$$\begin{aligned} \forall\, C \in \mathbb{Z} \,.\, (\exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \,.\, ((f(a,b) + C) \bmod 2^{k+1}) < 2^k \\ \iff \exists n \in [0, P-1] \,.\, ((f(p_n) + C) \bmod 2^{k+1}) < 2^k). \end{aligned} \tag{5.14}$$

Proof. Since each element of p is a pair $(a, b) \in (\gamma(\hat{a}), \gamma(\hat{b}))$, the backward direction is trivial. For the forward direction, use Lemma 5.4.1 to convert the sequence $(f(p_n) + C)_{n=0}^{P-1}$ to range $[f(p_0) + C, f(p_{P-1}) + C]$ and rewrite the forward direction as

$$\begin{aligned} &\forall\, C \in \mathbb{Z} \,.\, (\exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \,.\, ((f(a,b) + C) \bmod 2^{k+1}) < 2^k \implies \\ &\exists v \in \left[\min_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} (f(a,b) + C) \,,\, \max_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} (f(a,b) + C) \right] \,.\, (v \bmod 2^{k+1}) < 2^k). \end{aligned} \tag{5.15}$$

The implication clearly holds, completing the proof. □

While Theorem 5.4.2 forms a basis for the modular extreme-finding method, there are two problems. First, finding global extremes of a pseudo-Boolean function is not generally

trivial. Second, the *step condition*, that is, the absence of a step longer than 2^k in h, must be ensured. Otherwise, one of the inequality intervals could be "jumped over". For non-trivial operators, steps longer than 2^k surely are present in h for some k. However, instead of h, it is possible to use a tuple of functions $(h_k)_{k=0}^{M-1}$ where each one fulfils Equation 5.10 for a given k exactly when h does. This is definitely true if each h_k is congruent with h modulo 2^{k+1}.

Fast best abstract transformer algorithms can now be formed based on finding extremes of h_k, provided that h_k changes by at most 2^k when exactly one bit of input changes its value, which implies that a sequence p with properties required by Theorem 5.4.2 exists. For ease of expression of the algorithms, we define a function which discards bits of a number x below bit k (or, equivalently, performs integer division by 2^k),

$$\zeta_k(x) = \left\lfloor \frac{x}{2^k} \right\rfloor . \tag{5.16}$$

For conciseness, given inputs $\hat{a} \in (2^{\mathbb{B}})^N, \hat{b} \in (2^{\mathbb{B}})^N$, we also define

$$h_k^{\min} \stackrel{\text{def}}{=} \min_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} h_k(a,b), h_k^{\max} \stackrel{\text{def}}{=} \max_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} h_k(a,b), \tag{5.17}$$

Equation 5.10 then can be reformulated as follows: if $\zeta_k(h_k^{\min}) \neq \zeta_k(h_k^{\max})$, both inequalities are definitely fulfilled (as each one must be fulfilled by some element of the sequence) and output bit k is unknown. Otherwise, only one inequality is fulfilled, the output bit k is known and its value corresponds to $\zeta_k(h_k^{\min}) \bmod 2$. This forms the basis of Algorithm 5.1, which provides a general blueprint for fast abstract algorithms. Proper extreme-finding for the considered operation must be added to the algorithm, denoted by $(\dots)$ in the algorithm pseudocode. We will devise extreme-finding for fast abstract addition and multiplication operations in the rest of the paper.

Algorithm 5.1: Modular extreme-finding abstract algorithm blueprint

```
1: function MODULAR_ALGORITHM_BLUEPRINT(â, b̂)
2:     for k ∈ {0, ..., M − 1} do
3:         h_k^min ← (...)                          ▷ Compute extremes of h_k
4:         h_k^max ← (...)
5:         if ζ_k(h_k^min) ≠ ζ_k(h_k^max) then
6:             c_k ← X̂                               ▷ Set result bit unknown
7:         else
8:             c_k^m ← 0, c_k^v ← ζ_k(h_k^min) mod 2   ▷ Set value
9:         end if
10:    end for
11:    return ĉ
12: end function
```

5.5 Fast Abstract Addition

To express fast abstract addition using the modular extreme-finding technique, we first define a function expressing the unsigned value of a concrete bit-vector a with an arbitrary number of bits A,

$$\Phi(a) \stackrel{\text{def}}{=} \sum_{i=0}^{A-1} 2^i a_i. \tag{5.18}$$

Pseudo-Boolean addition is then defined simply as

$$h^+(a,b) \stackrel{\text{def}}{=} \Phi(a) + \Phi(b). \tag{5.19}$$

To fulfil the step condition, we define

$$h_k^+(a,b) = \Phi(a_{0..k}) + \Phi(b_{0..k}). \tag{5.20}$$

This is congruent with h^+ modulo 2^{k+1}. The step condition is trivially fulfilled for every function h_k^+ in $(h_k^+)_{k=0}^{M-1}$, as changing the value of a single bit of a or b changes the result of h_k^+ by at most 2^k. We note that this is due to h^+ having a special form where only single-bit summands with power-of-2 coefficients are present. Finding the global extremes is trivial as each summand only contains a single abstract bit. Recalling Subsection 5.2.1, the extremes can be obtained as

$$\begin{aligned} h_k^{+,\min} &\leftarrow \Phi(a_{0..k}^{\mathrm{v}}) + \Phi(b_{0..k}^{\mathrm{v}}), \\ h_k^{+,\max} &\leftarrow \Phi(a_{0..k}^{1}) + \Phi(b_{0..k}^{1}). \end{aligned} \tag{5.21}$$

The best abstract transformer for addition is obtained by combining Equation 5.21 with Algorithm 5.1. Time complexity is trivially $\Theta(N)$, proving Theorem 5.3.1. Similar reasoning can be used to obtain fast best abstract transformers for subtraction and general summation, only changing the computation of $h_k^{\min}$ and $h_k^{\max}$.

For further understanding, we will show how fast abstract addition behaves for "X0" + "11":

$$\begin{aligned} &k = 0 : \text{“0” + “1”},\ 1 = \zeta_0(0+1) = \zeta_0(0+1) = 1 \rightarrow r_0 = \text{‘1’}, \\ &k = 1 : \text{“X0” + “11”},\ 1 = \zeta_1(0+3) \neq \zeta_1(2+3) = 2 \rightarrow r_1 = \text{‘X’}, \\ &k = 2 : \text{“0X0” + “011”},\ 0 = \zeta_2(0+3) \neq \zeta_2(2+3) = 1 \rightarrow r_2 = \text{‘X’}, \\ &k > 2 : \zeta_k(h_k^{+,\min}) = \zeta_k(h_k^{+,\max}) = 0 \rightarrow r_k = \text{‘0’}. \end{aligned} \tag{5.22}$$

For $M = 2$, the result is "XX1". For $M > 2$, the result is padded by '0' to the left, preserving the unsigned value of the output. For $M < 2$, the addition is modular. This fully corresponds to the behaviour of concrete binary addition.

5.6 Fast Abstract Multiplication

Multiplication is typically implemented on microprocessors with three different input signedness combinations: unsigned × unsigned, signed × unsigned, and signed × signed, with signed variables using two's complement encoding. It is a well-known fact that the signed-unsigned and signed multiplication can be converted to unsigned multiplication by extending the signed multiplicand widths to product width using an arithmetic shift right. This could pose problems when the leading significant bit is 'X', but it can be split beforehand into two cases, '0' and '1'. This allows us to only consider unsigned multiplication in this section, signed multiplication only incurring a constant-time slowdown.

5.6.1 Obtaining a Best Abstract Transformer

Abstract multiplication could be resolved similarly to abstract addition by rewriting multiplication as an addition of a sequence of shifted summands (long multiplication) and performing fast abstract summation. However, this does not result in a best abstract transformer. The shortest counterexample is "11" · "X1". Here, the unknown bit b_1 is added twice before influencing r_2, once as a summand in the computation of r_2 and once as a carryover from r_1:

$$\begin{array}{cccc} (2^3) & (2^2) & (2^1) & (2^0) \\ & & 1 & 1 \\ \cdot & b_1 & 1 & \\ \hline (b_1) & (b_1) & b_1 & 1 \\ & b_1 & 1 & \\ \hline\hline b_1 & \mathbf{2b_1} & 1+b_1 & 1 \end{array}$$

In fast abstract summation, the summand b_1 is treated as distinct for each output bit computation, resulting in unnecessary overapproximation of multiplication.

Instead, to obtain a fast best abstract transformer for multiplication, we apply the modular extreme-finding technique to multiplication itself, without intermediate conversion to summation. Fulfilling the maximum 2^k step condition is not as easy as previously. The multiplication output function h^* is defined as

$$h^*(a,b) \stackrel{\text{def}}{=} \Phi(a) \cdot \Phi(b) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 2^{i+j} a_i b_j. \tag{5.23}$$

One could try to use congruences to remove some summands from h_k^* while keeping all remaining summands positive. This would result in

$$h_k(a,b) = \sum_{i=0}^{k} \sum_{j=0}^{k-i} 2^{i+j} a_i b_j. \tag{5.24}$$

Changing a single bit a_i would change the result by $\sum_{j=0}^{k-i} 2^{i+j} b_j$. This sums to at most $2^{k+1}-1$ and thus does not always fulfil the maximum 2^k step condition. However, the sign

of the summand $2^k a_i b_{k-i}$ can be flipped due to congruence modulo 2^{k+1}, after which the change of result from a single bit flip is always in the interval $[-2^k, 2^k - 1]$. Therefore, to fulfil the maximum 2^k step condition, we define $h_k^* : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{Z}$ as

$$h_k^*(a, b) \stackrel{\text{def}}{=} \left(-\sum_{i=0}^{k} 2^k a_i b_{k-i} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i b_j \right). \tag{5.25}$$

For more insight into this definition, we will return to the counterexample to the previous approach, "11" · "X1", which resulted in unnecessary overapproximation for $k = 2$. Writing h_2^* computation as standard addition similarly to the previously shown long multiplication, the carryover of b_1 is counteracted by the summand $-2^2 b_1$:

(2^3)	(2^2)	(2^1)	(2^0)
	(b_1)	b_1	1
	$\mathbf{-b_1}$	1	
	$\mathbf{0}$	$1 + b_1$	1

It is apparent that $\zeta_2(h_2^{\min}) = \zeta_k(h_2^{\max}) = 0$ and unnecessary overapproximation is not incurred. Using that line of thinking, the definition of h_k^* in Equation 5.25 can be intuitively regarded as ensuring that the carryover of an unknown bit into the k-th column is neutralised by a corresponding k-th column summand. Consequently, if the unknown bit can appear only in both of them simultaneously, no unnecessary overapproximation is incurred.

While the maximum 2^k step condition is fulfilled in Equation 5.25, extreme-finding is much more complicated than for addition, becoming heavily dependent on abstract input bit pairs of the form $(\hat{a}_i, \hat{b}_{k-i})$ where $0 \leq i \leq k$. Such pairs result in a summand $-2^k a_i b_{k-i}$ in h_k^*. When multiplication is rewritten using long multiplication as previously, this summand is present in the k-th column. We therefore name such pairs *k-th column pairs* for conciseness.

In Subsection 5.6.2, we show that if at most one k-th column pair where $\hat{a}_i = \hat{b}_{k-i} = \hat{X}$ (*double-unknown pair*) exists, extremes of h_k^* can be found easily. In Subsection 5.6.3, we prove that if at least two double-unknown pairs exist, $r_k = \hat{X}$. Taken together, this yields a best abstract transformer algorithm for multiplication. In Subsection 5.6.4, we discuss implementation considerations of the algorithm with emphasis on reducing computation time. Finally, in Subsection 5.6.5, we present the final algorithm.

5.6.2 At Most One Double-Unknown k-th Column Pair

An extreme is given by values $a \in \hat{a}, b \in \hat{b}$ for which the value $h_k^*(a, b)$ is minimal or maximal (Equation 5.17). We will show that such a, b can be found successively when at most one double-unknown k-th column pair is present.

First, for single-unknown k-th column pairs where $\hat{a}_i = \hat{X}$, $\hat{b}_{k-i} \neq \hat{X}$, we note that in Equation 5.25, the difference between h_k^* when $a_i = 1$ and when $a_i = 0$ is

$$h_k^*(a, b \mid a_i = 1) - h_k^*(a, b \mid a_i = 0) = -2^k b_{k-i} + \sum_{j=0}^{k-i-1} 2^{i+j} b_j. \tag{5.26}$$

Since the result of the sum over j must be in the interval $[0, 2^k - 1]$, the direction of the change (negative or non-negative) is uniquely given by the value of b_{k-i}, which is known. It is therefore sufficient to ensure $a_i^{\min} \leftarrow b_{k-i}$ when minimising and $a_i^{\min} \leftarrow 1 - b_{k-i}$ when maximising. Similar reasoning can be applied to single-unknown k-th column pairs where $\hat{a}_i \neq \hat{X}$, $\hat{b}_{k-i} = \hat{X}$.

After assigning values to all unknown bits in single-unknown k-th column pairs, the only still-unknown bits are the ones in the only double-unknown k-th column pair present. In case such a pair $\hat{a}_i = \hat{X}$, $\hat{b}_j = \hat{X}, j = k - i$ is present, the difference between h_k^* when a_i and b_j are set to arbitrary values and when they are set to 0 is

$$\begin{aligned} &h_k^*(a, b) - h_k^*(a, b \mid a_i = 0, b_j = 0) = \\ &- 2^k a_i b_j + 2^i a_i \left(\sum_{z=0}^{j-1} 2^z b_z \right) + 2^j b_j \left(\sum_{z=0}^{i-1} 2^z a_z \right). \end{aligned} \tag{5.27}$$

When minimising, it is clearly undesirable to choose $a_i^{\min} \neq b_j^{\min}$. Considering that the change should not be positive, $a_i^{\min} = b_j^{\min} = 1$ should be chosen if and only if

$$2^i \left(\sum_{z=0}^{j-1} 2^z b_z \right) + 2^j \left(\sum_{z=0}^{i-1} 2^z a_z \right) \leq 2^k. \tag{5.28}$$

When maximising, it is clearly undesirable to choose $a_i^{\max} = b_j^{\max}$. That said, $a_i^{\max} = 1, b_j^{\max} = 0$ should be chosen if and only if

$$2^j \left(\sum_{z=0}^{i-1} 2^z a_z \right) \leq 2^i \left(\sum_{z=0}^{j-1} 2^z b_z \right). \tag{5.29}$$

Of course, the choice is arbitrary when both possible choices result in the same change. After the case of the only double-unknown k-th column pair present is resolved, there are no further unknown bits and thus, the values of h_k^* extremes can be computed as

$$\begin{aligned} h_k^{*,\min} &= \left(-\sum_{i=0}^{k} 2^k a_i^{\min} b_{k-i}^{\min} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i^{\min} b_j^{\min} \right), \\ h_k^{*,\max} &= \left(-\sum_{i=0}^{k} 2^k a_i^{\max} b_{k-i}^{\max} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i^{\max} b_j^{\max} \right). \end{aligned} \tag{5.30}$$

5.6.3 Multiple Double-Unknown k-th Column Pairs

Lemma 5.6.1. Consider a sequence of integers $t = (t_0, t_1, \ldots, t_{T-1})$ that fulfils

$$\forall n \in [0, T-2] \,.\, |t_{n+1} - t_n| \leq 2^k, t_0 + 2^k \leq t_{T-1}. \tag{5.31}$$

Then,

$$\exists n \in [0, T-1] \,.\, (t_n \bmod 2^{k+1}) < 2^k. \tag{5.32}$$

Proof. Use Lemma 5.4.1 to transform the claim to equivalent

$$\exists v \in [\min t, \max t] \,.\, (v \bmod 2^{k+1}) < 2^k. \tag{5.33}$$

Since $[t_1, t_1 + 2^k] \subseteq [\min t, \max t]$, such claim is implied by

$$\exists v \in [t_0, t_0 + 2^k] \,.\, (v \bmod 2^{k+1}) < 2^k. \tag{5.34}$$

As $[t_0, t_0 + 2^k] \bmod 2^{k+1}$ has $2^k + 1$ elements and there are only 2^k elements that do not fulfil $(v \bmod 2^{k+1}) < 2^k$, Equation 5.34 holds due to the pigeonhole principle. □

Corollary 5.6.2. Given a sequence of integers $(t_0, t_1, \ldots, t_{T-1})$ that fulfils Lemma 5.6.1 and an arbitrary integer $C \in \mathbb{Z}$, the lemma also holds for sequence $(t_0+C, t_1+C, \ldots, t_{T-1}+C)$.

