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, March 2025

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. The created tool is free, open-source, and publicly available.

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. Vytvořený nástroj je bezplatně a veřejně dostupný, 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.

Věnování / Dedication

Mé babičce Jiřině

To my grandmother Jiřina

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

Commonly Used Abbreviations

API	Application Programming Interface
ABI	Application Binary Interface
ATP	Automated Theorem Prover
BDD	Binary Decision Diagram
CEGAR	Counterexample-guided Abstraction Refinement
CTL	Computation Tree Logic
GA	Generating Automaton
GPIO	General-Purpose Input/Output
ISA	Instruction Set Architecture
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. In particular, the technique of *abstraction refinement* is typically used in state-of-the-art tools for source-code and hardware systems to reduce the state space size, but using it while allowing verification of machine code for diverse processor architectures was previously not reasonably possible.

While there are publicly available tools for formal machine-code verification using theorem proving [1, 2], their use requires specialised knowledge and they are intended for verification of programs for computers rather than for embedded systems. The tool **Arcade.µC** [3, 4, 5, 6, 7] was devised for formal verification of machine-code programs for embedded systems using *model-checking with abstraction*, but its development has been since discontinued. In my diploma thesis [A.4], I created a tool inspired by **Arcade.µC**. However, it was not well-usable in practice due to the aforementioned challenges.

Research goal. My overarching goal has been to resolve the outstanding problems of formal verification of machine-code systems using model checking with abstraction by providing a solid yet flexible theoretical groundwork for model-checking embedded systems using abstraction refinement, allowing for flexibility in system specifications. I aimed to ensure that the techniques are practically viable and available to use.

To achieve my goal, 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.

[1]The official website of the tool is `https://machine-check.org`. The current release at the time of writing is 0.4.0, available at `https://crates.io/crates/machine-check/0.4.0`.

Thesis content. In this thesis, I comprehensively present the background of my work, state of the art in formal machine-code verification, the techniques I devised, as well as an experimental evaluation of the tool **machine-check** based on them. I was able to verify that various specifications hold in bare-metal programs for the AVR ATmega328 microcontroller using the tool. The techniques enable fully automatic verification in reasonable time and memory without problems of the previous model-checking tools, substantially improving the state of the art in formal verification of machine-code systems using model checking with abstraction.

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, there are cases where just one or two specific bits of a read input port influence whether the property holds or not, meaning that storing the other bits is wasteful and only contributes to exponential explosion [8]. Three-valued bit-vector abstraction solves this problem, but it was previously not possible to quickly compute useful results of arithmetic operations using it. 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. **Theoretical Background**: Introduces the reader to the necessary theoretical background.
3. **State of the Art in Digital System Verification**: Surveys the current state of the art in digital system verification, with emphasis on machine-code verification.
4. **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.
5. **Input-based Three-valued Abstraction Refinement**: Describes the idea of Three-valued Abstraction Refinement with refinement of inputs and states, followed by a framework formalisation, simpler than previous TVAR frameworks yet more powerful than the commonly used CEGAR 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].
6. **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].
7. **Created Machine-Code Formal Verification Tool machine-check**: Describes the combination of the techniques from Chapter 4, 5, and 6 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.
8. **Conclusion**: Summarises the results of my doctoral research, suggests possible topics of further research, and concludes the dissertation thesis.

Chapter 2

Theoretical Background

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 from July to August 2024 during the writing of this thesis,

- the outage caused by the CrowdStrike software glitch paralysed the global economy and resulted in losses estimated in billions of dollars [9],
- 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 [10],
- a vulnerability was found in AMD processors including the current generation [11], undetected for almost 20 years, allowing the planting of nearly undetectable malware once kernel-level access is obtained [12].

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, fundamental approaches to formal verification, and used specifications. After that, I will focus on formal verification using model checking with abstraction and abstraction refinement, which is the approach I use in my research and implementation of **machine-check**.

2.1 Digital Systems

As proven by Shannon [13], 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. Implementation-wise, processors implemented on a single Integrated Circuit (IC) chip but requiring external circuitry (including memory and storage) are typically called *microprocessors*, while fully integrated processors with little external requirements are called *microcontrollers*. Processors can also be implemented on FPGAs as *soft processors*.

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 [14]. 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.

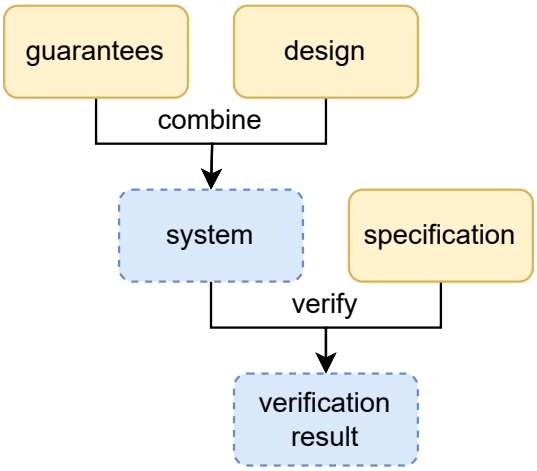


Figure 2.1: A high-level overview of formal verification of digital systems. The solid yellow cells represent verification inputs, while the dashed blue cells represent an automated combination or result. The guarantees and the design are combined together to form the system under verification. It is 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 [15, 16] that describes the whole system as a sequence of gates or the Btor2 format [17] 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 usage without an operating system, the machine code is typically in the Intel HEX format [18]. 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 formally[1], the informal

[1]For some processor architectures, the situation is starting to change for the part of guarantees that forms the Instruction Architecture Set (ISA), as noted in Section 3.1.

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 [14], 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 mix the characteristics:

- A *bytecode system* is a combination of the design in the form of bytecode for a Virtual Machine (VM), basic guarantees provided for the VM, 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 [19]. 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.

As this thesis is concerned with verification of machine-code systems, the other system levels will be discussed mainly in the context of their commonalities in the next subsection and of the state of the art in Chapter 3.

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 [20].

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 almost directly corresponds to the machine-code instructions) and assembled to machine code. The Instruction Set Architecture (ISA) description of the AVR architecture is publicly available [21].

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 [22]. However, these languages usually present some general overview of the system, not a fully specified system that can be used in the real world. In Chapter 3, I will discuss some languages for the formalisation of processor Instruction Set Architectures (ISAs). These are generally not directly possible to compile nor formally verify just using a standard compiler, requiring intricate translations to e.g. simulator programs or Automated Theorem Prover (ATP) formulas. I will show in Chapter 4 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 [23].

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 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 4.

2.1.3 Program Execution Environments

A typical microcontroller- or microprocessor-based digital system is logically divided into a processor core (or multiple cores), peripherals that communicate with the outside world, volatile Random Access Memory (RAM) and non-volatile storage. In the simplest setting (that nowadays occurs in *embedded systems* — that control devices from washing machines to spacecraft — rather than general-purpose computers), the processor executes a single machine-code program loaded from the non-volatile storage, which can run for an unbounded period of time (in practice, until the system is powered off). The program directly

interacts with processor peripherals, and is said to be running on *bare metal*: the processor itself (its core, peripherals, etc.) forms the *execution environment* of the program[2].

For user-operated computers, the limitation to a single program led to interpreters of e.g. the BASIC programming language, where a user could write and execute their own program. Unlike the interpreter, this program could *terminate*, returning back to the interpreter, which is more consistent with the original idea of computer programs for calculation as per Babbage and Turing. Interpreters as system-level programs were replaced by operating systems such as Unix, MS-DOS, Mac OS, and Windows, which themselves run as the system-level program and allow execution of other *user-mode* programs. In modern operating systems, the user-mode programs are completely separated from the processor peripherals and only interact with the execution environment in the form of the processor ISA and operating system Application Binary Interface (ABI).

The differences between the program execution environments lead to differences in verification mindset and parts of the systems considered. For conciseness, I will compare bare-metal and user-mode programs. System-level programs (in normal operating systems, system kernel and peripheral drivers) combine characteristics of both.

Termination. In bare-metal programs, there is no implicit notion of *termination*: the machine-code instructions are simply executed forever by unless the processor explicitly supports a switch to a different mode (e.g. sleep) and the program explicitly makes use of this. On the other hand, a user-mode program running in an operating system can terminate by performing a specified sequence of ABI operations.

Reasoning mindset. As a bare-metal machine-code program in an embedded system cannot terminate by normal means, it supports a mindset based on automata theory, where the system is an automaton that produces an infinite sequence of output symbols (describing e.g. high or low voltages on output processor pins). On the other hand, a user-mode program can (and is usually expected to) terminate at some point. This supports a mindset based on program analysis, where the traditional task is to determine if a program terminates and which outputs it produces depending on its inputs. Furthermore, it is possible to write user-mode programs that contain no loops at all (performing some calculation or task and terminating), while bare-metal programs must necessarily contain at least one loop to ensure that the system will be useful for an unbounded period of time.

Description of guarantees. In general, for all machine-code programs, it is necessary to describe the Instruction Set Architecture (ISA) so that the meaning of each executed program instruction is clear. This is sufficient for the verification of machine-code-program snippets that do not include any peripheral or ABI interaction. However, the peripheral or ABI interaction determines the input-output behaviour of the program and thus is critical for the verification of the program in full. For user-mode programs, the ABI is typically well-defined for each combination of a processor architecture and operating system family (e.g. x86, x86-64, AArch64 combined with Windows ABI or System-V ABI) and the system calls for console I/O operations are not exceedingly difficult to describe. On the other hand,

[2]Here, considering the program to be the *design* as defined at the start of this section, the execution environment is described by the *guarantees*.

the peripherals for bare-metal programs vary drastically for each processor model and may include complex state interactions, such as when the act of reading from or writing to one memory address modifies other memory addresses.

The discussed differences have far-reaching consequences for the verification approaches and system description choices, as will become apparent in Sections 2.2 and 3.1.

Example 2.1.6. In the ATmega328P datasheet [20], the system peripherals are described on approximately 270 pages. Even the fairly straightforward General Purpose I/O (GPIO) peripheral [20, p. 84-101] contains a subtle variation to functionality: reading from `PINx` registers reads the value present on the affected pins while writing to them instead toggles the bits in the corresponding `PORTx` register where the written value was 1 [20, p. 85]. This behaviour must be preserved during verification, as I did in my description of ATmega328P further discussed in Section 7.4.

2.1.4 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 2.1.7. 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. Programs that can terminate can be described as well, by outputting termination in the terminating state and remaining there.

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.

Note 2.1.8. This formalisation does not properly capture systems where incomplete guarantees are given. For example, the value in some register may not be specified after a division by zero, or different cores of a multicore system can perform reads and writes in an unspecified order. In such a case, we are technically not performing formal verification of a single system but a set of systems that fulfil the guarantees. We would then want to verify that all of the possible systems behave as required by the specification.

For common use-cases on single-core systems, it is enough to ensure that the incomplete guarantees cannot affect the result of verification so that traditional reasoning about a single system can be used. I will discuss how this is ensured in my tool **machine-check** in Subsection 7.3.1. General reasoning with incomplete guarantees is outside the scope of this thesis and is left for future work.

2.2 Fundamental Approaches to Formal Verification

The core task of programming and digital system design is implementing digital systems so that they work as intended. However, experience teaches us that this is not an easy task, and bugs crop up often. The easiest way to guard against bug is to try the system out with some chosen inputs: this is usually called *testing*. In software parlance[3], this can be considered *informal verification*, able to find bugs but not to guarantee their absence.

Formal verification allows us to guarantee that there are no bugs in the system that would make it violate the specification. While the implementing engineers typically use their intuition or manual formal proofs for some parts of the system to ensure correspondence to specification, formal verification techniques allow automation using computers. This requires formalising both the systems and specifications. I will give a cursory, mostly chronological overview of the development of the most fundamental approaches to formal verification. More information can be found e.g. in historical perspectives of principal authors [24, 25]. Introduced concepts most relevant to this thesis will be discussed in the next section.

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. Various approaches were devised for partially or completely automated verification of programs and systems.

The most important early work in formal verification of programs was the work by Floyd [26] and Hoare [27] in late 1960s, the introduced **Hoare logic** allowing verification of partial correctness (if the program returns an answer, it is correct) and total correctness (additionally, the program always returns). Notably, this reasoning is built on the theoretical concept of programs as algorithms (which can, and should, return) rather than as systems (which have no concept of "returning").

[3]In hardware parlance, *testing* determines whether the physical device matches the design (whether the system is manufactured correctly), while *verification* concerns matching the design against the specification (whether the system is designed correctly). In software, there is no manufacturing step.

Dijkstra called for limiting the scope of programs and proving their correctness as they are written in his 1972 Turing Award lecture [28]. While this is suitable for the most critical systems, the additional design complexity precludes the use in typical programs.

In 1975, Dijkstra provided a reformulation of Hoare's logic to *predicate transformer semantics*, providing an effective algorithm to convert imperative-language programs with assertions to formulas in predicate calculus [29]. Automated Theorem Provers (ATP) or Satisfiability Modulo Theories (SMT) solvers can then be used to prove or disprove the resulting formula. For programs with loops, *loop invariants* typically must be devised and provided by the user. Dijkstra used the notion of *weakest preconditions*, which correspond to the logic formula describing the program going backwards. The dual concept, describing the program going forwards, corresponds to later-introduced *strongest postconditions* [30].

Predicate transformer semantics are directly related to *symbolic execution* which produces symbolic states and results of the program similarly to the states and results arising from a standard program execution [31]. Traditional symbolic execution cannot formally verify programs with arbitrary loops, however, due to infinite execution possibilities.

Dissatisfied the need to manually devise loop invariants in predicate transformer semantics, Cousot & Cousot devised **abstract interpretation** inspired by compiler-related techniques of data-flow analysis. The techniques they introduced allow fully automatic verification of programs with loops [32].

The next breakthroughs came from research into concurrent programs and systems. Pnueli [33] used previous research on *modal logic* to express specification properties such as "eventually, ϕ will hold" or "from now on, ϕ will always hold", which later formed a type of *temporal logic*, the ubiquitous *Linear Time Logic*. In contrast to Pnueli, who considered the specification in context of linear program flow, Clarke and Emerson [34, 35] introduced the *Computation Tree Logic* logic based on the view of a branching tree of program executions. Since their verification task was to check whether a state-transition structure representing the concurrent system was a *model* of the specification, they called the corresponding problem **model checking**.

The major fundamental approaches (Hoare logic, abstract interpretation, and model checking) have enjoyed considerable popularity and extensions. It has also been shown that there is a strong link between model checking and abstract interpretation: for example, model-checking can resolve data-flow analysis in abstract interpretation [36]. While the three approaches have been historically developed by largely separate communities, they are becoming increasingly convergent [37, p. 2].

The most notable extensions to the approaches in the 21st century have been the introduction of *Counterexample-guided Abstraction Refinement* (CEGAR) for model checking, which will be discussed in Subsection 2.6.1, and *separation logic*, an extension of Hoare logic allowing reasoning about program components that use shared data structures [38].

Note 2.2.1. In my research, I have focused on machine-code verification using model checking, using the concepts from abstract interpretation for abstract domains where suitable. I did not focus on Hoare logic in my research and thus will not discuss it further except for its use in other tools introduced in in Chapter 3.

2.3 Property Specifications

Fundamentally, if we capture the exact behaviour of the system as a machine as per Subsection 2.1.4, we can try to prove or disprove (i.e. formally verify) **any** property of the system (model checking in the original mathematical meaning). However, formally verifying predicate calculus properties with respect to the machines **automatically** 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.

For practical model-checking, it is useful to write the specifications in some temporal logic instead. 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. They are a good compromise between the vast expressiveness of predicate calculus and low complexity of model-checking algorithms. 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 2.3.1. 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* [39, 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)[4]. 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 usual requirement is that the property must hold in all initial states to hold in the system itself [40, p. 57; 41, p. 392].

Example 2.3.2. Some of the more notable CTL* formula schemes are:

- **Safety.** $\mathbf{AG}[\phi]$, i.e. on all paths, ϕ holds forever.
- **Reachability.** $\mathbf{EF}[\phi]$, i.e. on some path, a state where ϕ holds is reached.
- **Recovery.** $\mathbf{AG}[\mathbf{EF}[\phi]]$, i.e. from every reachable state, there exists a path to a state where ϕ holds. 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.

Note 2.3.3. Safety and reachability can be used for verification of each other: if a signifies the state is a safe state, the system is safe exactly if all states are safe ($\mathbf{AG}[a]$) and, dually, the system is unsafe exactly if an unsafe state is reachable ($\mathbf{EF}[\neg a]$), meaning that it is safe exactly if $\neg\mathbf{EF}[\neg a]$.

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.

[4] 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 $\neg[(\neg\phi)\mathbf{U}(\neg\psi)]$.

- 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 **FG**ϕ could be more clearly expressed in CTL* as **AFG**ϕ [39, 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 [42, p. 907]. While the proofs in Chapter 5 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*.

Specifications in Hoare logic. In contrast to model checking, Hoare logic supports specifications expressed by arbitrary predicate calculus formulas at the expense of reduced automation. *Partial correctness* claims that if the program terminates, the execution fulfils the postcondition formula given the preconditions. *Total correctness* furthermore requires that the program always terminates. Total correctness is powerful enough that properties in linear-time temporal logics can be verified using it by encoding them to Hoare logic [43], but branching-time properties are generally not considered.

2.4 Formal Verification Using Model Checking

For finite-state state-transition systems, the model-checking approach can be used to prove or disprove the properties completely automatically. In the most basic view, the state space of the finite-state system is constructed and the properties are verified against the state space instead of the original system [39, 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 [39, 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 [39]. 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.

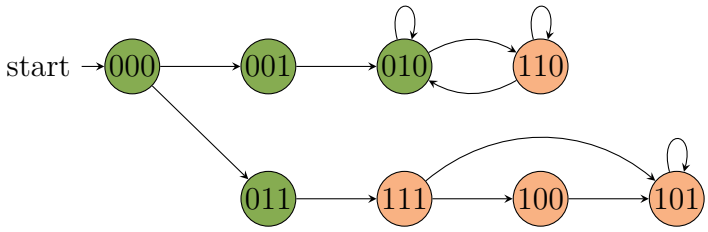


Figure 2.2: An example Kripke structure. This example will be reintroduced in Chapter 5.

2.4.1 Classic Model-Checking Formalisms

I will now introduce the classic formalisms for formal verification using model checking [39]. 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 2.4.1. 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 2.4.2. 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 2.6.

Example 2.4.3. Figure 2.2 shows a Kripke structure with $S = \{000, 001, \ldots, 111\}$, $S_0 = \{000\}$, $R = \{(000, 001), (001, 010), (010, 010), (010, 110), (110, 110), (110, 010), (000, 011), (011, 111), (111, 101), (111, 100), (100, 101), (101, 101)\}$. L labels the atomic proposition *msb* representing the most significant bit in states $\{000, 001, 010, 011\}$ as 0, and in $\{100, 101, 110, 111\}$ as 1.

The Moore machines from Section 2.1.4 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 5, 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 [39, p. 11].
- **LTL.** Running time depends linearly on the size of K and exponentially on the length of the LTL formula [39, p. 13].
- **CTL*.** The algorithms for CTL and LTL can be simply combined [40, 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 2.4.4. Let us consider formal verification of ATmega328P using naïve model checking. The I/O registers are reset during the device reset [20, 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.

2.5 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 [39, 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[5]. 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 [44] and later extended for use with SAT solvers [45]. 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.

Example 2.5.1. An example of a Binary Decision Diagram is shown in Figure 2.3. The table with 16 single-bit values indexed by four bits is stored using a tree structure. The value is determined by starting at the root node with indexing bit A and taking the left or right child according to the current node variable until a leaf node is reached, which determines the result value. This means that the example BDD returns 1 exactly if $(A = 1 \wedge B = 1) \vee C = 1 \vee D = 1$. This is useful for model checking as sets of states can be represented and manipulated using BDDs effectively, especially when bitwise operations are used.

[5]This is only generally possible for systems with left-total transition relations, as discussed by Dams & Grumberg [41, p. 392-393]. Fortunately, this includes practical digital systems.

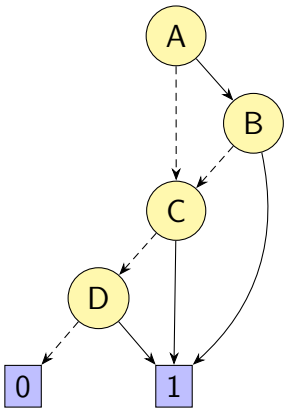


Figure 2.3: An example of a Binary Decision Diagram (BDD), of the classic Ordered Binary Decision Diagram type. The non-leaf nodes are shown as yellow circles and the leaf nodes are shown as blue squares. Each non-leaf node has two children, where the left child (dashed) corresponds to the node variable having the value 0 and the right child (solid) corresponds to the node variable having the value 1.

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 [46, 47], 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 the other tools discussed in Chapter 3.

2.6 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[6]. The three distinct valuations 0, 1, $\bot$ give rise to *three-valued logic*, which will be discussed in more detail later in Chapters 5 and 6.

[6]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.

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 2.6.1. 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 6). 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.

2.6.1 Methodologies

Now that the basic notions are in place, we can discuss the abstraction refinement methodologies. While Chapter 5 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 [41] 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. It is desirable (and usual) to keep $\hat{R}$ minimal under this condition while making sure to compute it efficiently [45].

Assuming that R is left-total[7], 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.

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 2.3.2, 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

[7]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 [41, p. 392-393]. Fortunately, digital systems expressible as automata have left-total transition relations (there is always at least one next state).

(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 5.

2.6.2 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 [48].
- **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 6.

Relational domains allow expressing relationships between variables, such as the octagon abstract domain [49]. The richest abstraction domain is predicate abstraction [50, 51, 52], where the abstract states retain information about whether some chosen predicate calculus formulas hold in them.

Note 2.6.2. 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[8]. 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 [3, 8, 5], an instance of this phenomenon already discussed in Example 2.4.4. 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 6.

Note 2.6.3. 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.

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 listed the main fundamental approaches to formal verification (Hoare logic, abstract interpretation, and model checking), focusing on the timeline of their introduction and the problems they were originally devised to solve.While developed largely separately, the approaches have been increasingly convergent. I discussed why standard predicate logic is not used for formal verification using model checking, noting the problems with feasibility

[8] 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.

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 the approach of model checking in more detail and defined the classic formalism based on Kripke structures. 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. The choice of abstraction domains is of crucial importance for efficient verification as well.

While I am focusing on formal verification via model checking, other tools for machine-code verification have been also devised using the other approaches, as will be described in Chapter 3. After discussing the state of the art, I will present the techniques that I devised during my studies in Chapters 4, 5, and 6, culminating in a odiscussion of my created tool **machine-check** that uses the techniques and a comparison to the previous tools in Chapter 7.

Chapter 3

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 focus on freely available verification tools where possible. Even though the basic techniques used in commercial tools may be similar to the ones in the freely available tools, the details are not well-known.

3.1 Machine-Code Systems

I identified three major, largely unconnected directions of research into formal machine-code verification. I will summarise the directions here and discuss them in the following subsections:

- Formal verification of simple embedded systems using model checking. It is possible to handle loops fully automatically and verify complex temporal properties at the danger of easily occurring exponential explosion preventing verification. Exemplified by the **Arcade.µC** tool [3, 4, 5, 6, 7].

- Machine-code program analysis, especially using abstract interpretation, used for decompilation and formal verification of user-mode executables for personal computers. Exemplified by the **CodeSurfer/x86** tool [53, 54, 55].

- Formal verification of machine-code snippets by translation to Hoare or similar logic formulas followed by proving properties via Automated Theorem Provers (ATP). Only safety properties are typically considered, and verifying programs with loops requires manual help from the user or is not supported at all. Exemplified by the **Islaris** [56, 1] and **Serval** [57, 2] tools.

As this thesis is concerned with machine-code verification using model-checking with abstraction, the first direction is the most relevant, inspiring the techniques used in **machine-check**. The direction using automated theorem proving is currently the most active and may be a good source of inspiration for future work.

It will be seen that in all directions, after initial attempts, the researchers tend to determine it is necessary to construct a language for describing the processor Instruction Set Architecture (ISA), and in the case of embedded system verification, also the peripherals.

3.1.1 Model-Checking Direction

A formal verification tool for embedded systems **HOIST** was described by Regehr and Reid [58], and used for stack size estimation [59]. 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 Regehr and Reid's 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 [60]. **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 [61]. 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 in 2006 by Schlich and Kowalewski [3][1], 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 [8], including delayed instantiation of the variables after masking of inputs by logical instructions so that state-space explosion is mitigated [4]. 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 [5, 6]. Interval abstraction was also added, working in concert with three-valued bit-vector abstraction to provide a further reduction of the abstract state space [62].

To reduce the difficulty of adding new microcontroller types and architectures to **Arcade.µC**, synthesis of state space generators was developed in 2014 by Gückel [7]. This

[1]The paper [3] 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.

approach was not entirely successful as Gückel was unable to implement abstraction in such a way that would not require the description writer to tailor the description to it [7, 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 [63]. Furthermore, while **Arcade.µC** used three-valued bit-vectors for abstraction [8], it was not possible to compute arithmetic operations in the abstraction, forcing exponential explosion.

In 2021, I created a C++ tool for verification of deadline specifications for my master thesis [A.4], using abstract state space and limited support for refinement, inspired especially by **Arcade.µC**. The tool used a special description language for processors and supported verifying whether an action ϕ is necessarily followed by a reaction ψ in a given time, as in $\mathbf{AG}[\phi \rightarrow \mathbf{AF}\ \psi]$ but with bounded time allowed for the reaction. While the tool was able to verify properties of very simple programs, the implementation of deadline checking was quadratic in the size of the state space rather than linear as with usual CTL or LTL checking algorithms, making its usability problematic. In addition, the custom specification language was bothersome to support and extend.

3.1.2 Program-Analysis Direction

In 2000, Xu et al. [64, 65] developed a tool to determine whether it was safe to run untrusted machine code in a host computer given source-level typestate properties, using a sequence of techniques including abstract interpretation and automated theorem proving. In 2004, Balakrishnan et al. introduced a static analysis tools for x86 executables **CodeSurfer/x86** [53, 54, 55] using overapproximation of possible program behaviours, and introduced model-checking using it [66]. Another tool **Device-Driver Analyzer for x86** (**DDA/x86**) was introduced by Balakrishnan and Reps [67, 68]. Both use static analysis with overapproximation and abstraction refinement [69].

Lim & Reps introduced a Transformer Specification Language (TSL) for writing ISA specifications to be used in static program analysis using abstract interpretation [70, 71]. The language was used in a machine-code verification tool **MCVETO** [72, 69], which combines overapproximation by abstraction with concrete program execution traces which under-approximate the program behaviour and guide the refinement as per the SYNERGY algorithm [73]. It seems that the focus of Reps et al. shifted to executable resynthesis after 2013 [74, 75, 76, 77] and largely away from machine code after 2017.

