FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	Deadline Verification Using Model Checking
Student:	Bc. Jan Onderka
Supervisor:	doc. Dipl.-Ing. Dr. techn. Stefan Ratschan
Study Programme:	Informatics
Study Branch:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	Until the end of winter semester 2020/21

Instructions

The objective of the thesis is to write a model checker for proving that a given program in machine code for the 8-bit processor ATmega328p reacts within a given time bound to certain user-defined events. In addition, the design of the model checker should allow easy extension to further RISC processors.

Methodology:
1) Search for possible approaches to this problem in the literature.
2) Implement a first version of the model checker that is designed to return sound results, but is not necessarily efficient.
3) Test the model checker against benchmarks.
4) Find algorithmic improvements for increasing the efficiency of the model checker.
5) Implement and test the improvements. Check against alternative approaches.
6) Document the result and release as open source software.

References

Will be provided by the supervisor.

doc. Ing. Hana Kubátová, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague July 30, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Deadline Verification Using Model Checking

Bc. Jan Onderka

Department of Digital Design

Supervisor: doc. Dipl.-Ing. Dr. techn. Stefan Ratschan

May 28, 2020

Declaration

In Prague on May 28, 2020 . . . .

Citation of this thesis

Onderka, Jan. *Deadline Verification Using Model Checking.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

V této práci je představena nová aplikace pro formální verifikaci splnění nejzazších termínů (deadlines) v jednoduchých programech pro mikrokontroléry, pracující na úrovni strojového kódu.

V práci jsou studovány dosavadní techniky a nástroje pro formální verifikaci. Jsou identifikovány jejich slabé stránky. Nevýhodou verifikačních technik pracujících na úrovni zdrojového kódu je zejména jejich neschopnost zaručit časy provádění na úrovni cyklů procesoru. Současných nástrojů pro verifikaci na úrovni strojového kódu je málo, nejsou široce dostupné a většinou jsou specificky navrženy pro konkrétní procesor, což velmi snižuje jejich užitečnost.

Aby nová aplikace nevykazovala nedostatky stávajících řešení, je navržen a implementován nový hybridní verifikační přístup. Techniky ověřování modelu na úrovni strojového kódu jsou použity pro reprezentaci stavového prostoru a verifikaci dodržení specifikace. Paměť mikrokontroléru a chování v rámci kroku jsou specifikovány pomocí jednoduchého imperativního jazyka, s kterým je možné manipulovat pomocí standardních technik na úrovni zdrojového kódu. Tím je umožněna kontrola splnění nejzazších termínů na úrovni cyklů procesoru, rozšiřitelnost na další mikrokontroléry vedle již implementovaného ATmega328P a implementace pokročilých technik bez závislosti na konkrétním použitém procesoru.

Vedle základní funkcionality programu jsou implementovány pokročilé techniky pro zacházení s nedeterminismem, generování řídícího toku a redukci cest pro jednoduché cykly. Aplikace je testována pro prokázání její užitečnosti pro verifikaci splnění nejzazších termínů pro jednoduché programy. Je diskutován dopad použitých technik a jsou identifikovány slibné cesty pro další zlepšení.

Klíčová slova Ověřování modelu, ověřování splnění nejzazších termínů, formální verifikace mikrokontrolérů, formální verifikace na úrovni strojového kódu, přesnost na úrovni cyklů, hybridní technika, nedeterminismus, generování řídícího toku, redukce cest pro jednoduché cykly

Abstract

In this thesis, a new utility is presented for performing formal deadline checking of simple microcontroller programs at machine code level.

The existing formal verification approaches and tools are studied and their weaknesses identified. Namely, source level techniques cannot guarantee cycle-count precise execution times, while machine code verification tools are few, not generally available, and usually heavily tailored to a specific processor, significantly reducing their usefulness.