Theorem 5.6.3. Let $\hat{r}_k^{*,\text{best}}$ be the best abstract transformer of multiplication. Let $\hat{a}$ and $\hat{b}$ be such that there are p_1, p_2, q_1, q_2 in $\{0, \ldots, k\}$ where

$$\begin{aligned} p_1 \neq p_2, p_1 + q_2 = k, p_2 + q_1 = k, \\ \hat{a}_{p_1} = \hat{X}, \hat{a}_{p_2} = \hat{X}, \hat{b}_{q_1} = \hat{X}, \hat{b}_{q_2} = \hat{X}. \end{aligned} \tag{5.35}$$

Then $\hat{r}_k^{best,*}(\hat{a}, \hat{b}) = \hat{X}$.

Proof. For an abstract bit-vector $\hat{c}$ with positions of unknown bits $u_1, \ldots, u_n$, denote the concrete bit-vector $c \in \gamma(\hat{c})$ for which $\forall i \in \{1, \ldots, n\} \,.\, c_{u_i} = s_i$ by $\gamma_{s_1,\ldots,s_n}(\hat{c})$. Let $\Phi_{s_1,\ldots,s_n}(\hat{c}) \stackrel{\text{def}}{=} \Phi(\gamma_{s_1,\ldots,s_n}(\hat{c}))$.

Now, without loss of generality, assume $\hat{a}$ only has unknown values in positions p_1 and p_2 and $\hat{b}$ only has unknown positions q_1, q_2 and $p_1 < p_2, q_1 < q_2$. Then, for $s_1, s_2, t_1, t_2 \in \mathbb{B}$, using $h(a, b) = \Phi(a) \cdot \Phi(b)$,

$$h(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b})) = (2^{p_1} s_1 + 2^{p_2} s_2 + \Phi_{00}(\hat{a})) \cdot (2^{q_1} t_1 + 2^{q_2} t_2 + \Phi_{00}(\hat{b})). \tag{5.36}$$

Define $A \stackrel{\text{def}}{=} \Phi_{00}(\hat{a})$ and $B \stackrel{\text{def}}{=} \Phi_{00}(\hat{b})$ and let them be indexable similarly to bit-vectors, i.e. $A_{0..z} = (A \bmod 2^{z+1})$, $A_z = \zeta_z(A_{0..z})$. Define

$$\begin{aligned} &h_k^{\text{proof}}(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b})) \stackrel{\text{def}}{=} \\ &\quad 2^{p_1+q_1} s_1 t_1 + 2^{p_1+q_2} s_1 t_2 + 2^{q_1} t_1 A_{0..p_2-1} + 2^{p_1} s_1 B_{0..q_2-1} + \\ &\quad 2^{p_2+q_1} s_2 t_1 + 2^{p_2+q_2} s_2 t_2 + 2^{q_2} t_2 A_{0..p_1-1} + 2^{p_2} s_2 B_{0..q_1-1} + AB. \end{aligned} \tag{5.37}$$

As $A_{p_1} = A_{p_2} = B_{q_1} = B_{q_2} = 0$, h_k^{proof} and h are congruent modulo 2^{k+1}. Define

$$D(s_1, s_2, t_1, t_2) \stackrel{\text{def}}{=} h_k^{\text{proof}}(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b})) \; - \; h_k^{\text{proof}}(\gamma_{00}(\hat{a}), \gamma_{00}(\hat{b})). \tag{5.38}$$

As $p_1 + q_2 = k$ and $p_2 + q_1 = k$,

$$\begin{aligned} D(s_1, s_2, t_1, t_2) = {} & 2^{p_1+q_1} s_1 t_1 + 2^k s_1 t_2 + 2^{q_1} t_1 A_{0..p_2-1} + 2^{p_1} s_1 B_{0..q_2-1} + \\ & 2^k s_2 t_1 + 2^{p_2+q_2} s_2 t_2 + 2^{q_2} t_2 A_{0..p_1-1} + 2^{p_2} s_2 B_{0..q_1-1}. \end{aligned} \tag{5.39}$$

Set s_1, s_2, t_1, t_2 to specific chosen values and obtain

$$\begin{aligned} D(1,1,0,0) &= D(1,0,0,0) + D(0,1,0,0), \\ D(0,0,1,1) &= D(0,0,1,0) + D(0,0,0,1), \\ D(1,0,0,1) &= 2^k + D(1,0,0,0) + D(0,0,0,1). \end{aligned} \tag{5.40}$$

Inspecting the various summands, note that

$$\begin{aligned} D(1,0,0,0) \in [0, 2^k - 1], \; D(0,1,0,0) &\in [0, 2^k - 1], \\ D(0,0,1,0) \in [0, 2^k - 1], \; D(0,0,0,1) &\in [0, 2^k - 1], \\ D(1,1,0,0) - D(1,0,0,0) &\in [0, 2^k - 1], \\ D(0,0,1,1) - D(0,0,1,0) &\in [0, 2^k - 1]. \end{aligned} \tag{5.41}$$

Recalling Equation 5.10, the best abstract transformer can be obtained as

$$\begin{aligned} 0 \in \hat{r}_k^{\text{best}} &\iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \; . \; (h_k^{\text{proof}}(a,b) \bmod 2^{k+1}) < 2^k, \\ 1 \in \hat{r}_k^{\text{best}} &\iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \; . \; ((h_k^{\text{proof}}(a,b) + 2^k) \bmod 2^{k+1}) < 2^k. \end{aligned} \tag{5.42}$$

Constructing a sequence of $h_k^{\text{proof}}(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b}))$ that fulfils the conditions of Lemma 5.6.1 then implies that both inequalities can be fulfilled due to Corollary 5.6.2, which will complete the proof. Furthermore, as $D(s_1, s_2, t_1, t_2)$ only differs from $h_k^{\text{proof}}(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b}))$ by the absence of summand AB that does not depend on the choice of s_1, s_2, t_1, t_2, it suffices to construct a sequence of $D(s_1, s_2, t_1, t_2)$ that fulfils Lemma 5.6.1 as well.

There is at least a 2^k step between $D(0,0,0,0)$ and $D(1,0,0,1)$. They will form the first and the last elements of the sequence, respectively. It remains to choose the elements in their midst so that there is at most 2^k step between successive elements.

Case 5.6.4. $D(0,1,0,0) \geq D(0,0,0,1)$. Considering Equations 5.40 and 5.41, a qualifying sequence is

$$(D(0,0,0,0), D(1,0,0,0), D(1,1,0,0), D(1,0,0,1)). \tag{5.43}$$

Case 5.6.5. $D(0,1,0,0) < D(0,0,0,1)$. Using Equation 5.39, rewrite the case condition to

$$2^{p_2-p_1} D(1,0,0,0) < 2^{q_2-q_1} D(0,0,1,0). \tag{5.44}$$

As $p_1 + q_2 = k, p_2 + q_1 = k$, it also holds that $q_2 - q_1 = p_2 - p_1$. Rewrite the case condition further to

$$2^{p_2-p_1} D(1,0,0,0) < 2^{p_2-p_1} D(0,0,1,0). \tag{5.45}$$

Therefore, $D(1,0,0,0) < D(0,0,1,0)$. Considering Equations 5.40 and 5.41, a qualifying sequence is

$$(D(0,0,0,0), D(0,0,1,0), D(0,0,1,1), D(1,0,0,1)). \tag{5.46}$$

This completes the proof. □

5.6.4 Implementation Considerations

There are some considerations to be taken into account for an efficient implementation of the fast multiplication algorithm.

The first question is how to detect the positions of single-unknown and double-unknown k-th column pairs. As such pairs have the form $2^k a_i b_{k-i}$, it is necessary to perform a bit reversal of one of the bit-vectors before bitwise logic operations can be used for position detection. Fortunately, it suffices to perform the reversal only once at the start of the computation. Defining the bit reversal of the first z bits of b as $\lambda(b,z) = (b_{z-1-i})_{i=0}^{z-1}$, when the machine word size $W \geq k+1$, reversal of the first $k+1$ bits (i.e. the bits in $b_{0..k}$) may be performed as

$$\lambda(b, k+1) = ((b_{k-i})_{i=0}^{k}) = ((b_{W-1-i})_{i=W-k-1}^{W-1}) = \lambda(b, W)_{W-k-1..W-1}. \tag{5.47}$$

It is thus possible to precompute $\lambda(b, W)$ and, for each k, obtain $\lambda(b, k+1)$ via a right shift through $W - k - 1$ bits, which can be performed in constant time. Furthermore, power-of-two bit reversals can be performed in logarithmic time on standard architectures [81, p. 33-35], which makes the computation of $\lambda(b, W)$ even more efficient.

The second problem is finding out whether multiple double-unknown k-th column pairs exist, and if there is only a single one, what is its position. While that can be determined trivially in linear time, a *find-first-set* algorithm can also be used, which can be implemented in logarithmic time on standard architectures [81, p. 9] and also is typically implemented as a constant-time instruction on modern processors.

The third problem, computation of h_k^* extremes in Equation 5.30, is not as easily mitigated. This is chiefly due to the removal of summands with coefficients above 2^k due to 2^{k+1} congruence. While typical processors contain a single-cycle multiplication operation, we have not found an efficient way to use it for the computation of Equation 5.25. To understand why this is problematic, computation of h_k^* with 3-bit operands and $k = 2$ can be visualised as

$$
\begin{array}{cccccc}
 & (2^4) & (2^3) & (2^2) & (2^1) & (2^0) \\
 & & & a_2 & a_1 & a_0 \\
\cdot & & & b_2 & b_1 & b_0 \\
\hline
 & & & (-a_2b_0) & a_1b_0 & a_0b_0 \\
 & & \cancel{a_2b_1} & (-a_1b_1) & a_0b_1 & \\
 & \cancel{a_2b_2} & \cancel{a_1b_2} & (-a_0b_2) & & \\
\hline\hline
 & & & & & \ldots
\end{array}
$$

The striked-out operands are removed due to 2^{k+1} congruence, while the k-th column pair summands are subtracted instead of adding them. These changes could be performed via some modifications of traditional multiplier implementation (resulting in a custom processor instruction), but are problematic when only traditional instructions can be performed in constant time. Instead, we propose computation of h_k^* via

$$h_k^*(a,b) = \sum_{i=0}^{k} a_i \left(-2^k b_{k-i} + 2^i \Phi(b_{0..k-i-1})\right). \tag{5.48}$$

As each summand over i can be computed in constant time on standard architectures, $h_k^*(a,b)$ can be computed in linear time. Modified multiplication techniques with lesser time complexity such as Karatsuba multiplication or Schönhage–Strassen algorithm [82] could also be considered, but they are unlikely to improve practical computation time when N corresponds to the word size of normal microprocessors, i.e. $N \leq 64$.

5.6.5 Fast Abstract Multiplication Algorithm

Applying the previously discussed improvements directly leads to Algorithm 5.2. For conciseness, in the algorithm description, bitwise operations are denoted by the corresponding logical operation symbol, shorter operands have high zeros added implicitly, and the bits of $a^{\min}, a^{\max}, b^{\min}, b^{\max}$ above k are not used, so there is no need to mask them to zero.

Algorithm 5.2: Fast abstract multiplication algorithm

1: **function** Fast_Abstract_Multiplication($\hat{a}, \hat{b}$)
2: $a^{\mathrm{v}}_{\mathrm{rev}} \leftarrow \lambda(b^{\mathrm{v}}, W)$ ▷ Compute machine-word reversals for word size W
3: $b^{\mathrm{v}}_{\mathrm{rev}} \leftarrow \lambda(b^{\mathrm{v}}, W)$
4: $a^{\mathrm{m}}_{\mathrm{rev}} \leftarrow \lambda(a^{\mathrm{m}}, W)$
5: $b^{\mathrm{m}}_{\mathrm{rev}} \leftarrow \lambda(b^{\mathrm{m}}, W)$
6: **for** $k \in \{0, \ldots, M\}$ **do**
7: $s_{\mathrm{a}} \leftarrow a^{\mathrm{m}} \wedge \neg b^{\mathrm{m}}_{\mathrm{rev},W-k-1..W-1}$ ▷ Single-unknown k-th c. pairs, 'X' in a
8: $a^{\min} \leftarrow a^{\mathrm{v}} \vee (s_{\mathrm{a}} \wedge b^{\mathrm{v}}_{\mathrm{rev},W-k-1..W-1})$ ▷ Minimise such pairs
9: $a^{\max} \leftarrow a^{\mathrm{v}} \vee (s_{\mathrm{a}} \wedge \neg b^{\mathrm{v}}_{\mathrm{rev},W-k-1..W-1})$ ▷ Maximise such pairs
10: $s_{\mathrm{b}} \leftarrow b^{\mathrm{m}} \wedge \neg a^{\mathrm{m}}_{\mathrm{rev},W-k-1..W-1}$ ▷ Single-unknown k-th c. pairs, 'X' in b
11: $b^{\min} \leftarrow b^{\mathrm{v}} \vee (s_{\mathrm{b}} \wedge a^{\mathrm{v}}_{\mathrm{rev},W-k-1..W-1})$ ▷ Minimise such pairs
12: $b^{\max} \leftarrow b^{\mathrm{v}} \vee (s_{\mathrm{b}} \wedge \neg a^{\mathrm{v}}_{\mathrm{rev},W-k-1..W-1})$ ▷ Maximise such pairs

13: $d \leftarrow a^{\mathrm{m}} \wedge b^{\mathrm{m}}_{\mathrm{rev},W-k-1..W-1}$ ▷ Double-unknown k-th column pairs
14: **if** $\Phi(d) \neq 0$ **then** ▷ At least one double-unknown 2^k pair
15: $i \leftarrow \textsc{Find\_First\_Set}(d)$
16: **if** $\Phi(d) \neq 2^i$ **then** ▷ At least two double-unknown k-th col. pairs
17: $c_k \leftarrow \hat{X}$ ▷ Theorem 5.6.3
18: **continue**
19: **end if**
20: $j \leftarrow k - i$ ▷ Resolve singular double-unknown k-th column pair
21: **if** $2^i\Phi(b^{\min}_{0..j-1}) + 2^j\Phi(a^{\min}_{0..i-1}) \leq 2^k$ **then** ▷ Equation 5.28
22: $a^{\min}_i \leftarrow 1$
23: $b^{\min}_j \leftarrow 1$
24: **end if**
25: **if** $2^j\Phi(a^{\max}_{0..i-1}) \leq 2^i\Phi(b^{\max}_{0..j-1})$ **then** ▷ Equation 5.29
26: $a^{\max}_i \leftarrow 1$
27: **else**
28: $b^{\max}_j \leftarrow 1$
29: **end if**
30: **end if**
31: $h^{*,\min}_k \leftarrow 0$ ▷ Computed $a^{\min}, b^{\min}$, compute minimum of h^*_k
32: $h^{*,\max}_k \leftarrow 0$ ▷ Computed $a^{\max}, b^{\max}$, compute maximum of h^*_k
33: **for** $i \in \{0, \ldots, k\}$ **do** ▷ Compute each row separately
34: **if** $a^{\min}_i = 1$ **then**
35: $h^{*,\min}_k \leftarrow h^{*,\min}_k - (2^k b^{\min}_{k-i}) + (2^i\Phi(b^{\min}_{0..k-i-1}))$
36: **end if**
37: **if** $a^{\max}_i = 1$ **then**
38: $h^{*,\max}_k \leftarrow h^{*,\max}_k - (2^k b^{\max}_{k-i}) + (2^i\Phi(b^{\max}_{0..k-i-1}))$
39: **end if**
40: **end for**
41: **if** $\zeta_k(h^{*,\min}_k) \neq \zeta_k(h^{*,\max}_k)$ **then**
42: $c_k \leftarrow \hat{X}$ ▷ Set result bit unknown
43: **else**
44: $c^{\mathrm{m}}_k \leftarrow 0, c^{\mathrm{v}}_k \leftarrow \zeta_k(h^{*,\min}_k) \bmod 2$ ▷ Set value
45: **end if**
46: **end for**
47: **return** $\hat{c}$
48: **end function**

Upon inspection, it is clear that the computation complexity is dominated by computation of $h^{\min}_k, h^{\max}_k$ and the worst-case time complexity is $\Theta(N^2)$, proving Theorem 5.3.2. Since the loops depend on M which does not change when signed multiplication is considered (only N does), signed multiplication is expected to incur at most a factor-of-4 slowdown when $2N$ fits machine word size, the possible slowdown occurring due to possible splitting of most significant bits of multiplicands (discussed at the start of this section).