3.1.3 Automated-Theorem-Proving Direction

Unlike the other two directions, which do not seem to currently enjoy much attention, formally verifying properties of machine-code programs and ISAs themselves using automated theorem provers, possible through translation to formulas by Hoare logic or similar approaches, has seen a lot of both non-recent and recent development. I will describe

selected historical developments and more recent state-of-the-art tools. Additional information can be found e.g. in the thesis of Sammler [78].

Nqthm. Boyer & Yu used an automated reasoning system Nqthm to verify machine-code programs for the Motorola 68020 processor, formalising a subset of its ISA[79]. They were able to formally verify various programs and snippets, including an implementation of the classic Euclidean Greatest Common Divisor algorithm, compiled classic C code such as the binary search and Quick Sort algorithm, and parts of the Berkeley Unix C string library, finding three C programming errors [79, p. 186-187]. However, the verification required a good amount of work: building the library of lemmas to support proving took many months, and using it, verification of correctness of another simple function would still take Boyer & Yu a few hours [79, p. 190].

Symbolic execution. Currie et al. [80] used symbolic execution to prove equivalences between pieces of machine code written for different architectures.

HOL4-assisted approach. Myreen et al. performed machine-code verification using the HOL4 theorem prover by translating the a snippet of machine code into a tail-recursive function, implementing the translation for the Arm, PowerPC, and x86 architectures [81, 82]. The snippet must be completely deterministic, and the program control flow is computed heuristically, with the possibility of problems caused by e.g. indirect procedure calls [82, p. 7].

L3 description language. Fox developed an ISA description language [83], later named L3, and combined the approach of Myreen et al. with L3 for easier addition of ISA support [84]. This has been used for the verified compilation of the ML language in the CakeML project, avoiding the problem of the CompCert compiler where the assembly-language-level compilation outputs are not verified [85].

x86 ISA in ACL2. Goel et al. [86] developed a formal specification for the x86-64 architecture for the ACL2 theorem prover. They only mentioned the actual verification of machine code in a cursory way, but were able to verify straight-line machine-code program snippets using SAT solving.

Sail description language. The Sail description language was introduced in 2015 to describe fragments of user-mode ISA of weakly consistent multiprocessors in the context of their concurrent behaviour [87, 88]. In 2019, the language was extended to support describing the full ISA of contemporary processor architectures including ARMv8A, RISC-V, and CHERI-MIPS [89], with automatic translation to sequential emulators as well as formulas for popular theorem provers OCaml, Isabelle/HOL, and HOL4. To demonstrate the usability for formal verification, a non-trivial address translation property of ARMv8A was proven in Isabelle/HOL [89, p. 25-26]. Following this, the RISC-V specification in the Sail language was selected as the official formal RISC-V specification [90].

Serval. Nelson et al. developed **Serval** [57, 2], a framework for verification of machine-code programs based on symbolic execution, built on top of the Rosette solver-aided programming language (which itself is backed by an SMT solver such as Z3). They emphasize the "push-button" approach to verification, where no user interaction is necessary, at the cost of simple properties (safety properties encoded in the instruction set description, a specification state machine, and additional predicates). Nelson et al. built interpreters of

RISC-V, x86-32, LLVM, and Berkeley Packet Filter (BPF), which are converted to verifiers by Rosette. They were able to verify properties of two *security monitor* programs that provide software isolation, porting them to a RISC-V platform. The fact that the security monitor programs have finite interfaces with bounded trace lengths allowed their verification with Serval. The Serval approach is severely limited in its verification power: in addition to only supporting safety properties, it only allows for programs with finite-length execution traces, disallowing any infinite loops. This precludes it from verifying many programs, including bare-metal programs intended to run for an unbounded amount of time (as discussed in Subsection 2.1.3).

Islaris. In 2021, Armstrong et al. introduced the Isla tool that allows symbolic execution of machine-code snippets using the Sail specification [91], intended for testing of memory concurrency behaviour. In 2022, Sammler et al. presented Islaris, verifying machine-code snippets for ARMv8A and RISC-V [56, 1]. They noted that the complexity of the specification translated to formulas precluded feasible verification using just a theorem prover [56, p. 826], instead using the Isla tool to generate symbolic-execution traces — using the Iris higher-order separation logic framework [92] — which are much simpler as they apply directly to the snippet under verification. The Coq theorem prover is then used to verify the trace, potentially using manually provided proof steps, ensuring that the assumptions used by the Isla tool hold and proving user-defined safety properties about externally visible program behaviour. Modified Hoare logic is used, with loop invariants necessary to be discovered (manually or with the aid of the theorem prover). Sammler et al. proved properties of 7 example snippets ranging from 1 to 47 assembly instructions, including a `memcpy` function, where it was necessary to manually give hints to the theorem prover to prove a loop invariant [56, p. 831].

Katamaran. The Katamaran project [93, 94] also uses symbolic execution for subsequent verification using Coq but is targeted towards verification of the guarantees of the ISA itself, such as secure enclaves or Capability Hardware Enhanced RISC Instructions (CHERI), against instruction semantics.

Notably, Islaris [1], Serval [2], and Katamaran [94] are available online as free and open-source projects. I have examined **Islaris** and **Serval** practically and was able to run the examples. Unfortunately, for actually specifying and proving properties, they require specialist knowledge of the Coq theorem prover and Rosette solver-aided programming language, respectively.

3.1.4 Comparison of Verification Directions

In each of the three directions, the approach used was determined by the specifics of the verification problem. The original model-checking and Hoare-logic approaches can be thought of as antipodes: naïve model checking is fully automated from the start, and Hoare logic is based on largely manual proofs, which can be automated to some extent by using Automated Theorem Provers. The program-analysis direction lies somewhere in the middle between them.

While the effort on previous tools from the model-checking and program-analysis directions was largely abandoned, the automated-theorem-proving direction currently enjoys much attention, notably spurred on by the work on RISC-V and Sail description language. The **Islaris** and **Serval** tools in particular are highly interesting for formal machine-code verification, although they require specialist knowledge of automated theorem proving (including the actual prover used) from the user. While model-checking tools may only be practically usable for simpler systems and specifications, they can be used for arbitrary systems that include loops fully automatically, without much specialist knowledge.

3.2 Other System Levels

Source-code and hardware verification tools are used in practice with a variety of competing tools [95]. While they posit substantially different challenges than machine code, inspiration can be taken from the tools and verification directions. I will also note bytecode and microcode verification efforts.

3.2.1 Source-Code

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 [96], 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 [96, 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 [97].

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 [98]. 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**[2] [99] and

[2]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

Ultimate Automizer, part of the **Ultimate** program analysis framework [100]. 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 [96, 101]. Further techniques are used to extend this basic concept or introduce additional improvements, but they are beyond the scope of this thesis.

Note 3.2.1. Bytecode was also used for verification. In the main part of the competition, the **DIVINE** model checker, which uses LLVM IR bytecode [102], 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** [103], **JBMC** [104], and **GDart** [105] all use JVM bytecode. All of the hors-concours tools used the JPF framework [96, p. 308-310], which works with JVM bytecode as well [106].

3.2.2 Hardware

Hardware verification has been widespread in industrial practice since the turn of the century [95]. Open-source tools are more scarce. The main hardware system formal verification competition is the Hardware Model Checking Competition (**HWMCC**), the latest **HWMCC** at the time of writing this thesis held in 2024 in Prague [107], with 13 teams participating, showing a distinctly lower popularity than source-code verification. Out of these tools, the model checker **rIC3** performed the best, followed by **AVR** (Abstractly Verifying Reachability) [108]. Both **rIC3** and **AVR** use a portfolio of verification strategies [107], employing the IC3 algorithm used with SMT solvers, Bounded Model Checking, and k-induction. Notably, the algorithms for **rIC3** are implemented in the Rust programming language.

The IC3 algorithm is based on to refining sets of states in steps reachable from the initial states, the sets of states determined by invariants that hold [109]. Bounded model checking allows to refute properties in a finite number of steps (especially using SAT solvers) [47], while k-induction [110] can also be applicable to prove they hold for an infinite number of steps using an inductive invariant strengthened to sequences of multiple (k) states [47, p].

Also notable is the **ABC** solver [111] used in the previous 2020 HWMCC [112], which is much more hardware-specific, based on single-bit handling via And-Inverter Graphs, using multiple algorithms to handle verification, one of them abstraction refinement [111, 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 [113]. 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.

verification of programs that use the standard C library memory-manipulation functions `memset`, `memcpy`, and `memmove`, and improved the treatment of quantifiers.

Note 3.2.2. Microcode can be considered to lie somewhere between hardware and machine-code verification. Little has been published on formal verification of microcode but fairly comprehensive summaries of related work have been presented by Davis et al. [114] and Goel et al. [115]. 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 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 [116].

3.3 Research Decisions

Formal verification of source-code systems is the most popular in the open-source community, with hardware systems more popular in industrial practice, with fewer open-source tools available. Formal verification of machine-code systems seems comparatively under-researched, although there have been significant steps in the Hoare logic direction. A large part of the difficulty 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 (or the size of formulas for Hoare-style approaches) is not infeasibly large but the verification is still useful. This contributes to the absence of standard test sets, as the machine-code is intrinsically tied to the underlying architecture (this is comparable to the source-code level, where test sets written in C dominate, followed by Java).

ISA descriptions. The research into formal ISA descriptions is significant especially in the Hoare logic direction, currently centred around the Sail description language in which the official formal description of RISC-V is formalised. However, the focus on more complex architectures and description of the ISA without peripheral considerations are problematic in the context of model-checking, which has been traditionally used for programs on embedded 8-bit microcontrollers due to state space explosion problems.

Existing machine-code verification tools. As for the tools such as **Serval** and **Islaris**, I believe that they are well-suited for verifying critical routines (especially in the context of possible bugs affecting security), the need to use special automated-proving-related tools and knowledge restricts them to the use by well-trained specialists for parts of truly critical systems. While the tools do not seem ready for industrial use yet, it seems prospective given further development. In addition, it also is sensible to develop tools that are fully automatic and as simple as possible to use even at the expense of the number of properties verifiable in reasonable time and memory, as exemplified by **Arcade.µC** for machine code and and the tools participating in SV-COMP and HWMCC, which can be used by less trained programmers for less critical systems as well.

Specifications. It is worrying to me that much research focuses on simple safety or (un)reachability properties, despite their lack of expressiveness. Specifically, branching-

time logics offer a different style of thinking than linear-time logics, which can be suitable to revealing different bugs. In Subsection 7.4.6, I will discuss how I found one such bug using a recovery property not expressible in a linear-time logic.

At the outset of my doctoral research, I was well-aware of the model-checking-style research into machine-code verification exemplified by **Arcade.µC**, on which I previously based the tool in my master thesis [A.4], and decided to focus on the points that limited the usability of my previous tool, devising novel techniques that will be discussed in the rest of this thesis. I will now give the reasons for my decisions in devising the techniques:

- In Chapter 4, I focus on enabling machine-code verification without tailoring to a specific architecture using translation of simulable processor descriptions that are written in the Rust language. The reason to use the Rust language instead of a specific description language[3] is simple: it is possible to compile it using a standard compiler (both the system description itself and the abstraction-refinement translations), which is not possible for description languages.

- In Chapter 5, I introduce a novel TVAR framework, so that arbitrary µ-calculus properties (including CTL*, CTL, and LTL properties) can be verified with abstraction refinement. This means the abstraction techniques can be used fully without falling back to a logic such as LTL or ACTL as in the case of **Arcade.µC** [4].

- In Chapter 6, 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. Importantly, this allows the TVAR framework from Chapter 5 to be useful in the face of combinations of bitwise manipulation and arithmetic operations.

In Chapter 7, I will discuss how I combined the techniques in my new tool **machine-check**, provide experimental results that show that is usable for machine-code verification, and compare its capabilities with other tools for machine-code formal verification I identified as important.

The introduced techniques allow **machine-check** to use arbitrary system descriptions and support abstraction refinement, something that was lacking in **Arcade.µC**. Unlike the tools such as **Islaris** or **Serval**, the verification is performed fully automatically without requiring much user training. Furthermore, the use of Three-valued Abstraction Refinement allows verification of branching-time verification properties. While the ATmega328P processor description was hand-written using the available documentation, translation of official formal ISA descriptions to **machine-check** descriptions is theoretically possible and may be added in the future.

[3]Such as the Sail description language, although I was not aware of it during much of writing the thesis.

Chapter 4

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 [22], 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 7.

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.

Note 4.0.1. Sections 4.1 and 4.2 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.

As **machine-check** is used as a library by the description writer, the functionality of the built verifier executable depends on the description writer. In simple cases such as Figure 4.3, the behaviour of **machine-check**, including the property to verify, is determined by command-line arguments. A Graphical User Interface (GUI) can also be opened for more comfortable verification. More information about these details can be found in the **machine-check** user guide[1]. I will not discuss them closely in this thesis.

4.1 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 4.1. 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.

[1]The user guide for **machine-check** is available at `https://book.machine-check.org`. The user guide for the latest version of **machine-check** at the time of writing is available at `https://book.machine-check.org/0.4.0`.

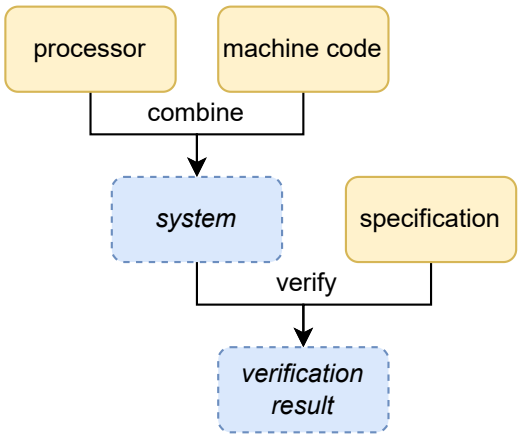


Figure 4.1: A high-level overview of formal verification of machine-code systems. The solid yellow cells represent inputs, while the dashed blue cells represent automated results. The processor and machine code are combined to form the system under verification. It is then determined if the specification holds or does not hold in the system. This figure is a specialisation of Figure 2.1 for machine-code systems.

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 4.2. 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 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) [34] 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).

4.2 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.

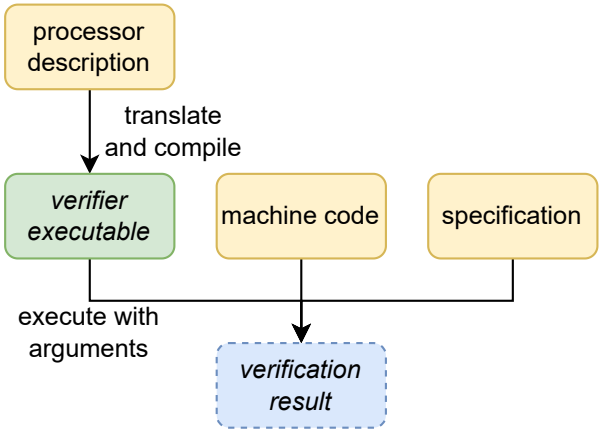


Figure 4.2: A high-level overview of **machine-check** machine-code system verification process. The processor description is translated to verification analogues, then compiled together with verification control algorithms to form a verifier executable for the given processor, visualised in a solid green cell. The verifier is executed with the machine code and specification given as arguments, performing formal verification as in Figure 4.1. The compilation step ensures a speed gain over interpretation. Additional guarantees beyond the processor descriptions are not considered in this chapter for simplicity.

A simulable description of a very simplified RISC microcontroller[2] is shown in Figure 4.3. The description is written in a subset of valid Rust code (which will be described later in Section 4.3), using specially provided **machine-check** types for simple transcription of behaviour from datasheets. The machine-code system described in Figure 4.3 can be immediately simulated in Rust by instantiating the `System` structure, with the machine 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 4.3, 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 4.3, 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