To counteract the weaknesses of current microcontroller verification tools, a novel hybrid approach is proposed and implemented. Machine code level model checking techniques are used for state space representation and verification of adherence to specification. Microcontroller memory and step behaviour is specified using a simple imperative language that can be manipulated using standard source code level techniques. This allows cycle-count based deadline checking, simple extension to other microcontrollers in addition to the implemented ATmega328P, and implementation of advanced techniques without dependence on the actual processor used.

In addition to the core functionality, advanced techniques for handling nondeterminism, control flow generation, and simple cycle path reduction are implemented. The utility is tested, showing its usefulness for simple program deadline verification. The impact of various techniques used is discussed and promising future improvements are identified.

Keywords Model checking, deadline checking, microcontroller formal verification, machine code level, cycle-accuracy, hybrid technique, nondeterminism, control flow generation, simple cycle path reduction

Contents

List of Figures

List of Tables

Introduction

Modern computer systems are becoming increasingly more complex. As the manufacturing technology progresses, more powerful devices can be used to aid with increasingly more difficult tasks. Despite this, there is a large amount of simple tasks that do not require an abundant amount of resources. These tasks are usually fulfilled by embedded devices that are as simple as possible to minimize cost per unit and power consumption.

When the amount of produced devices is not large enough to warrant an application-specific integrated circuit (ASIC), microcontrollers are the uncrowned kings of this field. They require little to no external components, can be programmed to fulfill the developer's needs, their cost is very low thanks to the economy of scale, and so can be their power consumption if an appropriate part is selected and good design practices are followed.

Unfortunately for developers, just the fact that the programmed device seemingly performs its task well does not mean that it will continue to do so indefinitely. While a manufacturer may guarantee that the microprocessor itself will fulfill its specification while inside its operation envelope (with a certain probability and subject to errata), the program itself is a weak point.

Empirical methods (setting device inputs to selected values and determining whether the device reacts in an expected way) can only take one so far. If a bug is found, the program must be fixed. If no bugs are found using a combination of inputs, not much can be said about the program: a different input could make the program react in an unexpected way. This could have a serious impact, compromising the function of the device (and consequently, its reliability), security, safety, etc.

While this is considered to be an extremely important issue for safety-critical, security-critical, and ultra-reliable systems, I feel that not enough is done for typical, consumer-targeted devices. A single software bug may impact thousands or millions of devices. As many embedded systems are not easily reprogrammable by the user (e.g. by over-the-air reprogramming or manufacturer-aided reflashing via a USB port), this may lead to high product recall costs, damages to the company's reputation, excessive electronic waste (as many companies choose to lock the microcontrollers from reprogramming in order to protect their intellectual property) and other undesirable effects.

Blame should not be put on the product developers, however. Using formal verification techniques that guarantee that the program always works according its specification is currently difficult, requiring considerable work by highly trained specialists. Tools are expensive or severely limited in usefulness.

This is why I decided to create a formal verification tool that would be useful for verification of microcontroller programs that perform simple tasks (henceforth referred to as a "deadline checker" for brevity). Its outward simplicity should aid its adoption not just by professionals, but also by students and hobbyists. This should ultimately lead to a wider adoption of formal verification techniques (as the students and hobbyists become professionals) and better, less faulty embedded products.

I have been convinced from early on that the tool should not be tied to a specific microcontroller or an architecture. Selecting the best device for a task is an integral step in the product development process. A tool that forces the developer to use a certain microcontroller is clearly flawed. As I found out, this requirement for extensibility unfortunately drastically increases the internal complexity of the implementation. Fortunately, this is paid off by the general applicability of each improvement instead of relying on various kludges that only work in specific cases.

Usage of advanced techniques is paramount for avoiding state space explosion. Naïve techniques that perform verification by building the whole state space and examining it are only sufficient for a highly limited subset of programs. Most programs (even simple ones) use various constructs that make the state space impractically big and verification unacceptably slow. Thanks to the generality of the implementation, I was able to add some general techniques to combat these scenarios. More can be added in the future.