5.7 Experimental Evaluation

We implemented the naïve universal algorithm, the fast abstract addition algorithm, and the fast abstract multiplication algorithm in the C++ programming language, without any parallelisation techniques used. In addition to successfully checking equivalence of naïve and fast algorithm outputs for $N \leq 9$, we measured the performance of algorithms with random inputs[1].

To ensure result trustworthiness, random inputs are uniformly distributed and generated using a C++ standard library Mersenne twister before the measurement. The computed outputs are assigned to a volatile variable to prevent their removal due to compile-time optimisation. Each measurement is taken 20 times and the corrected sample standard deviation is visualised.

The program was compiled by **GCC** 9.3.0, in 64-bit mode and with maximum speed optimisation level `-O3`. It was run on a virtual machine supplied by the conference where the original paper [A.1] was published, on an x86-64 desktop system with an AMD Ryzen 1500X processor.

5.7.1 Visualisation and Interpretation

We measured the CPU time taken to compute outputs for 10^6 random input combinations for all algorithms for $N \leq 8$, visualising the time elapsed in Figure 5.1. As expected, the naïve algorithm exhibits exponential dependency on N and the fast addition algorithm seems to be always better than the naïve one. The fast multiplication algorithm dominates the naïve one for $N \geq 6$. The computation time of the naïve algorithm makes its usage for $N \geq 16$ infeasible even if more performant hardware and parallelization techniques were used.

For the fast algorithms, we also measured and visualised the results up to $N = 32$ in Figure 5.2. Fast addition is extremely quick for all reasonable input sizes and fast multiplication remains quick enough even for $N = 32$. Fast multiplication results do not seem to exhibit a noticeable quadratic dependency. We consider it plausible that as N rises, so does the chance that there are multiple double-unknown k-th column pairs for an output bit and it is set to 'X' quickly, counteracting the worst-case quadratic computation time.

Finally, we fixed $N = 32$, changing the independent variable to the number of unknown bits in each input, visualising the measurements in Figure 5.3. As expected, the fast multiplication algorithm exhibits a prominent peak with the easiest instances being all-unknown, as almost all output bits will be quickly set to 'X' due to multiple double-unknown k-th column pairs. Even at the peak around $N = 6$, the throughput is still above one hundred thousand computations per second, which should be enough for model checking usage.

[1]The implementation and measurement scripts are available in an artefact at `https://doi.org/10.6084/m9.figshare.16622983.v1`.

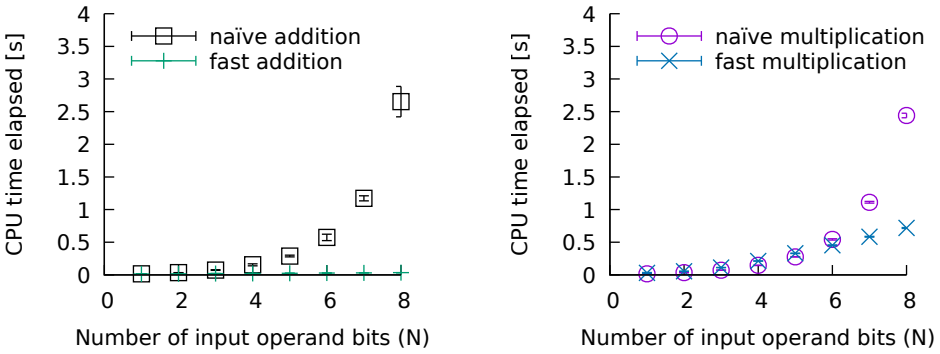

Figure 5.1: Measured computation times for 10^6 random abstract input combinations.

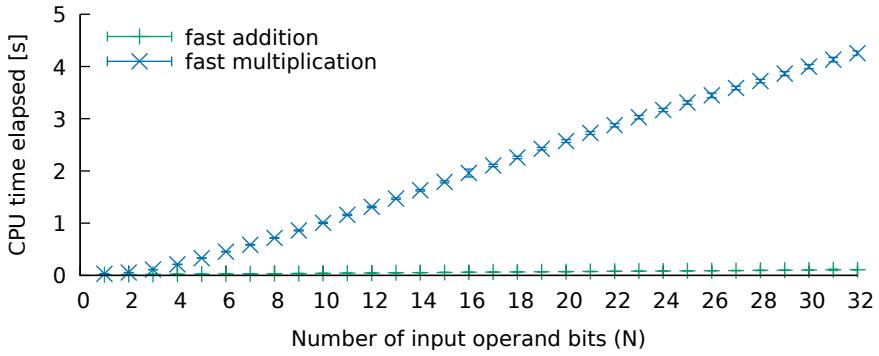

Figure 5.2: Measured computation time for 10^6 random abstract input combinations, fast algorithms only.

In summary, while the naïve algorithm is infeasible for usage even with 16-bit inputs, the fast algorithms remain quick enough even for 32-bit inputs.

5.8 Further Notes

We devised a new *modular extreme-finding technique* for the construction of fast algorithms which compute the best permissible three-valued abstract bit-vector result of concrete operations with three-valued abstract bit-vector inputs when the output is not restricted otherwise (*forward operation problem*). Using the introduced technique, we presented a linear-time algorithm for abstract addition and a worst-case quadratic algorithm for abstract multiplication. We implemented the algorithms and evaluated them experimentally, showing that their speed is sufficient even for 32-bit operations, for which naïve algorithms are infeasibly slow. As such, they may be used to improve the speed of model checkers which use three-valued abstraction.

There are various research paths that could further the results of this chapter. Lesser-used operations still remain to be inspected, most notably the division and remainder operations. Composing multiple abstract operations into one could also potentially reduce

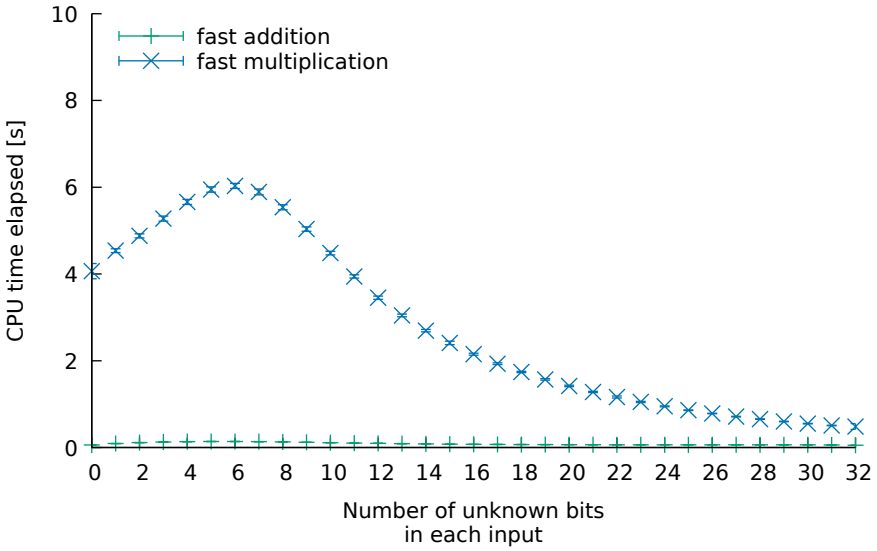

Figure 5.3: Measured computation times for 10^6 random abstract input combinations with fixed $N = 32$, while the number of unknown bits in each input varies.

overapproximation. Most interestingly, the forward operation problem could be augmented with pre-restrictions on outputs, which would allow not only fast generation of the state space in a forward fashion, but its fast pruning as well, allowing fast verification via state space refinement. Furthermore, verification of hardware containing adders and multipliers could be improved as well, e.g. by augmenting Boolean satisfiability solvers with algorithms that narrow the search space when such a structure is found.

Chapter 6

Created Formal Verification Tool machine-check

My publicly available, free, and open-source verification tool **machine-check**[1] is based on the techniques described in Chapters 3, 4, and 5, all of them working in concert to achieve state-of-the-art verification of machine-code systems.

In this chapter, I will first discuss combining the previously introduced techniques to be used in a verification tool without considering implementation specifics. I will then discuss the internal structure of **machine-check**, which uses the Rust language both for its implementation and the simulable descriptions, noting the added complications brought by the choice of the Rust language. Finally, I will present the evaluation of the current version of **machine-check** on machine-code programs written for the AVR ATmega328P microcontroller, using a simulable description of the microcontroller I wrote. I was able to verify interesting properties of several machine-code programs for ATmega328P. Most notably, I was able to find a bug in an oscillator calibration program I wrote during the work on my previous bachelor thesis [A.6], using its simplified version for verification.

6.1 Input-based Three-valued Abstraction Refinement Using Abstraction Analogues

To combine the techniques from Chapters 3 and 4, we need to begin with the fundamental building block, which is the input-based Three-valued Abstraction Refinement framework introduced in Chapter 4 that provides the verification results. The soundness, monotonicity, and completeness characteristics can then be either formally proven or only considered informally, guiding the implementation of the tool, with the possibility of bugs lessened by informal testing. Since I was concerned with creating a useful and easily extensible veri-

[1] The latest release of **machine-check** is available at `https://crates.io/crates/machine-check`. The current release at the time of writing this thesis, which will be discussed and evaluated in this chapter, is available at `https://crates.io/crates/machine-check/0.3.0`.

fication tool, not one that intensely adheres to formalisms, I will only discuss soundness with some degree of formality in this section. In this section, while I will use **machine-check** as an example, I will focus on sensible choices rather than implementation-specific considerations.

Recalling the definition of generating automata from Section 4.4, a generating automaton (GA) is a tuple $G = (S, s_0, I, q, f, L)$ where

- S is the set of automaton states,
- $s_0 \in S$ is the initial state,
- I is the set of all possible step inputs,
- $q : S \to 2^I \setminus \{\emptyset\}$ is the input qualification function,
- $f : S \times I \to S$ is the step function, mapping the combination of the current state and step input to the next state,
- $L : S \times \mathbb{A} \to \{0, 1, \bot\}$ is a labelling function.

Implementing the framework as per Figure 4.4, we choose the model-checking algorithms based on the specification formalism.

Example 6.1.1. In **machine-check**, I chose Computation Tree Logic (CTL) and implemented the model-checking algorithms as discussed in Section 4.6, using a previously introduced algorithm that combines two passes of the classic explicit-state CTL checking algorithm to obtain the three-valued result [68, p. 173-174]. The property is first transformed to a positive normal form with no negations, the negations of atomic properties changed to complementary atomic properties, and the properties are then model-checked using a pessimistic Kripke structure, where the atomic properties with valuation $\bot$ are coerced to 0, and an optimistic Kripke structure, where they are coerced to 1. If the results are the same, that gives the three-valued result. If they differ, the three-valued result is $\bot$.

To connect the framework to the system, we have to provide:

- The concrete generating automaton (CGA) $G = (S, s_0, I, \{(s, I) \mid s \in S\}, f, L)$, which represents the original (concrete) system under verification.
- The initial abstract generating automaton (AGA) $\hat{G}^0 = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}^0, \hat{f}^0, \hat{L})$.
- The algorithm for the refinement of the abstract generating automaton that, in each refinement loop iteration with index $n \in \mathbb{N}_0$, manipulates $\hat{q}^n$ and $\hat{f}^n$ to produce the refined AGA $\hat{G}^{n+1} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}^{n+1}, \hat{f}^{n+1}, \hat{L})$.

Considering the simulable descriptions from Chapter 3, the CGA is directly given by the system comprised of the described finite-state machine (FSM) and the system instantiation parameters (for machine-code systems, the processor behaviour and the machine code, respectively). S contains the FSM states, and I contains the FSM inputs.

The reachable state space is determined by s_0 and f. As the initialisation and step behaviour of digital systems tends to be very different, I chose to provide two functions `init` and `next` for the simulable description FSMs, as shown in Figure 3.3. In addition to the input and, in the case of the `next` function, the current state, the functions take the system instantiation parameters so that the behaviour can change with e.g. different machine code.

Treating s_0 as a dummy start state for simplicity (skipped in model checking) and naming the system instantiation parameters p, I define f as

$$f(s,i) \stackrel{\text{def}}{=} \begin{cases} \texttt{init}(p,i) & s = s_0, \\ \texttt{next}(p,s,i) & s \neq s_0. \end{cases} \tag{6.1}$$

It remains to define the labellings. While any function $L : S \times \mathbb{A} \to \{0, 1, \bot\}$ can be used, it is sensible to at least allow questions about state variables based on relational operators (equality and inequalities). The choice of labellings guides the specification writers, preventing them from writing specifications likely to result in exponential explosion.

Example 6.1.2. In the current version of **machine-check**, the right side of relational operators in the property must be a constant, preventing comparing e.g. equality of two fields, which tends to result in severe exponential explosion with only three-valued bit-vector domains used. In a future version, this limitation could be dropped, allowing for richer specifications.

6.1.1 Abstract Generating Automatons and Soundness

In the input-based framework, a sequence of abstract generating automatons is used as the refinement commences. Each AGA uses the same set of abstract states $\hat{S}$ and the same set of abstract inputs $\hat{I}$, both of which are determined by the abstract domains used, and the same abstract labelling function $\hat{L}$. In other words, in each iteration of the refinement loop $n \in \mathbb{N}_0$, the AGA used is defined as $\hat{G}^n = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}^n, \hat{f}^n, \hat{L})$, where $\hat{q}^n : \hat{S} \to 2^{\hat{I}} \setminus \{\emptyset\}$ and $\hat{f}^n : \hat{S} \times \hat{I} \to \hat{S}$.