[2]The whole description is available at `https://docs.rs/crate/machine-check/0.4.0/source/examples/simple_risc.rs`.

```
#[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 4.3: Example description of a simplified RISC microcontroller as a finite-state machine. Less important code details are omitted for conciseness and readability.

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 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 4.4, the example system from Figure 4.3 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 4.3 and 4.4 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, 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.

4.3 Subset of the Rust Language Usable in Descriptions

Having shown an example of a simulable description in Figure 4.3, 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 4.4.

Note 4.3.1. 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[3]. In the electronic version of this thesis, the underlined constructs link to the appropriate parts of the Rust Reference.

[3]The latest version of the Rust Reference is available at `https://doc.rust-lang.org/stable/reference/`. In **machine-check** 0.4.0, the current version at the time of writing of this thesis, the minimum supported Rust version is 1.83.0, with the corresponding version of the Rust Reference at `https://doc.rust-lang.org/1.83.0/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 4.4: Example of code for verification of a machine-code system based on the simplified RISC processor from Figure 4.3. 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 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.

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 4.3.2. In Figure 4.3 and Figure 4.4, 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 4.3, 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[4], as well as unary arithmetic negation (`-`) and bitwise NOT (`!`). Special 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.

4.4 Further Notes

This chapter was a light introduction to the simulable descriptions of processors from the viewpoint 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, although

[4]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.

recompilation is necessary each time the system is changed. A more straightforward use of this possibility is the translation of a processor description (or a part of it) from another formally specified language, such as the Sail language discussed in Subsection 3.1.3. This could be used to translate e.g. parts of the official RISC-V Instruction Set Architecture (ISA) description from Sail in the future.

The implementation of the translation in the macro `machine_description` that produces the verification analogues will be described in more detail in Chapter 7, once the interacting techniques from Chapters 5 and 6 are also introduced.

Chapter 5

Input-based Three-valued Abstraction Refinement

In this chapter, I describe a novel abstraction framework based on Three-Valued Abstraction Refinement (TVAR) that I devised during my doctoral study, with formal proofs that it can be used for model checking with abstraction refinement. The framework 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 is implemented in my free and open-source verification tool **machine-check**.

Note 5.0.1. This chapter is based on the contents of an available preprint [A.3], reworked for inclusion in this thesis. As the paper was a joint work with my supervisor, 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 simplification of formalisms and proofs and overall readability.

Abstraction-refinement methodologies are ubiquitous in formal verification, the foremost being Counterexample-guided Abstraction Refinement (CEGAR) [45, 44]. Unfortunately, CEGAR does not support the whole propositional µ-calculus or even Computation Tree Logic (CTL), being in principle suitable for logics such as linear-time µ-calculus, ACTL*, ACTL, or LTL [41, p. 404]. This leaves a large class of potentially crucial non-linear-time-properties unverifiable. Three-valued Abstraction Refinement (TVAR) is able to verify full µ-calculus.

Previous TVAR frameworks refined abstract states, necessitating state space formalisms based on modal transitions. Frameworks based on simple formalisms [117, 118] are not monotone: previously provable properties may no longer be provable after refinement. Intricate monotone formalisms [119, 120, 121] were devised, their specialised semantics complicating the use of standard model-checking algorithms. We conjecture that the conceptual complexity of those approaches is one of the reasons for the lack of available TVAR tools (with one exception [122]), compared to CEGAR.

Inspired by simulation-oriented abstract state space generation in (Generalized) Symbolic Trajectory Evaluation [123, 124, 125, 126] and Delayed Nondeterminism [4], which

were both restricted to proving linear-time properties, we present a novel TVAR framework that does not use modal transitions. The framework is based on a simple Partial Kripke Structure (PKS), allowing direct application of standard model-checking algorithms. It also allows for precise control of the size of the reachable abstract state space during each refinement. We prove that the introduced framework is sound, monotone, and complete for μ-calculus properties and existential abstraction domains, provided simple requirements are met.

We implemented a machine-code verification tool that instantiates our framework and successfully used it to find a bug in a machine-code program using a property unverifiable by CEGAR-based tools. We also show effective mitigation of state-space explosion by evaluating on synthetic automata.

The generality of the framework allows its instantiation to arbitrary systems based on automata, its conceptual simplicity and the possibility to directly use standard model-checking algorithms are beneficial for the implementation of corresponding tools, and its flexibility allows the design of efficient heuristics.

5.1 Previous Work

In this section, we list previous relevant work on TVAR in roughly chronological order, with additional information available in summarising papers [41, 127]. After that, we discuss similar work based on abstract simulation.

Example 5.1.1. Consider a finite-state machine in Figure 5.1, representing e.g. a controller of aircraft landing gear retraction: if the most significant bit (*msb*) of the state is 0, the landing gear is extended; if 1, it is retracted. The system is required to follow a single-bit input from the gear lever, with some slack for responses. There is a critical bug, occurring if the aircraft loses power in flight when the landing gear is retracted, and the controller restarts in the state 000 after the power is regained: since the landing gear lever is set to retraction, the controller goes through the states 011 and 111, causing a **total loss of capability** to extend the gear again unless the controller is turned off and on again.

The bug is not just dangerous, but also sneaky, as it does not occur in normal aircraft operation. To protect ourselves against it, we can verify the property "from every reachable system state, it should be possible to reach a state where the landing gear is extended" holds in the system. This is formalised by a *recovery property* $\mathbf{AG}[\mathbf{EF}[\neg msb]]$ in CTL, not possible to check using CEGAR.

Since the system is buggy, the property should be disproved[1]. We will instead reason about proving a dual property, $\mathbf{EF}[\mathbf{AG}[msb]]$. In Figure 5.1, *msb* definitely holds in the state 101, and since 101 just loops on itself, $\mathbf{AG}[msb]$ holds in it. As 101 is reachable from 000, $\mathbf{EF}[\mathbf{AG}[msb]]$ holds, and the bug is found. While such reasoning is easy for simple

[1]In our terminology, *proving* the property determines it holds in the system. *Disproving* it determines it does not. *Verification* aims to either prove or disprove it.

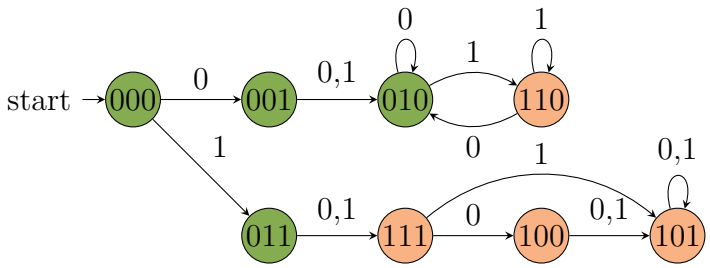


Figure 5.1: Example system expressed as a finite-state machine. The states where $\neg msb$ holds are drawn green while the states where msb holds are drawn orange.

systems, in real life, the controller may have billions of possible states, requiring us to abstract some information away[2].

Partial Kripke Structures (PKS). In verification on *partial state spaces* [128], some information is disregarded to produce a smaller state space. PKS enrich standard Kripke structures (KS) by allowing unknown state labellings.

Definition 5.1.2. A partial Kripke structure (PKS) is a tuple (S, S_0, R, L) over a set of atomic propositions $\mathbb{A}$ with the elements

- S (the *set of states*),
- $S_0 \subseteq S$ (the *set of initial states*),
- $R \subseteq S \times S$ (the *transition relation*),
- $L : S \times \mathbb{A} \to \{0, 1, \bot\}$ indicating for each atomic proposition whether it holds, does not hold, or its truth value is unknown (the *labelling function*).

A Kripke Structure (KS) is a PKS with L restricted to $S \times \mathbb{A} \to \{0, 1\}$.

Example 5.1.3. While Figure 5.1 shows a finite-state machine, it can be converted to a Kripke structure by discarding the inputs, with $S = \{000, 001, \ldots, 111\}$, $S_0 = \{000\}$, and R given by the transitions in Figure 5.1. L labels msb in states $\{000, 001, 010, 011\}$ as 0, and in $\{100, 101, 110, 111\}$ as 1.

For proving $\mathbf{EF}[\mathbf{AG}[msb]]$, such a KS is unnecessarily detailed. Using PKS, we could e.g. combine 010 and 110 into a single *abstract* state where it is unknown what the value of the most significant bit is, and the labelling of msb is $\bot$.

Existential abstraction. In TVAR, *existential abstraction* is used, where the abstract states in set $\hat{S}$ are related to the original concrete states in S by a function $\gamma : \hat{S} \to 2^S$, the abstract state $\hat{s} \in \hat{S}$ representing some (not fixed) concrete state in $s \in \gamma(\hat{s})$ in each system execution instant. This is a generalisation of Abstract Interpretation domains, also allowing e.g. wrap-around intervals [129].

[2]The example is directly inspired by a bug we found, discussed in Subsection 7.4.6.

Example 5.1.4. In the examples, we will use the three-valued bit-vector domain, where each element is a tuple of three-valued bits, each with value '0' (definitely 0), '1' (definitely 1), or 'X' (possibly 0, possibly 1). Except for figures, we write three-valued bit-vectors in quotes, e.g. $\gamma(\text{“0X1”}) = \{001, 011\}$. The bits can also refer to a predicate rather than a specific value. For example, '1' could mean that $v > 5$ holds, '0' that its negation holds, and 'X' that we do not know.

5.1.1 Previous TVAR Frameworks

Building on the work of Bruns & Godefroid [128, 130, 131], Godefroid et al. [117] introduced TVAR by refining the abstract state set, using a state space formalism based on modal transitions. Early TVAR approaches [117, 132, 118, 133, 134] were based on Kripke Modal Transition Structures (KMTS) and did not guarantee previously provable properties stay provable after refinement, i.e. were not monotone.

Definition 5.1.5. A Kripke Modal Transition Structure (KMTS) is a tuple $(S, S_0, R^{\text{may}}, R^{\text{must}}, L)$ where S, S_0, and L follow Definition 5.1.2, and

- $R^{\text{may}} \subseteq S \times S$ 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.

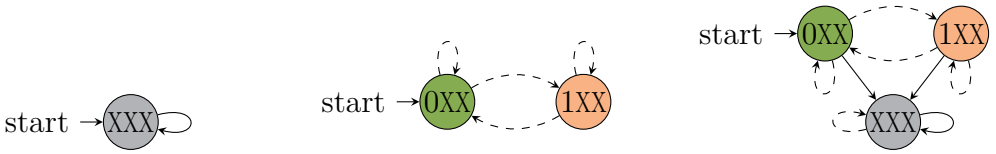
Intuitively, KMTS allow for transitions with unknown presence ($R^{\text{may}} \setminus R^{\text{must}}$). PKS can be trivially converted to KMTS by setting $R^{\text{may}} = R^{\text{must}} = R$. While it is possible to convert a KMTS to an equally expressive PKS by moving the transition presence into the states [135], it requires the set of states to be modified.

Monotone frameworks. Godefroid et al. recognised non-monotonicity as a problem and suggested keeping previous states when refining [117, p. 3-4]. However, Shoham & Grumberg showed the approach was not sufficient, as the refinements did not allow verifying more properties, and introduced a monotone TVAR framework using Generalized KMTS for CTL [119], later extended to µ-calculus [136]. Gurfinkel & Chechik introduced a framework for verification of CTL properties on Boolean programs using Mixed Transition Systems [120], later extended to lattice-based domains [137][3]. Wei et al. introduced a TVAR framework using Reduced Inductive Semantics for µ-calculus under which the results of model-checking GKMTS and MixTS are equivalent [121].

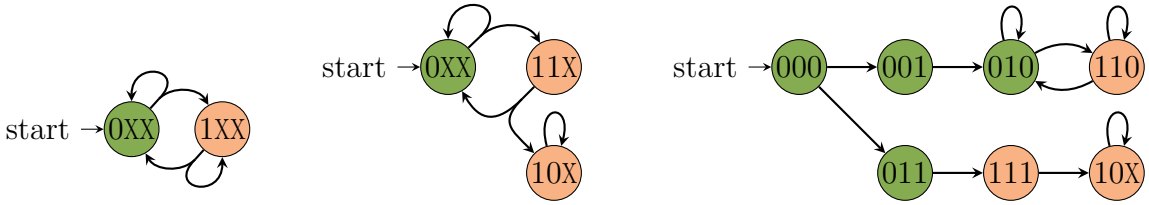
Definition 5.1.6. A Generalized KMTS (GKMTS) is a tuple $(S, S_0, R^{\text{may}}, R^{\text{must}}, L)$ where S, S_0, R^{may}, and L follow Definition 5.1.5 and $R^{\text{must}} : S \times 2^S$ is the set of *hyper-transitions*, where $\forall (a, B) \in R^{\text{must}} \,.\, \forall b \in B \,.\, (a, b) \in R^{\text{may}}$.

Definition 5.1.7. A Mixed Transition System (MixTS) is a tuple $(S, S_0, R^{\text{may}}, R^{\text{must}}, L)$ where S, S_0, R^{may}, and L follow Definition 5.1.5 and $R^{\text{must}} \subseteq S \times S$.

[3] An instance of the framework is implemented in the tool Yasm [122], available online at the time of writing [138]. However, we found that it does not support language elements such as bitwise-operation statements (e.g. $x = x \,\&\, 1$), which our machine-code verification tool focuses on.



(a) Only state "XXX". (b) KMTS after refining (a). (c) MixTS after refining (a).



(d) GKMTS after refining (a). (e) GKMTS after refining "1XX" in (d). (f) GKMTS after refining "0XX" in (e) by splitting to "000", "001", "010", and "011".

Figure 5.2: State-based refinement with hyper-transitions based on Generalised KMTS. Implied may-transitions, present in all sub-figures except for (c), are not drawn. The states where it is unknown whether msb or $\neg msb$ holds are drawn grey.

Example 5.1.8. We will prove **EF**[**AG**[msb]] using state-based TVAR over the system from Figure 5.1, starting with abstract state set {"XXX"}. Clearly, we both **may** and **must** transition from "XXX" to "XXX", visualised in Figure 5.2a. Since msb is unknown in "XXX", the model-checking result is unknown and we refine.

Suppose we decide to split to {"0XX", "1XX"}, starting in "0XX". From "0XX", we **may** transition either to "0XX" (e.g. by $000 \to 001$ or $010 \to 010$) or "1XX" (e.g. by $010 \to 110$), but cannot conclude that e.g. a transition from "0XX" to itself **must** exist: $011 \in \gamma(\text{"0XX"})$ only transitions to $111 \notin \gamma(\text{"0XX"})$.

PKS cannot be used as they cannot describe unknown-presence transitions. KMTS allow this, producing Figure 5.2b. However, it is not possible to prove e.g. **EX**[**true**], which was possible in Figure 5.2a, i.e. the refinement is not monotone. Using MixTS, we retain "XXX" and the must-transitions to it, producing a *forced choice* in Figure 5.2c. Using GKMTS, we obtain Figure 5.2d instead. In both, it is possible to prove **EX**[**true**], but not **EF**[**AG**[msb]]. Refining further using GKMTS, we obtain Figure 5.2e, where it is still not possible to prove **EF**[**AG**[msb]]: the hyper-transitions do not imply that the path ("0XX","11X","10X") corresponds to a concrete path. The property is only proven after additional refinement to Figure 5.2f. While the final GKMTS trivially corresponds to KMTS or PKS, in general, fewer refinements may be needed using GKMTS or MixTS, and they may guide the refinement better due to monotonicity.

Model checking. µ-calculus properties can be model-checked on PKS and KMTS by a simple conversion to two KS, applying standard model-checking algorithms, and combining the results [130, 132]. Similar conversions are also possible for multi-valued logics [139, 140], although a multi-valued model-checker was used for the MixTS approach [120].

Discussion of previous TVAR frameworks. It was recognised early on that using KMTS with non-monotone refinement is problematic [117, 118]. GKTMS seem more susceptible to exponential explosion as MixTS can make use of abstract domains. However, specialised algorithms must be used for MixTS to obtain GKMTS-equivalent results [121]. A drawback of all mentioned approaches is their conceptual complexity which, in our opinion, is the main reason for the dearth of available TVAR tools and test sets, compared to CEGAR. This has also made the analysis of these methods difficult, as illustrated by the subtle differences in definitions of expressiveness identified by Gazda & Willemse [141].

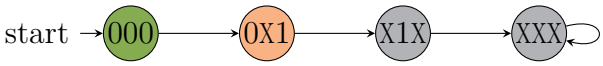
5.1.2 Simulation-Splitting Approaches

The mentioned TVAR frameworks first choose the set of abstract states, then compute the transitions. We identified this as the reason why KMTS, GKMTS, or MixTS are necessary, as opposed to simple PKS. We now discuss non-TVAR abstract techniques that start in the initial state(s) and build the state space iteratively, by *simulation*. Simply building the state space as in Figure 5.3a produces a PKS. The question is how to refine without modal transitions. We consider two notable approaches where soundness is guaranteed only when proving linear-time properties (no "false positives"), focusing on how the simulation is split.

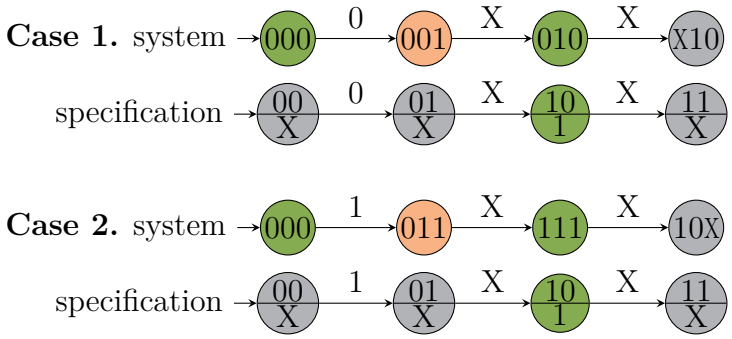
Trajectory Evaluation. Bryant used three-valued simulation, widely available in logic simulators, for formal verification of hardware circuits [123, 125]. He showed linear-time properties (expressed by specification machines or circuit assertions) can be proven using a set of three-valued input sequences that together cover all concrete inputs [123, p. 320] by generating permissible state sequences, i.e. *trajectories* [124]. *Symbolic trajectory evaluation (STE)* is an extension that allows parametrisation of the introduced *trajectory formulas* [124]. However, the STE formalism drops the distinction between inputs and states, treating inputs as a part of the previous state. We refer to Melham [142] for a discussion of STE and extensions. Notably, Generalized STE [143] allows verification of ω-regular properties using infinite trajectories, corresponding to acceptance by Büchi automata and linear-time μ-calculus [144]. While manual refinement was originally needed, automatic refinement was proposed for both STE [145] and GSTE [146].

Delayed Nondeterminism. Noll & Schlich [4] verified machine-code programs by model-checking an abstract state space generated by a simulation-based approach. Each input bit was read as 'X' and split to '0' and '1' only when it was decided to in a subsequent step (e.g. if it was an argument of a branch instruction). This allowed e.g. splitting only one bit of a read 8-bit port if the other bits were masked out by a constant first, soundly proving ACTL properties.

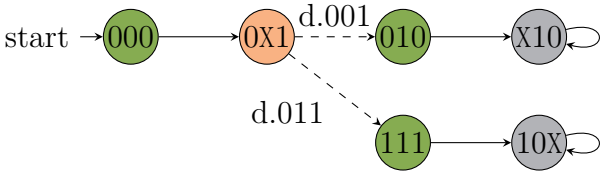
Example 5.1.9. Due to the approach restrictions, we will illustrate proving the property "in two steps from the initial state, the most significant bit corresponds to the least significant bit", i.e. $\mathbf{A}[\mathbf{X}[\mathbf{X}[msb \Leftrightarrow lsb]]]$. Simulating without splitting, we produce Figure 5.3a, unable to prove the property.



(a) Abstract state space corresponding to the system in Figure 5.1, computed by simulation with unconstrained inputs ('X')



(b) Trajectory evaluation: The verification is split into cases, ensuring the abstract input sequences together cover all possible concrete input sequences. Unlike Bryant, we use initial states for consistency with other approaches.



(c) Delayed nondeterminism augmented with must-transitions: the 'X' in state "0X1" is split to '0' and '1' before computing the successor

Figure 5.3: Simulation-based approaches proving $\mathbf{A}[\mathbf{X}[\mathbf{X}[msb \Leftrightarrow lsb]]]$. (a) can be considered PKS or KMTS, and (c) KTMS. (b) contains four trajectories. Specially in this figure, system states are drawn green if $msb \Leftrightarrow lsb$ holds, orange if it does not, and grey if it is unknown. Specification states are coloured according to the output.

To visualise Bryant's trajectory evaluation approach with explicitly considered inputs [123], we encode the specification as a finite-state machine with two bits containing an initially-zero saturating counter. The system output function is $msb \Leftrightarrow lsb$. The specification outputs '1' iff the counter is 10 and 'X' otherwise. To prove the property, we split verification into two cases based on the value of the first input, and obtain simulated trajectories of both machines in Figure 5.3b. The property is proven as the trajectories are long enough (at least 3 for the given property) and the specification output always covers the system output.

To better understand Delayed Nondeterminism, we augment with must-transitions where possible. Splitting "0X1" from Figure 5.3a, we obtain Figure 5.3c, where "010" is obtained as a direct successor of "001", and "111" as a direct successor of "011". We cannot augment during the split as 'X' might not generally correspond to a unique input, potentially e.g. being copied before splitting.

5.2 Input-based Abstraction Refinement

We propose a framework that eliminates the need for modal transitions in TVAR by combining the ideas from the discussed approaches: using TVAR, build the abstract state space by simulation and split **inputs** instead of states. We also allow for refining the **step function** to avoid simulating uninteresting details.

Example 5.2.1. We return to the original problem of proving $\mathbf{EF}[\mathbf{AG}[msb]]$. Using the simulation-based approach, we initially build the abstract state space as shown in Figure 5.3a. After that, we decide (using e.g. a heuristic, machine-learning or human guidance) that the input after "000" should be split. We regenerate the abstract state space as shown in Figure 5.4a. We are immediately able to prove $\mathbf{EF}[\mathbf{AG}[msb]]$ holds, meaning the system from Figure 5.1 contains a bug.

The part of the abstract state space in Figure 5.4a starting with "001" is unnecessarily large for proving the property, potentially causing exponential explosion problems. To prevent them, we also introduce a way to soundly and precisely regulate the outgoing states of transitions, allowing us to e.g. replace "001" by "XXX" as in Figure 5.4b when generating the abstract state space, *decaying* to less information. Only one refinement was necessary compared[4] to multiple in Example 5.1.8, with the final state space in Figure 5.4b smaller than in Figure 5.2f. However, this depends on the correct choice to decay "001" and not "011".

The generated state spaces are PKS, which allows us to use previous work on PKS, KMTS, GKMTS, and MixTS, as PKS are trivially convertible to all. This notably includes model-checking using standard formalisms [130, 132] and refinement guidance [118, 133, 134], with the caveat that we need to select an input instead of a state to refine. Unlike (G)STE and Delayed Nondeterminism which were limited to linear-time properties, the approach can be used for the full µ-calculus. Our framework also allows for precise control of the number of reachable abstract states and transitions: we can split inputs up to one by one, and decay any newly reachable states before refining the decay.

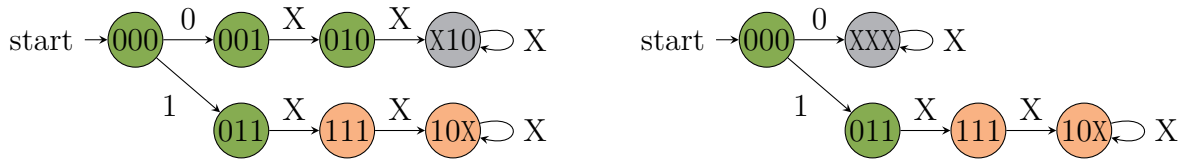
We will now give the framework formalism and simple requirements for its instances to be sound, monotone, and complete, proving that the requirements are sufficient in Section 5.3. Finally, we will evaluate an implementation of an instance of our framework in **machine-check** in Section 5.4.

5.2.1 Framework Formalism

We assume that the original Kripke Structure has only one initial state[5], i.e. the structure is $K = (S, \{s_0\}, R, L)$. We write the result of model-checking a property ϕ against K as $[\![\phi]\!](K)$, which returns 0 or 1. For a PKS $\hat{K}$, $[\![\phi]\!](\hat{K})$ returns 0, 1, or $\bot$.

[4]General verification performance depends drastically on abstraction, refinement, and implementation choices, further discussed in Section 5.4.

[5]This is merely a formal choice. For multiple initial states, a dummy initial state can be introduced before them and the verified property ϕ converted to $\mathbf{AX}[\phi]$.



(a) Refining the input after "000" while keeping all others 'X'.

(b) A smaller reachable state space using a *decayed* step function.

Figure 5.4: Input-based Abstraction Refinement.

We consider the original (concrete) system to be an automaton and will also use automata for abstracting the system, introducing the formalism of *generating automata* that can generate partial Kripke structures.

Definition 5.2.2. A *generating automaton* (GA) is a tuple $G = (S, s_0, I, q, f, L)$ with the elements

- S (the *set of automaton states*),
- $s_0 \in S$ (the *initial state*),
- I (the *set of all step inputs*),
- $q : S \to 2^I \setminus \{\emptyset\}$ (the *input qualification function*),
- $f : S \times I \to S$ (the *step function*),
- $L : S \times \mathbb{A} \to \{0, 1, \bot\}$ (the *labelling function*).

Definition 5.2.3. For a generating automaton (S, s_0, I, q, f, L), we define the PKS-generating function Γ as

$$\Gamma((S, s_0, I, q, f, L)) \stackrel{\text{def}}{=} (S, \{s_0\}, R, L) \tag{5.1}$$

$$\text{where } R = \{(s, f(s, i)) \mid s \in S, i \in q(s)\}. \tag{5.2}$$

We call a generating automaton $G = (S, s_0, I, q, f, L)$ *concrete* if the labelling function $L : S \times \mathbb{A} \to \{0, 1\}$ (disallowing the value $\bot$) and for all $s \in S$, $q(s) = I$. A concrete GA corresponds to a Moore machine with the output of each state mapping each atomic proposition from $\mathbb{A}$ to either 0 or 1.

Algorithm 5.1 describes our framework. Given a concrete GA, it abstracts it to an *abstract* generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$, successively refining the input qualification function $\hat{q}$ and step function $\hat{f}$ until the result of model-checking is non-$\bot$. $\hat{S}$ and $\hat{I}$ are related to S and I by a state concretization function[6] $\gamma : \hat{S} \to 2^S \setminus \{\emptyset\}$ and an input concretization function $\zeta : \hat{I} \to 2^I \setminus \{\emptyset\}$.

[6]We forbid abstract elements with no concretizations as they do not represent any concrete element. Practically speaking, this does not disqualify abstract domains with such elements, we just require such elements are not produced by $\hat{s}_0$, $\hat{q}$, or $\hat{f}$.

Algorithm 5.1: Input-based Three-valued Abstraction Refinement Framework

Require: a concrete generating automaton (S, s_0, I, q, f, L), a µ-calculus property ϕ
Ensure: return $[\![\phi]\!]((S, s_0, I, q, f, L))$ ▷ If requirements are fulfilled, see Corollary 5.2.8

$(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L}) \leftarrow \textsc{Abstract}(S, s_0, I, q, f, L)$
while $(r \leftarrow [\![\phi]\!](\Gamma((\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L}))) = \bot$ **do**
  $(\hat{q}, \hat{f}) \leftarrow \textsc{Refine}(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$
end while
return r

Unlike state-based TVAR, the set of abstract states $\hat{S}$ does not change during refinement. The number of states to be considered is limited by the codomain of $\hat{f}$, allowing structures such as Binary Decision Diagrams to be used. The abstract state space can be built quickly by forward simulation. For backward simulation, care must be taken to pair the states according to inputs.