I shall further discuss the theoretical underpinnings used to develop the tool, current alternatives, and the tool itself. In Chapter 1, I discuss the terms I use when concerned with ensuring that a device actually does what it is supposed to do. In Chapter 2, I introduce approaches to formal verification (that is, checking that a specification truly holds for all inputs) with a focus on model checking. In Chapter 3, I discuss the specifics of formal verification of microcontroller programs as opposed to other levels and parts of systems that may be verified. In Chapter 4, I examine the current tools available in detail and the lessons I learned from them. In Chapter 5, I explain the precise goals that I chose to aim for during development. In Chapter 6, I describe usage of the tool from user's point of view. In Chapter 7, I discuss design choices made when developing the tool. In Chapter 8, I describe specific parts of implementation in detail. In Chapter 9, I describe advanced techniques implemented. In Chapter 10, I determine how well the tool performs using simple testing programs and discuss areas for potential improvements.

Chapter 1

Definitions of terms used

Many terms are commonly used concerning creation of devices that should behave in a certain way and determination of whether they actually do. Unfortunately, the exact definitions used are usually rather contentious, as seen and discussed in [1]. I shall therefore define how I use them in this thesis. The definitions are based on a "systems" point of view, particularly as used for embedded development. I shall also give an explanation of why I consider them to be the best fit for my purposes.

1.1 Systems, models and specifications

First, I introduce some core terms:

- A *physical system* is a "system" in the physical sense, i.e. "a group of things, pieces of equipment, etc. that are connected or work together" [2].
- A *model* is "a simple description of a system, used for explaining how something works or calculating what might happen, etc." [2].
- A *conceptual model*, also termed *mental model*, is a model comprised by a set of thoughts about the expected properties and behaviour of a physical system (e.g. what it should cost, what task it should fulfill, how should it react depending on an input).
- A *specification* is "a detailed description of how something is, or should be, designed or made" [2].
- A *requirements specification* is a specification that reflects the requirements imposed by the conceptual model.
- A *system model* is a model that can be, in an automated manner, converted to a manufacturing specification.

- A *manufacturing specification* is a specification that contains all needed information for manufacturing the actual physical system.

1.2 Production process

I find the aforementioned definitions most useful in the context of a *production process*. Its ultimate goal is production of an actual physical system that behaves according to the conceptual model. This process is divided into two stages, a *design process* and a *manufacturing process*. The design process is comprised by three steps and the manufacturing process by the fourth one:

1. The conceptual model is turned into a requirements specification.
2. The requirements specification is used to create a system model.
3. The system model is converted into a manufacturing specification.
4. The manufacturing specification is used to create the actual physical systems that may be sold to customers.

I consider the differences between the words *model* and *specification* to be rather semantic in nature: a model describes something, while a specification requires something to perform in a certain way. However, while one can create a product without actually writing down specifications (taking a rather informal approach), the conceptual model (what is wanted) and the system model (how it should work) cannot be skipped.

These definitions are not universally fit for everything and everyone. Mathematicians typically are not concerned with product development. They thus use different definitions, sometimes colliding with the previously discussed definitions (such as they do for the term *model*, which shall be further discussed in the context of model checking in Chapter 2). The production process becomes a bit different in software development, but the core ideas remain the same.

1.2.1 Software development production process

Traditional computer software (e.g. word processors, video editing software, computer games, web browsers, etc.) is not tightly coupled with the hardware that it runs on, using an abstraction layer such as operating system application programming interface (API). This is highly beneficial for common tools since it enables a single program to be run on a multitude of different devices, differing in processor capabilities, peripherals, etc. The production process for software becomes a bit muddled since the manufacturing process is no longer well defined. I consider the manufacturing process in the traditional software case to consist of running the program on the target machine, enabling the computer executing the program to become the physical system fulfilling the specifications. In this case, the previously defined terms have this correspondence:

1. The conceptual model is a set of ideas about how should a computer running the program should behave.

2. The requirements specification is a specification detailing the suitable computer configurations and usable APIs (*execution environment*) and the required behaviour of the compiled program when executed on one of these computer configurations.