Example 6.1.3. In **machine-check**, the currently supported types of state and input variables are bit-vectors and bit-vector arrays. Formally, the variables can be flattened to a single bit-vector. Naming the width of the state bit-vector w and the width of the input bit-vector as y, the sets S and I are defined as

$$\begin{aligned} S &\stackrel{\text{def}}{=} \{(b_0, b_1, \ldots, b_{w-1}) \mid \forall k \in [0, w-1] \,.\, b_k \in \{0,1\}\}, \\ I &\stackrel{\text{def}}{=} \{(b_0, b_1, \ldots, b_{y-1}) \mid \forall k \in [0, y-1] \,.\, b_k \in \{0,1\}\}. \end{aligned} \tag{6.2}$$

Representing '0' by $\{0\}$, '1' by $\{1\}$, and 'X' by $\{0,1\}$, the sets $\hat{S}$ and $\hat{I}$ are defined as

$$\begin{aligned} \hat{S} &\stackrel{\text{def}}{=} \{(\hat{b}_0, \hat{b}_1, \ldots, \hat{b}_{w-1}) \mid \forall k \in [0, w-1] \,.\, \hat{b}_k \in \{\{0\},\{1\},\{0,1\}\}\}, \\ \hat{I} &\stackrel{\text{def}}{=} \{(\hat{b}_0, \hat{b}_1, \ldots, \hat{b}_{y-1}) \mid \forall k \in [0, y-1] \,.\, \hat{b}_k \in \{\{0\},\{1\},\{0,1\}\}\}. \end{aligned} \tag{6.3}$$

Recalling Theorem 4.5.2, soundness of verification using the input-based TVAR framework is ensured if the following three formulas hold for each $n \in \mathbb{N}_0$,

$$\begin{aligned}
&\forall \hat{s} \in \hat{S} \,.\, \forall s \in \gamma(\hat{s}) \,.\, \forall a \in \mathcal{A} \,.\, (\hat{L}(\hat{s}, a) \neq \bot \Rightarrow \hat{L}(\hat{s}, a) = L(s, a)),\\
&\forall (\hat{s}, i) \in \hat{S} \times I \,.\, \exists \hat{i} \in \hat{q}^n(\hat{s}) \,.\, i \in \zeta(\hat{i}),\\
&\forall (\hat{s}, \hat{i}) \in \hat{S} \times \hat{I} \,.\, \forall (s, i) \in \gamma(\hat{s}) \times \zeta(\hat{i}) \,.\, f(s, i) \in \gamma(\hat{f}^n(\hat{s}, \hat{i})),
\end{aligned} \tag{6.4}$$

which we called the *labelling soundness*, *full input coverage*, and *step soundness*, respectively. Labelling soundness is independent of the refinement loop iteration and can be achieved fairly easily by defining a reasonable $\hat{L}$, given a simple labelling function and reasonable abstract domains, and I will thus not discuss it in detail.

Input qualification. Full input coverage is a bit more involved, but not overly so. We will first set $\hat{i}_{\text{initial}} \in 2^{\hat{I}} \setminus \{\emptyset\}$ that will be the result of $\hat{q}^n$ for abstract states where the input qualification was never refined, defining $\hat{q}^0$ as

$$\hat{q}^0 \stackrel{\text{def}}{=} \{(\hat{s}, \hat{i}_{\text{initial}}) \mid \hat{s} \in \hat{S}\}. \tag{6.5}$$

To ensure full input coverage, we require that

$$\forall i \in I \,.\, i \in \zeta(\hat{i}_{\text{initial}}). \tag{6.6}$$

Then, when refining qualified inputs from a given state $\hat{s}_{\text{from}}$, we will set $\hat{q}^{n+1}$ so that it has the same results as $\hat{q}^n$ except for $\hat{q}^{n+1}(\hat{s}_{\text{from}})$, which we will set so that it produces more information. For full input coverage, we again require

$$\forall i \in I \,.\, i \in \zeta(\hat{q}^{n+1}(\hat{s}_{\text{from}})). \tag{6.7}$$

Example 6.1.4. In the naïve strategy for **machine-check** discussed in Section 4.6, for each three-valued bit, the qualified inputs in $\hat{i}_{\text{initial}}$ are '0' and '1'. In the input-splitting strategy (both with and without decay), the only qualified input in $\hat{i}_{\text{initial}}$ is 'X'. Both of these ensure full input coverage, whereas e.g. only '0' would not cover the concrete inputs where the given bit is set to 1.

Step decay. The step function is the crucial part where the framework interacts with the description and translated analogues. To separate our concerns, we will provide a fixed *step behaviour function* $\hat{f} : \hat{S} \times \hat{I} \to \hat{S}$. In every refinement loop iteration n, we will provide a *decay function* $\hat{d}^n : \hat{S} \to \hat{S}$. We define $\hat{f}^n$ as

$$\hat{f}^n \stackrel{\text{def}}{=} \{((\hat{s}, \hat{i}), \hat{d}^n(\hat{f}(\hat{s}, \hat{i}))) \mid \hat{s} \in \hat{S}, \hat{i} \in \hat{I}\}. \tag{6.8}$$

We will require the decay function outputs to cover its inputs, so that it can only either retain the information or decay to an abstract state containing less information, i.e.

$$\forall \hat{s} \in \hat{S} \,.\, \gamma(\hat{s}) \subseteq \gamma(\hat{d}(\hat{s})). \tag{6.9}$$

Using Equation 6.9, we can strengthen the step soundness requirement from Equation 6.4 to use $\hat{f}$ instead of $\hat{f}^n$, i.e.

$$\forall(\hat{s},\hat{i}) \in \hat{S} \times \hat{I} \,.\, \forall(s,i) \in \gamma(\hat{s}) \times \zeta(\hat{i}) \,.\, f(s,i) \in \gamma(\hat{f}(\hat{s},\hat{i})). \tag{6.10}$$

As such, we managed to separate the step behaviour described by $\hat{f}$ from the concern of mitigating the state-space explosion (and the subsequent refinement) in $\hat{d}^n$. While the input-based framework allows us to keep the concerns mixed together, their separation allows for easier reasoning.

Example 6.1.5. In the **machine-check** strategy of input-splitting with decay discussed in Section 4.6, the initial decay function d^0 decays each three-valued bit to 'X'. The decay function can later be refined not to decay bits determined to be important. This prevents irrelevant computations from causing state-space explosion but can increase the number of necessary refinement loop iterations drastically. In the other strategies discussed in Section 4.6, the decay function is an identity function in every refinement loop iteration.

Step behaviour. The step behaviour function $\hat{f}$ should be computable reasonably fast while fulfilling Equation 6.10. The function $f : S \times I \to S$ is defined in terms of the description functions `init` and `next`, so we will define the function $\hat{f} : \hat{S} \times \hat{I} \to \hat{S}$ as

$$\hat{f}(\hat{s},\hat{i}) \stackrel{\text{def}}{=} \begin{cases} \hat{\texttt{init}}(\hat{p},\hat{i}) & \hat{s} = \hat{s}_0, \\ \hat{\texttt{next}}(\hat{p},\hat{s},\hat{i}) & \hat{s} \neq \hat{s}_0. \end{cases} \tag{6.11}$$

The abstract system instantiation parameters $\hat{p}$ can be produced from p by simply converting the concrete variables to single-concretization abstract variables (assuming single-concretization abstractions exist in the domain). The functions $\hat{\texttt{init}}$ and $\hat{\texttt{next}}$ are verification analogues of the functions `init` and `next` used for computing abstract states (they are the *abstract analogues* of the concrete functions). Rewriting step soundness from Equation 6.10, we require that

$$\begin{aligned} &\forall\hat{i} \in \hat{I} \,.\, \forall i \in \zeta(\hat{i}) \,.\, \texttt{init}(p,i) \in \gamma(\hat{\texttt{init}}(\hat{p},\hat{i})), \\ &\forall(\hat{s},\hat{i}) \in (\hat{S} \setminus \{\hat{s}_0\}) \times \hat{I} \,.\, \forall(s,i) \in \gamma(\hat{s}) \times \zeta(\hat{i}) \,.\, \texttt{next}(p,s,i) \in \gamma(\hat{\texttt{next}}(\hat{p},\hat{s},\hat{i})). \end{aligned} \tag{6.12}$$

For each function in the simulable description $g : V \to R$, its abstract analogue will have the appropriate abstract domains $\hat{V}$ and $\hat{R}$ with concretization functions $\gamma_V : \hat{V} \to 2^V \setminus \{\emptyset\}$ and $\gamma_R : \hat{R} \to 2^R \setminus \{\emptyset\}$, resulting in $\hat{g} : \hat{V} \to \hat{R}$. We can strengthen the requirement in Equation 6.12 to require each such function to fulfil

$$\forall\hat{v} \in \hat{V} \,.\, \forall v \in \gamma_V(\hat{v}) \,.\, g(v) \in \gamma_R(\hat{g}(\hat{v})). \tag{6.13}$$

This is a very nice and natural requirement for the abstract analogues of the functions in the simulable description. Furthermore, we can see that for bit-vectors, the requirement corresponds to the concept of the approximate abstract transformer from Subsection 5.2.2. As the signatures of the functions do not change except for the types, we can translate to abstract analogues by simply translating the types in the simulable description to their abstract domains, ensuring that Equation 6.13 holds for each one.

6.1.2 The Refinement Algorithm

While the refinement algorithm has no impact on soundness, it is crucial for the reduction of state space explosion. As such, we want to choose intelligently, deducting how the AGA should be changed so that we strike a balance between the state space size and refinement speed. In other words, we need a good heuristic.

An unknown result of verification will be caused by some culprit, a path that ends with an unknown labelling that prevents a known result of model-checking. We are trying to refine $\hat{q}^n$ and $\hat{d}^n$ to $\hat{q}^{n+1}$ and $\hat{d}^{n+1}$ so that the culprit will hopefully disappear, with the offending state replaced by states where the labelling is known.

For practical systems, we can purely structurally deduce that many of the inputs in $\hat{q}^n$ and decayed parts of states in $\hat{d}^n$ cannot cause the culprit labelling to be unknown, because they do not act as inputs of any operations that play a role in the part of the state responsible for the labelling result. As such, we can use a fairly simple *marking* algorithm for the refinement. After finding the culprit with path $(\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_c)$:

1. Mark the variables of the last state of the culprit $\hat{s}_c$ that can have an effect on the unknown labelling.
2. Set k equal to c.
3. If k is zero or $\hat{s}_k$ is fully unmarked, stop.
4. Mark the decay of variables in $\hat{d}^n(\hat{f}(\hat{s}_{k-1}))$ that could have had an effect on the unknown labelling.
5. Mark through $\hat{f}$ backwards, starting with marked parts of $\hat{s}_k$, marking the inputs of operations in $\hat{f}$ that could have had an effect on marked operation outputs, until obtaining the marking of $\hat{s}^{k-1}$ and the output of $\hat{q}^n(\hat{s}^{k-1})$.
6. Decrement k and go to Step 3.

After stopping, we will have the candidates for refinement of $\hat{q}^n$ and $\hat{d}^n$ marked and can choose some of them to perform the actual refinement, during which we ensure that Equations 6.7 and 6.9 hold to avoid loss of soundness.

Example 6.1.6. Let us consider an example system that reads a Boolean value v from the input during initialisation and later uses it after an initially-zeroed counter t counts to 3, disregarding the inputs after initialisation:

$$\begin{aligned} &\texttt{init\_ex}(p, i) = (0, i), \\ &\texttt{next\_ex}(p, (t, v), i) = (\min(t + 1, 3), v). \end{aligned} \tag{6.14}$$

Let us assume that we use a three-valued abstraction for v. We want to verify that it is possible to reach a state where t is 3 and v is 1, i.e. $\mathbf{EF}[t = 3 \wedge v = 1]$. Using the input splitting strategy, we first construct the reachable state space as in Figure 6.1a).

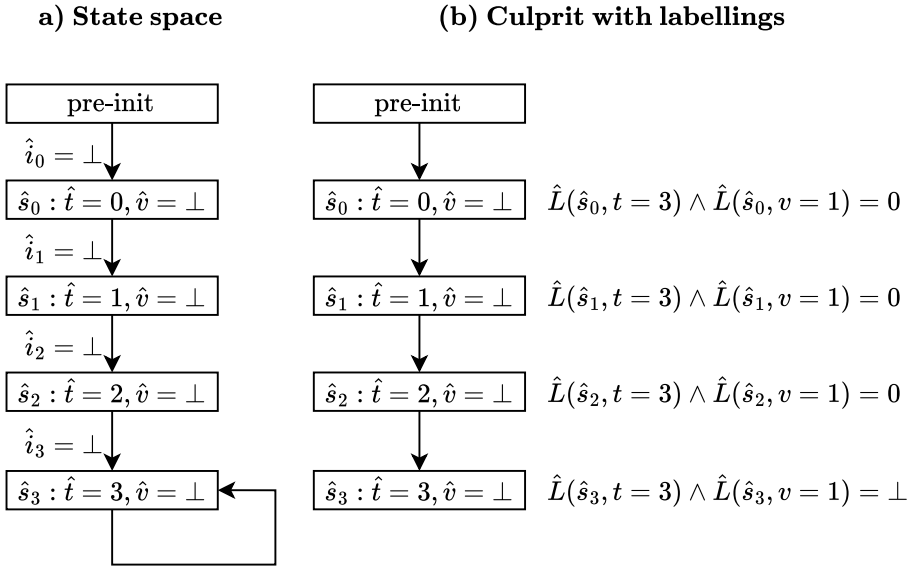

Figure 6.1: An example of a lasso-shaped state space before refinement and the culprit.

Model-checking produces the result $\bot$. If, instead of $\hat{s}_3$, we reached another state $\hat{s}$ where $\hat{L}(\hat{s}, v = 1) \neq \bot$, the property could have been determined to hold. As such, we want to make sure that the culprit shown in Figure 6.1b), the path $(\hat{s}_0, \hat{s}_1, \hat{s}_2, \hat{s}_3)$ with the labelling $\hat{L}(\hat{s}_3, v = 1) = \bot$, is no longer present in the state space after refinement. As for labellings, $\hat{L}(\hat{s}_3, v = 1)$ only depends on $\hat{v}$. Therefore, starting the marking loop:

1. We mark $\hat{v}$ in $\hat{s}_3$.
2. There is no decay, so we mark backwards through $\hat{\texttt{step\_ex}}$, where the marked $\hat{v}$ only depends on $\hat{v}$ in $\hat{s}_2$. Therefore, we end up with marked $\hat{v}$ in $\hat{s}_2$.
3. We mark backwards twice in the same fashion, marking $\hat{v}$ in $\hat{s}_1$ and later in $\hat{s}_0$.
4. We now have $\hat{v}$ marked in $\hat{s}_0$. Since $\hat{s}_0$ was constructed using the $\hat{\texttt{init\_ex}}$ function, we mark backwards through $\hat{\texttt{init\_ex}}$, marking the input $\hat{i}_0$.
5. We are done with marking. The only marked input is $\hat{i}_0$, so we choose it for refinement.

After refinement, it will be possible to verify that $\mathbf{EF}[t = 3 \wedge v = 1]$ holds.

The marking algorithm can be further improved not to deduce only structurally, but to also consider the abstract values of the inputs of the abstract operations in $\hat{f}$. For example, with reasonable decisions for the generating automatons, it is unnecessary to mark variables with a single-concretization abstract value, as no refinement of the variables they are dependent on will refine the single-concretization abstract value further.

Example 6.1.7. Let us consider that a logical AND is performed with two abstract values. It is reasonable to ensure that if both have a single concretization, we produce a result with a single concretization, corresponding to the concrete logical AND operation. After we ensure this, when performing the logical AND where only one of the abstract values has a single concretization, we can mark only the value with multiple concretizations, knowing that the single-concretization value is not the one that may cause a result with multiple concretizations. This will be later expanded upon in Subsection 6.2.1.

To combine the improved marking algorithm with the translation of simulable descriptions, in addition to the abstract analogues, we will also create verification analogues to structures and functions that will make it possible to execute the marking algorithm, called the *refinement analogues*.