Example 5.2.4. In **machine-check**, states and inputs are composed of bit-vector and bit-vector-array variables, formally represented by flattened $S = \{0,1\}^w$ and $I = \{0,1\}^y$ for finite state width w and finite input width y. A dummy s_0 precedes the actual initial system states, f is a function written in an imperative programming language, and L computes relational operations on state variables.

For the abstract GA, we use three-valued bit-vector abstraction [8, 4] with fast abstract operations [A.1], abstracting as

$$\gamma^{\text{bit}}(\hat{a}) = \{v \in \mathbb{B} \mid (v = 0 \Rightarrow \hat{a} \neq \text{`1'}) \wedge (v = 1 \Rightarrow \hat{a} \neq \text{`0'})\}, \tag{5.3a}$$

$$\hat{S} = \{\text{`0'}, \text{`1'}, \text{`X'}\}^w,\ \gamma(\hat{s}) = \{s \in S \mid \forall k \in [0, w-1]\ .\ s_k \in \gamma^{\text{bit}}(\hat{s}_k)\}, \tag{5.3b}$$

$$\hat{I} = \{\text{`0'}, \text{`1'}, \text{`X'}\}^y,\ \zeta(\hat{i}) = \{i \in I\ .\ \forall k \in [0, y-1]\ .\ i_k \in \gamma^{\text{bit}}(\hat{i}_k)\}. \tag{5.3c}$$

Again, $\hat{s}_0$ is a dummy state with $\gamma(\hat{s}_0) = \{s_0\}$. We rewrite the step function f into an abstract function $\hat{f}^{\text{basic}} : \hat{S} \times \hat{I} \to \hat{S}$. To formalise the manipulation in Figure 5.4, we use an *input precision function* $\hat{p}_{\hat{q}} : \hat{S} \to \{0,1\}^y$ and a *step precision function* $\hat{p}_{\hat{f}} : \hat{S} \to \{0,1\}^w$. The value 1 means we must keep the corresponding value of the bit in input or the result of f^{basic}, respectively, precise, without replacing it with 'X'.

For monotonicity, we ensure that we forbid clearing bits of $\hat{p}_{\hat{q}}$ and $\hat{p}_{\hat{f}}$ in subsequent refinements after setting them to 1, and that we always set at least one bit of $\hat{p}_{\hat{q}}$ or $\hat{p}_{\hat{f}}$ in each refinement, setting the bits until R in $\gamma(\hat{G})$ changes. We also define the monotone versions of the step and input precision functions as $\hat{m}_{\hat{q}} : \hat{S} \to \{0,1\}^y$ and $\hat{m}_{\hat{f}} : \hat{S} \to \{0,1\}^w$, respectively, defining their result in each bit k as

$$\hat{m}_{\hat{q}}(\hat{s})_k = 1 \Leftrightarrow (\exists \hat{s}^* \in \hat{S}\ .\ \gamma(\hat{s}^*) \subseteq \gamma(\hat{s}) \wedge \hat{p}_{\hat{q}}(\hat{s})_k = 1), \tag{5.4a}$$

$$\hat{m}_{\hat{f}}(\hat{s})_k = 1 \Leftrightarrow (\exists \hat{s}^* \in \hat{S}\ .\ \gamma(\hat{s}^*) \subseteq \gamma(\hat{s}) \wedge \hat{p}_{\hat{f}}(\hat{s})_k = 1). \tag{5.4b}$$

We will show why $\hat{m}_{\hat{q}}, \hat{m}_{\hat{f}}$ are important for monotonicity in Example 5.2.14. We define the result of $\hat{q}$ and $\hat{f}$ in each bit k by

$$(\hat{m}_{\hat{q}}(\hat{s})_k = 0 \Rightarrow \hat{q}(\hat{s})_k = \{\text{`X'}\}) \wedge (\hat{m}_{\hat{q}}(\hat{s})_k = 1 \Rightarrow \hat{q}(\hat{s})_k = \{\text{`0'}, \text{`1'}\}), \tag{5.5a}$$

$$(\hat{m}_{\hat{f}}(\hat{s})_k = 0 \Rightarrow \hat{f}(\hat{s}, \hat{i})_k = \text{`X'}) \wedge (\hat{m}_{\hat{f}}(\hat{s})_k = 1 \Rightarrow \hat{f}(\hat{s}, \hat{i})_k = \hat{f}^{\text{basic}}(\hat{s}, \hat{i})_k). \tag{5.5b}$$

The usage of $\hat{p}_{\hat{q}}$ and $\hat{p}_{\hat{f}}$ allows fairly precise control of the size of the reachable abstract state space. For example, if $\hat{p}_{\hat{q}}(\hat{s}) = (0)^y$, there is exactly one outgoing transition from $\hat{s}$ in $\hat{R}$ generated by $\Gamma(\hat{G})$. Each bit set to 1 increases that up to a factor of 2. Similarly, if $\hat{p}_{\hat{f}}(\hat{s}) = (0)^w$, there is exactly one outgoing transition to the "most-decayed" state $(\text{`X'})^w$.

5.2.2 Soundness, Monotonicity, and Completeness

In this subsection, we state the requirements sufficient to ensure soundness (the algorithm returns the correct result if it terminates), monotonicity (refinements never lose any information), and completeness (the algorithm always terminates). We defer the proofs to Section 5.3.

To intuitively describe the requirements, we formalise the concept of *coverage*. An abstract state $\hat{s}$ or input $\hat{i}$ *covers* a concrete $s \in S$ or $i \in I$ exactly when $s \in \gamma(\hat{s})$ or $i \in \zeta(\hat{i})$, respectively, and it *covers* another abstract state $\hat{s}^* \in \hat{S}$ or input $\hat{i}^* \in \hat{I}$ exactly when $\gamma(\hat{s}^*) \subseteq \gamma(\hat{s})$ or $\zeta(\hat{i}^*) \subseteq \zeta(\hat{i})$, respectively.

We want abstraction to preserve the truth value of μ-calculus properties in the following sense:

Definition 5.2.5. A partial Kripke structure $K^{\uparrow}$ is *sound* with respect to a partial Kripke structure $K^{\downarrow}$ if, for every property ϕ of μ-calculus over the set of atomic propositions $\mathbb{A}$, it holds that

$$[\![\phi]\!](K^{\uparrow}) \neq \bot \Rightarrow [\![\phi]\!](K^{\downarrow}) = [\![\phi]\!](K^{\uparrow}). \tag{5.6}$$

Intuitively, $K^{\uparrow}$ can contain less information than $K^{\downarrow}$, turning some non-$\bot$ proposition results to $\bot$. No other differences are possible.

To ensure the soundness of Algorithm 5.1, we use the following requirements. Soundness is ensured with any refinement heuristic as long as they are met.

Definition 5.2.6. A generating automaton $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ is a *soundness-guaranteeing* (γ, ζ)*-abstraction* of a concrete generating automaton $G = (S, s_0, I, q, f, L)$ iff

$$\gamma(\hat{s}_0) = \{s_0\}, \tag{5.7a}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall s \in \gamma(\hat{s}) \,.\, \forall a \in \mathbb{A} \,.\, (\hat{L}(\hat{s}, a) \neq \bot \Rightarrow \hat{L}(\hat{s}, a) = L(s, a)), \tag{5.7b}$$

$$\forall (\hat{s}, i) \in \hat{S} \times I \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, i \in \zeta(\hat{i}), \tag{5.7c}$$

$$\forall (\hat{s}, \hat{i}) \in \hat{S} \times \hat{I} \,.\, \forall (s, i) \in \gamma(\hat{s}) \times \zeta(\hat{i}) \,.\, f(s, i) \in \gamma(\hat{f}(\hat{s}, \hat{i})). \tag{5.7d}$$

Informally, the four requirements express the following:

(a) **Initial state concretization.** The abstract initial state has exactly the concrete initial state in its concretization.

(b) **Labelling soundness.** Each abstract state labelling must either correspond to the labelling of all concrete states it covers or be unknown.

(c) **Full input coverage.** In every abstract state, each concrete input must be covered by some qualified abstract input.

(d) **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.

The requirements ensure the soundness of the used abstractions as follows.

Theorem 5.2.7 (Soundness)**.** For every generating automaton $\hat{G}$ and concrete generating automaton G, state concretization function γ, and input concretization function ζ such that $\hat{G}$ is a soundness-guaranteeing (γ, ζ)-abstraction of G, the partial Kripke structure $\Gamma(\hat{G})$ is sound with respect to $\Gamma(G)$.

Corollary 5.2.8. Assume that the functions Abstract and Refine ensure that the generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ in Algorithm 5.1 is always a soundness-guaranteeing (γ, ζ)-abstraction of (S, s_0, I, q, f, L). Then, if the algorithm terminates, its result is correct.

Example 5.2.9. Continuing from Example 5.2.4, (5.7a) is fulfilled trivially. (5.7c) is fulfilled due to (5.3c) and (5.5a). From (5.5b), it is apparent that

$$\forall(\hat{s}, \hat{i}) \in (\hat{S}, \hat{I}) \,.\, \gamma(\hat{f}^{\text{basic}}(\hat{s}, \hat{i})) \subseteq \gamma(\hat{f}(\hat{s}, \hat{i})), \tag{5.8}$$

i.e. results of $\hat{f}$ cover results of $\hat{f}^{\text{basic}}$ that abstracts f. We carefully implemented the translation of f to $\hat{f}^{\text{basic}}$ and $\hat{L}$ so that (5.7b) and (5.7d) hold.

Next, we turn to monotonicity, which ensures no algorithm loop iteration loses information. We give the requirements for the refinement to guarantee it.

Definition 5.2.10. A generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ is monotone wrt. (γ, ζ)-coverage iff

$$\begin{aligned} &\forall(\hat{s}, \hat{s}', a) \in \hat{S} \times \hat{S} \times \mathbb{A} \,. \\ &\quad ((\gamma(\hat{s}') \subseteq \gamma(\hat{s}) \wedge \hat{L}(\hat{s}, a) \neq \bot) \Rightarrow \hat{L}(\hat{s}, a) = L(\hat{s}', a)), \end{aligned} \tag{5.9a}$$

$$\begin{aligned} &\forall(\hat{s}, \hat{s}', \hat{i}, \hat{i}') \in \hat{S} \times \hat{S} \times \hat{I} \times \hat{I} \,. \\ &\quad ((\gamma(\hat{s}') \times \zeta(\hat{i}') \subseteq \gamma(\hat{s}) \times \zeta(\hat{i})) \Rightarrow \gamma(\hat{f}(\hat{s}', \hat{i}')) \subseteq \gamma(\hat{f}(\hat{s}, \hat{i})). \end{aligned} \tag{5.9b}$$

Informally, we require that each abstract state has at least as much labelling information as each abstract state it covers, and the abstract step function result covers each result produced using covered arguments.

Definition 5.2.11. The generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}', \hat{f}', \hat{L})$ is a (γ, ζ)-monotone refinement of the generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ iff it is monotone wrt. (γ, ζ)-coverage and

$$\forall \hat{s} \in \hat{S} \,.\, \forall \hat{i}' \in \hat{q}'(\hat{s}) \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \zeta(\hat{i}') \subseteq \zeta(\hat{i}), \tag{5.10a}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \exists \hat{i}' \in \hat{q}'(\hat{s}) \,.\, \zeta(\hat{i}') \subseteq \zeta(\hat{i}), \tag{5.10b}$$

$$\forall (\hat{s}, \hat{i}) \in \hat{S} \times \hat{I} \,.\, \gamma(\hat{f}'(\hat{s}, \hat{i})) \subseteq \gamma(\hat{f}(\hat{s}, \hat{i})). \tag{5.10c}$$

Informally, in addition to the monotonicity wrt. coverage, we also require:

(a) **New qualified inputs are not spurious.** Each new qualified input is covered by at least one old qualified input.

(b) **Old qualified inputs are not lost.** Each old qualified input covers at least one new qualified input.

(c) **New step function covered by old.** The result of the new step function is always covered by the result of the old step function.

The need for both quantifier combinations in the first two requirements in Equation 5.10 may be surprising. Their violations correspond to transition addition and removal, respectively, which could make a previously non-$\bot$ property $\bot$.

Theorem 5.2.12 (Monotonicity)**.** If the generating automaton $\hat{G}'$ is a (γ, ζ)-monotone refinement of the generating automaton $\hat{G}$, then for every µ-calculus property ψ for which $[\![\psi]\!](\Gamma(\hat{G})) \neq \bot$, it also holds $[\![\psi]\!](\Gamma(\hat{G}')) \neq \bot$.

Corollary 5.2.13. If the update in the loop of Algorithm 5.1 performs a (γ, ζ)-monotone refinement of the generating automaton $(\hat{S}, \hat{s}_0, \hat{q}, \hat{f}, \hat{L})$ and for a µ-calculus property ψ, it held that $[\![\psi]\!](\Gamma((\hat{S}, \hat{s}_0, \hat{q}, \hat{f}, \hat{L}))) \neq \bot$ before the loop iteration, then this is also the case after the iteration.

Example 5.2.14. Continuing from Example 5.2.9, to ensure monotonicity, we have to ensure that (5.9) holds for each generating automaton used, and the (5.10) holds for every refinement. We implemented $\hat{L}$ and $\hat{f}^{\text{basic}}$ so that they would fulfil (5.9a) and (5.9b). Still, (5.9b) poses a major danger: if we used $\hat{p}_{\hat{q}}$, $\hat{p}_{\hat{f}}$ instead of $\hat{m}_{\hat{q}}$, $\hat{m}_{\hat{f}}$ in (5.5), it would be possible to e.g. set them to all-zeros in a state $\hat{s}_a$ and to all-ones in $\hat{s}_b$ that covers it, possibly violating (5.9b) for $\hat{f}$. Combining (5.4) and (5.5) ensures (5.9b) holds, with no violation possible: $\hat{m}_{\hat{q}}$, $\hat{m}_{\hat{f}}$ would be all-ones for both $\hat{s}_a$ and $\hat{s}_b$ in this situation.

As we have forbidden clearing bits in $\hat{p}_q$ after setting them, (5.10a) and (5.10b) follow from (5.4a) and (5.5a). Analogously, as we have forbidden clearing bits in $\hat{p}_f$, (5.10c) follows from (5.4b) and (5.5b).

Definition 5.2.15. The generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}', \hat{f}', \hat{L})$ is a *strictly* (γ, ζ)-monotone refinement of the generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ if it is a monotone refinement and either

$$\exists \hat{s} \in \hat{S} \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \forall \hat{i}' \in \hat{q}'(\hat{s}) \,.\, \exists i \in \zeta(\hat{i}) \,.\, i \notin \zeta(\hat{i}'), \tag{5.11}$$

$$\text{or } \exists (\hat{s}, \hat{i}) \in \hat{S} \times \hat{I} \,.\, \exists s \in \gamma(\hat{f}(\hat{s}, \hat{i})) \,.\, s \notin \gamma(\hat{f}'(\hat{s}, \hat{i})). \tag{5.12}$$

Example 5.2.16. Continuing from Example 5.2.14, we know each refinement is monotone. In Example 5.3, we mentioned that we set at least one bit in $\hat{p}_{\hat{q}}$ or $\hat{p}_{\hat{f}}$ and we set them until R in $\Gamma(\hat{G})$ changes (just setting bits in $\hat{p}_{\hat{f}}$ does not necessarily mean $\hat{f}$ changes). Consequently, $\hat{q}$ or $\hat{f}$ change as per (5.11) or (5.12), and the refinement is strictly monotone.

Theorem 5.2.17 (Completeness)**.** If $\hat{S}$ and $\hat{I}$ are finite, there is no infinite sequence of generating automata that are soundness-guaranteeing (γ, ζ)-abstractions of some G such that all subsequent pairs in the sequence are strictly (γ, ζ)-monotone refinements.

Corollary 5.2.18. If $\hat{S}, \hat{I}$ are finite, the functions Abstract and Refine in Algorithm 5.1 ensure that $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ always is a *soundness-guaranteeing* (γ, ζ)*-abstraction* of (S, s_0, I, q, f, L), and calls of Refine perform strict (γ, ζ)-monotone refinements, then the algorithm returns the correct result in finite time.

Fulfilling the requirements of Corollary 5.2.18 is not trivial. We must exclude the situation when no strict (γ, ζ)-monotone refinement is possible any more while a non-$\bot$ result has not yet been reached. We propose Lemma 5.2.20 for easier reasoning.

Definition 5.2.19. A generating automaton $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ is (γ, ζ)-terminating wrt. a concrete generating automaton $G = (S, s_0, I, q, f, L)$ if it is a (γ, ζ)-abstraction of G monotone wrt. (γ, ζ)-coverage and

$$\forall (s, \hat{s}) \in S \times \hat{S} \,.\, (\gamma(\hat{s}) = \{s\} \Rightarrow \hat{L}(\hat{s}) = L(s)), \tag{5.13a}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \exists i \in I \,.\, \zeta(\hat{i}) = \{i\}, \tag{5.13b}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall i \in I \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \zeta(\hat{i}) = \{i\}, \tag{5.13c}$$

$$\forall (\hat{s}, \hat{i}, s, i) \in \hat{S} \times \hat{I} \times S \times I \,.\, ((\gamma(\hat{s}), \zeta(\hat{i})) = (\{s\}, \{i\}) \Rightarrow \gamma(\hat{f}(\hat{s}, \hat{i})) = \{f(s, i)\}). \tag{5.13d}$$

Lemma 5.2.20 (Strict Refinement)**.** If $\hat{G}$ is (γ, ζ)-terminating wrt. G, then for every µ-calculus property ψ, $[\![\psi]\!](\Gamma(\hat{G})) = [\![\psi]\!](\Gamma(G))$. Furthermore, if $\hat{G}$ is a soundness-guaranteeing (γ, ζ)-abstraction of G for which some (γ, ζ)-monotone refinement is (γ, ζ)-terminating wrt. G, then $\hat{G}$ itself is (γ, ζ)-terminating wrt. G or the refinement is strict.

Corollary 5.2.21. If every generating automaton constructed in Algorithm 5.1 fulfils the conditions of Lemma 5.2.20, calls to Refine can always perform a *strict* (γ, ζ)-monotone refinement to a soundness-guaranteeing (γ, ζ)-abstraction.

Example 5.2.22. Continuing from Example 5.2.14, we implemented $\hat{L}, \hat{f}^{\text{basic}}$ so that they fulfil (5.13a) and (5.13d). Since (5.13b) and (5.13c) hold due to (5.5a), $\hat{G}^*$ obtained by $\hat{p}_{\hat{q}} = (1)^y, \hat{p}_{\hat{f}} = (1)^w$ is (γ, ζ)-terminating. Inspecting (5.10), $\hat{G}^*$ is monotone wrt. all other applicable GA, and Corollary 5.2.21 holds. As soundness was already ensured in Example 5.2.9 and $\hat{S}$, $\hat{I}$ are finite by definition, Corollary 5.2.18 holds.

5.3 Proofs of Soundness, Monotonicity, and Completeness

In this section, we will prove Theorems 5.2.7, 5.2.12, and 5.2.17. Since directly proving on µ-calculus is cumbersome, we will typically prove the theorems by showing their requirements imply that some PKS is sound wrt. another PKS as per Definition 5.2.6 using a well-known property of modal simulation [41, p. 408-410], which we simplified from KMTS to PKS by setting $R = R_{may} = R_{must}$.

Definition 5.3.1. Let $K^\downarrow = (S^\downarrow, S_0^\downarrow, R^\downarrow, L^\downarrow)$ and $K^\uparrow = (S^\uparrow, S_0^\uparrow, R^\uparrow, L^\uparrow)$ be PKS over the set of atomic propositions $\mathbb{A}$. Then $H \subseteq S^\downarrow \times S^\uparrow$ is a modal simulation from $K^\downarrow$ to $K^\uparrow$ if and only if all of the following hold,

$$\forall (s^\downarrow, s^\uparrow) \in H \,.\, \forall a \in \mathbb{A} \,.\, (L^\uparrow(s^\uparrow, a) \neq \bot \Rightarrow L^\downarrow(s^\downarrow, a) = L^\uparrow(s^\uparrow, a)), \tag{5.14a}$$

$$\forall (s^\downarrow, s^\uparrow) \in H \,.\, \forall s'^\downarrow \in S^\downarrow \,.\, (R^\downarrow(s^\downarrow, s'^\downarrow) \Rightarrow \exists s'^\uparrow \in S^\uparrow \,.\, (R^\uparrow(s^\uparrow, s'^\uparrow) \wedge H(s'^\downarrow, s'^\uparrow))), \tag{5.14b}$$

$$\forall (s^\downarrow, s^\uparrow) \in H \,.\, \forall s'^\uparrow \in S^\uparrow \,.\, (R^\uparrow(s^\uparrow, s'^\uparrow) \Rightarrow \exists s'^\downarrow \in S^\downarrow \,.\, (R^\downarrow(s^\downarrow, s'^\downarrow) \wedge H(s'^\downarrow, s'^\uparrow))). \tag{5.14c}$$

Informally, H relates the states of $K^\downarrow$ and $K^\uparrow$ so that the states in the fine $K^\downarrow$ preserve all known labellings of their related states from the coarse $K^\uparrow$, 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 5.3.2. A PKS $K^\downarrow$ with initial state $s_0^\downarrow$ is modal-simulated by a PKS $K^\uparrow$ with initial state $s_0^\uparrow$, denoted $K^\downarrow \preceq K^\uparrow$, if there is a modal simulation H from $K^\downarrow$ to $K^\uparrow$ and furthermore,

$$\forall s_0^\downarrow \in S_0^\downarrow \,.\, \exists s_0^\uparrow \in S_0^\uparrow \,.\, H(s_0^\downarrow, s_0^\uparrow). \tag{5.15}$$

Property 5.3.3. $K^\uparrow$ is sound wrt. $K^\downarrow$ if $K^\downarrow \preceq K^\uparrow$ [41, p. 410] (original argument by Huth et al. [147, p. 161]).

To simplify the proofs further, we first prove a criterion for generating automata that ensures their corresponding Kripke structures are sound.

Lemma 5.3.4. The PKS $\Gamma(S^\uparrow, s_0^\uparrow, q^\uparrow, f^\uparrow, L^\uparrow)$ is sound wrt. $\Gamma(S^\downarrow, s_0^\downarrow, q^\downarrow, f^\downarrow, L^\downarrow)$ if there exists a relation $H \subseteq S^\downarrow \times S^\uparrow$ where

$$(s_0^\downarrow, s_0^\uparrow) \in H, \tag{5.16a}$$
$$\forall (s^\downarrow, s^\uparrow) \in H \,.\, \forall a \in \mathbb{A} \,.\, (L^\uparrow(s^\uparrow, a) \neq \bot \Rightarrow L^\downarrow(s^\downarrow, a) = L^\uparrow(s^\uparrow, a)), \tag{5.16b}$$
$$\forall (s^\downarrow, s^\uparrow) \in H \,.\, \forall i^\downarrow \in q^\downarrow(s^\downarrow) \,.\, \exists i^\uparrow \in q^\uparrow(s^\uparrow) \,.\, H(f^\downarrow(s^\downarrow, i^\downarrow), f^\uparrow(s^\uparrow, i^\uparrow)), \tag{5.16c}$$
$$\forall (s^\downarrow, s^\uparrow) \in H \,.\, \forall i^\uparrow \in q^\uparrow(s^\uparrow) \,.\, \exists i^\downarrow \in q^\downarrow(s^\downarrow) \,.\, H(f^\downarrow(s^\downarrow, i^\downarrow), f^\uparrow(s^\uparrow, i^\uparrow)). \tag{5.16d}$$

Proof. Let $K^\uparrow = \Gamma(S^\uparrow, s_0^\uparrow, q^\uparrow, f^\uparrow, L^\uparrow)$ and $K^\downarrow = \Gamma(S^\downarrow, s_0^\downarrow, q^\downarrow, f^\downarrow, L^\downarrow)$. We assume a relation $H \subseteq S^\downarrow \times S^\uparrow$ satisfying (5.16a) to (5.16d) and use it to prove $K^\downarrow \preceq K^\uparrow$ which implies soundness due to Property 5.3.3. Due to (5.16a), (5.15) holds, and it remains to prove (5.14). (5.14a) directly follows from (5.16b). To prove (5.14b) and (5.14c), we respectively expand $R^\downarrow$ and $R^\uparrow$ according to Definition 5.2.3 to

$$\begin{aligned}&\forall (s^\downarrow, s^\uparrow) \in H \,.\, \forall s'^\downarrow \in S^\downarrow \,. \\ &\quad ((\exists i^\downarrow \in q^\downarrow(s^\downarrow) \,.\, s'^\downarrow = f^\downarrow(s^\downarrow, i^\downarrow)) \Rightarrow \exists s'^\uparrow \in S^\uparrow \,.\, (R^\uparrow(s^\uparrow, s'^\uparrow) \wedge H(s'^\downarrow, s'^\uparrow))),\end{aligned} \tag{5.17a}$$
$$\begin{aligned}&\forall (s^\downarrow, s^\uparrow) \in H \,.\, \forall s'^\uparrow \in S^\uparrow \,. \\ &\quad ((\exists i^\uparrow \in q^\uparrow(s^\uparrow) \,.\, s'^\uparrow = f^\uparrow(s^\uparrow, i^\uparrow)) \Rightarrow \exists s'^\downarrow \in S^\downarrow \,.\, (R^\downarrow(s^\downarrow, s'^\downarrow) \wedge H(s'^\downarrow, s'^\uparrow))).\end{aligned} \tag{5.17b}$$

We move the $i^\downarrow, i^\uparrow$ quantifiers out of the implication, negating them due to moving out of antecedent. $s'^\downarrow$ and $s'^\uparrow$ must be equal to $f^\downarrow(s^\downarrow, i^\downarrow)$ and $f^\uparrow(s^\uparrow, i^\uparrow)$, respectively, so we replace them and eliminate the quantifiers, obtaining

$$\forall (s^\downarrow, s^\uparrow) \in H. \,\forall i^\downarrow \in q^\downarrow(s^\downarrow) \,.\, \exists s'^\uparrow \in S^\uparrow \,.\, (R^\uparrow(s^\uparrow, s'^\uparrow) \wedge H(f^\downarrow(s^\downarrow, i^\downarrow), s'^\uparrow)), \tag{5.18a}$$
$$\forall (s^\downarrow, s^\uparrow) \in H. \,\forall i^\uparrow \in q^\uparrow(s^\uparrow) \,.\, \exists s'^\downarrow \in S^\downarrow \,.\, (R^\downarrow(s^\downarrow, s'^\downarrow) \wedge H(s'^\downarrow, f^\uparrow(s^\uparrow, i^\uparrow))). \tag{5.18b}$$

We then respectively insert the definition of $R^\uparrow$ and $R^\downarrow$, pull the $i^\uparrow, i^\downarrow$ quantifiers outside, and eliminate the $s'^\uparrow, s'^\downarrow$ variables, obtaining

$$\forall (s^\downarrow, s^\uparrow) \in H \,.\, \forall i^\downarrow \in q^\downarrow(s^\downarrow) \,.\, \exists i^\uparrow \in q^\uparrow(s^\uparrow) \,.\, H(f^\downarrow(s^\downarrow, i^\downarrow), f^\uparrow(s^\uparrow, i^\uparrow)), \tag{5.19a}$$
$$\forall (s^\downarrow, s^\uparrow) \in H \,.\, \forall i^\uparrow \in q^\uparrow(s^\uparrow) \,.\, \exists i^\downarrow \in q^\downarrow(s^\downarrow) \,.\, H(f^\downarrow(s^\downarrow, i^\downarrow), f^\uparrow(s^\uparrow, i^\uparrow)), \tag{5.19b}$$

which correspond to the assumed (5.16c) and (5.16d). □

5.3.1 Proof of Soundness

Proof (Theorem 5.2.7). Assume generating automata $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ and $G = (S, s_0, I, q, f, L)$, state concretization function γ, and input concretization function ζ such that $\hat{G}$ is a soundness-guaranteeing (γ, ζ)-abstraction of G. Our goal is to prove that $\Gamma(\hat{G})$ is sound wrt. $\Gamma(G)$. For this, we prove that

$$H = \{(s, \hat{s}) \in S \times \hat{S} \mid s \in \gamma(\hat{s})\} \tag{5.20}$$

satisfies the conditions of Lemma 5.3.4 which implies that $\Gamma(\hat{G})$ is sound wrt. $\Gamma(G)$. Conditions (5.16a) and (5.16b) hold due to (5.7a) and (5.7b), respectively. We use the definition of H and the fact that G is concrete to rewrite (5.16c) and (5.16d) to

$$\forall \hat{s} \in \hat{S} \,.\, \forall s \in \gamma(\hat{s}) \,.\, \forall i \in I \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, f(s,i) \in \gamma(\hat{f}(\hat{s},\hat{i})), \tag{5.21a}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall s \in \gamma(\hat{s}) \,.\, \forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \exists i \in I \,.\, f(s,i) \in \gamma(\hat{f}(\hat{s},\hat{i})). \tag{5.21b}$$

Informally, from each concrete state covered by an abstract state, (5.21a) requires that each concrete step result is covered by some abstract step result, and (5.21b) requires that each abstract step result covers some concrete step result.

To prove (5.21a), we assume $\hat{s} \in \hat{S}, s \in \gamma(\hat{s}), i \in I$ to be arbitrary but fixed. From (5.7c), we know that we can choose some $\hat{i} \in \hat{q}(\hat{s})$ for which $i \in \zeta(\hat{i})$.

To prove (5.21b), we assume $\hat{s} \in \hat{S}, s \in \gamma(\hat{s}), \hat{i} \in \hat{q}(\hat{s})$ to be arbitrary but fixed. As $\zeta : \hat{I} \to 2^I \setminus \{\emptyset\}$, we can always choose some $i \in \zeta(\hat{i})$.

In both situations, our assumptions include $s \in S, \hat{s} \in \gamma(\hat{s}), \hat{i} \in \hat{q}(\hat{s}), i \in \zeta(\hat{i})$, and it remains to prove $f(s,i) \in \gamma(\hat{f}(\hat{s},\hat{i}))$. This follows from (5.7d). □

5.3.2 Proof of Monotonicity

Proof (Theorem 5.2.12). We assume that $\hat{G}' = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}', \hat{f}', \hat{L})$ is a (γ,ζ)-monotone refinement of the generating automaton $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$. We prove that for every µ-calculus property ψ for which $[\![\psi]\!](\Gamma(\hat{G})) \neq \bot$, also $[\![\psi]\!](\Gamma(\hat{G}')) \neq \bot$. For this, it suffices to prove that $\Gamma(\hat{G})$ is sound wrt. $\Gamma(\hat{G}')$. Using

$$H \stackrel{\text{def}}{=} \{(\hat{s}', \hat{s}) \in \hat{S} \times \hat{S} \mid \gamma(\hat{s}') \subseteq \gamma(\hat{s})\}, \tag{5.22}$$

we will prove that (5.16) holds for $G^{\downarrow} \stackrel{\text{def}}{=} \hat{G}'$, $G^{\uparrow} \stackrel{\text{def}}{=} \hat{G}$. Formula (5.16a) holds trivially, and (5.16b) holds due to (5.9a). Assuming that $\hat{s}' \in \hat{S}, \hat{s} \in \hat{S}$ are arbitrary but fixed and $\gamma(\hat{s}') \subseteq \gamma(\hat{s})$ holds, we rewrite (5.16c) and (5.16d) to

$$\forall \hat{i}' \in \hat{q}'(\hat{s}') \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \gamma(\hat{f}'(\hat{s}',\hat{i}')) \subseteq \gamma(\hat{f}(\hat{s},\hat{i})), \tag{5.23a}$$

$$\forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \exists \hat{i}' \in \hat{q}'(\hat{s}') \,.\, \gamma(\hat{f}'(\hat{s}',\hat{i}')) \subseteq \gamma(\hat{f}(\hat{s},\hat{i})). \tag{5.23b}$$

To prove (5.23a), we assume $\hat{i}' \in \hat{q}'(\hat{s}')$ arbitrary but fixed and choose $\hat{i} \in \hat{q}(\hat{s})$ for which $\zeta(\hat{i}') \subseteq \zeta(\hat{i})$, which exists due to (5.10a).

To prove (5.23a), we assume $\hat{i} \in \hat{q}(\hat{s})$ arbitrary but fixed and choose $\hat{i}' \in \hat{q}'(\hat{s}')$ for which $\zeta(\hat{i}') \subseteq \zeta(\hat{i})$, which exists due to (5.10b).

In both situations, it remains to prove $\gamma(\hat{f}'(\hat{s}',\hat{i}')) \subseteq \gamma(\hat{f}(\hat{s},\hat{i}))$. From the assumed (5.10c), we strengthen this to $\gamma(\hat{f}(\hat{s}',\hat{i}')) \subseteq \gamma(\hat{f}(\hat{s},\hat{i}))$. This follows from (5.9b), given our other assumptions about $\hat{s}, \hat{s}', \hat{i}, \hat{i}'$. □

5.3.3 Proof of Completeness

Proof (Theorem 5.2.17). We assume a sequence satisfying the preconditions of the theorem and show that it cannot be infinite. For each $\hat{G}$ in the sequence, we define a function $M_{\hat{G}}(\hat{s}) = (Q_{\hat{G}}(\hat{s}), F_{\hat{G}}(\hat{s}))$ where

$$Q_{\hat{G}}(\hat{s}) = \{\zeta(\hat{i}) \mid \hat{i} \in \hat{q}_{\hat{G}}(\hat{s})\},\ F_{\hat{G}}(\hat{s}) = \{(\hat{i} \in \hat{I}, \gamma(\hat{f}_{\hat{G}}(\hat{s}, \hat{i})))\}, \tag{5.24}$$