3. The system model is the source code of the program paired with the knowledge of the guaranteed behaviour of the programming language used and execution environment.

4. The manufacturing specification is a compiled program binary. The actual physical system is created ("manufactured") by running the program binary on a suitable computer.

To enable further discussion of system model checking without excluding checking compiled program binaries, I note that they can be considered source code with an identity translation programming language.

These definitions are useful for me as they set up the notion is that actual software never exists in a vacuum: it must always be paired with a proper execution environment. This is a fundamental concept for my further discussion. By combining execution environment guarantees, the guarantees specified by the programming language used (in case the analysis is not performed on the machine code itself), and the program, it may be possible to prove or disprove some properties about this combination. Then, we might be certain that the actual system will behave as intended as long as all of the guarantees hold.

For microcontroller programs, the process is very similar to traditional software, with the important distinction that no APIs and abstraction layers are used for simple programs ("bare metal" programming). For more complex applications, a lightweight operating system may be used, such as FreeRTOS [3]. These operating systems usually are not a distinct program that executes other programs at its will, but behave like libraries in that they are compiled with the program, forming a single binary file to be uploaded to the microcontroller. These lightweight operating systems may therefore be considered a part of the program for analysis instead of being an external API. This practically means that for the resultant program, the execution environment is still the microcontroller and its peripherals, not some abstract APIs. The knowledge of processor behaviour is therefore crucial for determining interesting properties.

1.3 Verification and validation

Having defined the core terms, I shall discuss what steps can be taken to ensure that the final physical system behaves according to the conceptual model.

First, I shall define the top-level terms:

- *Validation* is the process of manual evaluation whether a model, a specification, or a physical system behaves according to the conceptual model ("are we building the right system?").

- *Verification* is the process of evaluation whether a model or a physical system behaves according to the corresponding specification ("are we building the system right?"). It may be manual or automated.

In my view, these constructs are disjoint (no technique is both a verification and a validation technique). Validation falls outside the scope of this thesis, which is concerned mainly about general program behaviour instead of concrete program examples.

Verification can be done "informally" or "formally". Informal techniques restrict themselves to a subset of possible cases that can happen (*execution paths*); therefore, while they may find a counterexample, that is, a case which shows that a model does not fulfill its specification, they cannot fully assure the designer that a model does fulfill its specification. Formal verification provides a *formal proof* that given the previous specification, the verified model always fulfills it.

1.4 Informal verification, testing

Various audits, code reviews, etc. may be instances of *informal verification* if adherence to corresponding specification is checked. Another important informal verification technique is *testing*, a process of evaluation whether a model or an actual system behaves as intended for a limited (usually finite) subset of possible executions. This is important since complex models and systems usually cannot be fully formally verified. Also, since the behaviour of actual systems is not governed by guarantees, but by physics, formal verification of actual physical systems is impossible, while testing is certainly possible.

This can be illustrated by an example: if I insert an appropriate amount of coins into a soda machine and press the button with my favourite drink pictured on it, the drink and corresponding change must come out of the machine for it to behave according to my conceptual model of soda machines. There is no "formal" way of verification that the machine does not become faulty due to its internal components breaking, however. Ordering and receiving a drink is a *test* that, while not guaranteeing that the machine will behave properly if another drink is ordered, yields a high degree of confidence that it will.

1.5 Formal verification, its advantages and limitations

Formal verification, in contrast to informal verification, may prove that some property is true for a given exactly described model with certainty, deducing that the property must naturally follow from its guarantees. This may be used to verify adherence of a model to its specification.