6.2 Translation to Abstraction and Refinement Analogues

I will now show how to perform translation to abstraction and refinement analogues. I will assume that the underlying operations for the variables are implemented, and will focus on how to rewrite the description code.

6.2.1 Functions without Control Flow

Let us first consider the function FN1 in Figure 6.2, which simply performs a logical AND of its two inputs and returns the output. As discussed in Subsection 6.1.1, it is enough to convert the types and ensure that the function FN1_ABSTR fulfils Equation 6.13. Clearly, with no control flow, it should be enough to ensure that each called function fulfils the equation. The abstraction analogues of functions in the description must fulfil it, so we can call them, and it is only necessary to consider the operations on the abstract data types such as $\hat{a}\hat{\&}\hat{b}$.

Example 6.2.1. For variables $\hat{a}$ and $\hat{b}$ in a three-valued bit-vector domain, we can use the three-valued logic for logical operations and the algorithms introduced in Chapter 5 for arithmetic operations. For example, in the abstract analogue, the function FN1_ABSTR called with arguments $\hat{a}$ = "XXXXXXXX" and $\hat{b}$ = "00001111" will produce "0000XXXX".

The refinement analogue, named FN1_REFIN in Figure 6.2, performs backwards marking, using abstract variable values for added deductive capability. The refinement analogues of the `init` and `next` functions are called in Step 5 of the refinement algorithm in Subsection 6.1.2. To ensure that functions can call each other within the refinement analogues, each function has a refinement analogue, similarly to the abstract analogues.

Similarly to FN1_ABSTR, FN1_REFIN has two inputs $\hat{a}$ and $\hat{b}$, which are used to allow marking based on abstract variable values. The third input is $\hat{c}_{\text{mark}}$, which contains the marking of the result of FN1_ABSTR. The task is to compute the markings of the inputs $\hat{a}$ and $\hat{b}$ of FN1_ABSTR.

function FN1(a, b)
  $c \leftarrow a \& b$ ▷ Compute the logical AND
  return c ▷ Return the result
end function

function FN1_ABSTR$(\hat{a}, \hat{b})$
  $\hat{c} \leftarrow \hat{a} \hat{\&} \hat{b}$ ▷ Compute the logical AND in abstract domain
  return $\hat{c}$ ▷ Return the result
end function

function FN1_REFIN$(\hat{a}, \hat{b}, \hat{c}_{\text{mark}})$
  $\hat{c} \leftarrow \hat{a} \hat{\&} \hat{b}$ ▷ Compute the abstract variables
  $\hat{a}_{\text{mark},0} \leftarrow$ UNMARKED ▷ Have all markings except $\hat{c}_{\text{mark}}$ unmarked
  $\hat{b}_{\text{mark},0} \leftarrow$ UNMARKED
  $(\hat{a}_{\text{mark},1}, \hat{b}_{\text{mark},1}) \leftarrow$ MARKLOGICALAND$(\hat{a}, \hat{b}, \hat{c}_{\text{mark}})$ ▷ Compute operation markings
  $\hat{a}_{\text{mark},2} \leftarrow$ JOINMARKINGS$(\hat{a}_{\text{mark},0}, \hat{a}_{\text{mark},1})$ ▷ Join them with previous markings
  $\hat{b}_{\text{mark},2} \leftarrow$ JOINMARKINGS$(\hat{b}_{\text{mark},0}, \hat{b}_{\text{mark},1})$
  return $(\hat{a}_{\text{mark},2}, \hat{b}_{\text{mark},2})$ ▷ Return marks of the original inputs
end function

Figure 6.2: A function without control flow and its abstract and refinement analogues.

As we want to use the abstract variables for the computation of markings, we compute the values of the non-input abstract variables first (here, $\hat{c}$). We then consider all markings except for the result marking to be unmarked at first and start marking in a backward order of operations. Since variables can be used multiple times as operation inputs, it is necessary to join all the markings introduced by operations. At the end, markings of the abstract inputs are returned.

Example 6.2.2. Using three-valued bit-vector abstraction, it is reasonable to mark each bit separately. Let us continue from Example 6.2.1 and suppose that we marked all bits of the output of FN1_ABSTR, which I will write as $c_{\text{mark}} = 11111111_2$, and we want to propagate the marking to its inputs.

Deducing mentally from FN1_ABSTR with arguments $\hat{a}$ = "XXXXXXXX" and $\hat{b}$ = "00001111", we see that that b is fully known, so it does not need to be considered further. We put $b_{\text{mark}} = 00000000_2$. While $\hat{a}$ is fully unknown, considering the logic AND operation, the upper four bits of $\hat{a}$ could not have caused the problem as the zeroed upper four bits of $\hat{b}$ ensure they have no impact. Therefore, we put $a_{\text{mark}} = 00001111_2$. That is also what FN1_REFIN will return, provided the functions MARKLOGICALAND and JOINMARKINGS are implemented reasonably.

```
function FN2(a, b)
    if a ≤ b then                                   ▷ Compute the minimum
        c ← a
    else
        c ← b
    end if
    return c                                        ▷ Return the minimum
end function

function FN2_ABSTR(â, b̂)
    ŵ ← â ≤ b̂                                       ▷ Three-valued condition
    if CanBeTrue(ŵ) then
        ĉ1 ← Taken(â)                               ▷ the then branch can be taken
    else
        ĉ1 ← NotTaken                               ▷ the then branch cannnot be taken
    end if
    if CanBeFalse(ŵ) then
        ĉ2 ← Taken(b̂)                               ▷ the else branch can be taken
    else
        ĉ2 ← NotTaken                               ▷ the else branch cannot be taken
    end if
    ĉ ← φ(ĉ1, ĉ2)                                   ▷ Use the φ function
    return ĉ                                        ▷ Return the result
end function
```

Figure 6.3: A function with branching and its abstract analogue.

6.2.2 Functions with Conditional Branches

Control flow is a notable complication to translation to the abstract analogue: the underlying description language only supports concrete control flow (e.g. exactly one branch is taken in conditional branch statements), not abstract control flow (where both branches can be taken). Therefore, the constructs must be rewritten. As I will only consider conditional branches, we can cleanly resolve this with some inspiration from static program analysis [83, 84].

Let us consider the function FN2 in Figure 6.3, where the value of c depends on the branch taken. Since the value of $\hat{a} \leq \hat{b}$ is a Boolean, its most reasonable abstraction is a three-valued Boolean (functionally equivalent to a single-bit three-valued bit-vector). Of course, our description language cannot branch based on that, so we duplicate the branches: one of them will be taken depending on if $\hat{a} \leq \hat{b}$ can be true, the other depending on if it can be false. To ensure that $\hat{c}$ has the correct value afterwards, we will combine the values assigned to in the duplicate branches, combining them using a phi function [83]. To avoid the need for a special abstract domain value signifying that the branch was not taken, we

can wrap the value in an enumeration that will either be `Taken` (with the given value) or `NotTaken` (with no value).

While the construction of the abstract analogue is heavily complicated by control flow, the refinement analogue is already based on the abstract analogue and it is therefore almost unaffected. It is, however, necessary to mark the condition variable if the branch taken could have affected some later variable that is marked.

Example 6.2.3. Continuing with three-valued bit-vector abstraction, let us consider that FN2_ABSTR is called with arguments $\hat{a}$ = "X0000000" and $\hat{b}$ = "00001111". The condition $\hat{w}$ is 'X' and therefore, $\hat{c}_1$ = `Taken`("X0000000") and $\hat{c}_2$ = `Taken`("00001111"). As both are taken, they will be combined by the ϕ function to "X0001111". In this instance, the marking will be propagated back to $\hat{a}$, but if $\hat{c}_1$ was instead set e.g. to a constant, it would not be. As such, it is necessary to ensure the marking is propagated to $\hat{a}$ through $\hat{w}$ if $\hat{c}$ is marked.

6.3 Implementation Specifics

I chose to implement **machine-check** in the compiled programming language Rust and represent the descriptions in Rust as well, motivated by many factors[2] including

- its meta-programming support,
- the similarity to the ubiquitously used C language in simpler constructs but simpler syntax without many pitfalls,
- its applicability to embedded programming making it worthwhile for processor description writers to learn,
- the availability of fast standard containers and well-supported libraries for e.g. Rust abstract syntax tree parsing.

The choice to use the Rust language heavily improved the speed of development compared to my previous model checker written in the C++ language [A.4]. Furthermore, the verification analogues are compiled, allowing the use of compiler optimisations for quicker verification. For the main focus, which is machine-code verification, the compilation is not problematic as the processor description is compiled once and the machine code is provided as a parameter to the resulting executable, as discussed in Chapter 3.

To solve the problems of library and binary distribution in languages such as C, the Rust language features a built-in package system. A package contains at most one *library crate* and an arbitrary number of *binary crates* and can be published in a public package repository, identified by its name. The default package repository for Rust is `crates.io`, where the crates comprising **machine-check** are published.

[2]More details on Rust can be found e.g. in the Rust language book or the Rust reference, both available online from `https://www.rust-lang.org/learn`.

The package organisation has allowed me to expose the types and functions available to the description writer in the `machine-check` package (which only contains a library crate since it has to be combined with a system description and construction), while the implementation details are hidden in other packages. The interface is as streamlined as possible, consisting of custom data types, macros `machine_description` and `bitmask_switch`, the `run` function yield the constructed system to **machine-check**, and other functions to support the construction of systems based on command-line arguments.

The internal implementation of **machine-check** is split between three basic concepts, organised in three Rust packages:

- **State space generation and model checking** (`machine-check-exec`). Implements an instance of input-based Three-valued Abstraction Refinement framework described in Chapter 4, with explicit abstract states.
- **Abstraction and refinement domains** (`mck`[3]). Implements the abstraction and refinement analogues of bit-vectors and bit-vector arrays. Fast bit-vector arithmetic described in Chapter 5 is implemented here.
- **Translation to verification analogues** (`machine-check-machine`). Implements the translation, described in Chapter 3 and Section 6.2, in the `machine_description` macro.

The lines of code grouped by the concepts can be seen in Figure 6.4. It can be seen that the translation is the most demanding, followed by the implementation of the abstraction and refinement domains. The implementation of state space generation and model checking is third by far. Common third-party Rust libraries with permissive licences are used when possible, such as for abstract trees of Rust code (the `syn` crate) or command-line argument parsing (the `clap` crate), so that the least amount of custom code is necessary, increasing speed of development and reducing support baggage.

6.3.1 Resolution of Introduced Complications

While the Rust language constructs are fairly regular compared to C, the description code in the `machine_description` macro still must be coerced to a Single Static Assignment form with simple constructs before performing the translation to abstraction and refinement analogues, which is complicated by some issues.

Panic. In Rust, the concept of *panic* forms a deviation from function behaviour following their signature. Panic can occur everywhere and results in program termination by default. Panics are highly useful in practice, allowing immediate termination due to an unexpected situation or a detected bug, and they are useful for processor descriptions as well: for example, calling an illegal instruction can result in a panic. In **machine-check**,

[3]The package name `mck` was used instead of the more conventional name `machine-check-types` as it appears ubiquitously in translated system descriptions, and a short name improves their readability and reduces their length drastically.

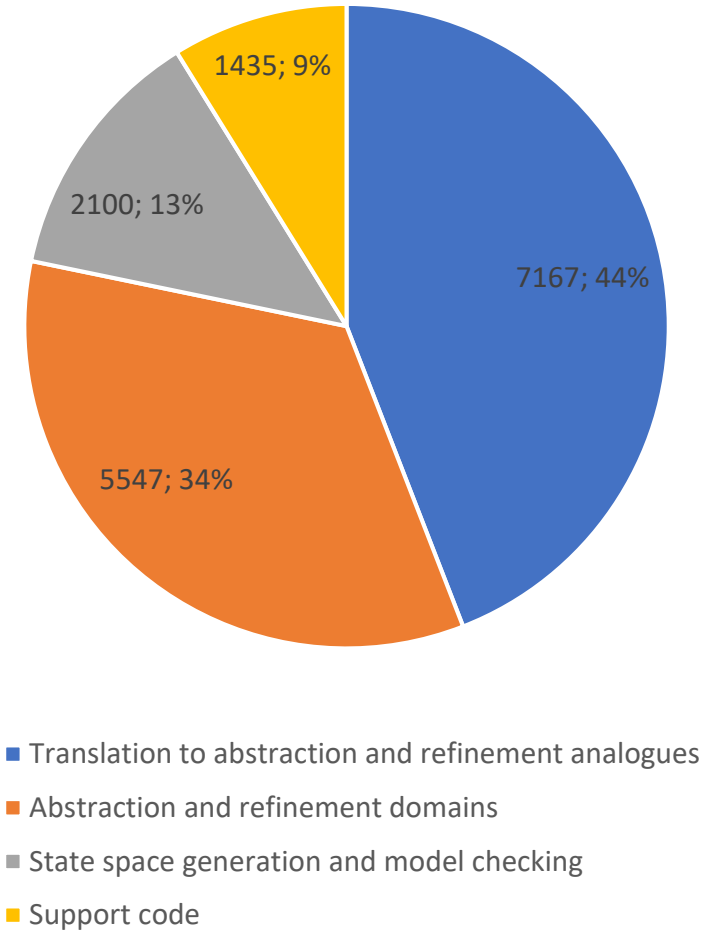

Figure 6.4: Categories of lines of Rust code in **machine-check** version 0.3.0. There are 16249 lines of code in total. Blank and comment lines are not counted.

I decided to support panics by always checking an *inherent property* of the system $\mathbf{AG}\phi$, where ϕ means no panic is issued in the given state. That way, verification of illegally-formed systems (e.g. machine-code systems with illegal machine-code instructions) always produces a special inherent violation result. To implement panic checking, I change the return value of every function before translation to include the information about whether panic was raised and rewrite function calls to propagate the information.

Inside macro expansion. As the Rust macro model guarantees that macros are expanded outside-in and there is currently no way to arbitrarily expand the macros inside, the expansion of macros inside `machine_description` must be done manually. This means that only the `bitmask_switch` macro is supported, together with some simple standard Rust macros invoking panics (`panic`, `unimplemented`, `todo`). In conjunction with the macro expansion, use declarations are resolved, so e.g.

```
1 use ::machine_check::Bitvector;
2 fn example_3(a: Bitvector<8>) {}
```

turns into

```
1 use ::machine_check::Bitvector;
2 fn example_3(a: ::machine_check::Bitvector<8>>) {}
```

After macro expansion and use resolving, language constructs are normalised, variables are uniquely named (eliminating the need for scopes) and converted into the Single Static Assignment (SSA) form which is easy to work with. As Rust variable types do not have to be explicitly stated, being inferred otherwise, basic type inference is done before finishing. Panic conversion is handled throughout.

The chosen strategy of meta-programming with macros is remarkably drop-in: standard Rust code can be written with little regard to formal verification. Incorrect Rust code will produce an error, as will code that cannot be translated, with a more-or-less helpful code span and reason. Only a simple imperative subset of Rust code (described in Section 3.3) is currently supported, but more features can be supported in the future for simpler and more elegant descriptions.

Bit-vector arrays. While implementing the operations for three-valued bit-vectors and bit-vector marking is fairly simple, a problem that can slow down the verification drastically arises when indexing arrays. In case the index variable is partially or even fully unknown, a large amount of elements must be considered during the read operations (which must join all of them to obtain the result) and the write operations (which must, for all of the elements, join the previous value and the written value, as it is unknown whether the specific element will be actually written to). As elements with the same values can be considered together, I implemented the arrays by only storing the leftmost elements with the same value in an indexed map and operating on ranges of the same elements, avoiding many unnecessary computations in practice.