and will prove it cannot be equal for two different elements of the sequence named $\hat{G}$ and $\hat{G}^*$. We name the element after $\hat{G}$ as $\hat{G}'$, it is a strict (γ, ζ)-monotone refinement of $\hat{G}$. $\hat{G}^*$ is a (γ, ζ)-monotone refinement of both $\hat{G}$ and $\hat{G}'$ as Definition 5.2.11 is clearly transitive and reflexive.

Case 1. The strict refinement is due to (5.11). For some $\hat{s} \in \hat{S}$, there is an $\hat{i} \in \hat{q}(\hat{s})$ not covered by any $\hat{i}' \in \hat{q}(\hat{s})$, so $\zeta(\hat{i}) \in Q_{\hat{G}}(\hat{s})$ but $\zeta(\hat{i}) \notin Q_{\hat{G}'}(\hat{s})$. Due to (5.10a) from $\hat{G}'$ to $\hat{G}^*$, $\zeta(\hat{i}) \notin Q_{\hat{G}^*}(\hat{s})$, so $M_{\hat{G}} \neq M_{\hat{G}^*}$.

Case 2. The strict refinement is due to (5.12). There is some pair $(\hat{s}, \hat{i}) \in \hat{S} \times \hat{I}$ where for some $s \in \gamma(\hat{f}(\hat{s}, \hat{i}))$, $s \notin \gamma(\hat{f}'(\hat{s}, \hat{i}))$. As such, $s \in F_{\hat{G}}(\hat{s})(\hat{i})$ but $s \notin F_{\hat{G}'}(\hat{s})(\hat{i})$. Due to (5.10c) from $\hat{G}'$ to $\hat{G}^*$, $s \notin F_{\hat{G}^*}(\hat{s})(\hat{i})$, so $M_{\hat{G}} \neq M_{\hat{G}^*}$.

For each $\hat{s} \in \hat{S}$, $\hat{Q}(\hat{s})$ has less than $2^{|\hat{I}|}$ valuations. For each pair $(\hat{s}, \hat{i}) \in \hat{S} \times \hat{I}$, $F(\hat{s})(\hat{i})$ has less than $2^{|\hat{S}|}$ valuations. As such, there are less than $|S|(2^{|\hat{I}|} + |I|2^{|\hat{S}|})$ valuations of $\hat{M}$. Since we assume $\hat{S}, \hat{I}$ finite, this completes the proof. □

Proof (Lemma 5.2.20).

First part. We will first prove the claim that if $\hat{G}$ is (γ, ζ)-terminating wrt. G, then for every µ-calculus property ψ, $[\![\psi]\!](\Gamma(\hat{G})) = [\![\psi]\!](\Gamma(G))$. Since G is a concrete generating automaton, $[\![\psi]\!](\Gamma(G)) \neq \bot$, and it suffices to prove $\Gamma(G)$ is sound wrt. $\Gamma(\hat{G})$. Using Lemma 5.3.4, we define H as

$$H \stackrel{\text{def}}{=} \{(\hat{s}, s) \mid \gamma(\hat{s}) = \{s\}\}. \tag{5.25}$$

(5.16a) and (5.16b) hold due to (5.7a) and (5.13a), respectively. We rewrite (5.16c) and (5.16d) as

$$\forall (\hat{s}, s) \in \hat{S} \times S\,.\,(\gamma(\hat{s}) = \{s\} \Rightarrow \forall \hat{i} \in \hat{q}(\hat{s})\,.\,\exists i \in I\,.\,\gamma(\hat{f}(\hat{s}, \hat{i})) = \{f(s, i)\}), \tag{5.26a}$$

$$\forall (\hat{s}, s) \in \hat{S} \times S\,.\,(\gamma(\hat{s}) = \{s\} \Rightarrow \forall i \in I\,.\,\exists \hat{i} \in \hat{q}(\hat{s})\,.\,\gamma(\hat{f}(\hat{s}, \hat{i})) = \{f(s, i)\}). \tag{5.26b}$$

We assume $(\hat{s}, s) \in \hat{S} \times S$ arbitrary but fixed and $\gamma(\hat{s}) = \{s\}$ to hold.

To prove (5.26a), we assume $\hat{i} \in \hat{q}(\hat{s})$ arbitrary but fixed and choose $i \in I$ for which $\zeta(\hat{i}) = \{i\}$, which exists due to (5.13b).

To prove (5.26b), we assume $i \in I$ arbitrary but fixed and choose $\hat{i} \in \hat{q}(\hat{s})$ for which $\zeta(\hat{i}) = \{i\}$, which exists due to (5.13c).

Second part. We turn to the claim that if $\hat{G}$ is a soundness-guaranteeing (γ, ζ)-abstraction of G for which some (γ, ζ)-monotone refinement $\hat{G}^*$ is (γ, ζ)-terminating wrt. G, then either $\hat{G}$ itself is (γ, ζ)-terminating wrt. G or the refinement is strict. We will prove

this by assuming $\hat{G}$ is not (γ, ζ)-terminating and proving that $\hat{G}^*$ is a strict (γ, ζ)-monotone refinement of $\hat{G}$.

As per Definition 5.2.15, we are to prove either (5.11) or (5.12) holds for the refinement from $\hat{G}$ to $\hat{G}^*$. Since $\hat{G}$ is not guaranteed-terminating, at least one of (5.13a), (5.13b), (5.13c), (5.13d) does not hold.

Case (a). (5.13a) does not hold for $\hat{G}$. However, $\hat{S}$, $\hat{I}$, and $\hat{L}$ are the same for $\hat{G}^*$, and we already assumed (5.13a) for $\hat{G}^*$. This is a contradiction and so this case cannot occur.

Case (b). (5.13b) does not hold for $\hat{G}$, but we have assumed it holds for $\hat{G}^*$, i.e.

$$\exists \hat{s} \in \hat{S} \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \forall i \in I \,.\, \zeta(\hat{i}) \neq \{i\}. \tag{5.27a}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall \hat{i^*} \in \hat{q}^*(\hat{s}) \,.\, \exists i^* \in I \,.\, \zeta(\hat{i^*}) = \{i^*\}. \tag{5.27b}$$

We will prove that (5.11) holds for the refinement from $\hat{G}$ to $\hat{G}^*$, i.e.

$$\exists \hat{s} \in \hat{S} \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \forall \hat{i'} \in \hat{q}^*(\hat{s}) \,.\, \exists i \in \zeta(\hat{i}) \,.\, i \notin \zeta(\hat{i'}). \tag{5.28}$$

We fix $\hat{s}$, $\hat{i}$ from (5.27a) and choose them in (5.28). For all $\hat{i'} \in \hat{q}^*(\hat{s})$, clearly $|\gamma(\hat{i'})| = 1$ due to (5.27b). However, due to (5.27a) and ζ not returning $\emptyset$, $|\gamma(\hat{i})| > 1$. As such, for any $\hat{i'} \in \hat{q}^*(\hat{s})$, there exists $i \in \zeta(\hat{i})$ such that $i \notin \zeta(\hat{i'})$.

Case (c). (5.13c) does not hold for $\hat{G}$, i.e.

$$\exists \hat{s} \in \hat{S} \,.\, \exists i \in I \,.\, \forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \zeta(\hat{i}) \neq \{i\}. \tag{5.29}$$

Since $\hat{G}$ is assumed to be a soundness-guaranteeing (γ, ζ)-abstraction of G, it must fulfil the full input coverage requirement from (5.7c), i.e.

$$\forall (\hat{s}, i) \in \hat{S} \times I \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, i \in \zeta(\hat{i}). \tag{5.30}$$

Fixing $\hat{s} \in \hat{S}, i \in I$ from (5.29) and choosing them in (5.30), we combine both formulas to obtain

$$(\forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \zeta(\hat{i}) \neq \{i\}) \wedge (\exists \hat{i} \in \hat{q}(\hat{s}) \,.\, i \in \zeta(\hat{i})). \tag{5.31}$$

We can weaken this to

$$\exists \hat{i} \in \hat{q}(\hat{s}) \,.\, (i \in \zeta(\hat{i}) \wedge \zeta(\hat{i}) \neq \{i\}). \tag{5.32}$$

This implies $\exists \hat{i} \in \hat{q}(\hat{s}) \,.\, |\gamma(\hat{i})| > 1$, where i no longer appears. We reintroduce an existential quantifier for $\hat{s}$,

$$\exists \hat{s} \in \hat{S} \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, |\zeta(\hat{i})| > 1. \tag{5.33}$$

This can be weakened to

$$\exists \hat{s} \in \hat{S} \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \forall i \in I \,.\, \zeta(\hat{i}) \neq \{i\}, \tag{5.34}$$

which is the same as (5.27a) and we can complete proving this case using the argument from Case (b).

Case (d). (5.13d) does not hold for $\hat{G}$ but holds for $\hat{G}^*$, i.e. it holds that

$$\exists(\hat{s},\hat{i},s,i) \in \hat{S} \times \hat{I} \times S \times I \,.\, ((\gamma(\hat{s}),\zeta(\hat{i})) = (\{s\},\{i\}) \wedge \gamma(\hat{f}(\hat{s},\hat{i})) \neq \{f(s,i)\}), \tag{5.35a}$$

$$\forall(\hat{s},\hat{i},s,i) \in \hat{S} \times \hat{I} \times S \times I \,.\, ((\gamma(\hat{s}),\zeta(\hat{i})) = (\{s\},\{i\}) \Rightarrow \gamma(\hat{f}^*(\hat{s},\hat{i})) = \{f(s,i)\}). \tag{5.35b}$$

We fix $\hat{s},\hat{i},s,i$ from (5.35a) and choose them in (5.35b), obtaining

$$(\gamma(\hat{s}),\zeta(\hat{i})) = (\{s\},\{i\}) \wedge \gamma(\hat{f}(\hat{s},\hat{i})) \neq \{f(s,i)\}, \tag{5.36a}$$

$$(\gamma(\hat{s}),\zeta(\hat{i})) = (\{s\},\{i\}) \Rightarrow \gamma(\hat{f}^*(\hat{s},\hat{i})) = \{f(s,i)\}. \tag{5.36b}$$

We therefore know $\gamma(\hat{f}(\hat{s},\hat{i})) \neq \gamma(\hat{f}^*(\hat{s},\hat{i}))$ and $|\gamma(\hat{f}^*(\hat{s},\hat{i}))| = 1$. Due to assumed monotonicity, we also know from (5.10c) that $\gamma(\hat{f}^*(\hat{s},\hat{i})) \subseteq \gamma(\hat{f}(\hat{s},\hat{i}))$. Therefore, it must be the case that

$$\exists s \in \gamma(\hat{f}^*(\hat{s},\hat{i})) \,.\, s \notin \gamma(\hat{f}^*(\hat{s},\hat{i})). \tag{5.37a}$$

Reintroducing the quantifiers for $\hat{s},\hat{i}$, we obtain (5.12) from $\hat{G}$ to $\hat{G}^*$, which completes the proof. □

5.3.4 Proof of the Strict Refinement Lemma

Proof (Lemma 5.2.20).

First part. We will first prove the claim that if $\hat{G}$ is (γ,ζ)-terminating wrt. G, then for every µ-calculus property ψ, $[\![\psi]\!](\Gamma(\hat{G})) = [\![\psi]\!](\Gamma(G))$. Since G is a concrete generating automaton, $[\![\psi]\!](\Gamma(G)) \neq \bot$, and it suffices to prove $\Gamma(G)$ is sound wrt. $\Gamma(\hat{G})$. Using Lemma 5.3.4, we define H as

$$H \stackrel{\text{def}}{=} \{(\hat{s},s) \mid \gamma(\hat{s}) = \{s\}\}. \tag{5.38}$$

(5.16a) and (5.16b) hold due to (5.7a) and (5.13a), respectively. We rewrite (5.16c) and (5.16d) as

$$\forall(\hat{s},s) \in \hat{S} \times S \,.\, (\gamma(\hat{s}) = \{s\} \Rightarrow \forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \exists i \in I \,.\, \gamma(\hat{f}(\hat{s},\hat{i})) = \{f(s,i)\}), \tag{5.39a}$$

$$\forall(\hat{s},s) \in \hat{S} \times S \,.\, (\gamma(\hat{s}) = \{s\} \Rightarrow \forall i \in I \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \gamma(\hat{f}(\hat{s},\hat{i})) = \{f(s,i)\}). \tag{5.39b}$$

We assume $(\hat{s},s) \in \hat{S} \times S$ arbitrary but fixed and $\gamma(\hat{s}) = \{s\}$ to hold.

To prove (5.26a), we assume $\hat{i} \in \hat{q}(\hat{s})$ arbitrary but fixed and choose $i \in I$ for which $\zeta(\hat{i}) = \{i\}$, which exists due to (5.13b).

To prove (5.26b), we assume $i \in I$ arbitrary but fixed and choose $\hat{i} \in \hat{q}(\hat{s})$ for which $\zeta(\hat{i}) = \{i\}$, which exists due to (5.13c).

Second part. We turn to the claim that if $\hat{G}$ is a soundness-guaranteeing (γ,ζ)-abstraction of G for which some (γ,ζ)-monotone refinement $\hat{G}^*$ is (γ,ζ)-terminating wrt. G, then either $\hat{G}$ itself is (γ,ζ)-terminating wrt. G or the refinement is strict. We will prove

this by assuming $\hat{G}$ is not (γ, ζ)-terminating and proving that $\hat{G}^*$ is a strict (γ, ζ)-monotone refinement of $\hat{G}$.

As per Definition 5.2.15, we are to prove either (5.11) or (5.12) holds for the refinement from $\hat{G}$ to $\hat{G}^*$. Since $\hat{G}$ is not guaranteed-terminating, at least one of (5.13a), (5.13b), (5.13c), (5.13d) does not hold.

Case (a). (5.13a) does not hold for $\hat{G}$. However, $\hat{S}$, $\hat{I}$, and $\hat{L}$ are the same for $\hat{G}^*$, and we already assumed (5.13a) for $\hat{G}^*$. This is a contradiction and so this case cannot occur.

Case (b). (5.13b) does not hold for $\hat{G}$, but we have assumed it holds for $\hat{G}^*$, i.e.

$$\exists \hat{s} \in \hat{S} \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \forall i \in I \,.\, \zeta(\hat{i}) \neq \{i\}. \tag{5.40a}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall \hat{i^*} \in \hat{q}^*(\hat{s}) \,.\, \exists i^* \in I \,.\, \zeta(\hat{i^*}) = \{i^*\}. \tag{5.40b}$$

We will prove that (5.11) holds for the refinement from $\hat{G}$ to $\hat{G}^*$, i.e.

$$\exists \hat{s} \in \hat{S} \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \forall \hat{i'} \in \hat{q}^*(\hat{s}) \,.\, \exists i \in \zeta(\hat{i}) \,.\, i \notin \zeta(\hat{i'}). \tag{5.41}$$

We fix $\hat{s}$, $\hat{i}$ from (5.27a) and choose them in (5.28). For all $\hat{i'} \in \hat{q}^*(\hat{s})$, clearly $|\gamma(\hat{i'})| = 1$ due to (5.27b). However, due to (5.27a) and ζ not returning $\emptyset$, $|\gamma(\hat{i})| > 1$. As such, for any $\hat{i'} \in \hat{q}^*(\hat{s})$, there exists $i \in \zeta(\hat{i})$ such that $i \notin \zeta(\hat{i'})$.

Case (c). (5.13c) does not hold for $\hat{G}$, i.e.

$$\exists \hat{s} \in \hat{S} \,.\, \exists i \in I \,.\, \forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \zeta(\hat{i}) \neq \{i\}. \tag{5.42}$$

Since $\hat{G}$ is assumed to be a soundness-guaranteeing (γ, ζ)-abstraction of G, it must fulfil the full input coverage requirement from (5.7c), i.e.

$$\forall (\hat{s}, i) \in \hat{S} \times I \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, i \in \zeta(\hat{i}). \tag{5.43}$$

Fixing $\hat{s} \in \hat{S}, i \in I$ from (5.29) and choosing them in (5.30), we combine both formulas to obtain

$$(\forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \zeta(\hat{i}) \neq \{i\}) \wedge (\exists \hat{i} \in \hat{q}(\hat{s}) \,.\, i \in \zeta(\hat{i})). \tag{5.44}$$

We can weaken this to

$$\exists \hat{i} \in \hat{q}(\hat{s}) \,.\, (i \in \zeta(\hat{i}) \wedge \zeta(\hat{i}) \neq \{i\}). \tag{5.45}$$

This implies $\exists \hat{i} \in \hat{q}(\hat{s}) \,.\, |\gamma(\hat{i})| > 1$, where i no longer appears. We reintroduce an existential quantifier for $\hat{s}$,

$$\exists \hat{s} \in \hat{S} \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, |\zeta(\hat{i})| > 1. \tag{5.46}$$

This can be weakened to

$$\exists \hat{s} \in \hat{S} \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \forall i \in I \,.\, \zeta(\hat{i}) \neq \{i\}, \tag{5.47}$$

which is the same as (5.27a) and we can complete proving this case using the argument from Case (b).

Case (d). (5.13d) does not hold for $\hat{G}$ but holds for $\hat{G}^*$, i.e. it holds that

$$\exists(\hat{s},\hat{i},s,i) \in \hat{S} \times \hat{I} \times S \times I \;.\; ((\gamma(\hat{s}),\zeta(\hat{i})) = (\{s\},\{i\}) \wedge \gamma(\hat{f}(\hat{s},\hat{i})) \neq \{f(s,i)\}), \tag{5.48a}$$

$$\forall(\hat{s},\hat{i},s,i) \in \hat{S} \times \hat{I} \times S \times I \;.\; ((\gamma(\hat{s}),\zeta(\hat{i})) = (\{s\},\{i\}) \Rightarrow \gamma(\hat{f}^*(\hat{s},\hat{i})) = \{f(s,i)\}). \tag{5.48b}$$

We fix $\hat{s},\hat{i},s,i$ from (5.35a) and choose them in (5.35b), obtaining

$$(\gamma(\hat{s}),\zeta(\hat{i})) = (\{s\},\{i\}) \wedge \gamma(\hat{f}(\hat{s},\hat{i})) \neq \{f(s,i)\}, \tag{5.49a}$$

$$(\gamma(\hat{s}),\zeta(\hat{i})) = (\{s\},\{i\}) \Rightarrow \gamma(\hat{f}^*(\hat{s},\hat{i})) = \{f(s,i)\}. \tag{5.49b}$$

We therefore know $\gamma(\hat{f}(\hat{s},\hat{i})) \neq \gamma(\hat{f}^*(\hat{s},\hat{i}))$ and $|\gamma(\hat{f}^*(\hat{s},\hat{i}))| = 1$. Due to assumed monotonicity, we also know from (5.10c) that $\gamma(\hat{f}^*(\hat{s},\hat{i})) \subseteq \gamma(\hat{f}(\hat{s},\hat{i}))$. Therefore, it must be the case that

$$\exists s \in \gamma(\hat{f}^*(\hat{s},\hat{i})) \;.\; s \notin \gamma(\hat{f}^*(\hat{s},\hat{i})). \tag{5.50a}$$

Reintroducing the quantifiers for $\hat{s},\hat{i}$, we obtain (5.12) from $\hat{G}$ to $\hat{G}^*$, which completes the proof. □

5.4 Implementation and Experimental Evaluation

We implemented an instance of our proposed framework in the publicly available, free, and open-source tool **machine-check**. In this section, we show how our framework is able to mitigate exponential explosion[7].

Implementation. We use three-valued bit-vector abstraction with explicit-state forward simulation to build the state space, using CTL model-checking [148] to model-check PKS [132]. We currently support and will experimentally evaluate three basic strategies for the initial GA:

- **Naïve.** Initially $\hat{p}_{\hat{q}}(\hat{s}) = (1)_{k=0}^{y-1}$, $\hat{p}_{\hat{f}}(\hat{s}) = (1)_{k=0}^{w-1}$ from all states $\hat{s} \in \hat{S}$. The model-checking result is immediately non-$\bot$ (no refinements occur).
- **Input splitting.** Instead, $\hat{p}_{\hat{q}}(\hat{s}) = (0)_{k=0}^{y-1}$, i.e. the qualified inputs are initially not split, and a lasso-shaped PKS is generated initially, like in Figure 5.3a.
- **Input splitting with decay.** In addition, $\hat{p}_{\hat{f}}(\hat{s}) = (0)_{k=0}^{w-1}$, i.e. all step results initially decay to the state $(\text{'X'})^w$.

We deduce the refinements to be performed from the computed labellings, performing a breadth-first search until a $\bot$ labelling forcing the verified property to be $\bot$ is found. We then deduce which input or state bits imprecise due to $\hat{p}_{\hat{q}},\hat{p}_{\hat{f}}$ could have caused that, choose the refinement, update the state space (retaining unchanged parts), and recompute all

[7]The evaluation results together with the source code and scripts for their reproduction are available in an artefact located at `https://doi.org/10.5281/zenodo.15109048`.

labellings. We discuss the abstraction and refinement choices in more detail in Chapter 7. Irrespective of refinement choices, **machine-check** is designed to be sound, monotone, and complete, as discussed in Examples 5.2.4-5.2.22, notwithstanding possible bugs.

The choice of the implementation of monotone versions of input and step precision from Example 5.2.4 is non-trivial: since the functions must be computed for each state when building the state space, the complexity impact can be severe. We implemented fast computation of them using a transitive reduction graph of the states with non-default precisions and some of their covering states. When computing $\hat{m}_{\hat{q}}$ or $\hat{m}_{\hat{f}}$ as in (5.4), it is then possible not to consider states where some state covering them is already known not to be covered by the queried state.

Parametric systems. We verify recovery properties on a set of systems parametrised by naturals V, U, C, and a Boolean determining if they are recoverable or not. Each input is a tuple $i = (n, z, r)$ where n has V bits, z has U bits, and r is a single bit that determines whether the system should be reset. Each state is a tuple $s = (v, u, c)$, where the V-bit variable v is the running maximum of the input n, the U-bit variable u is loaded from the input z in every step but otherwise unused and irrelevant, and the C-bit variable c is an irrelevant free-running counter. The initial state is $(0, 0, 0)$. The step functions are

$$\begin{aligned} f_{\text{non-recoverable}}((v,u,c),(n,w,r)) &\stackrel{\text{def}}{=} (\max(v,n), w, c+1 \bmod 2^C), \\ f_{\text{recoverable}}((v,u,c),(n,w,r)) &\stackrel{\text{def}}{=} ((1-r)\max(v,n), w, c+1 \bmod 2^C), \end{aligned} \tag{5.51}$$

Using **machine-check**, we verified $\mathbf{AG}[\mathbf{EF}[v = 0]]$ for varying parameters, and visualised the elapsed wall-time in Figure 5.5. Our framework implementation is fairly well-behaved in regard to the parameters, with clear trends shown by the strategies.

- **Naïve.** Susceptible to exponential explosion in all shown cases, expected as it corresponds to model-checking without abstraction.
- **Input splitting.** Not susceptible to exponential explosion due to irrelevant unused input assignments (middle column of Figure 5.5).
- **Input splitting with decay.** In addition, not susceptible to exponential explosion due to irrelevant computations (right column of Figure 5.5).

The evaluation confirms generally expected TVAR behaviour, mitigating exponential explosion stemming from irrelevant information. The price of refining, seen in the left column of Figure 5.5, a shift of the trend curve upwards when introducing input splitting and then also decay, is overshadowed by the mitigation effectiveness in the other columns.

The trends of the non-naïve strategies where mitigation did not succeed are super-linear. This is due to the introduction of the monotone versions of input and step precision in (5.4). We consider this to be a tradeoff for the granularity of precision: the problem would not exist if the precisions were constant for all states within each refinement.

Experimental comparison to previous abstraction-refinement frameworks would be problematic, as the results would be dominated by the abstraction and refinement choices. That said, CEGAR- and (G)STE-based approaches would be incapable of producing results comparable to Figure 5.5 because they cannot soundly verify $\mathbf{AG}[\mathbf{EF}[v = 0]]$ at all.

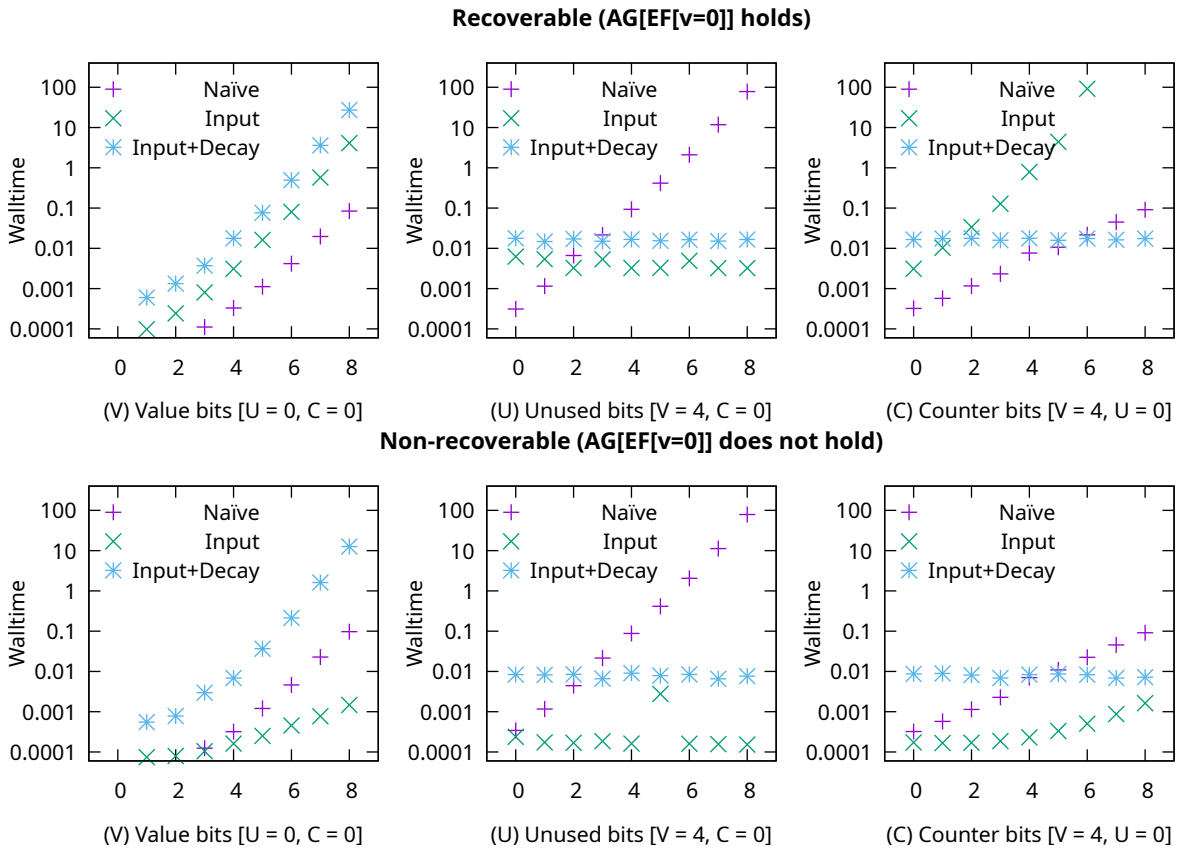


Figure 5.5: Wall-time elapsed during verification of the recovery property on a PC with Ryzen 5600 CPU. In the upper-left corner, $N > 6$ was not measured for the input strategy for feasibility reasons. Rising-line trends correspond to exponential explosion. Horizontal-line trends show complete insensitivity 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 quickly reaching a state where $\mathbf{EF}[v = 0]$ does not hold.

5.5 Further Notes

The introduced input-based TVAR framework forms the basis of **machine-check**, allowing sound, monotone, and complete formal verification while being fairly simple to work with. Unlike CEGAR-based frameworks, it can be used to verify branching-time properties. While only verification of CTL properties is currently implemented, it could be enhanced in the future, as the framework is general enough for the whole μ-calculus.

For quick verification of systems such as those used in the evaluation while using three-valued abstraction, it is necessary to compute the results of arithmetic operations quickly on the three-valued bit-vectors that form the states. In Chapter 6, I will present a technique that accomplishes that. After that, I will discuss the implementation of **machine-check** in more detail in Chapter 7, including an experimental evaluation on machine-code systems.

Chapter 6

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 [8, 4].

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 6.0.1. 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 6.6.

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.

6.1 Related Work

Many-valued logics have been extensively studied on their own, including Kleene logic [149] used for three-valued model checking [8]. Previously, three-valued logic was used for static program analysis of 8-bit microcontroller programs [58]. 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 [150] 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. [151] 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 [152]. 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.

6.2 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

$$\text{'0'} \stackrel{\text{def}}{=} \{0\}, \text{'1'} \stackrel{\text{def}}{=} \{1\}, \text{'X'} \stackrel{\text{def}}{=} \{0, 1\}. \tag{6.1}$$

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.

6.2.1 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

$$a_i^0 = 1 \iff 0 \in \hat{a}_i,\ a_i^1 = 1 \iff 1 \in \hat{a}_i, \tag{6.2}$$

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 [8]. 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

$$a_i^{\mathrm{m}} = 1 \iff 0 \in \hat{a}_i \wedge 1 \in \hat{a}_i,\ a_i^{\mathrm{v}} = 1 \iff 0 \notin \hat{a}_i \wedge 1 \in \hat{a}_i, \tag{6.3}$$

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

$$\begin{aligned} a_i^0 = 1 &\iff a_i^{\mathrm{m}} = 1 \vee a_i^{\mathrm{v}} = 0,\ a_i^1 = 1 \iff a_i^{\mathrm{m}} = 1 \vee a_i^{\mathrm{v}} = 1, \\ a_i^{\mathrm{m}} = 1 &\iff a_i^0 = 1 \wedge a_i^1 = 1,\ a_i^{\mathrm{v}} = 1 \iff a_i^0 = 0 \wedge a_i^1 = 1. \end{aligned} \tag{6.4}$$

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. $\hat{a}$.

6.2.2 Abstract Transformers

We borrow the notions defined in this subsection from abstract interpretation [153, 154], 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* $\gamma : (2^{\mathbb{B}})^A \to 2^{(\mathbb{B}^A)}$,

$$\gamma(\hat{a}) \stackrel{\text{def}}{=} \{a \in \mathbb{B}^A \mid \forall i \in \{0, \ldots, A-1\} \,.\, a_i \in \hat{a}_i\}. \tag{6.5}$$

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,

$$\forall \hat{a} \in (2^{\mathbb{B}})^N, \hat{b} \in (2^{\mathbb{B}})^N \,.\, \{r(a,b) \mid a \in \gamma(\hat{a}), b \in \gamma(\hat{b})\} \subseteq \gamma(\hat{r}(\hat{a}, \hat{b})). \tag{6.6}$$

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

$$\forall i \in \{0, \ldots, A-1\} \,.\, (\alpha^{\text{best}}(C))_i \stackrel{\text{def}}{=} \{c_i \in \mathbb{B} \mid c \in C\}. \tag{6.7}$$

By using α^{best} to perform the abstraction on the minimal set of concrete results from Equation 6.6, 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

$$\hat{r}_k^{\text{best}}(\hat{a}, \hat{b}) = \alpha^{\text{best}}(\{r_k(a, b) \mid a \in \gamma(\hat{a}), b \in \gamma(\hat{b})\}). \tag{6.8}$$

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.

6.2.3 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.

6.2.4 Naïve Universal Abstract Algorithm

Equation 6.8 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.

6.3 Formal Problem Statement

Theorem 6.3.1. The best abstract transformer of abstract bit-vector addition is computable in linear time.

Theorem 6.3.2. The best abstract transformer of abstract bit-vector multiplication is computable in worst-case quadratic time.

In Section 6.4, 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 6.3.1 and 6.3.2 by constructing corresponding algorithms in Sections 6.5 and 6.6, respectively. We will experimentally evaluate the presented algorithms to demonstrate their practical efficiency in Section 6.7.

6.4 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

$$\begin{aligned} &\forall a \in \mathbb{B}^N, b \in \mathbb{B}^N, \forall k \in \{0, \ldots, M-1\} \; . \\ &\qquad r_k(a,b) = 1 \iff (h(a,b) \bmod 2^{k+1}) \geq 2^k. \end{aligned} \tag{6.9}$$

The best abstract transformer definition in Equation 6.8 is then equivalent to

$$\begin{aligned} &\forall k \in \{0, \ldots, M-1\} \; . \\ &\qquad (0 \in \hat{r}_k^{\text{best}} \iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \; . \; (h(a,b) \bmod 2^{k+1}) < 2^k) \wedge \\ &\qquad (1 \in \hat{r}_k^{\text{best}} \iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \; . \; (h(a,b) \bmod 2^{k+1}) \geq 2^k). \end{aligned} \tag{6.10}$$

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 6.10 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 6.4.1. Consider a sequence of integers $t = (t_0, t_1, \ldots, t_{T-1})$ that fulfils

$$\forall n \in [0, T-2] \,.\, |t_{n+1} - t_n| \leq 2^k. \tag{6.11}$$

Then,

$$\begin{aligned} &\exists v \in [\min t, \max t] \,.\, (v \bmod 2^{k+1}) < 2^k \iff \\ &\exists n \in [0, T-1] \,.\, (t_n \bmod 2^{k+1}) < 2^k. \end{aligned} \tag{6.12}$$

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 6.4.2. 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

$$\begin{aligned} &\forall n \in [0, P-2] \,.\, |f(p_{n+1}) - f(p_n)| \leq 2^k, \\ &f(p_0) = \min_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} f(a, b), \\ &f(p_{P-1}) = \max_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} f(a, b). \end{aligned} \tag{6.13}$$

Then,

$$\begin{aligned} \forall\, C \in \mathbb{Z} \,.\, (\exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \,.\, ((f(a,b) + C) \bmod 2^{k+1}) < 2^k \\ \iff \exists n \in [0, P-1] \,.\, ((f(p_n) + C) \bmod 2^{k+1}) < 2^k). \end{aligned} \tag{6.14}$$

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 6.4.1 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

$$\begin{aligned} &\forall\, C \in \mathbb{Z} \,.\, (\exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \,.\, ((f(a,b) + C) \bmod 2^{k+1}) < 2^k \implies \\ &\exists v \in \left[\min_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} (f(a,b) + C) \,,\, \max_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} (f(a,b) + C) \right] \,.\, (v \bmod 2^{k+1}) < 2^k). \end{aligned} \tag{6.15}$$

The implication clearly holds, completing the proof. □

While Theorem 6.4.2 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 6.10 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 6.4.2 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),

$$\zeta_k(x) = \left\lfloor \frac{x}{2^k} \right\rfloor . \tag{6.16}$$

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

$$h_k^{\min} \stackrel{\text{def}}{=} \min_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} h_k(a,b), h_k^{\max} \stackrel{\text{def}}{=} \max_{\substack{a \in \gamma(\hat{a}) \\ b \in \gamma(\hat{b})}} h_k(a,b), \tag{6.17}$$

Equation 6.10 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 6.1, 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 6.1: 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
```