The ability of formal verification to provide a proof that a model fulfills a specification is important in critical systems and infrastructure. Failures may lead to loss of lives and costly damages, making every bit of assurance that they shall not fail important. With non-critical systems such as most consumer devices, this is not strictly required, but given that the failure might occur within all devices manufactured to the same specification, pursuing formal verification techniques may be a viable alternative to costly product recalls.

I believe it is important to note that while formal verification is a powerful technique, it does not always mean that the resulting system will behave correctly. There are many cases where it might fail to ensure actual compliance:

- The checked model operates with guarantees while the actual system is physical. If the guarantees do not hold, the system may fail. This may be due to non-correspondence between the actual system and its guarantees or due to a reliability issue.

- Any requirements that are not a part of the specification are not checked. This is typically due to the specification writer not realizing that they are necessary.

- The transition from a model to a manufacturing specification and to the system itself may be a point where deviations from specification may occur. In software, even a program with source code that provably conforms to its specification might not conform after a compiler pass (if the compiler does not fulfill its guarantees) or after a faulty upload to the embedded device.

- The actual formal verification tool used is a program and as such may be faulty or executed in an environment that does not fulfill its guarantees. This is true even if a formal verification algorithm is executed by hand. Errors may be introduced and a wrong result produced.

This means that while formal verification may increase our trust in the checked model (and consequently, the physical system), it is not a silver bullet. Proper engineering and quality management is always necessary to ensure that the physical system works as intended.

1.6 Deadline checking using formal verification techniques

A deadline is “a point in time by which something must be done” [2]. In this thesis, I consider *deadline checking* to be a process of checking action-reaction *rules*, whether an action being performed in a model of an embedded system will always result in a corresponding reaction being performed within a certain time (the deadline).

This checking is performed on a system model against a requirements specification containing the rules. As previously discussed, this model must comprise of the execution environment, the program checked, and the used programming language behaviour (if machine code is not used). The specification must also be well-defined (written in an unambiguous language) to enable proving the adherence of the model to the rules.

While it is critical that the deadline checker only proves that the deadline rule was fulfilled (the reaction to the action always occurs before the deadline) if it it actually true, it is not required to prove it for all cases when it is true. In these cases, a false counterexample may be returned.

This corresponds to the intended usage of the deadline checker: if it is able to prove that the specification is fulfilled, it forms a *guarantee* of the behaviour and could influence decisions about releasing a product to customers. There are no significant risks associated with returning a false counterexample. It may result in not releasing the product or abandoning the use of the deadline checker. I do not consider these possibilities to be critical.

Chapter 2

Approaches to formal program verification

Formal program verification is a long-standing field of computer science. In his 1972 Turing Award lecture, Edsger W. Dijkstra devised a scheme that would allow humans to build complex systems without the fear of programming bugs and the costs associated: by providing a mathematical proof of a program's correctness while building it, bugs would not appear later. In fact, by constructing a proof, the implementation would follow naturally [4]. Ultimately, this did not catch on as constructing a proof for a program is not particularly scalable. A different approach, computer-aided verification, proved to be more suitable [5, p. 1-2]. System designers produce a program and a specification it should fulfill; the difficult task of finding whether the program actually fulfills it is offloaded to a computer.

Different approaches for formal program verification exist and various classifications may be used. I introduce a basic categorization of the techniques. They may be variously combined, as is often done in practice. [5, p. 4-5].

2.1 Formal proof techniques

Formal proof techniques aim to produce a formal proof of some property of a program by mathematically defining programming language semantics and, consequently, the mathematical meaning of the program itself. Once the program is expressed in a suitable logic framework, the proof may be constructed from the appropriate axioms [6] [7].

Unfortunately, while such methods do exist and may be automated, their scalability to more complex program structures is still limited. As with Dijkstra's approach, a proof must be somehow devised; moving its construction to computer essentially moves it to the writers of the verification tool. This results in limited usage of these techniques [5, p. 1-2].