Candidate importance. For machine-code systems, it is typically desirable to refine the Program Counter before any other variables. While the concept of the Program Counter is unknown to **machine-check**, I was successful in achieving the behaviour generally, by adding an importance counter to variable markings. When marking an indexing operation with a not-fully-known index, the index is marked with an incremented importance. The candidate for refinement with the highest importance is selected at the end of the refinement algorithm. As the Program Counter is typically used to index the program memory in order to retrieve the instruction to be executed, this simple improvement is enough for reasonable refinement choices in simple machine-code programs. Of course, further improvements may be added in the future.

6.4 Verification of AVR Programs

To show the feasibility of using **machine-check** for verification of machine-code programs on actual processors, I have written a **machine-check** description of the 8-bit micro-controller AVR ATmega328P, and compiled it to create the tool **machine-check-avr**. I evaluated the verification capability using simple programs, experimentally verifying some

of their interesting properties[4]. Notably, I was able to find a bug in a real-world program I have written previously by verifying its simplified version.

6.4.1 Description Details and Evaluation Setup

The system description consists of approximately 3000 lines, out of which approximately 2000 are lines of Rust code, the rest are comments and blank lines. The description could be made more compact in the future by increasing the amount of constructs that can be translated. That said, the AVR instruction set itself contains around a hundred instructions [13], the exact number depending on whether instructions sharing operation codes are counted multiple times, and whether variants of the same instruction with different behaviour are counted separately. The description is limited: enabling interrupts is not supported, and only the General-Purpose I/O (GPIO) peripheral is supported. Using an unimplemented feature or an illegal operation (such as the execution of an illegal instruction) results in a panic.

A distinct complication to describing real-world processors is the presence of memory addresses that have special behaviour when read or written, usually as parts of memory-mapped peripherals. For example, in ATmega328P, the GPIO peripheral address `PINB` is usually used for reading the state of the pins on the microcontroller port B, but writing to it toggles the output values of the pins where the bit value 1 is written. This behaviour can be described easily using **machine-check** descriptions.

The description was written without considerations for formal verification with one exception: applying the `xor` instruction on the same register with itself is a common way to set it to zero, but is problematic for verification using only three-valued bit-vector abstraction, so I added a kludge that immediately sets the register to zero in this case.

The length of the system description in the macro `machine_description` is not problematic for the Rust compiler, but the **rust-analyzer** extension of **Visual Studio Code**, which I use for development, produces an error due to too many tokens generated by the macro[5]. This means that a manual compilation is necessary after modifying the description instead of background compilation as the code is written, which reduces ease of development. This problem could be resolved in the future by e.g. reducing the number of tokens after translating or improving support for splitting the translation into multiple `machine_description` macros.

The evaluation of **machine-check-avr** was performed on a machine with the Ryzen 5600 processor in a Linux virtual machine with 8 GB of 3200 MT/s RAM available. The programs were compiled/assembled using Microchip Studio 7.0.132. The tool was built in release configuration using Rust 1.80.0. Building from a clean slate, the core **machine-check** libraries were built in 18 seconds, and building **machine-check-avr** after that

[4]The programs, the evaluation script containing the properties, and the reference measurements are available at `https://doi.org/10.5281/zenodo.13377030`.

[5]I use the version 0.3.2029 of **rust-analyzer**. The same error is described in `https://github.com/rust-lang/rust-analyzer/issues/10855`, and it seems the token limit currently cannot be changed.

took 59 seconds. The built **machine-check-avr** executable can verify properties of ATmega328P machine-code programs. The name of the machine-code Intel HEX file and the property to be verified are supplied on the command line. For evaluation, the default strategy of input splitting with no decay was used.

6.4.2 Toy Programs

In my diploma thesis [A.4], I evaluated my previous model checker on simple toy programs, and it was able to verify some of their action-reaction deadline properties [A.4, p. 49-60]. Concisely, the programs are:

- **Basic branch.** Checks the value of an input pin and sets the output pin value accordingly.
- **Blink.** Toggles an output pin (using `PINB`) with a 5-millisecond delay between successive toggles.
- **Gate array.** Emulates five classic logic gates (buffer, inverter, AND, OR, XOR) using GPIO.
- **Switch with momentary selection.** A mode input determines if the primary input should behave as a momentary or toggle switch of the output pin. Inspired by microcontroller-assisted relay switching schemes for e.g. guitar pedals.
- **Independent nondeterminism.** Uses single-bit branches to set different register bits in succession, providing a verification challenge to the previous tool due to the amount of non-determinism.

With the exception of the basic branch program, the toy programs were implemented in assembly language. The basic branch program was implemented in C and is compiled to different machine code in the debug and release configurations, so I used both for verification. As the deadline properties in the previous checker are incomparable to CTL properties, I have used these CTL properties for verification with **machine-check-avr**:

- **Reachability of the loop start with appropriate GPIO direction settings.** $\mathbf{AF}[(\mathrm{PC} = w) \wedge a \wedge b]$, where w is the program counter at the start of the main program loop, a represents the equality of all used GPIO direction registers to their intended values, and b represents the equality of all used GPIO output registers to their intended values at the start of the program loop.
- **Invariant lock.** $\mathbf{AG}[a \Rightarrow \mathbf{AG}[a]]$. Once the used GPIO direction registers are set to the appropriate values, they never change.
- **Recovery.** $\mathbf{AG}[\mathbf{EF}[(\mathrm{PC} = w) \wedge b]]$. It is always possible to return to the start of the program loop with the used GPIO output registers set to their intended values at the start of the program loop.

Table 6.1: Measurements of machine-code verification of toy programs using **machine-check-avr**. For the states and transitions, the first number shows the total generated number, while the second number shows the number in the final state space.

Program name	Property name	Result	Refin.	States	Transitions	CPU time [s]	Memory[MB]
Basic branch (debug)	Inherent	✓	2	22 / 22	26 / 25	<0.01	6.26
Basic branch (debug)	Reachability	✓	0	13 / 13	14 / 14	<0.01	3.75
Basic branch (debug)	Invariant lock	✓	2	23 / 23	28 / 26	0.01	6.20
Basic branch (debug)	Recovery	✓	2	23 / 23	28 / 26	<0.01	6.15
Basic branch (release)	Inherent	✓	2	22 / 22	26 / 25	<0.01	6.24
Basic branch (release)	Reachability	✓	0	13 / 13	14 / 14	<0.01	3.75
Basic branch (release)	Invariant lock	✓	2	23 / 23	28 / 26	0.01	6.15
Basic branch (release)	Recovery	✓	2	23 / 23	28 / 26	<0.01	6.20
Blink	Inherent	✓	0	513 / 513	514 / 514	<0.01	4.46
Blink	Reachability	✓	0	513 / 513	514 / 514	<0.01	4.56
Blink	Invariant lock	✓	0	513 / 513	514 / 514	<0.01	4.32
Blink	Recovery	✓	0	513 / 513	514 / 514	<0.01	4.32
Gate array	Inherent	✓	987	4796 / 4030	6761 / 5018	11.56	14.32
Gate array	Reachability	✓	0	8 / 8	9 / 9	<0.01	3.76
Gate array	Invariant lock	✓	987	4796 / 4027	6771 / 5015	13.39	14.66
Gate array	Recovery	✓	987	4798 / 3828	6773 / 4816	12.19	14.13
Independent nondet.	Inherent	✓	384	1228 / 909	1987 / 1294	2.7	8.63
Independent nondet.	Reachability	✓	0	5 / 5	6 / 6	<0.01	3.67
Independent nondet.	Invariant lock	✓	384	1228 / 910	1997 / 1295	2.81	8.59
Independent nondet.	Recovery	✗	193	624 / 474	1011 / 668	1.3	7.65
Momentary selection	Inherent	✓	29	1878 / 1858	1930 / 1888	1.84	9.40
Momentary selection	Reachability	✓	0	7 / 7	8 / 8	<0.01	3.77
Momentary selection	Invariant lock	✓	29	1879 / 1859	1938 / 1889	1.87	9.66
Momentary selection	Recovery	✓	29	1879 / 1860	1938 / 1890	1.78	9.52

The inherent property of the machine-code system, ensuring that no panics occur, is verified separately from other properties. While verifying the other properties, it is assumed that the inherent property holds.

Note 6.4.1. In the independent nondeterminism program, no output was used, so instead of the output value register, the relevant working register was used for verification.

The results for the toy programs are shown in Table 6.1. All of the properties were verified in fairly little time and memory. Only the verification result of the recovery of the independent nondeterminism program is false, which I investigated. As the program zeroes the working register and conditionally performs inclusive-OR to it in the program loop, recovery to a zeroed working register can be impossible, so the verification result is correct.

```c
#include <avr/io.h>
int factorial(uint8_t n) {
    if (n == 0) {
        return 1; // factorial of 0 is 1
    }
    return n * factorial(n - 1); // compute the factorial recursively
}
int main(void) {
    DDRD |= 0xFF; // set port D as output
    while (1) {
        // get the value 0-7 from the lower 3 bits of port B
        uint8_t read_value = PINB & 0x07;
        // write the factorial result to port D
        PORTD = factorial(read_value);
    }
}
```

Figure 6.5: The factorial program, written in C.

6.4.3 Factorial: Stack Overflow Avoidance

To show a more interesting application of machine-code verification, I wrote a program that computes the factorial of an input number between 0 and 7, shown in Figure 6.5. The output overflows above $5! = 120$, but the focus here is on recursion behaviour rather than the output. The program uses recursive calls, which grow the program stack, threatening to overwrite other variables located in memory (*stack overflow*). It is impossible to verify that stack overflow cannot occur using source-code verification unless the verification is tightly integrated into the compiler. A machine-code verifier, on the other hand, can verify the impossibility of stack overflow exactly using the property $\mathbf{AG}[t]$, where t determines the stack pointer position.

In the AVR architecture, there is a single stack that grows downwards, typically from the end of the memory, and the Stack Pointer (SP) is pre-decremented and post-incremented, i.e. it always points to the byte below the innermost stack byte [13]. As such, it is possible to verify that the stack overflow does not occur by ensuring $\mathrm{SP} \geq v$, where v is the highest non-stack variable byte. As the Stack Pointer is 16-bit but retained in two 8-bit registers SPL (Stack Pointer Low) and SPH (Stack Pointer High) on AVR, $\mathrm{SP} \geq v$ can be rewritten to $\mathrm{SPH} > v_{\mathrm{high}} \vee (\mathrm{SPH} = v_{\mathrm{high}} \wedge \mathrm{SPL} \geq v_{\mathrm{low}})$. Using binary search, it is possible to find the highest value of v where no stack overflow occurs.

The results of verification of the factorial program are shown in Table 6.2. In addition to the properties verified in the toy programs, I also verified stack properties. The compiled machine code uses the recursive calls in the debug target without optimisation. In the release target, the factorial function is transformed into iterative computation in the resultant machine code, which reduces the maximum stack size.

The inherent, reachability, and invariant lock properties hold, which is expected. It is also expected that recovery is impossible: the output is zero at the start of the main program loop, but non-zero afterwards, and it is impossible for the output to become zero again, even though the output value is factorial modulo 256: factorials up to $5! = 120$ are non-zero modulo 256 trivially, $6! \bmod 256 = 208$, and $7! \bmod 256 = 176$. I found the

Table 6.2: Measurements of machine-code verification of the factorial program using **machine-check-avr**.

Target	Property name	Result	Refinements	States	Transitions	CPU time [s]	Memory [MB]
Debug	Inherent	✓	576	68027 / 56847	70700 / 58192	68.32	107.96
Debug	Reachability	✓	0	20 / 20	21 / 21	<0.01	3.69
Debug	Invariant lock	✓	576	68027 / 56847	70716 / 58192	93.99	113.02
Debug	Recovery	✗	576	68027 / 56847	70716 / 58192	85.07	109.15
Debug	Stack above 0x08DD	✓	576	68027 / 56847	70716 / 58192	84.15	111.61
Debug	Stack above 0x08DE	✗	3	840 / 748	855 / 756	0.05	7.83
Release	Inherent	✓	45	5883 / 4272	6090 / 4378	1.08	14.89
Release	Reachability	✓	0	20 / 20	21 / 21	<0.01	3.81
Release	Invariant lock	✓	45	5883 / 4272	6094 / 4378	1.19	15.55
Release	Recovery	✗	45	5883 / 4272	6094 / 4378	1.2	15.18
Release	Stack above 0x08FB	✓	45	5883 / 4272	6094 / 4378	1.18	15.26
Release	Stack above 0x08FC	✗	0	20 / 20	21 / 21	<0.01	3.78

maximum stack size needed by manually binary-searching until I determined the properties that together give the maximum value of v.

As the maximum SRAM location for ATmega328P is `0x08FF`, the maximum stack size is only 4 bytes for the release target, which corresponds to a call to the `main` function from the initialisation code and a later call from `main` to `factorial`. For the debug target, the maximum stack size is 34 bytes. In addition to the 2 bytes corresponding to the call to `main`, the factorial function is called at most 8 times (decreasing from a value of 7 to a value of 0 inclusively). It pushes two registers to the stack in its prelude, so that each call of the function together with the prelude takes up 4 bytes, resulting in $2 + 8 \cdot 4 = 34$.

These maximum stack sizes might seem small as ATmega328P has 2048 bytes of SRAM, but excessive recursion might preclude the use of a cheaper microcontroller. For example, the related AVR ATtiny24A has only 128 bytes of SRAM, so even 34 bytes used by the stack can be problematic. Using **machine-check-avr**, it is possible to select the appropriate device while ensuring the stack never overwrites other variables.

6.4.4 Digital Calibration: Finding a Bug in a Realistic Program

In my previous bachelor thesis, I used the AVR ATtiny24A microcontroller for digital calibration of an analog Voltage-Controlled Oscillator (VCO) [A.6, p. 28-30, 42-43, 48-50]. The calibration is based on monotonicity of adjustment: as a digital potentiometer controlling the VCO input voltage is adjusted in a specified direction, the VCO output frequency rises. The optimal potentiometer setting is found using binary search based on whether the VCO frequency is lower or higher than desired. The calibration program was able to adjust the VCO satisfactorily for musical audio and is a practical example of where a low-cost microcontroller may be used.

I simplified the calibration program, using the core calibration routine and replacing the frequency estimation using a timer peripheral (input) and SPI digital potentiometer

```c
#define F_CPU 1000000
#include <avr/io.h>
#include <util/delay.h>

int main(void) {
    DDRC |= 0x01;
    DDRD |= 0xFF;
    while (1) {
        while ((PINC & 0x2) == 0) {}
        // signify we are calibrating
        PORTC |= 0x01;

        // start with MSB of calibration
        uint8_t search_bit = 7;
        uint8_t search_mask = (1 << search_bit);
        uint8_t search_val = search_mask;

        while (1) {
            // wait a bit
            _delay_us(10);
            // write the current search value
            PORTD = search_val;
            // wait a bit
            _delay_us(10);

            // get input value and compare it to desired
            uint8_t input_value = PINB;

            if ((input_value & 0x80) == 0) {
                // input value lower than desired
                // we should lower the calibration value
                search_val &= ~search_mask;
            }

            if (search_bit == 0) {
                // all bits have been set, stop
                break;
            }

            search_bit -= 1;
            // continue to next bit
            search_mask >>= 1;
            // update the search value with the next bit set
            search_val |= search_mask;
        }
        // calibration complete, stop signifying that we are calibrating
        PORTC &= ~0x01;
    }
}
```

Figure 6.6: The simplified calibration program, written in C.

control (output) with GPIO read and write, respectively, so that I could verify the program using the current **machine-check-avr**. I considered only a single calibration for easier verification. The simplified program is shown in Figure 6.6.