6.5 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,

$$\Phi(a) \stackrel{\text{def}}{=} \sum_{i=0}^{A-1} 2^i a_i. \tag{6.18}$$

Pseudo-Boolean addition is then defined simply as

$$h^+(a,b) \stackrel{\text{def}}{=} \Phi(a) + \Phi(b). \tag{6.19}$$

To fulfil the step condition, we define

$$h_k^+(a,b) = \Phi(a_{0..k}) + \Phi(b_{0..k}). \tag{6.20}$$

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 6.2.1, the extremes can be obtained as

$$\begin{aligned} h_k^{+,\min} &\leftarrow \Phi(a^{\text{v}}_{0..k}) + \Phi(b^{\text{v}}_{0..k}), \\ h_k^{+,\max} &\leftarrow \Phi(a^{1}_{0..k}) + \Phi(b^{1}_{0..k}). \end{aligned} \tag{6.21}$$

The best abstract transformer for addition is obtained by combining Equation 6.21 with Algorithm 6.1. Time complexity is trivially $\Theta(N)$, proving Theorem 6.3.1. 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":

$$\begin{aligned} &k = 0 : \text{“0” + “1”},\ 1 = \zeta_0(0+1) = \zeta_0(0+1) = 1 \rightarrow r_0 = \text{‘1’}, \\ &k = 1 : \text{“X0” + “11”},\ 1 = \zeta_1(0+3) \neq \zeta_1(2+3) = 2 \rightarrow r_1 = \text{‘X’}, \\ &k = 2 : \text{“0X0” + “011”},\ 0 = \zeta_2(0+3) \neq \zeta_2(2+3) = 1 \rightarrow r_2 = \text{‘X’}, \\ &k > 2 : \zeta_k(h_k^{+,\min}) = \zeta_k(h_k^{+,\max}) = 0 \rightarrow r_k = \text{‘0’}. \end{aligned} \tag{6.22}$$

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.

6.6 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.

6.6.1 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:

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

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

$$h^*(a,b) \stackrel{\text{def}}{=} \Phi(a) \cdot \Phi(b) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 2^{i+j} a_i b_j. \tag{6.23}$$

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

$$h_k(a,b) = \sum_{i=0}^{k} \sum_{j=0}^{k-i} 2^{i+j} a_i b_j. \tag{6.24}$$

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

$$h_k^*(a, b) \stackrel{\text{def}}{=} \left(-\sum_{i=0}^{k} 2^k a_i b_{k-i} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i b_j \right). \tag{6.25}$$

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 6.25 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 6.25, 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 6.6.2, 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 6.6.3, 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 6.6.4, we discuss implementation considerations of the algorithm with emphasis on reducing computation time. Finally, in Subsection 6.6.5, we present the final algorithm.

6.6.2 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 6.17). 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 6.25, the difference between h_k^* when $a_i = 1$ and when $a_i = 0$ is

$$h_k^*(a, b \mid a_i = 1) - h_k^*(a, b \mid a_i = 0) = -2^k b_{k-i} + \sum_{j=0}^{k-i-1} 2^{i+j} b_j. \tag{6.26}$$

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

$$\begin{aligned} &h_k^*(a, b) - h_k^*(a, b \mid a_i = 0, b_j = 0) = \\ &- 2^k a_i b_j + 2^i a_i \left(\sum_{z=0}^{j-1} 2^z b_z \right) + 2^j b_j \left(\sum_{z=0}^{i-1} 2^z a_z \right). \end{aligned} \tag{6.27}$$

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

$$2^i \left(\sum_{z=0}^{j-1} 2^z b_z \right) + 2^j \left(\sum_{z=0}^{i-1} 2^z a_z \right) \leq 2^k. \tag{6.28}$$

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

$$2^j \left(\sum_{z=0}^{i-1} 2^z a_z \right) \leq 2^i \left(\sum_{z=0}^{j-1} 2^z b_z \right). \tag{6.29}$$

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

$$\begin{aligned} h_k^{*,\min} &= \left(-\sum_{i=0}^{k} 2^k a_i^{\min} b_{k-i}^{\min} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i^{\min} b_j^{\min} \right), \\ h_k^{*,\max} &= \left(-\sum_{i=0}^{k} 2^k a_i^{\max} b_{k-i}^{\max} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i^{\max} b_j^{\max} \right). \end{aligned} \tag{6.30}$$

6.6.3 Multiple Double-Unknown k-th Column Pairs

Lemma 6.6.1. Consider a sequence of integers $t = (t_0, t_1, \ldots, t_{T-1})$ that fulfils

$$\forall n \in [0, T-2] \,.\, |t_{n+1} - t_n| \leq 2^k, t_0 + 2^k \leq t_{T-1}. \tag{6.31}$$

Then,

$$\exists n \in [0, T-1] \,.\, (t_n \bmod 2^{k+1}) < 2^k. \tag{6.32}$$

Proof. Use Lemma 6.4.1 to transform the claim to equivalent

$$\exists v \in [\min t, \max t] \,.\, (v \bmod 2^{k+1}) < 2^k. \tag{6.33}$$

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

$$\exists v \in [t_0, t_0 + 2^k] \,.\, (v \bmod 2^{k+1}) < 2^k. \tag{6.34}$$

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 6.34 holds due to the pigeonhole principle. □

Corollary 6.6.2. Given a sequence of integers $(t_0, t_1, \ldots, t_{T-1})$ that fulfils Lemma 6.6.1 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 6.6.3. 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

$$\begin{gathered} p_1 \neq p_2, p_1 + q_2 = k, p_2 + q_1 = k, \\ \hat{a}_{p_1} = \hat{X}, \hat{a}_{p_2} = \hat{X}, \hat{b}_{q_1} = \hat{X}, \hat{b}_{q_2} = \hat{X}. \end{gathered} \tag{6.35}$$

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 $h(a, b) = \Phi(a) \cdot \Phi(b)$,

$$h(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b})) = (2^{p_1} s_1 + 2^{p_2} s_2 + \Phi_{00}(\hat{a})) \cdot (2^{q_1} t_1 + 2^{q_2} t_2 + \Phi_{00}(\hat{b})). \tag{6.36}$$

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

$$\begin{aligned} &h_k^{\text{proof}}(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b})) \stackrel{\text{def}}{=} \\ &\quad 2^{p_1+q_1} s_1 t_1 + 2^{p_1+q_2} s_1 t_2 + 2^{q_1} t_1 A_{0..p_2-1} + 2^{p_1} s_1 B_{0..q_2-1} + \\ &\quad 2^{p_2+q_1} s_2 t_1 + 2^{p_2+q_2} s_2 t_2 + 2^{q_2} t_2 A_{0..p_1-1} + 2^{p_2} s_2 B_{0..q_1-1} + AB. \end{aligned} \tag{6.37}$$

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

$$D(s_1, s_2, t_1, t_2) \stackrel{\text{def}}{=} h_k^{\text{proof}}(\gamma_{s_1,s_2}(\hat{a}), \gamma_{t_1,t_2}(\hat{b})) \; - \; h_k^{\text{proof}}(\gamma_{00}(\hat{a}), \gamma_{00}(\hat{b})). \tag{6.38}$$

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

$$\begin{aligned} D(s_1, s_2, t_1, t_2) = {} & 2^{p_1+q_1} s_1 t_1 + 2^k s_1 t_2 + 2^{q_1} t_1 A_{0..p_2-1} + 2^{p_1} s_1 B_{0..q_2-1} + \\ & 2^k s_2 t_1 + 2^{p_2+q_2} s_2 t_2 + 2^{q_2} t_2 A_{0..p_1-1} + 2^{p_2} s_2 B_{0..q_1-1}. \end{aligned} \tag{6.39}$$

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

$$\begin{aligned} D(1,1,0,0) &= D(1,0,0,0) + D(0,1,0,0), \\ D(0,0,1,1) &= D(0,0,1,0) + D(0,0,0,1), \\ D(1,0,0,1) &= 2^k + D(1,0,0,0) + D(0,0,0,1). \end{aligned} \tag{6.40}$$

Inspecting the various summands, note that

$$\begin{aligned} D(1,0,0,0) \in [0, 2^k - 1], \; D(0,1,0,0) &\in [0, 2^k - 1], \\ D(0,0,1,0) \in [0, 2^k - 1], \; D(0,0,0,1) &\in [0, 2^k - 1], \\ D(1,1,0,0) - D(1,0,0,0) &\in [0, 2^k - 1], \\ D(0,0,1,1) - D(0,0,1,0) &\in [0, 2^k - 1]. \end{aligned} \tag{6.41}$$

Recalling Equation 6.10, the best abstract transformer can be obtained as

$$\begin{aligned} 0 \in \hat{r}_k^{\text{best}} &\iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \; . \; (h_k^{\text{proof}}(a,b) \bmod 2^{k+1}) < 2^k, \\ 1 \in \hat{r}_k^{\text{best}} &\iff \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \; . \; ((h_k^{\text{proof}}(a,b) + 2^k) \bmod 2^{k+1}) < 2^k. \end{aligned} \tag{6.42}$$

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 6.6.1 then implies that both inequalities can be fulfilled due to Corollary 6.6.2, 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 6.6.1 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 6.6.4. $D(0,1,0,0) \geq D(0,0,0,1)$. Considering Equations 6.40 and 6.41, a qualifying sequence is

$$(D(0,0,0,0), D(1,0,0,0), D(1,1,0,0), D(1,0,0,1)). \tag{6.43}$$

Case 6.6.5. $D(0,1,0,0) < D(0,0,0,1)$. Using Equation 6.39, rewrite the case condition to

$$2^{p_2-p_1} D(1,0,0,0) < 2^{q_2-q_1} D(0,0,1,0). \tag{6.44}$$

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

$$2^{p_2-p_1} D(1,0,0,0) < 2^{p_2-p_1} D(0,0,1,0). \tag{6.45}$$

Therefore, $D(1,0,0,0) < D(0,0,1,0)$. Considering Equations 6.40 and 6.41, a qualifying sequence is

$$(D(0,0,0,0), D(0,0,1,0), D(0,0,1,1), D(1,0,0,1)). \tag{6.46}$$

This completes the proof. □

6.6.4 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

$$\lambda(b, k+1) = ((b_{k-i})_{i=0}^{k}) = ((b_{W-1-i})_{i=W-k-1}^{W-1}) = \lambda(b, W)_{W-k-1..W-1}. \tag{6.47}$$

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 [155, 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 [155, 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 6.30, 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 6.25. 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

$$h_k^*(a,b) = \sum_{i=0}^{k} a_i \left(-2^k b_{k-i} + 2^i \Phi(b_{0..k-i-1})\right). \tag{6.48}$$

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 [156] 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. $N \leq 64$.

6.6.5 Fast Abstract Multiplication Algorithm

Applying the previously discussed improvements directly leads to Algorithm 6.2. 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 6.2: 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 6.6.3
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 6.28
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 6.29
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 6.3.2. 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).

6.7 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.

6.7.1 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 6.1. 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 6.2. 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 6.3. 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.

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.

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

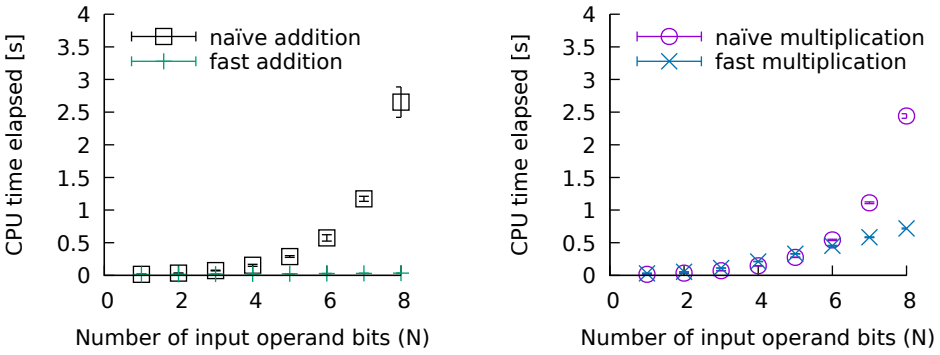


Figure 6.1: Measured computation times for 10^6 random abstract input combinations.

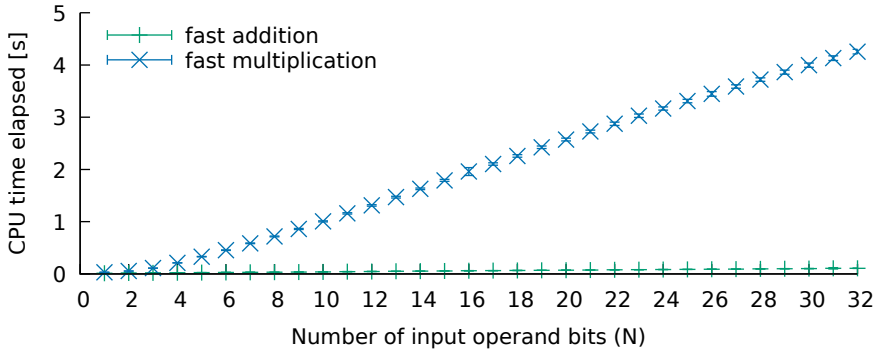


Figure 6.2: Measured computation time for 10^6 random abstract input combinations, fast algorithms only.

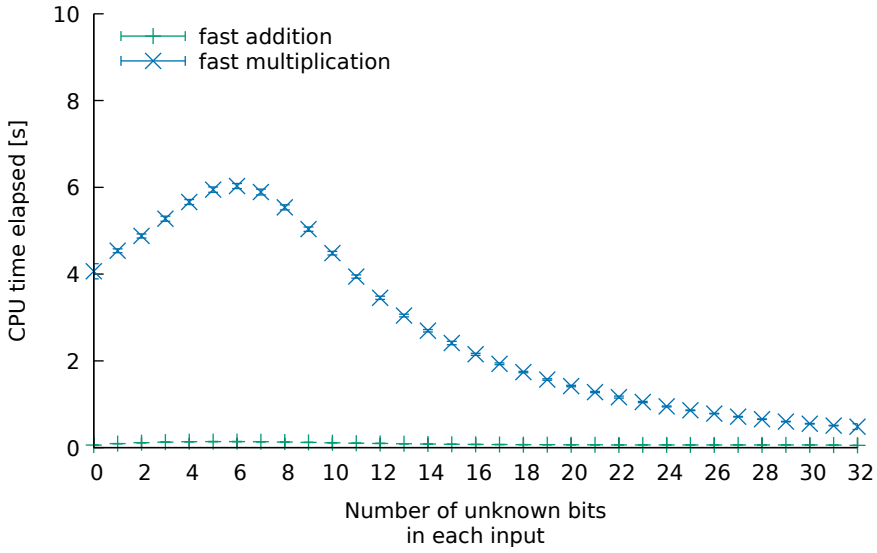


Figure 6.3: Measured computation times for 10^6 random abstract input combinations with fixed $N = 32$, while the number of unknown bits in each input varies.

6.8 Further Notes

In this chapter, a new *modular extreme-finding technique* was introduced 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. Also presented was a linear-time algorithm for abstract addition and a worst-case quadratic algorithm for abstract multiplication, with the experimental evaluation 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.

In **machine-check**, addition, subtraction, and multiplication are currently resolved according to the modular blueprint presented in Algorithm 6.1. While the best algorithm for multiplication could be used in the future, the use of the abstraction refinement framework from Chapter 5 makes it a bit less important to immediately compute the best results. The insights from Section 6.6 into which bits may not have the best result computed using Algorithm 6.1 are important, as they offer us an option to switch to Algorithm 6.2 in the future for the cases where not having the best result may be problematic. In Chapter 7, the interplay between the introduced techniques will be discussed in more detail.

Chapter 7

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 4, 5, and 6, 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 **machine-check** without considering the implementation specifics too much, giving a high-level overview of the interactions between the formalisms and the implementation before discussing how the translation is accomplished. 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.

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

To combine the techniques from Chapters 4, 5 and 6, we need to begin with the fundamental building block, which is the input-based Three-valued Abstraction Refinement framework introduced in Chapter 5 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 ex-

[1]The official website of the tool is `https://machine-check.org`. 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/0.4.0`.

tensible verification tool, not one that intensely adheres to formalisms, I will only discuss soundness, monotonicity, and completeness with some degree of formality in this section.

Recalling the definition of generating automata from Section 5.2, 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.

To instantiate the framework as per Algorithm 5.1, it is first needed to provide the model-checking algorithm based on the specification formalism. In **machine-check**, I chose Computation Tree Logic (CTL) and implemented the model-checking algorithms as discussed in Section 5.4, using a previously introduced algorithm that combines two passes of the classic explicit-state CTL checking algorithm to obtain the three-valued result [130, 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 $\bot$.

In addition to the model-checking algorithm, it is necessary to consider:

- 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 4, the CGA in **machine-check** is directly given by the system comprised of the described finite-state machine (FSM) and other system instantiation data. S contains the FSM states, and I contains the FSM inputs. Treating s_0 as a dummy start state for simplicity and also considering an arbitrary *system instantiation object* z, I define f as

$$f(s, i) \stackrel{\text{def}}{=} \begin{cases} \texttt{init}(z, i) & s = s_0, \\ \texttt{next}(z, s, i) & s \neq s_0. \end{cases} \tag{7.1}$$

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 4.3. In addition to the input and, in the case of the `next` function, the current state, the functions take a system instantiation object z so that the behaviour can change when e.g. different machine code is loaded. This does not play a formal role, as z is constant during the verification.

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 allow questions about state variables based on relational operators (equality and signed or unsigned comparisons). The choice of available labellings can guide the specification writers, preventing them from writing specifications likely to result in exponential explosion. In the current version of **machine-check**, there must be a constant on the right side of the operator, preventing comparing e.g. the equality of two fields, which tends to result in severe exponential explosion when only the three-valued bit-vector domain is used. In a future version, this limitation could be dropped, allowing for richer specifications.

7.1.1 Abstraction Soundness

Let us now consider soundness in the context of a single refinement with the abstract generating automaton $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$. 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. Recalling Example 5.2.4, naming the width of the state bit-vector w and the width of the input bit-vector as y, the sets S and I with the corresponding concretization functions γ, ζ are defined as

$$\begin{aligned}
&\gamma^{\text{bit}}(\hat{a}) = \{v \in \mathbb{B} \mid (v = 0 \Rightarrow a \neq \text{`1'}) \wedge (v = 1 \Rightarrow a \neq \text{`0'})\},\\
&\hat{S} = \{\text{`0'}, \text{`1'}, \text{`X'}\}^w,\ \gamma(\hat{s}) = \{s \in S \mid \forall k \in [0, w-1] \,.\, s_k \in \gamma^{\text{bit}}(\hat{s}_k)\},\\
&\hat{I} = \{\text{`0'}, \text{`1'}, \text{`X'}\}^y,\ \zeta(\hat{i}) = \{i \in I \,.\, \forall k \in [0, y-1] \,.\, i_k \in \gamma^{\text{bit}}(\hat{i}_k)\}.
\end{aligned} \tag{7.2}$$

The step function is the crucial part where the framework interacts with the description and translated analogues. The translated analogue of f becomes $\hat{f}^{\text{basic}}$: recalling Example 5.2.9, for soundness, the translation must ensure that

$$\forall(\hat{s}, \hat{i}) \in \hat{S} \times \hat{I} \,.\, \forall(s, i) \in \gamma(\hat{s}) \times \zeta(\hat{i}) \,.\, f(s, i) \in \gamma(\hat{f}^{\text{basic}}(\hat{s}, \hat{i})). \tag{7.3}$$

How this is managed will be shown informally in Section 7.2. Examples 5.2.4 and 5.2.9 discuss the way soundness is achieved for $\hat{f}$ itself. As for the labelling function $\hat{L}$, the implementation also must ensure that it fulfils

$$\forall \hat{s} \in \hat{S} \,.\, \forall s \in \gamma(\hat{s}) \,.\, \forall a \in \mathbb{A} \,.\, (\hat{L}(\hat{s}, a) \neq \bot \Rightarrow \hat{L}(\hat{s}, a) = L(s, a)). \tag{7.4}$$

Since the current implementation only supports labellings induced by equality and inequality comparisons of three-valued bit-vector fields with constants, this is easy to achieve by taking the minimum and maximum value of the field and determining the three-valued result of the comparison.

7.1.2 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, deducing 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 three-valued model-checking will be caused by some *culprit*, a path that ends with an unknown labelling that prevents the model-checking result from being non-$\bot$. We want to refine $\hat{p}_{\hat{q}}^n$ and $\hat{p}_{\hat{f}}^n$ to $\hat{p}_{\hat{q}}^{n+1}$ and $\hat{p}_{\hat{f}}^{n+1}$ so that the culprit ultimately disappears when the offending state is replaced by states where the labelling is known.

Marking. For practical systems, we can purely structurally deduce that many of the inputs in $\hat{p}_{\hat{q}}^n$ and decayed parts of states in $\hat{p}_{\hat{f}}^n$ 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 bits of step precision $\hat{p}_{\hat{f}}^n(\hat{f}(\hat{s}_{k-1}))$ that could have affected 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 affected marked operation outputs, until obtaining the marking of $\hat{s}_{k-1}$ and the marking of input precision $\hat{p}_{\hat{q}}^n(\hat{s}_{k-1})$.
6. Decrement k and go to Step 3.

After stopping, we will have the candidates for refinement of $\hat{p}_{\hat{f}}^n$ and $\hat{p}_{\hat{q}}^n$ marked and can choose to set some of the candidate bits in candidate states.