2.2 Model checking

Model checking takes advantage of the fact that practical digital systems only have a finite number of possible states. This makes it possible to represent the states, transitions and properties of the system model by using a graph-like structure. This structure can then be manipulated to prove that the system model fulfills the specification [5, p. 2-3].

The classic formalism for the graph-like structure used is a Kripke structure. The structure is constructed over a set A of atomic propositions as a triple $K = (S, R, L)$ where

- S is a finite set of states (state space),
- $R \subseteq S \times S$ is a set of (directed) transitions,
- L is a labeling function $L : S \rightarrow 2^A$ which associates each state with a set of atomic propositions [5].

Furthermore, the dynamic behaviour of the system corresponds to a path within the directed graph (S, R). Using this representation, the model checker checks whether the specification expressed the Kripke structure K is a model (in the sense of model theory) of as a temporal-logic formula ϕ, that is, $K \models \phi$ [5, p. 2-3]. Unlike the rest of this thesis, the word *model* is used in the mathematical sense. This is where model checking gets its name from. In practice, there is fortunately not much confusion between the system model (in the studied case, the processor description and binary program code) and the graph-like structure it generates, which is subsequently checked against specification.

The specifications are usually described using *temporal logic*. Various types of temporal logic are used, such as Linear temporal logic (LTL) or Computation tree logic (CTL). Usage of these logics allows describing highly complex behaviours. Unfortunately, they are not very intuitive for an average programmer to use. As I do not use them in the implementation, I shall not describe them further. A detailed description of temporal logics may be found in [8].

While the task of finding out whether a model fulfills its specification is decidable if the number of states is finite, it is infeasible to do so by naïve exhaustive state space generation due to *state space explosion*, also known as *combinatorial explosion*. Multiple approaches have been devised to prevent it [5, p. 14-22].

Practically speaking, the state space explosion in actual processor-based systems typically exhibits itself when reading (often unrelated) nondeterministic inputs and assigning them to separate variables. For example, if an 8-bit (completely unknown) variable is read from input and assigned to a register, the naïve exhaustive generator must generate 2^8 states from a previous one, one each for every possible 8-bit combination. After that, reading from input to another variable generates another 2^8 states from each one, resulting in $2^8 * 2^8 = 2^{16}$ new states after this is done. In this manner, the state space size increases exponentially, resulting in unacceptable computational requirements extremely quickly. This is why prevention of this behaviour is a critical problem when model checking is performed.

2.2.1 Symbolic model checking

Symbolic model checking approaches do not store the explicitly enumerated set of states, instead storing them as symbolic logic expressions. The first important symbolic model checking technique was model checking with binary decision diagrams (BDDs), essential for success of model checking in the hardware industry. The other important technique in this category is SAT-based model checking which converts the model checking problem to the boolean satisfability problem, which can be then solved using advanced SAT solvers. SAT-based model checking may be bounded (only finding counterexamples of finite length) or unbounded (finding all counterexamples) [5, p. 16].

2.2.2 Abstract model checking

Abstract model checking uses a different approach to combat the state space explosion, using a structure that *overapproximates* the actual Kripke structure. There are multiple definitions for the notion of overapproximation, as seen in [9], but the fundamental notion is that overapproximation must preserve the original behaviours but may introduce additional behaviours.

Abstraction techniques based on three-valued logic are widely used, particularly in the context of circuit verification [10] and microcontroller program verification [11]. In this scheme, each bit of original system state is assigned one of three values: ’0’, ’1’, or ’X’. While values of ’0’ and ’1’ represent the bit being in one of the original values (logic zero or logic one), value of ’X’ means that states with both values are present. Bit operations can be performed using three-valued logic, resulting in an overapproximation of the state space. These abstractions preserve *soundness* (also called soundness for true) [10]. Soundness means that while a counterexample may be generated by the model checker even if the system model fulfills its specification, but it may not prove adherence to specification if the system model does not fulfill it.