I used the release configuration of the calibration program and verified the same kinds of properties as in the factorial example, as shown in the columns in Figure 6.3 denoted as Original. The inherent, reachability, and invariant lock properties hold as expected. The maximum stack size is 2 bytes, caused by a call to `main` from initialisation code.

Table 6.3: Measurements of machine-code verification of the calibration program using **machine-check-avr**.

Program	Property name	Result	Refin.	States	Transitions	CPU time [s]	Memory [MB]
Original (debug)	Inherent	✓	513	13575 / 13213	14599 / 13727	25.75	31.55
Original (debug)	Reachability	✓	0	17 / 17	18 / 18	<0.01	3.64
Original (debug)	Invariant lock	✓	513	13575 / 13213	14602 / 13727	31.43	32.61
Original (debug)	Recovery	✗	513	13575 / 13213	14602 / 13727	28.82	31.62
Original (debug)	Stack above 0x08FD	✓	513	13575 / 13213	14602 / 13727	29.66	32.16
Original (debug)	Stack above 0x08FE	✗	0	17 / 17	18 / 18	<0.01	3.72
Original (release)	Inherent	✓	512	11253 / 10830	12275 / 11343	24.53	27.32
Original (release)	Reachability	✓	0	15 / 15	16 / 16	<0.01	3.86
Original (release)	Invariant lock	✓	512	11253 / 10830	12278 / 11343	29.33	28.39
Original (release)	Recovery	✗	512	11253 / 10830	12278 / 11343	27.12	27.45
Original (release)	Stack above 0x08FD	✓	512	11253 / 10830	12278 / 11343	27.8	28.12
Original (release)	Stack above 0x08FE	✗	0	15 / 15	16 / 16	<0.01	3.70
Fixed (debug)	Inherent	✓	513	13575 / 13213	14599 / 13727	26.21	31.40
Fixed (debug)	Reachability	✓	0	17 / 17	18 / 18	<0.01	3.81
Fixed (debug)	Invariant lock	✓	513	13575 / 13213	14602 / 13727	31.26	32.69
Fixed (debug)	Recovery	✓	513	13575 / 13213	14602 / 13727	29.11	31.72
Fixed (debug)	Stack above 0x08FD	✓	513	13575 / 13213	14602 / 13727	29.49	32.26
Fixed (debug)	Stack above 0x08FE	✗	0	17 / 17	18 / 18	<0.01	3.80
Fixed (release)	Inherent	✓	512	12526 / 12158	13548 / 12671	24.9	29.68
Fixed (release)	Reachability	✓	0	15 / 15	16 / 16	<0.01	3.76
Fixed (release)	Invariant lock	✓	512	12526 / 12158	13551 / 12671	29.66	30.73
Fixed (release)	Recovery	✓	512	12526 / 12158	13551 / 12671	27.3	29.76
Fixed (release)	Stack above 0x08FD	✓	512	12526 / 12158	13551 / 12671	28.23	30.62
Fixed (release)	Stack above 0x08FE	✗	0	15 / 15	16 / 16	<0.01	3.74

While I did expect the recovery property to hold, **machine-check-avr** determined that it does not hold. I investigated further and realised that the lowest output bit is cleared in `search_val` but not in `PORTD`. As the output can never recover to being fully zero, the recovery property is violated.

I fixed the problem by writing `search_val` to `PORTD` before breaking from the loop. The results are shown in the columns in Figure 6.3 denoted as Fixed. The recovery property is no longer violated.

The bug affects the original calibration program by reducing the number of used digital potentiometer values[6] from 256 to 128, which is hard to find because the output quality is degraded but not significantly enough to be noticeable without special care. It is slightly lucky that the reachability property uncovered the bug: no bug would be uncovered if the initial value had the lowest bit set to 1. To thoroughly reveal the problems with unusable output values, the recovery property would ideally be parameterised, so it could be verified that all output values from 0 to 255 are used.

[6]The MCP4251 digital potentiometer has 257 steps, but one step is ignored in the calibration program.

The bug would be hard to find when using the ubiquitous source-code verification with LTL properties. While it would be possible to rewrite the code so it does not use AVR-specific peripherals and functions, the need to write an LTL property that would detect the bug, e.g. $\mathbf{F}(\texttt{PORTD} = 2)$, is not obvious when we do not know about the bug yet. Furthermore, the reachability property does not detect a bug where output values become blocked indefinitely for some reason. On the other hand, the property $\mathbf{AG}[\mathbf{EF}[\texttt{PORTD}{=}x]]$ parameterised with x from 0 to 255 directly corresponds to ensuring all output values can be used without being blocked indefinitely, which is what we would ideally want.

6.4.5 Assessment of Capabilities and Possible Improvements

The tool I have created during my doctoral studies, **machine-check**, is capable of verifying CTL properties of machine-code systems. I was able to verify important properties including maximum stack size and recovery in programs for the AVR ATmega328P microcontroller. The programs were simpler than typical real-world AVR programs, with only general-purpose I/O peripherals used, but their intricacy was approaching real-world programs ideal for the low-cost and low-power ATtiny devices. The verification results matched expectations, except for one case where the unexpected result was caused by a previously undiscovered bug in the verified program, which demonstrates the capability for bug-finding.

Compared to previous work on machine-code verification, **machine-check** is built around general principles rather than tailored to a specific architecture or device. As such, there is notably no special handling of e.g. the Program Counter or Stack Pointer registers. This can lead to sub-optimal choices of refinement but makes the reasoning straightforward without architecture-dependent decisions. There is little need to consider the verification internals when writing the processor description.

Overall, I am satisfied with the theoretical and practical groundwork used to implement **machine-check**, with the combination of translation of simulable descriptions presented in Chapter 3 and discussed more extensively in this chapter, the input-based TVAR framework presented in Chapter 4, and the bit-vector abstraction with fast abstract arithmetic presented in Chapter 5. However, for real-world use, further improvements are necessary:

- Working with **machine-check** is currently not very user-friendly. Currently, verification is done using a command-line tool that takes the machine code and the property in a very simple format and outputs the result and further information when done. The property format could be made more user-friendly, minimal witnesses of the result could be given, a graphical user interface could be added for visualisation, etc. Special care should be taken so that more processor peripherals can be easily implemented so that real-world programs can be verified without any simplification.
- While the time and memory necessary for verification were reasonable for simple programs, it would not be for more complex programs. The abstraction domains and refinement strategies can be extended and tuned generally without any system-dependent information so that the number of states and refinements is lower. While

I would expect general improvements to provide usable results for verification of real-world machine-code programs, the descriptions could be also extended with verification hints (e.g. that the Program Counter is the most important variable) if the general approach is insufficient.

All things considered, the techniques I have introduced during my doctoral studies and implemented in **machine-check** make it possible to verify interesting and useful properties of machine-code programs. There are still practical issues precluding serious non-academic use of the tool. However, in my opinion, it is only a matter of time and work before machine-code verification can be successfully used to improve practical systems.

Chapter 7

Conclusion

At the outset of my doctoral studies, I set out to improve the state of the art in formal verification of programs in machine code, which had been under-researched. During my studies, I have successfully introduced three novel techniques to resolve known challenges, published them, and implemented them in my free and open-source formal verification tool **machine-check**. To my knowledge, it is the first publicly available, free, and open-source tool of its kind. The introduced techniques and their implementations form a solid foundation of machine-code verification through abstraction-based model checking.

7.1 Summary

In Chapter 1, I introduced the concept of and need for formal verification of machine-code programs, enumerated my contributions, and provided an overview of the organisation of this thesis.

In Chapter 2, I introduced the background of the work. I showed how source-code, machine-code, and hardware systems have some basic commonalities despite their differences, and noted that machine-code systems are formed by machine-code programs together with the guarantees about the processor they are executed on, with possible additional guarantees. I introduced basic formalisms for model checking and discussed the grouping of advanced techniques. I introduced the concept of abstraction refinement. I discussed the state of the art in verification of digital systems and noted that abstraction refinement is commonly used in the best verification tools to mitigate the problem of exponential explosion. I noted that verification of machine-code systems is highly under-researched and that a major problem specific to machine-code verification has been the difficulty of supporting easy writing of processor descriptions while ensuring that advanced verification techniques such as abstraction refinement can be used.

In Chapter 3, I presented my technique that solves the difficulty. The descriptions in are written in the Rust programming language and meta-programming is used to transform them into verification equivalents, usable for model-checking with abstraction refinement. I discussed how this solution is fully automatic and opaque to the processor

description writer, allowing those not familiar with formal verification to write the descriptions. I showed an example of a simplified RISC processor described in a way it can be transformed into verification equivalents and discussed the subset of the Rust language that can be used in the descriptions.

In Chapter 4, I discussed a novel Three-Valued Abstraction Refinement (TVAR) framework I introduced to be able to verify properties that are not verifiable by the conventional Counterexample-guided Abstraction Refinement (CEGAR). I noted the problems of previous TVAR frameworks and based my framework on a novel input-based approach. I formally proved that my framework produces correct results and does so in finite time for finite systems as long as requirements are met. I experimentally evaluated an implementation of an instance of the introduced framework in **machine-check**, showing its potential to reduce exponential explosion on simple digital systems.

In Chapter 5, I showed how I and my supervisor resolved a problem in previous approaches to machine-code model checking that used three-valued bit-vector abstraction, namely that it was not possible to adequately resolve arithmetic operations. We were able to devise an algorithm that produces correct results with linear complexity provided the original non-abstract operation has constant complexity. We then proved that the operation results are optimal for addition, subtraction, and general summation. For multiplication, we devised an algorithm that produces optimal results with quadratic complexity.

In Chapter 6, I discussed my publicly available, free, and open-source tool **machine-check** that I created during my doctoral research, implementing the techniques from Chapters 3, 4, and 5. I discussed how the techniques are combined, and noted some further implementation difficulties and their resolution. I evaluated the tool using machine-code programs for the ATmega328P microcontroller and was able to verify their interesting properties. Notably, I found a bug in a real-world program from my previous bachelor thesis [A.6] using a simplified program that retained its core behaviour.

7.2 Contributions of the Dissertation Thesis

In my dissertation thesis, the results of my doctoral research are described and brought into general context. In Chapter 2, the theoretical background and the state of the art are discussed, and valuable insights frame the subject matter:

- All digital systems have basic commonalities: finite-length bit-vector variables, indexable arrays of bit-vectors, and fixed-point bit-vector operations. These commonalities are instrumental for the effective expression of the system using other system levels. The systems can be formalised as Moore machines.

- The commonly used temporal logics are Computation Tree Logic (CTL), Linear Time Logic (LTL), and CTL*, especially as there are known algorithms for model-checking against them that depend only linearly on the size of the state space.

- Model-checking is often used with abstraction, which can further be extended for abstraction refinement. The two main methodologies for abstraction refinement are Counterexample-guided Abstraction Refinement (CEGAR) and Three-valued Abstraction Refinement (TVAR).

- Use of CEGAR is very common in state-of-the-art verification tools, typically combined with symbolic verification techniques and using Satisfiability Modulo Theories (SMT) solvers.

- The focus on CEGAR and degenerate temporal properties, especially reachability, in source-code verification tools may present blind spots in verification, as can the reliance on compilers to produce the correct machine code that is not formally verified. Using Three-valued Abstraction Refinement (TVAR) makes it possible to verify properties not verifiable using CEGAR, and using machine-code verification can reveal problems not possible to find using source-code verification.

In the following chapters, I described three novel techniques I devised during the course of my research:

- Simulable machine-code system descriptions translated to their verification equivalents by meta-programming, combining a previously published overview [A.2] with added material original to the thesis.

- A Three-Valued Abstraction Refinement framework (TVAR) using input-based instead of state-based splitting to resolve problems of previous TVAR frameworks and provide a simpler representation, containing material available as a preprint [A.3].

- Fast algorithms for computation of the best results of arithmetic operations on three-valued abstract bit-vectors, containing previously published material [A.1].

Finally, I described the combination of the techniques in my implementation of the formal verification tool **machine-check** that I created during my doctoral research, and showed that the tool is capable of machine-code program verification.

7.3 Future Work

There are possibilities for further research based on the foundations laid by the work described in this thesis, although hampered by the fact that no machine-code verification tool other than **machine-check** is currently practically available, and **machine-check** is currently more limited by practical rather than theoretical problems. The time and memory requirements of formal verification are the main theoretical problem: even the source-code verification tools, despite decades of research, have problems formally verifying some rather simple programs. Possible extensions of machine-code verification, such as verification of real-time properties, have been already researched formally in other contexts, such as

hardware verification. There might be some adjustments necessary for the machine-code level, but I would not expect them to be drastic due to the digital system commonalities.

Despite the fact that **machine-check** is able to verify simple digital systems including some machine-code systems, its practical usability is currently not good enough yet for serious use. Currently, the most significant problems are practical, such as the user-friendliness of the interface or the ability to write descriptions more elegantly and use more advanced Rust constructs. Programmers unfamiliar with formal verification will only start using formal verification tools if there is a simple and understandable process for obtaining useful verification results, helping them with development. Only then will we be closer to the overarching goal of formal verification, designing safe, secure, and useful systems.

Bibliography

[1] B. Fung. We finally know what caused the global tech outage - and how much it cost. *CNN Business*, July 2024. URL `https://edition.cnn.com/2024/07/24/tech/crowdstrike-outage-cost-cause/index.html`. Accessed on 23 August 2024.

[2] T. Hannaford. Microcode (0x129) update for Intel Core 13th and 14th Gen desktop processors. URL `https://community.intel.com/t5/Processors/Microcode-0x129-Update-for-Intel-Core-13th-and-14th-Gen-Desktop/m-p/1624688`. Accessed on 23 August 2024.

[3] AMD-SB-7014. SMM lock bypass. URL `https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7014.html`. Popularly known as the SinkClose vulnerability. Accessed on 23 August 2024.

[4] B. Toulas. New AMD SinkClose flaw helps install nearly undetectable malware. *Bleeping Computer*, August 2024. URL `https://www.bleepingcomputer.com/news/security/new-amd-sinkclose-flaw-helps-install-nearly-undetectable-malware/`. Accessed on 23 August 2024.

[5] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57(12):713–723, 1938. doi:10.1109/T-AIEE.1938.5057767.

[6] International Organization for Standardization. ISO/IEC 9899:1999. Standard, International Organization for Standardization, Geneva, Switzerland, December 1999. Informally known as the C99 standard of the C language.

[7] A. Biere. The AIGER And-Inverter Graph (AIG) format version 20071012. *FMV Technical Reports*, 07/1, 2007. doi:10.35011/fmvtr.2007-1.

[8] A. Biere, K. Heljanko, and S. Wieringa. AIGER 1.9 and beyond. *FMV Technical Reports*, 11/2, 2011. doi:10.35011/fmvtr.2011-2.

[9] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. Btor2, BtorMC and Boolector 3.0. In H. Chockler and G. Weissenbacher, editors, *Proceedings of the 30th International Conference on Computer Aided Verification, CAV 2018*, pages 587–595, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96145-3. doi:10.1007/978-3-319-96145-3_32.

[10] Intel Corporation. Hexadecimal object file format specification. Technical report, Intel Corporation, 1988. URL `https://archive.org/details/IntelHEXStandard`.

[11] The LLVM Compiler Infrastructure. LLVM bitcode file format. URL `https://llvm.org/docs/BitCodeFormat.html`. Accessed on 24 August 2024.

[12] *ATmega48A/PA/88A/PA/168A/PA/328/P Data Sheet*. Microchip Technology Inc., October 2018. URL `http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf`. DS40002061A.

[13] *AVR Instruction Set Manual*. Microchip Technology Inc., February 2021. URL `https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf`. DS40002198B.