Example 7.1.1. 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:

$$\begin{aligned} &\texttt{init\_ex}(z, i) = (0, i), \\ &\texttt{next\_ex}(z, (t, v), i) = (\min(t + 1, 3), v). \end{aligned} \tag{7.5}$$

We use three-valued abstraction for v and 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 7.1a). Model-checking produces the result $\bot$. If, instead of $\hat{s}_4$, we reached another state $\hat{s}$ where $\hat{L}(\hat{s}, v = 1) \neq \bot$, the property

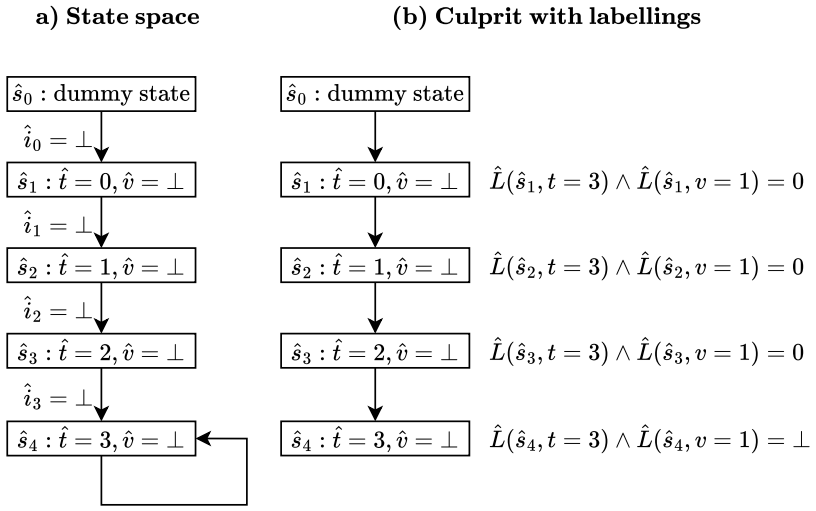


Figure 7.1: An example of a lasso-shaped state space before refinement and the culprit, which consists of a path ending with a state with unknown labelling that forces the model-checking result to be unknown.

could have been determined to hold. As such, we want to make sure that the culprit shown in Figure 7.1b, the path $(\hat{s}_0, \hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4)$ with the labelling $\hat{L}(\hat{s}_4, v = 1) = \bot$, is no longer present in the state space after refinement. As for the labelling, $\hat{L}(\hat{s}_4, v = 1)$ only depends on $\hat{v}$. Therefore, starting the marking loop:

1. We mark $\hat{v}$ in $\hat{s}_4$.
2. Only input splitting is used, so we just mark backwards through $\widehat{\texttt{step\_ex}}$, where the marked $\hat{v}$ only depends on $\hat{v}$ in $\hat{s}_3$. Therefore, we end up with marked $\hat{v}$ in $\hat{s}_3$.
3. We mark backwards two times in the same fashion, marking $\hat{v}$ in $\hat{s}_2$ and then in $\hat{s}_1$.
4. We now have $\hat{v}$ marked in $\hat{s}_1$. Since $\hat{s}_1$ was constructed using the $\widehat{\texttt{init\_ex}}$ function, we mark backwards through $\widehat{\texttt{init\_ex}}$, marking the bit in $\hat{p}^0_{\hat{q}}(\hat{s}_0)$ that caused $\hat{i}_0 = \bot$.
5. We are done with marking. There is only one marked bit of $\hat{p}^0_{\hat{f}}$ or $\hat{p}^0_{\hat{q}}$, the one in $\hat{p}^0_{\hat{q}}(\hat{s}_0)$, so we choose it for refinement.

After the refinement, it will be possible to verify that $\mathbf{EF}[t = 3 \wedge v = 1]$ holds using the newly generated Partial Kripke structure $\Gamma(\hat{G}^1)$.

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 $\hat{f}$. Notably, if the abstract operations do not spuriously generate abstract values with multiple concretizations

when all inputs only had a single one, 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.

Choosing the refinement. After the candidates are found by the marking algorithm, it is necessary to choose which bits in which state to refine. In the current version of **machine-check**, a simple heuristic is used, implemented after I tried out machine-code verification with trivial refinement choices and noticed a problematic pattern that precluded feasible verification due to not refining the Program Counter (PC) state variable first.

For machine-code systems, it is typically desirable to refine the Program Counter before any other variables. However, the concept of the Program Counter is unknown to **machine-check**, as it is simply just another field in the state structure. I was successful in achieving the behaviour generally by adding an importance counter to variable markings. When marking an indexing operation with a not-completely-known index, the index is marked with an incremented importance. The candidate for refinement with the highest importance is selected for refinement. In case there are still multiple candidate bits, the most significant bit of an arbitrarily but deterministically selected field will be refined. As the Program Counter is 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 machine-code programs, as will be shown in Section 7.4. Of course, further improvements may be added in the future.

Ensuring monotonicity. Recalling Examples 5.2.14 and 5.2.16, to ensure monotonicity in **machine-check**, a single bit of the current $\hat{p}_{\hat{q}}$ or $\hat{p}_{\hat{f}}$ is changed after marking and choosing the refinement, and the whole process commences until $\hat{R}$ in $\Gamma(\hat{G})$, i.e. the state space graph, changes. It is also required $\hat{L}$ and $\hat{f}^{basic}$ fulfil

$$\begin{aligned} &\forall(\hat{s}, \hat{s}', a) \in \hat{S} \times \hat{S} \times \mathbb{A} \,. \\ &\qquad ((\gamma(\hat{s}') \subseteq \gamma(\hat{s}) \wedge \hat{L}(\hat{s}, a) \neq \bot) \Rightarrow \hat{L}(\hat{s}, a) = L(\hat{s}', a)), \end{aligned} \tag{7.6}$$

$$\begin{aligned} &\forall(\hat{s}, \hat{s}', \hat{i}, \hat{i}') \in \hat{S} \times \hat{S} \times \hat{I} \times \hat{I} \,. \\ &\qquad ((\gamma(\hat{s}') \times \zeta(\hat{i}') \subseteq \gamma(\hat{s}) \times \zeta(\hat{i})) \Rightarrow \gamma(\hat{f}^{\text{basic}}(\hat{s}', \hat{i}')) \subseteq \gamma(\hat{f}^{\text{basic}}(\hat{s}, \hat{i})). \end{aligned} \tag{7.7}$$

Ensuring completeness. Recalling Example 5.2.22, to achieve completeness, $\hat{L}$ and $\hat{f}^{\text{basic}}$ must fulfil

$$\forall(s, \hat{s}) \in S \times \hat{S} \,.\, (\gamma(\hat{s}) = \{s\} \Rightarrow \hat{L}(\hat{s}) = L(s)), \tag{7.8}$$

$$\forall(\hat{s}, \hat{i}, s, i) \in \hat{S} \times \hat{I} \times S \times I.((\gamma(\hat{s}), \zeta(\hat{i})) = (\{s\}, \{i\}) \Rightarrow \gamma(\hat{f}(\hat{s}, \hat{i})) = \{f(s, i)\}). \tag{7.9}$$

The requirements are easy enough to fulfil for $\hat{L}$ with the chosen atomic properties: simply provide the best comparison possible. I will discuss $\hat{f}^{\text{basic}}$ in the context of translation to the abstract analogues.

7.2 Translation to Abstract and Refinement Analogues

I will now show how to perform translation to abstraction and refinement analogues by rewriting the code of the simulable description. The refinement analogue has no distinct formal requirements as long as it chooses something to refine. As for the abstraction analogue, from the previous section, there are the following requirements for $\hat{f}^{\text{basic}}$:

$$\forall(\hat{s},\hat{i}) \in \hat{S} \times \hat{I} \,.\, \forall(s,i) \in \gamma(\hat{s}) \times \zeta(\hat{i}) \,.\, f(s,i) \in \gamma(\hat{f}^{\text{basic}}(\hat{s},\hat{i})), \tag{7.10a}$$

$$\forall(\hat{s},\hat{s}',\hat{i},\hat{i}') \in \hat{S} \times \hat{S} \times \hat{I} \times \hat{I} \,. \\ ((\gamma(\hat{s}') \times \zeta(\hat{i}') \subseteq \gamma(\hat{s}) \times \zeta(\hat{i})) \Rightarrow \gamma(\hat{f}^{\text{basic}}(\hat{s}',\hat{i}')) \subseteq \gamma(\hat{f}^{\text{basic}}(\hat{s},\hat{i})), \tag{7.10b}$$

$$\forall(\hat{s},\hat{i},s,i) \in \hat{S} \times \hat{I} \times S \times I.((\gamma(\hat{s}),\zeta(\hat{i})) = (\{s\},\{i\}) \Rightarrow \gamma(\hat{f}^{\text{basic}}(\hat{s},\hat{i})) = \{f(s,i)\}). \tag{7.10c}$$

The requirements are very natural: the result of the abstract analogue must cover every result of the original step function to achieve soundness, it must be monotone with respect to coverage to achieve monotonicity, and it must preserve a single concretization to achieve completeness.

With an appeal to intuition[2] supported by classical static program analysis [157, 158], due to the non-relational nature of the three-valued bit-vector abstraction, $\hat{f}^{\text{basic}}$ can be implemented by manipulating variables in an analogous manner to f provided each basic operation fulfils its version of Equation 7.10. Using three-valued bit-wise operations, ensuring this is not problematic for bitwise operations. For the fast arithmetic operations, the notions from Chapter 6 can be used. Based on this treatment, I will now discuss how the translations are performed.

7.2.1 Functions without Control Flow

Let us first consider the function FN1 in Figure 7.2, which simply performs a logical AND of its two inputs and returns the output. For its abstract analogue FN1_ABSTR, the variables are translated to their abstract analogues, replacing Boolean bit-vectors by three-valued bit-vectors, and abstract operations are used instead of standard ones.

Example 7.2.1. The abstract analogue FN1_ABSTR, upon being called with arguments $\hat{a}$ = "XXXXXXXX" and $\hat{b}$ = "00001111", produces "0000XXXX".

The refinement analogue, named FN1_REFIN in Figure 7.2, 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 7.1.2. 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

[2] A more formal treatment could be based on abstract interpretation in the context of static program analysis [157, p. 211-282].

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
    ▷ (Note $\hat{c}$ is unused in this example as no operation uses $\hat{c}$ as an argument.)
  $\hat{a}_{\text{mark},0} \leftarrow$ 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 7.2: A function without control flow and its abstract and refinement analogues.

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.

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 the 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, the markings of the abstract inputs are returned.

Example 7.2.2. Continuing from Example 7.2.1, 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 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 7.3: A function with branching and its abstract analogue.

7.2.2 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 further inspiration from static program analysis [159, 157, 158].

Let us consider the function FN2 in Figure 7.3, 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 in the duplicate branches, combining them using a *phi function* [159]. 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 7.2.3. 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.

7.3 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[3] 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 usually provided as a command-line argument to the resulting executable, as discussed in Chapter 4.

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.

[3] 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 used to yield the constructed system to **machine-check**, and other functions to support the construction of systems based on command-line arguments. A Graphical User Interface can also be used, implemented in the package `machine-check-gui`.

The internal implementation of verification in **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 5, with abstract states stored explicitly, i.e. without use of Binary Decision Diagrams or similar structures.
- **Abstraction and refinement domains** (`mck`[4]). Implements the abstraction and refinement analogues of bit-vectors and bit-vector arrays. Fast bit-vector arithmetic described in Chapter 6 is implemented here.
- **Translation to verification analogues** (`machine-check-machine`). Implements the translation, described in Chapter 4 and Section 7.2, in the `machine_description` macro.

The lines of code grouped by the concepts can be seen in Figure 7.4. Disregarding the Graphical User Interface code that has no impact on verification, 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.

7.3.1 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. There are also some parts of the code where the non-trivial algorithms and structures must be used to achieve good verification performance.

[4]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.

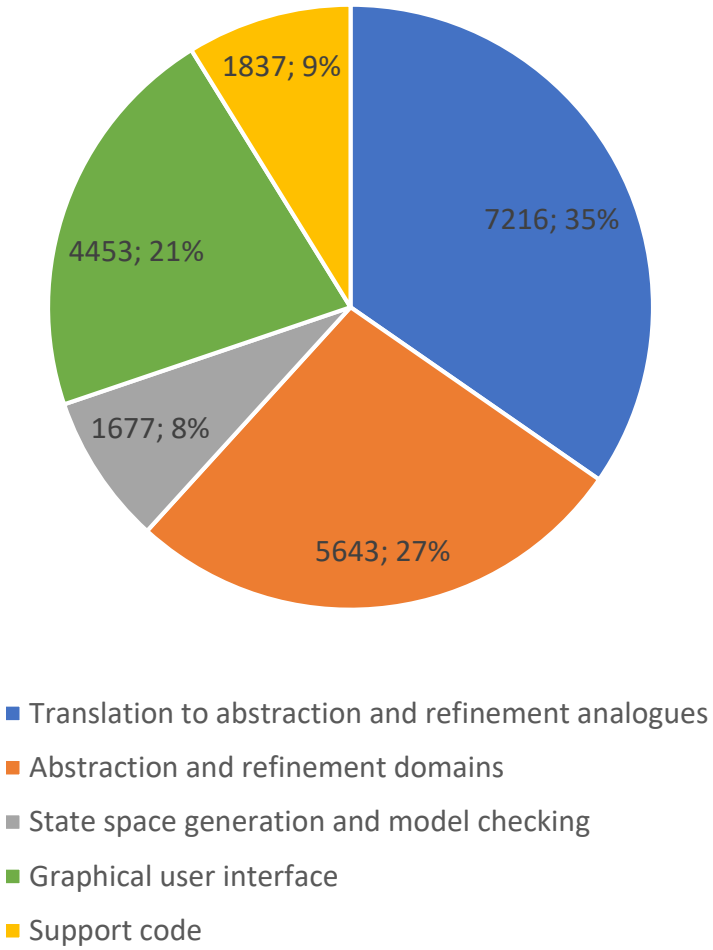


Figure 7.4: Categories of lines of Rust code in **machine-check** version 0.4.0. There are 20826 lines of code in total. Blank and comment lines are not counted.

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. Furthermore, processor instructions and peripherals that are not used can be left unimplemented in the description, raising a panic instead. In **machine-check**, 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 changed the return value of every function before translation to include the information about whether panic was raised and rewrote 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 resolution, 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 4.3) 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.

Computation of monotone precision functions. As discussed in Examples 5.2.4 and 5.2.14, the monotone versions of the precision functions have been introduced to achieve monotonicity. Problematically, since their result must be computed during the construction of the abstract state space for every abstract state, and their trivial implementation requires taking every abstract state into account. The resolution to this was discussed in Section 5.4: a transitive reduction graph is used for states with non-default precisions, allowing a significant improvement over the trivial implementation. While this does not resolve the problem completely, in most scenarios I experimented with, it makes it possible to use fairly fine precision adjustments and preserve monotonicity without sacrificing too much computation speed and memory.

7.4 Verification of AVR Programs

To show the feasibility of using **machine-check** for verification of machine-code programs on actual processors, I wrote a **machine-check** description of the 8-bit microcontroller 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[5]. I used 16 programs in total. Out of these, 11 were introduced in my diploma thesis [A.4]: 6 were single-instruction programs where the inherent property should be immediately violated and 5 were more interesting toy programs. In addition, I introduced a new Voltage-Controlled Oscillator calibration program, a simplified version of a real-world program with 4 variations, and a factorial-computation program, the verification times of both of which approach the limits of what I consider acceptable for formal verification during development (verification under a minute for each property). Notably, I was able to find a bug in the calibration program, determining it was also present in the original real-world version.

7.4.1 Description Details

The system description of ATmega328P 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 number of constructs that can be translated. That said, the AVR instruction set itself contains around a hundred instructions [21], 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 theoretically problematic for the Rust compiler, but can pose usability concerns, especially for the

[5]The programs, the evaluation script containing the properties, and the reference measurements are available in an artefact located at `https://doi.org/10.5281/zenodo.15109092`.

development of the processor description themselves. The full compilation of **machine-check-avr** takes over 1 minute on the used personal computer, slowing the rapid development process. Furthermore, in the version 0.3.2029 of 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[6]. This means that a manual compilation was necessary after modifying the description instead of background compilation as the code is written, which reduced the ease of development. This problem disappeared after updating to version 0.3.2319 of **rust-analyzer**, but shows the potential issues with more complicated and better-described architectures. These issues could be mitigated by extending the expressiveness of the supported Rust subset (leading to more compact descriptions) and improving the performance of the translation and refinement analogues.

7.4.2 Evaluation Setup

The evaluation of **machine-check-avr** was performed on a personal computer 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 1.83.0. Building from a clean slate, the **machine-check-avr** executable was built in 2 minutes and 12 seconds in the release mode. The built executable can verify properties of ATmega328P machine-code programs. For evaluation, the name of the machine-code Intel HEX file and the property to be verified are supplied on the command line, with the default strategy of input splitting with no decay. The inherent property was verified separately from others. When verifying the other properties, it was assumed that the inherent property holds.

7.4.3 Inherent-Violation Programs

From my diploma thesis, I took 6 simple single-instruction programs that implement some behaviours that should result in **machine-check-avr** determining that the inherent property does not hold:

- Getting a value from a location that is not described.
- Setting a value from a location that is not described.
- Processing an instruction that is not described.
- Jumping outside the loaded program instructions.
- Setting a value to a bit restricted for writing. AVR architecture typically has unused bits of registers in the memory map restricted in a way that a logic 1 should not be written to it. This restriction is guaranteed in the processor description, raising an error when this occurs.

[6] An issue with the same error is described in `https://github.com/rust-lang/rust-analyzer/issues/10855`.

Table 7.1: Measurements of machine-code verification of inherent-violation 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	Prop. name	Result	Refin.	States	Transitions	CPU time [s]	Mem. [MB]
Get from undescribed	Inherent	✗	0	4 / 3	4 / 4	<0.01	3.46
Set to undescribed	Inherent	✗	0	3 / 2	3 / 3	<0.01	3.46
Undescribed instruction	Inherent	✗	0	3 / 2	3 / 3	<0.01	3.58
Jump outside	Inherent	✗	0	16386 / 16385	16386 / 16386	0.04	36.33
Set to restricted	Inherent	✗	0	5 / 4	5 / 5	<0.01	3.46
Set global interrupt flag	Inherent	✗	0	4 / 3	4 / 4	<0.01	3.58

- Enabling the global interrupt flag, as the processor behaviour while interrupts are enabled is currently not described.

The results of measurements are visualised in Table 7.1. As expected, all of the programs have been shown not to uphold the inherent property. The only program deserving its own mention is the jump outside the loaded program instructions: while the jump outside the loaded program instructions immediately results in generation of a state in which it is asserted that the panic occurred, the current version of **machine-check** does not immediately short-circuit the verification of the inherent-violation property, constructing the whole space as the Program Counter still changes. The number of states and transitions follows the 32 kB size of the ATmega328P instruction memory: as the Program Counter indexes 16-bit instructions, it is 14 bits wide, with $2^{14} = 16384$.

Note 7.4.1. While interrupts are not currently supported in **machine-check-avr**, in one of the previous versions, an inherent panic was not asserted upon enabling interrupts by setting the Global Interrupt Flag, resulting in the verification of the inherent property in the corresponding program to return that the property holds. This illustrates how an incorrectly written processor description can result in verification results not corresponding to the underlying device, even if the verification tool is sound. This is a great motivation towards official formal descriptions of the processor behaviour (which are unfortunately currently not available for the AVR architecture).

Note 7.4.2. In my diploma thesis, I additionally had a program that would set a register to an initially undefined memory value. This resulted in a violation in the deadline checker from the diploma thesis, but I opted to treat the initial memory values as inputs in the new **machine-check-avr** instead. This is a system description decision based on the possibility of programs being compiled to e.g. perform a read of uninitialised memory for some reason but do not use the result further. As such, I have not included the program in this set.

7.4.4 Toy Programs

In my diploma thesis, 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**:

- **Initialisation 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.

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

Table 7.2: 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]	Mem. [MB]
Basic branch (debug)	Inherent	✓	2	26 / 19	28 / 22	<0.01	6.02
Basic branch (debug)	Initialisation	✓	0	14 / 13	14 / 14	<0.01	3.97
Basic branch (debug)	Invariant lock	✓	2	26 / 19	28 / 22	<0.01	5.89
Basic branch (debug)	Recovery	✓	2	26 / 19	28 / 22	<0.01	6.02
Basic branch (release)	Inherent	✓	2	26 / 19	28 / 22	<0.01	5.89
Basic branch (release)	Initialisation	✓	0	14 / 13	14 / 14	<0.01	3.84
Basic branch (release)	Invariant lock	✓	2	26 / 19	28 / 22	<0.01	6.02
Basic branch (release)	Recovery	✓	2	26 / 19	28 / 22	<0.01	6.02
Blink	Inherent	✓	0	514 / 513	514 / 514	<0.01	4.61
Blink	Initialisation	✓	0	514 / 513	514 / 514	<0.01	4.99
Blink	Invariant lock	✓	0	514 / 513	514 / 514	<0.01	4.74
Blink	Recovery	✓	0	514 / 513	514 / 514	<0.01	4.74
Gate array	Inherent	✓	987	5784 / 3799	6771 / 4787	3.74	17.02
Gate array	Initialisation	✓	0	9 / 8	9 / 9	<0.01	3.97
Gate array	Invariant lock	✓	987	5784 / 3799	6771 / 4787	4.32	17.02
Gate array	Recovery	✓	987	5785 / 3799	6772 / 4787	4.34	16.51
Independent nondet.	Inherent	✓	384	1613 / 834	1997 / 1219	0.8	9.22
Independent nondet.	Initialisation	✓	0	6 / 5	6 / 6	<0.01	3.97
Independent nondet.	Invariant lock	✓	384	1613 / 834	1997 / 1219	0.78	9.09
Independent nondet.	Recovery	✗	193	819 / 423	1012 / 617	0.33	7.68
Momentary selection	Inherent	✓	29	1909 / 1842	1938 / 1872	0.56	10.11
Momentary selection	Initialisation	✓	0	8 / 7	8 / 8	<0.01	3.84
Momentary selection	Invariant lock	✓	29	1909 / 1842	1938 / 1872	0.5	10.37
Momentary selection	Recovery	✓	29	1909 / 1842	1938 / 1872	0.45	10.11

The results for the toy programs are shown in Table 7.2. 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.

7.4.5 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 7.5. The output overflows above $5! = 120$, but the focus here is on recursion behaviour rather than

```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 7.5: The factorial program, written in C.

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 a 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 [21]. As such, it is possible to verify that the stack overflow does not occur by ensuring $\text{SP} \geq v$, 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, $\text{SP} \geq v$ can be rewritten to $\text{SPH} > v_{\text{high}} \vee (\text{SPH} = v_{\text{high}} \wedge \text{SPL} \geq v_{\text{low}})$. 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 7.3. 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, initialisation, 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 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

Table 7.3: 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	68604 / 51595	70716 / 52940	24.37	233.28
Debug	Initialisation	✓	0	21 / 20	21 / 21	<0.01	3.84
Debug	Invariant lock	✓	576	68604 / 51595	70716 / 52940	35.05	238.51
Debug	Recovery	✗	576	68604 / 51595	70716 / 52940	35.29	234.48
Debug	Stack above 0x08DD	✓	576	68604 / 51595	70716 / 52940	32.86	237.08
Debug	Stack above 0x08DE	✗	3	844 / 748	855 / 756	0.01	8.58
Release	Inherent	✓	45	5917 / 4272	6082 / 4378	0.22	18.82
Release	Initialisation	✓	0	21 / 20	21 / 21	<0.01	3.84
Release	Invariant lock	✓	45	5917 / 4272	6082 / 4378	0.24	19.20
Release	Recovery	✗	45	5917 / 4272	6082 / 4378	0.27	18.94
Release	Stack above 0x08FB	✓	45	5917 / 4272	6082 / 4378	0.25	18.94
Release	Stack above 0x08FC	✗	0	21 / 20	21 / 21	<0.01	3.71

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.

7.4.6 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 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 7.6.

```
#define F_CPU 1000000
#include <avr/io.h>
#include <util/delay.h>
// COMPLICATION: global variables that will store the reads
// volatile uint8_t irrelevant[8];
// volatile uint8_t pinb;
// volatile uint8_t pinc;

int main(void) {
    // COMPLICATION: read to global variables
    // for (uint8_t i = 0; i < 8; ++i) { irrelevant[i] = PINB; }
    DDRC |= 0x01;
    DDRD |= 0xFF;
    while (1) {
        // wait until we should calibrate
        while ((PINC & 0x2) == 0) {}
        // COMPLICATION: replace the line above with
        // while 1 {
        //     pinc = PINC;
        //     if ((pinc & 0x2) != 0) { break; }
        // }

        // 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;
            // COMPLICATION: replace the line above with
            // pinb = PINB; 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
                // FIX: add the following line
                // PORTD = search_val;
                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 7.6: The source code of the calibration program (with the replaced peripheral interactions) which was compiled to produce the machine code under verification. The fixed versions have the line below `FIX` uncommented, and the versions with complications are changed as per the `COMPLICATION` comments.

Table 7.4: Measurements of machine-code verification of the calibration program using **machine-check-avr**, without complications.

Version	Target	Property name	Result	Refin.	States	Transitions	CPU time [s]	Mem. [MB]
Original	Debug	Inherent	✓	513	14090 / 13059	14603 / 13573	6.65	42.50
Original	Debug	Initialisation	✓	0	18 / 17	18 / 18	<0.01	3.84
Original	Debug	Invariant lock	✓	513	14090 / 13059	14603 / 13573	8.86	43.65
Original	Debug	Recovery	✗	513	14090 / 13059	14603 / 13573	8.04	42.62
Original	Debug	Stack above 0x08FD	✓	513	14090 / 13059	14603 / 13573	7.68	43.22
Original	Debug	Stack above 0x08FE	✗	0	18 / 17	18 / 18	<0.01	3.71
Original	Release	Inherent	✓	512	11767 / 10738	12279 / 11251	7.17	36.35
Original	Release	Initialisation	✓	0	16 / 15	16 / 16	<0.01	3.84
Original	Release	Invariant lock	✓	512	11767 / 10738	12279 / 11251	9.48	37.50
Original	Release	Recovery	✗	512	11767 / 10738	12279 / 11251	8.58	36.48
Original	Release	Stack above 0x08FD	✓	512	11767 / 10738	12279 / 11251	8.5	37.12
Original	Release	Stack above 0x08FE	✗	0	16 / 15	16 / 16	<0.01	3.97
Fixed	Debug	Inherent	✓	513	14090 / 13059	14603 / 13573	6.34	42.36
Fixed	Debug	Initialisation	✓	0	18 / 17	18 / 18	<0.01	3.97
Fixed	Debug	Invariant lock	✓	513	14090 / 13059	14603 / 13573	8.78	43.63
Fixed	Debug	Recovery	✓	513	14090 / 13059	14603 / 13573	7.32	42.75
Fixed	Debug	Stack above 0x08FD	✓	513	14090 / 13059	14603 / 13573	7.33	43.26
Fixed	Debug	Stack above 0x08FE	✗	0	18 / 17	18 / 18	<0.01	3.84
Fixed	Release	Inherent	✓	512	13040 / 12011	13552 / 12524	7.13	39.81
Fixed	Release	Initialisation	✓	0	16 / 15	16 / 16	<0.01	3.97
Fixed	Release	Invariant lock	✓	512	13040 / 12011	13552 / 12524	7.3	40.96
Fixed	Release	Recovery	✓	512	13040 / 12011	13552 / 12524	6.94	40.19
Fixed	Release	Stack above 0x08FD	✓	512	13040 / 12011	13552 / 12524	6.8	40.58
Fixed	Release	Stack above 0x08FE	✗	0	16 / 15	16 / 16	<0.01	3.97

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 Table 7.4 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.

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. Notably, this is exactly the same failure mode as in Example 5.1.1, but has less severe consequences as the output is only slightly degraded. However, the degradation is more difficult to detect and means that the system always performs below its capabilities (as if we used a 7-bit rather than an 8-bit digital potentiometer). I do not believe that the problem would be detected without formal verification or an actual problem caused by the degradation.

Table 7.5: Measurements of machine-code verification of the calibration program **with complications introduced**, using **machine-check-avr**.