Soundness is an expected behaviour for typical model checkers of complex systems: if a proof is made, the system definitely adheres to the specification. If a counterexample is found, it should be investigated to determine its actual behaviour in system model. There is a possibility that it is a *false counterexample*, not being a counterexample for the original, non-abstracted system. In that case, the model checker does not tell us anything interesting. This is closely related to the (rather informal) notion of *usefulness*. While a model checker that always fails to prove adherence to specification is trivially sound, it is not useful at all.

Chapter 3

Specifics of microcontroller code verification

Traditional computer programs are usually verified using their source code (or its simplified representation). For embedded systems, a binary code approach might be more beneficial, prompting the existence both types of verification tools. I shall examine the differences further in this chapter.

3.1 Source code verification

In traditional computer programs, the interest lies mainly in producing the correct outputs. Microcontroller-based embedded systems usually require precise timings of those outputs as they interact with other system components and outside world in real time. This means that analysis of source code (usually written in C language) might not be sufficient because the language provides no timing guarantees. A “high-level” approximate timing specification schemes may be produced, but the actual timing depends on the compiler.

Furthermore, if the compiler itself does not conform to the specification of the programming language, the program itself may be faulty even though the source code fulfills the specification used on its own. This problem may be alleviated in some way, for example by using the formally verified CompCert compiler [12].

An interesting take on the notion of compiler trust is demonstrated in Thompson [13], where a C compiler binary is made so that it injects a vulnerability into a specific piece of code it is compiling; it also injects compiler source code it is compiling so that it injects compiled programs the same way. Verification after compilation may detect this, but the same argument raises the question of trusting the verification tool itself. That said, it would seem rather unlikely for both a compiler and a verification tool to be infected in a way that an infected program passes the verification check.

In any way, the main reason why source code is unsuited for the task at hand, deadline checking, is not trust, but the timing issue. Actual timing can be only determined when both the execution environment timing and its relationship to the program is known. This might be done inside a compiler using various existing techniques [14, ch. 1]. It does, however, tie the programmer to a specific compiler and, consequently, to the processors supported and optimizations implemented by that compiler, in addition to restricting the ability to use assembly language for time-critical operations (unless it is also supported for consideration by the compiler). I did not choose to pursue this approach due to these limitations.

3.2 Binary code verification

Binary code verification requires a slightly different mindset from the source code approach. While the higher-level languages are usually understood as a logical construct independent of the architecture, binary code is intristically tied to the concrete processor. This means that the verification tool must consider the processor in conjunction with the binary code.

This could be approached from two different angles. From hardware point of view, a "virtual" processor might be constructed from Register-Transfer Level (RTL) blocks (especially D flip-flops and logic gates); the program would be stored in the register elements and RTL-level verification could be performed. From software point of view, "virtual" registers and code describing processor step behaviour also might be constructed, allowing the use of source-code-like verification techniques. Unfortunately, while RTL level formal verification is widely used nowadays in hardware verification [15, p. 782-786], the high-level relationships are muddled, precluding high-level optimization that may drastically reduce computational complexity.

Compared to the source code approach, timed verification is very much possible, especially when the processor architecture is simple. With advanced pipelining or caching, the worst-case timing may be heavily overapproximated, decreasing the ability to prove adherence to specification.

As the conversion to binary code loses information, the verification tool is in a harder high-level construct optimization position than a source code tool. This means that source code verification is still a more reasonable approach for general computer applications. Binary code verification still has distinct advantages for checking of simple embedded programs on microcontrollers, especially when processor step behaviour is described instead of RTL-level behaviour.

Chapter 4

Current binary-level model checkers for microcontrollers

Currently, RTL level and source-level verification is widely used in industry with a variety of competing tools; further discussion of them may be found in [15, p. 782-786]. Unfortunately, as I discussed in Chapter 3, source code verification and RTL-level verification are not very suitable for the special case of programs for simple microcontrollers. This restricts the interesting tools to a few binary-level verification tools, which are generally built on model checking due to the suitability of this method for this application.