[14] S. A. Seshia, N. Sharygina, and S. Tripakis. Modeling for verification. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 75–105. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:https://doi.org/10.1007/978-3-319-10575-8_3.

[15] H. S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012. ISBN 0321842685.

[16] E. M. Clarke, T. A. Henzinger, and H. Veith. Introduction to model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8_1.

[17] J. Bradfield and I. Walukiewicz. The mu-calculus and model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 871–919. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8_26.

[18] N. Piterman and A. Pnueli. Temporal logic and fair discrete systems. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 27–73. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8_2.

[19] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the*

12th International Conference on Computer Aided Verification, CAV 2000, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45047-4. doi:10.1007/10722167_15.

[20] E. Clarke, A. Gupta, and O. Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(7):1113–1123, 2004. doi:10.1109/TCAD.2004.829807.

[21] S. Chaki and A. Gurfinkel. BDD-based symbolic model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 219–245. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8_8.

[22] A. Biere and D. Kröning. SAT-based model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8_10.

[23] D. Dams and O. Grumberg. Abstraction and abstraction refinement. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 385–419. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8_13.

[24] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Interval analysis and machine arithmetic: Why signedness ignorance is bliss. *ACM Transactions on Programming Languages and Systems*, 37(1):1:1–1:35, 2014. doi:10.1145/2651360.

[25] A. Mine. The octagon abstract domain. In *Proceedings of the 8th Working Conference on Reverse Engineering, WCRE 2001*, pages 310–319, 2001. doi:10.1109/WCRE.2001.957836.

[26] B. Schlich and S. Kowalewski. [mc]square: A model checker for microcontroller code. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISOLA 2006*, pages 466–473, 2006. doi:10.1109/ISoLA.2006.62.

[27] T. Reinbacher, M. Horauer, and B. Schlich. Using 3-valued memory representation for state space reduction in embedded assembly code model checking. In *Proceedings of the 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems, DDECS 2009*, pages 114–119, 2009. doi:10.1109/DDECS.2009.5012109.

[28] B. Schlich. Model checking of software for microcontrollers. *ACM Transactions on Embedded Computing Systems*, 9(4), April 2010. ISSN 1539-9087. doi:10.1145/1721695.1721702.

[29] R. P. Kurshan. Transfer of model checking to industrial practice. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 763–793. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8_23.

[30] D. Beyer. State of the art in software verification and witness validation: SV-COMP 2024. In B. Finkbeiner and L. Kovács, editors, *Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2024*, pages 299–329, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-57256-2. doi:10.1007/978-3-031-57256-2_15.

[31] L. Westhofen, P. Berger, and J.-P. Katoen. Benchmarking software model checkers on automotive code. In R. Lee, S. Jha, A. Mavridou, and D. Giannakopoulou, editors, *Proceedings of the 12th International Symposium on NASA Formal Methods, NFM 2020*, pages 133–150, Cham, 2020. Springer International Publishing. ISBN 978-3-030-55754-6. doi:10.1007/978-3-030-55754-6_8.

[32] D. Beyer. SV-COMP 2024: Benchmark verification tasks. URL `https://sv-comp.sosy-lab.org/2024/benchmarks.php`. Accessed on 12 June 2024.

[33] CPAchecker: A software verification tool for configurable program analyses. URL `https://cpachecker.sosy-lab.org/`. Accessed on 25 August 2024.

[34] Ultimate program analysis framework. URL `https://ultimate-pa.org/`. Accessed on 25 August 2024.

[35] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate Automizer with SMTInterpol. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2013*, pages 641–643, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-36742-7. doi:10.1007/978-3-642-36742-7_53.

[36] Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai, and V. Štill. Model checking of C and C++ with DIVINE 4. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis, ATVA 2017*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017. doi:10.1007/978-3-319-68167-2_14.

[37] Machine learning based symbolic execution (MLB-SE). URL `https://github.com/MLB-SE/Experiment`. Accessed on 25 August 2024.

[38] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik. JBMC: A bounded model checking tool for verifying Java bytecode. In H. Chockler and G. Weissenbacher, editors, *Proceedings of the 30th International Conference on Computer Aided Verification, CAV 2018*, pages 183–190, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96145-3. doi:10.1007/978-3-319-96145-3_10.

[39] M. Mues and F. Howar. GDart: An ensemble of tools for dynamic symbolic execution on the Java Virtual Machine (competition contribution). In D. Fisman and G. Rosu, editors, *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2022*, pages 435–439, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99527-0. doi:10.1007/978-3-030-99527-0_27.

[40] C. Artho, P. Parízek, D. Qu, V. Galgali, and P. L. Yi. JPF: From 2003 to 2023. In B. Finkbeiner and L. Kovács, editors, *Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2024*, pages 3–22, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-57249-4. doi:10.1007/978-3-031-57249-4_1.

[41] A. Biere, N. Froleyks, and M. Preiner. Hardware Model Checking Competition 2020, HWMCC 2020. URL `https://fmv.jku.at/hwmcc20/`. Accessed on 12 June 2024.

[42] A. Goel and K. Sakallah. AVR: Abstractly verifying reachability. In A. Biere and D. Parker, editors, *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2020*, pages 413–422, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-45190-5_23.

[43] A. R. Bradley. Understanding IC3. In A. Cimatti and R. Sebastiani, editors, *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing, SAT 2012*, pages 1–14, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-31612-8_1.

[44] R. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In T. Touili, B. Cook, and P. Jackson, editors, *Proceedings of the 22nd International Conference on Computer Aided Verification, CAV 2010*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14295-6. doi:10.1007/978-3-642-14295-6_5.

[45] D. Beyer, P.-C. Chien, and N.-Z. Lee. Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator. In S. Sankaranarayanan and N. Sharygina, editors, *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023*, pages 152–172, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-30820-8. doi:10.1007/978-3-031-30820-8_12.

[46] J. Regehr and A. Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. *ACM SIGOPS Operating Systems Review*, 38(5):133–143, October 2004. ISSN 0163-5980. doi:10.1145/1037949.1024410.

[47] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems*, 4(4):751–778, November 2005. ISSN 1539-9087. doi:10.1145/1113830.1113833.

[48] E. G. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *Proceedings of the 12th International SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*, pages 251–265, San Francisco, USA, August 2005. Springer. doi:10.1007/11537328_20.

[49] T. Mehler. *Challenges and Applications of Assembly-Level Software Model Checking*. Dissertation thesis, University of Dortmund, 2006. URL `http://hdl.handle.net/2003/22435`.

[50] T. Noll and B. Schlich. Delayed nondeterminism in model checking embedded systems assembly code. In K. Yorav, editor, *Proceedings of the 3rd Haifa Verification Conference, HVC 2008*, pages 185–201, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-77966-7. doi:10.1007/978-3-540-77966-7_16.

[51] T. Reinbacher, J. Brauer, M. Horauer, and B. Schlich. Refining assembly code static analysis for the Intel MCS-51 microcontroller. In *Proceedings of the Fourth IEEE International Symposium on Industrial Embedded Systems, SIES 2009*, pages 161–170, 2009. doi:10.1109/SIES.2009.5196212.

[52] J. Brauer, T. Noll, and B. Schlich. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems, SCOPES 2010*, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300841. doi:10.1145/1811212.1811216.

[53] D. Gückel. *Synthesis of State Space Generators for Model Checking Microcontroller Code*. Dissertation thesis, Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen, November 2014. URL `http://aib.informatik.rwth-aachen.de/2014/2014-15.pdf`.

[54] S. Biallas, J. Brauer, and S. Kowalewski. Arcade.PLC: a verification platform for programmable logic controllers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012*, pages 338–341, 2012. doi:10.1145/2351676.2351741.

[55] J. Davis, A. Slobodova, and S. Swords. Microcode verification – another piece of the microprocessor verification puzzle. In G. Klein and R. Gamboa, editors, *Proceedings of the 5th International Conference on Interactive Theorem Proving, ITP 2014*, pages 1–16, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08970-6. doi:10.1007/978-3-319-08970-6_1.

[56] S. Goel, A. Slobodova, R. Sumners, and S. Swords. Verifying x86 instruction implementations. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, page 47–60, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370974. doi:10.1145/3372885.3373811.

[57] CVE-2022-22819. LPC55Sxx SB2 loader vulnerability. URL `https://community.nxp.com/t5/LPC-Microcontrollers-Knowledge/LPC55Sxx-SB2-loader-vulnerability/ta-p/1433661`. CVE-2022-22819. Accessed on 18 February 2024.

[58] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982. doi:10.1016/0167-6423(83)90017-5.

[59] S. Shoham and O. Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. In W. A. Hunt and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification, CAV 2003*, pages 275–287, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45069-6. doi:10.1007/978-3-540-45069-6_28.

[60] O. Grumberg, M. Lange, M. Leucker, and S. Shoham. Don't Know in the µ-calculus. In R. Cousot, editor, *Proceeding of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2005*, pages 233–249, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30579-8. doi:10.1007/978-3-540-30579-8_16.

[61] O. Grumberg, M. Lange, M. Leucker, and S. Shoham. When not losing is better than winning: Abstraction and refinement for the full µ-calculus. *Information and Computation*, 205(8):1130–1148, 2007. ISSN 0890-5401. doi:10.1016/j.ic.2006.10.009.

[62] P. Godefroid. May/must abstraction-based software model checking for sound verification and falsification. In O. Grumberg, H. Seidl, and M. Irlbeck, editors, *Software Systems Safety*, volume 36 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–16. IOS Press, 2014. doi:10.3233/978-1-61499-385-8-1.

[63] F. Belardinelli, A. Ferrando, and V. Malvone. An abstraction-refinement framework for verifying strategic properties in multi-agent systems with imperfect information. *Artificial Intelligence*, 316:103847, 2023. doi:10.1016/J.ARTINT.2022.103847.

[64] S. C. Kleene. On notation for ordinal numbers. *The Journal of Symbolic Logic*, 3(4): 150–155, 1938. ISSN 00224812. doi:10.2307/2267778.

[65] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In N. Halbwachs and D. Peled, editors, *Proceedings on 11th International*

Conference on Computer Aided Verification, CAV 1999, pages 274–287, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-48683-6_25.

[66] M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In D. Sands, editor, *Proceedings of the 10th European Symposium on Programming, ESOP 2001*, pages 155–169, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45309-3. doi:10.1007/3-540-45309-1_11.

[67] K. Larsen and B. Thomsen. A modal process logic. In *Proceedings of the Third Annual Symposium on Logic in Computer Science, LICS 1988*, pages 203–210, 1988. doi:10.1109/LICS.1988.5119.

[68] G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In C. Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory, CONCUR 2000*, pages 168–182, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44618-7. doi:10.1007/3-540-44618-4_14.

[69] P. Godefroid and R. Jagadeesan. Automatic abstraction using generalized model checking. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002*, pages 137–151, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45657-0_11.

[70] P. Godefroid and R. Jagadeesan. On the expressiveness of 3-valued models. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2003*, pages 206–222, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36384-2. doi:10.1007/3-540-36384-X_18.

[71] S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In K. Jensen and A. Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2004*, pages 546–560, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24730-2. doi:10.1007/978-3-540-24730-2_40.

[72] S. Shoham and O. Grumberg. 3-valued abstraction: More precision at less cost. *Information and Computation*, 206(11):1313–1333, 2008. ISSN 0890-5401. doi:10.1016/j.ic.2008.07.004.

[73] O. Kupferman. Automata theory and model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 107–151. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8_4.

[74] D. Dams and K. S. Namjoshi. Automata as abstractions. In R. Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2005*, pages 216–232, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30579-8. doi:10.1007/978-3-540-30579-8_15.

[75] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The Cyber-Physical Systems Series. MIT Press, 1st edition, 1999. ISBN 9780262032704.

[76] Institute of Electrical and Electronics Engineers. IEEE standard multivalue logic system for VHDL model interoperability (std_logic_1164). *IEEE Std 1164-1993*, pages 1–24, 1993. doi:10.1109/IEEESTD.1993.115571.

[77] S. Yamane, R. Konoshita, and T. Kato. Model checking of embedded assembly program based on simulation. *IEICE Transactions on Information and Systems*, E100.D (8):1819–1826, 2017. doi:10.1587/transinf.2016EDP7452.

[78] E. Boros and P. L. Hammer. Pseudo-Boolean optimization. *Discrete Applied Mathematics*, 123(1):155–225, 2002. ISSN 0166-218X. doi:10.1016/S0166-218X(01)00341-9.

[79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1979*, page 269–282, New York, NY, USA, 1979. Association for Computing Machinery. ISBN 9781450373579. doi:10.1145/567752.567778.

[80] T. Reps and A. Thakur. Automating abstract interpretation. In B. Jobstmann and K. R. M. Leino, editors, *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 3–40, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49122-5. doi:10.1007/978-3-662-49122-5_1.

[81] J. Arndt. *Bit wizardry*, pages 2–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-14764-7. doi:10.1007/978-3-642-14764-7_1.

[82] S. S. Skiena. *Introduction to Algorithm Design*, pages 3–30. Springer London, London, 2008. ISBN 978-1-84800-070-4. doi:10.1007/978-1-84800-070-4_1.

[83] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988*, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912527. doi:10.1145/73560.73562.

[84] A. Møller and M. I. Schwartzbach. Static program analysis. URL `http://cs.au.dk/~amoeller/spa/`. Department of Computer Science, Aarhus University. Accessed on 20 August 2024.

Reviewed Publications of the Author Relevant to the Thesis

[A.1] Onderka, J., Ratschan, S. Fast three-valued abstract bit-vector arithmetic. In Finkbeiner, B., Wies, T., editors, *Proceedings of the 23rd International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2022,* pages 242–262. Springer Nature Switzerland, Cham, 2022. ISBN: 978-3-030-94583-1. doi:10.1007/978-3-030-94583-1_12.

The paper has been cited in:

- Vishwanathan, H., Shachnai, M., Narayana, S., Nagarakatte, S. Verifying the verifier: eBPF range analysis verification. In Enea, C., Lal, A., editors, *Proceedings of the 35th International Conference on Computer Aided Verification, CAV 2023.* Springer, Cham, 2023. ISBN: 978-3-031-37709-9. doi:10.1007/978-3-031-37709-9_12.
- Shachnai, M., Vishwanathan, H., Narayana, S., Nagarakatte, S. Fixing Latent Unsound Abstract Operators in the eBPF Verifier of the Linux Kernel (preprint). Accepted and scheduled to be presented at the *Static Analysis Symposium, SAS 2024.* `https://mshachnai.github.io/pubs/sas_24.pdf`

[A.2] Onderka, J. Formal verification of machine-code systems by translation of simulable descriptions. In *Proceedings of the 13th Mediterranean Conference on Embedded Computing, MECO 2024.* Budva, Montenegro, 2024. doi:10.1109/MECO62516.2024.10577942.

The paper has received the MECO 2024 conference award *The Best Paper in Software and Algorithms*.

Remaining Publications of the Author Relevant to the Thesis

[A.3] Onderka, J. Input-based framework for three-valued abstraction refinement (preprint). arXiv:2408.12668 [cs.LO]. 2024. https://arxiv.org/abs/2408.12668. The current version at the time of writing this thesis is available at https://arxiv.org/abs/2408.12668v2.

[A.4] Onderka, J. Deadline verification using model checking. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology. Prague, 2020. http://hdl.handle.net/10467/87989.

Remaining Publications of the Author

[A.5] Onderka, J. Pitch shifting of audio signals in real time using STFT on a Digital Signal Processor. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology. Prague, 2018. http://hdl.handle.net/10467/77279.

[A.6] Onderka, J. Analog modular music synthesizer with digital control. Bachelor's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering. Prague, 2022. http://hdl.handle.net/10467/101676.