Version	Target	Property name	Result	Refin.	States	Transitions	CPU time [s]	Mem. [MB]
Original	Debug	Inherent	✓	771	20674 / 17330	21445 / 18102	19.06	72.52
Original	Debug	Initialisation	✓	0	176 / 175	176 / 176	<0.01	4.22
Original	Debug	Invariant lock	✓	771	20674 / 17330	21445 / 18102	30.34	74.13
Original	Debug	Recovery	✗	770	20671 / 17333	21441 / 18104	29.6	72.69
Original	Debug	Stack above 0x08FD	✓	771	20674 / 17330	21445 / 18102	30.09	73.74
Original	Debug	Stack above 0x08FE	✗	0	176 / 175	176 / 176	<0.01	4.48
Original	Release	Inherent	✓	771	18865 / 15780	19636 / 16552	19.55	67.72
Original	Release	Initialisation	✓	0	172 / 171	172 / 172	<0.01	4.48
Original	Release	Invariant lock	✓	771	18865 / 15780	19636 / 16552	24	69.53
Original	Release	Recovery	✗	770	18862 / 15783	19632 / 16554	24.07	68.24
Original	Release	Stack above 0x08FD	✓	771	18865 / 15780	19636 / 16552	22.22	68.92
Original	Release	Stack above 0x08FE	✗	0	172 / 171	172 / 172	<0.01	4.35
Fixed	Debug	Inherent	✓	771	20674 / 17330	21445 / 18102	25.76	72.53
Fixed	Debug	Initialisation	✓	0	176 / 175	176 / 176	<0.01	4.48
Fixed	Debug	Invariant lock	✓	771	20674 / 17330	21445 / 18102	31.72	74.24
Fixed	Debug	Recovery	✓	771	20674 / 17330	21445 / 18102	27.53	73.27
Fixed	Debug	Stack above 0x08FD	✓	771	20674 / 17330	21445 / 18102	28.17	73.51
Fixed	Debug	Stack above 0x08FE	✗	0	176 / 175	176 / 176	<0.01	4.48
Fixed	Release	Inherent	✓	771	19121 / 16036	19892 / 16808	20.06	68.67
Fixed	Release	Initialisation	✓	0	172 / 171	172 / 172	<0.01	4.35
Fixed	Release	Invariant lock	✓	771	19121 / 16036	19892 / 16808	22.18	70.24
Fixed	Release	Recovery	✓	771	19121 / 16036	19892 / 16808	16.83	69.00
Fixed	Release	Stack above 0x08FD	✓	771	19121 / 16036	19892 / 16808	16.25	69.84
Fixed	Release	Stack above 0x08FE	✗	0	172 / 171	172 / 172	<0.01	4.48

I fixed the problem by writing `search_val` to `PORTD` before breaking from the loop, as shown in Figure 7.6. The results are shown in the columns in Table 7.4 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[7] 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 parametric, so it could be verified that all output values from 0 to 255 are used.

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

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

the bug, e.g. **F**(`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 **AG**[**EF**[`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.

Introducing complications. To test the robustness of verification using **machine-check**, I also introduced two versions with complications that would make naïve verification without abstraction completely infeasible, the modifications shown in Figure 7.6: there is an an initial 64-bit read, and explicit read of an 8-bit input port to a variable before obtaining the single bit that determines whether the calibration should start. The variables are declared as `volatile` to ensure they are not optimised away, resulting in are in more than 2^{80} reachable states even if not considering the uninitialised memory values. As seen in Table 7.5, this results in the verification time rising to a factor of at most 4.1 and the memory usage rising to a factor of at most 1.9. I consider this to be a success as it shows the tool is behaving reasonably well in the face of irrelevant modifications even with the simple abstraction domain and system-independent heuristics.

7.5 Assessment of Capabilities and Comparison to Other Tools

The tool I created during my doctoral studies, **machine-check**, is capable of verifying CTL properties of machine-code systems, as evidenced by the results presented in Section 7.4. 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 generally matched the 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.

While the results cannot be directly compared to those of any other tool identified in Section 3.1 due to the differences in capabilities, system descriptions, specifications, public availability, I will compare **machine-check** to the other tools qualitatively, focusing on the comparison of their capabilities.

7.5.1 Model-Checking Direction

The model-checking direction is exemplified by the **Arcade.µC** tool, which is unfortunately no longer in development and is not publicly available, and which directly inspired the predecessor of **machine-check**. Both are targeted at verification of embedded systems, especially on 8-bit AVR microcontrollers. The techniques I introduced in Chapter 4, 5, and 6 directly improve on the capabilities of **Arcade.µC**:

- Thanks to the description capability introduced in Chapter 4, instead of hard-coded processor simulators or state-space generators that considered the processor intricacies [7], systems described by arbitrary finite-state machines can be verified.
- Due to the TVAR framework from Chapter 5, **machine-check** is theoretically capable of verifying CTL properties of **any** such system in finite time (of course, practically limited by available time and memory). On the other hand, **Arcade.µC** was limited to ACTL properties if abstraction was used, and the abstraction could be too coarse, preventing verification [4].
- In the three-valued abstraction currently used in **machine-check**, not just bitwise, but also arithmetic operations can be computed quickly thanks to the techniques introduced in Chapter 6.

While there are results of benchmarking versions of **Arcade.µC**, I do not feel it is possible to directly compare the given results due to the differences in capabilities. For example, in the ATmega16 case studies by Gückel [7, p. 145-149], it can be seen that the choice of the abstraction heavily impacts the verification time and memory, and it is unclear which one should be chosen for comparison. In the absence of having the tool available, it is unclear how much the abstraction strategies would impact different programs under verification, such as the ones in Section 7.4. As such, I refrained from making direct comparisons.

7.5.2 Program-Analysis Direction

In the program-analysis direction, the static analysis is exemplified by **CodeSurfer/x86** [53, 54, 55] and verification is exemplified by the tool **MCVETO** [72, 69]. Both are specialised towards x86 user-mode executables, posing different problems to embedded-system verification: the 32-bit register width makes explicit verification without abstraction much less feasible. Only safety properties are considered. While **CodeSurfer/x86** can determine some properties using the value-set analysis even for programs with 100 thousand instructions [53, p. 15], it is questionable how this would be comparable, as the value-set analysis can be highly overapproximative. This should be resolved by **MCVETO**, but the practical usability is questionable without quantitative data given. In any case, the tools seem quite dependent on the conventional Application Binary Interface (ABI) usage so that static analysis can be performed, which is not a problem for **machine-check**.

7.5.3 Automated-Theorem-Proving Direction

The tools based translation to predicate calculus formulas and solving using Automated Theorem Provers are strong potential contenders to **machine-check**. I especially consider **Islaris** [1] and **Serval** [2] to be of interest, and will consider their capabilities first:

- **Islaris** and **Serval** support the Sail language for the Instruction Set Architecture (ISA) description, intended more towards usage for a general computer rather than embedded programs due to lack of peripheral interactions in the used descriptions.

- **Islaris** supports program loops via loop invariants (that may be discovered automatically or require hints), while **Serval** does not support loops, precluding use for whole systems rather than programs, but in theory allowing completely automatic verification of programs that do not contain loops.
- Both **Islaris** and **Serval** currently only support safety properties.

The use of Hoare logic and Automated Theorem Proving mean that the tools are much better suited for verification of higher-level properties such as the function `memcpy` behaving according to its contract using **Islaris** [56], which requires automated reasoning with some aid from the tool user. The weakness is that the user must have specialist knowledge of ATP to guide the proof. Even **Serval** is affected, requiring the specifications in the purely functional language Rosetta, which would be unfamiliar to non-specialists. In **machine-check**, only some general knowledge of CTL is needed, and the supported fairly imperative subset of the Rust language should be more understandable by typical developers.

In my opinion, the addition of TVAR and the focus on generality draws **machine-check** a bit closer to theoretical proving than the previous model-checkers. The choices of refinement must be done carefully, resembling the choice of how to continue with a proof. I reckon that further development of **machine-check** will result in more focus on the choices of abstraction and refinement, with the great potential for inspiration by the more proof-based techniques. In time, the directions may complement each other, model-checking tools supporting development of both non-critical and critical programs by performing fully automatic verification where it is possible in reasonable time, with ATP-based tools usable for critical programs and properties that the model-checking tools could not feasibly verify using the implemented techniques.

7.5.4 General Assessment

Overall, I consider the goal of the research presented in this thesis, given in Chapter 1 as constructing a solid yet flexible theoretical groundwork, fulfilled. The combination of the translation of simulable descriptions presented in Chapter 4 and discussed more extensively in this chapter, the input-based TVAR framework presented in Chapter 5, and the bit-vector abstraction with fast abstract arithmetic presented in Chapter 6, has resulted in a fairly well-behaved and extensible tool. For real-world use, I believe furtherr improvements are necessary:

- Custom-written descriptions can be prone to errors, as illustrated in Section 7.4.3. I feel that the move toward first-party formal descriptions of ISAs and, hopefully, whole systems including peripherals, is desirable. As such, in addition to the currently supported custom-written ATmega328P description, I plan to add translation of Sail descriptions to the descriptions of finite-state machines supported by **machine-check**. This is completely possible in theory, although the 32-bit architectures can significantly deteriorate the feasibility of machine-code program verification, which was my reason for starting with an 8-bit architecture in the first place.

- While the time and memory necessary for verification were reasonable for the machine-code programs presented, they would not be for more complex ones. The abstraction domains and refinement strategies could 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 also be extended with verification hints (e.g. that the Program Counter is the most important variable) if the general approach is insufficient.
- Working with **machine-check** could be made more user-friendly. While I did not discuss the user experience in detail in this thesis, only lightly touching on it in Chapter 4, user-friendliness is key to the adoption of the tool by others. I have begun the work on this, creating a Graphical User Interface in addition to the command-line experience, and setting up a website for the project, but further work is necessary.

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 labour before machine-code verification based on model checking with abstraction is successfully and consistently used to improve practical systems.

Chapter 8

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 introduced three novel techniques to resolve known challenges, published two of them, and implemented them in my free and open-source formal verification tool **machine-check**. The introduced techniques and their implementations form a solid foundation of machine-code verification through abstraction-based model checking.

8.1 Summary

In Chapter 1, I introduced the concept of and the 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, discussed the grouping of advanced techniques, and introduced the concept of abstraction refinement.

In Chapter 3, I discussed the state of the art in verification of digital systems and noted that abstraction refinement is commonly used in the best software- and hardware-verification tools to mitigate the problem of exponential explosion. I noted that verification of machine-code systems is under-researched in the context of model checking, and that a major problem specific to machine-code verification has been the difficulty of supporting easily written processor descriptions while ensuring that advanced model-checking techniques such as abstraction refinement can be used.

In Chapter 4, I presented my technique that solves the difficulty. The descriptions 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 5, I discussed a novel Three-Valued Abstraction Refinement (TVAR) framework that I introduced together with my supervisor, which can 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. We formally proved that the framework produces correct results in finite time for finite systems as long as requirements are met. We experimentally evaluated an implementation of an instance of the introduced framework, showing its potential to reduce exponential explosion on simple digital systems.

In Chapter 6, 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 7, I discussed my publicly available, free, and open-source tool **machine-check** that I created during my doctoral research, implementing the techniques from Chapters 4, 5, and 6. 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. I discussed the capabilities of **machine-check** compared to other machine-code verification tools.

8.2 Contributions of the Dissertation Thesis

In my dissertation thesis, the results of my doctoral research are described and brought into a general context. In Chapters 2 and 3, 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.

8.3 Future Work

In this thesis, I have laid the groundwork for verification of machine code using model checking with abstraction refinement, implementing the techniques introduced in my tool **machine-check**. I aim to continue in this work further. While **machine-check** is currently able to verify simple digital systems, including some machine-code systems, it is still necessary to improve the practical usability. Based on the research and experiments presented in this thesis, I have identified the following major areas of further interest:

- **Abstraction and refinement choices and strategies.** As seen in Section 5.4 and Subsection 7.1.2, the choices of good abstraction, initial generating automaton, and

the refinement heuristics are crucial for good practical performance of model checking with abstraction refinement. More powerful abstraction domains might help where the three-valued bit-vector abstraction domain does not help with mitigating exponential explosion. I also anticipate that abstract-state decay introduced in Chapter 5 offers much yet-untapped potential for a reduction of abstract state space sizes.

- **Support of official formal processor descriptions.** I believe that the recent work on formally specifying processor Instruction Set Architecture (ISA) noted in Section 3.1 is of major interest, especially the official RISC-V ISA specification in the Sail language [90]. I plan to investigate the automatic translation of the specification to the subset of the Rust language supported by **machine-check**.
- **Parametric systems.** As noted in Subsection 2.1.4, some systems are given with incomplete guarantees, so we are in fact trying to prove or disprove a property for a class of systems rather than a single system. While I have partially alleviated the problem in **machine-check** by disallowing some operations using inherent properties, I believe that it is important to be able to generally verify in the context of incomplete guarantees, where the systems can be described as parametric. This would be especially convenient in the context of processor peripherals, where they could be described partially without precluding verification unless the property under verification depended on the exact behaviour not described.

Building on the groundwork laid down in this thesis allows for further formal-verification research backed with a practical component through extending **machine-check**. There are also more practical problems to solve, 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] Rigorous Engineering of Mainstream Systems Project. Islaris: verification of machine code against authoritative ISA semantics. URL `https://github.com/rems-project/islaris`. Retrieved 2025-02-23.

[2] The UNSAT group. Serval. URL `https://unsat.cs.washington.edu/projects/serval/`. Retrieved 2025-02-23.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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`.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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

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

[20] *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.

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

[22] 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:10.1007/978-3-319-10575-8_3.

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

[24] E. M. Clarke. The birth of model checking. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking: History, Achievements, Perspectives*, pages 1–26, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-69850-0. doi:10.1007/978-3-540-69850-0_1.

[25] P. Cousot. A personal historical perspective on abstract interpretation. In B. Meyer, editor, *The French School of Programming*, pages 205–239, Cham, 2024. Springer International Publishing. ISBN 978-3-031-34518-0. doi:10.1007/978-3-031-34518-0_9.

[26] R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12 (10):576–580, October 1969. ISSN 0001-0782. doi:10.1145/363235.363259.

[28] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, October 1972. ISSN 0001-0782. doi:10.1145/355604.361591.

[29] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. doi:10.1145/360933.360975.

[30] M. Gordon and H. Collavizza. Forward with hoare. In A. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, pages 101–121, London, 2010. Springer London. ISBN 978-1-84882-912-1. doi:10.1007/978-1-84882-912-1_5.

[31] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. doi:10.1145/360248.360252.

[32] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373500. doi:10.1145/512950.512973.

[33] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977. doi:10.1109/SFCS.1977.32.

[34] 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.

[35] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986. ISSN 0164-0925. doi:10.1145/5397.5399.

[36] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, page 38–48, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919793. doi:10.1145/268946.268950.

[37] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), October 2009. ISSN 0360-0300. doi:10.1145/1592434.1592438.

[38] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Trans. Program. Lang. Syst.*, 11(1):147–167, January 1989. ISSN 0164-0925. doi:10.1145/59287.62028.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In O. Grumberg, editor, *Computer Aided Verification*, pages 72–83, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69195-2. doi:10.1007/3-540-63166-6_10.

[51] H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification*, pages 443–454, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48683-1. doi:10.1007/3-540-48683-6_38.

[52] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. *SIGPLAN Not.*, 37(5):57–68, May 2002. ISSN 0362-1340. doi:10.1145/543552.512538.

[53] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In E. Duesterwald, editor, *Compiler Construction*, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24723-4. doi:10.1007/978-3-540-24723-4_2.

[54] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In R. Bodik, editor, *Compiler Construction*, pages 250–254, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31985-6. doi:10.1007/978-3-540-31985-6_19.

[55] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 202–213, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-69149-5. doi:10.1007/978-3-540-69149-5_22.

[56] M. Sammler, A. Hammond, R. Lepigre, B. Campbell, J. Pichon-Pharabod, D. Dreyer, D. Garg, and P. Sewell. Islaris: verification of machine code against authoritative ISA semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 825–840, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi:10.1145/3519939.3523434.

[57] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 225–242, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi:10.1145/3341301.3359641.

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

[63] 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.

[64] Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 70–82, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581131992. doi:10.1145/349299.349313.

[65] Z. Xu, T. Reps, and B. P. Miller. Typestate checking of machine code. In D. Sands, editor, *Programming Languages and Systems*, pages 335–351, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45309-3. doi:10.1007/3-540-45309-1_22.

[66] G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C. H. Chen, and T. Teitelbaum. Model checking x86 executables with codesurfer/x86 and wpds++. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification*, pages 158–163, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31686-2. doi:10.1007/11513988_17.

[67] G. Balakrishnan and T. Reps. Analyzing stripped device-driver executables. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–140, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. doi:10.1007/978-3-540-78800-3_10.

[68] G. Balakrishnan and T. Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6), August 2010. ISSN 0164-0925. doi:10.1145/1749608.1749612.

[69] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed proof generation for machine code. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, pages 288–305, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14295-6. doi:10.1007/978-3-642-14295-6_27.

[70] J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In L. Hendren, editor, *Compiler Construction*, pages 36–52, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78791-4. doi:10.1007/978-3-540-78791-4_3.

[71] J. Lim and T. Reps. Tsl: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Trans. Program. Lang. Syst.*, 35(1), April 2013. ISSN 0164-0925. doi:10.1145/2450136.2450139.

[72] T. Reps, J. Lim, A. Thakur, G. Balakrishnan, and A. Lal. There's plenty of room at the bottom: Analyzing and verifying machine code. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, pages 41–56, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14295-6. doi:10.1007/978-3-642-14295-6_6.

[73] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, page 117–127, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934685. doi:10.1145/1181775.1181790.

[74] V. Srinivasan and T. Reps. Synthesis of machine code from semantics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 596–607, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi:10.1145/2737924.2737960.

[75] V. Srinivasan and T. Reps. An improved algorithm for slicing machine code. *SIGPLAN Not.*, 51(10):378–393, October 2016. ISSN 0362-1340. doi:10.1145/3022671.2984003.

[76] V. Srinivasan, T. Sharma, and T. Reps. Speeding up machine-code synthesis. *SIGPLAN Not.*, 51(10):165–180, October 2016. ISSN 0362-1340. doi:10.1145/3022671.2984006.

[77] V. Srinivasan, A. Vartanian, and T. Reps. Model-assisted machine-code synthesis. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133885.

[78] M. J. Sammler. *Automated and foundational verification of low-level programs*. PhD thesis, Saarland University, 2023. URL `https://plv.mpi-sws.org/sammler-thesis/`.

[79] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, January 1996. ISSN 0004-5411. doi:10.1145/227595.227603.

[80] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *Int. J. Parallel Program.*, 34(1):61–91, February 2006. ISSN 0885-7458. doi:10.1007/s10766-005-0004-8.

[81] M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 568–582, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-71209-1. doi:10.1007/978-3-540-71209-1_44.

[82] M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In *2008 Formal Methods in Computer-Aided Design*, pages 1–8, 2008. doi:10.1109/FMCAD.2008.ECP.24.

[83] A. Fox. Directions in ISA specification. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving*, pages 338–344, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32347-8. doi:10.1007/978-3-642-32347-8_23.

[84] A. Fox. Improved tool support for machine-code decompilation in HOL4. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving*, pages 187–202, Cham, 2015. Springer International Publishing. ISBN 978-3-319-22102-1. doi:10.1007/978-3-319-22102-1_12.

[85] A. Fox, M. O. Myreen, Y. K. Tan, and R. Kumar. Verified compilation of CakeML to multiple machine-code targets. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, page 125–137, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450347051. doi:10.1145/3018610.3018621.

[86] S. Goel, W. A. Hunt, and M. Kaufmann. Engineering a formal, executable x86 isa simulator for software verification. In M. Hinchey, J. P. Bowen, and E.-R. Olderog, editors, *Provably Correct Systems*, pages 173–209, Cham, 2017. Springer International Publishing. ISBN 978-3-319-48628-4. doi:10.1007/978-3-319-48628-4_8.

[87] K. E. Gray, G. Kerneis, D. Mulligan, C. Pulte, S. Sarkar, and P. Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 635–646, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340342. doi:10.1145/2830772.2830775.

[88] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. *SIGPLAN Not.*, 51(1):608–621, January 2016. ISSN 0362-1340. doi:10.1145/2914770.2837615.

[89] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, volume 3 of *POPL*, New York, NY, USA, January 2019. Association for Computing Machinery. doi:10.1145/3290384.

[90] RISC-V Archive. ISA formal spec public review. `https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review`.

[91] A. Armstrong, B. Campbell, B. Simner, C. Pulte, and P. Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In A. Silva and K. R. M. Leino, editors, *Computer Aided Verification*, pages 303–316, Cham, 2021. Springer International Publishing. ISBN 978-3-030-81685-8. doi:10.1007/978-3-030-81685-8_14.

[92] Iris Project. Iris. URL `https://iris-project.org/`. Retrieved 2025-02-28.

[93] G. L. Steven Keuchel and D. Devriese. Katamaran: semi-automated verification of ISA specifications. In *REMS-DeepSpec workshop*, 2020. URL `https://katamaran-project.github.io/articles/2020-06-remsdeepspec-katamaran.pdf`. Extended abstract.

[94] Katamaran Project. Katamaran. URL `https://github.com/katamaran-project/katamaran`. Retrieved 2025-02-28.

[95] 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.

[96] 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.

[97] 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.

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

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

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

[101] 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.

[102] 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.

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

[104] 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.

[105] 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.

[106] 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.

[107] A. Biere, N. Froleyks, and M. Preiner. Hardware Model Checking Competition 2024, HWMCC 2024. URL `https://hwmcc.github.io/2024/`. Accessed on 2025-03-03.

[108] 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.

[109] 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.

[110] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design*, pages 127–144, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-40922-9. doi:10.1007/3-540-40922-X_8.

[111] 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.

[112] 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.

[113] 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.

[114] 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.

[115] 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.

[116] 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.

[117] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In K. G. Larsen and M. Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory, CONCUR 2001*, pages 426–440, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44685-9. doi:10.1007/3-540-44685-0_29.

[118] 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.

[119] 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.

[120] A. Gurfinkel and M. Chechik. Why waste a perfectly good abstraction? In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 212–226, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33057-8. doi:10.1007/11691372_14.

[121] O. Wei, A. Gurfinkel, and M. Chechik. On the consistency, expressiveness, and precision of partial modeling formalisms. *Information and Computation*, 209(1):20–47, 2011. ISSN 0890-5401. doi:10.1016/j.ic.2010.08.001.

[122] A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A software model-checker for verification and refutation. In T. Ball and R. B. Jones, editors, *Computer Aided Verification*, pages 170–174, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37411-4. doi:10.1007/11817963_18.

[123] R. E. Bryant. A methodology for hardware verification based on logic simulation. *J. ACM*, 38(2):299–328, April 1991. ISSN 0004-5411. doi:10.1145/103516.103519.

[124] R. E. Bryant and C.-J. H. Seger. Formal verification of digital circuits using symbolic ternary system models. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification*, pages 33–43, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-38394-9. doi:10.1007/BFb0023717.

[125] R. Bryant. Formal verification of memory circuits by switch-level simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1): 94–102, 1991. doi:10.1109/43.62795.

[126] R. Bryant, C.-J. Seger, and D. Beatty. Formal hardware verification by symbolic ternary trajectory evaluation. In *28th ACM/IEEE Design Automation Conference*, pages 397–402, 1991. doi:10.1145/127601.127701.

[127] 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.

[128] 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.

[129] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Interval analysis and machine arithmetic: Why signedness ignorance is bliss. *ACM Trans. Program. Lang. Syst.*, 37(1), January 2015. ISSN 0164-0925. doi:10.1145/2651360.

[130] 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.

[131] G. Bruns and P. Godefroid. Temporal logic query checking. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, pages 409–417. IEEE Computer Society, 2001. doi:10.1109/LICS.2001.932516.

[132] 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.

[133] 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.

[134] 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.

[135] 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.

[136] 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.

[137] A. Gurfinkel, O. Wei, and M. Chechik. Model checking recursive programs with exact predicate abstraction. In S. S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, editors, *Automated Technology for Verification and Analysis*, pages 95–110, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-88387-6. doi:10.1007/978-3-540-88387-6_9.

[138] A. Gurfinkel. Yasm: Software model-checker. URL `https://www.cs.toronto.edu/~arie/yasm/`. Retrieved on 2025-01-17. The 2008 version was used for evaluation.

[139] B. Konikowska and W. Penczek. Reducing model checking from multi-valued ctl* to ctl*. In L. Brim, M. Křetínský, A. Kučera, and P. Jančar, editors, *CONCUR 2002 — Concurrency Theory*, pages 226–239, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45694-0. doi:10.1007/3-540-45694-5_16.

[140] A. Gurfinkel and M. Chechik. Multi-valued model checking via classical model checking. In R. Amadio and D. Lugiez, editors, *CONCUR 2003 - Concurrency Theory*, pages 266–280, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45187-7. doi:10.1007/978-3-540-45187-7_18.

[141] M. Gazda and T. A. Willemse. Expressiveness and completeness in abstraction. *Electronic Proceedings in Theoretical Computer Science*, 89:49–64, August 2012. ISSN 2075-2180. doi:10.4204/eptcs.89.5.

[142] T. Melham. Symbolic trajectory evaluation. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 831–870. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8_25.

[143] J. Yang and C.-J. Seger. Introduction to generalized symbolic trajectory evaluation. In *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*, pages 360–365, 2001. doi:10.1109/ICCD.2001.955052.

[144] M. Dam. Fixed points of Büchi automata. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 39–50, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-47507-1. doi:10.1007/3-540-56287-7_93.

[145] R. Tzoref and O. Grumberg. Automatic refinement and vacuity detection for symbolic trajectory evaluation. In T. Ball and R. B. Jones, editors, *Computer Aided Verification*, pages 190–204, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37411-4. doi:10.1007/11817963_20.

[146] Y. Chen, Y. He, F. Xie, and J. Yang. Automatic abstraction refinement for generalized symbolic trajectory evaluation. In *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*, pages 111–118. IEEE Computer Society, 2007. doi:10.1109/FAMCAD.2007.11.

[147] 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.

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

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

[150] 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.

[151] 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.

[152] 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.

[153] 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.

[154] 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.

[155] 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.

[156] 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.

[157] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999. ISBN 978-3-540-65410-0. doi:10.1007/978-3-662-03811-6.

[158] 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.

[159] 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.

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. In: Giacobazzi, R., Gorla, A. (eds) *Proceedings of the 31st International Static Analysis Symposium, SAS 2024.* Lecture Notes in Computer Science, vol 14995. Springer, Cham. doi:10.1007/978-3-031-74776-2_15.

[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., Ratschan, S. Input-based three-valued abstraction refinement (preprint). arXiv:2408.12668 [cs.LO]. 2024. https://arxiv.org/abs/2408.12668. The version that corresponds to the material used in this thesis is located at https://arxiv.org/abs/2408.12668v4.

[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.