4.1 State Exploring Assembly Model Checker (StEAM)

The StEAM checker described in [16] is rather unique among binary-level checkers since it does not aim to perform verification for specific hardware. Instead, it compiles the C/C++ program it is trying to verify using a virtual machine architecture as a target. This results in it having similar properties to source code level checkers. While the methods used are different, the verification is ultimately not done using the actual target architecture. This model checker unfortunately does not seem to be publicly available at the time of writing.

4.2 Estes

In [17], model checker Estes is introduced. This model checker uses the `gdb` debugger to step through states and thus can, in theory, support multiple processor models as long as `gdb` supports them. Unfortunately, this is not a drop-in scenario in practice as the paper discusses extensive changes necessary to adapt the debugger to the task of model checking on Motorola 68hc11 processor.

While this approach might be interesting for complex architectures where writing the processor code would be handy, this is not the case for simple microcontrollers. Furthermore, this model checker also does not seem to be publicly available at the time of writing.

4.3 Arcade.µC (formerly [mc]square)

The Arcade.µC is a long-running effort to create a usable binary-level checker for microcontrollers at RWTH Aachen. The checker, formerly known as [mc]square, was introduced in [18]. A previous similar checker called MCESS was also mentioned there but seems to be obsoleted by Arcade.µC. This model checker builds the state space directly using a custom simulator written for the specific processor. It checks whether Computation Tree Logic (CTL) formulae are valid without handling timing.

The Arcade.µC model checker was refined after the initial paper, exhibiting a definite trend towards using advanced abstraction and static analysis techniques. One of the most important techniques for state space reduction is the *delayed nondeterminism*. Using this technique, each single bit of state is represented by three-valued logic: 0, 1, or unknown [11]. Static analysis techniques are also used [19].

A longstanding problem of Arcade.µC is the difficulty of adding new microcontroller types and architectures. As stated in [20]:

> [mc]square is an assembly code model checker, which builds state spaces using simulators, which are used to simulate microcontrollers. Based on experience from adding support for new platforms, it is possible to estimate the effort for manually implementing such a simulator. The Atmel ATmega128 and XC167 simulators were implemented by diploma thesis students [citation omitted], and the 8051 simulator was implemented by Reinbacher partly before and partly within a master's thesis [citation omitted]. At the end of these theses, the simulators were operative, but not all of the peripherals had been implemented. Depending on whether there was interest in further uses of the simulator, it was then necessary to continue development, ending up in a total effort of about one year for a single full-time developer.

This was the reason for development of "state space generators synthesis" in [20, p. 26]. Unfortunately, this approach was not entirely successful as the author was unable to implement the delayed nondeterminism technique in a way that would not require the author of the microcontroller description to manage the instances of nondeterminism [20, p. 121].

The decision to not to use timing is also problematic. This disqualifies the checker from performing deadline checking where timing is crucial. This problem was examined in [21] and new methods for introduction of timing were proposed.

Ultimately, the most significant problem lies in the fact that Arcade.µC is not available to general public; only a demo is provided on the official website `https://arcade.embedded.rwth-aachen.de/doku.php?id=arcade.uc`.

In the end, after researching the available literature, I did not find any binary-level verification utilities that could be used by general public without extensive knowledge of model checking. Therefore, I decided to create a new one.

Chapter 5

Goals for a new deadline verification utility

As a student and an enthusiast, I have been interested in microcontroller-based devices for a long time. My experience with writing microcontroller programs and developing devices has been, unfortunately, less than stellar. I remember multiple instances of a device seemingly working as intended, then failing to do so afterwards. Sometimes, it was due to bad connections to peripherals, sometimes due to the parts themselves being damaged (usually after being subjected to voltages or currents outside their maximum ratings). The especially tricky problems were usually caused by software bugs, however; sometimes in my code, sometimes in the libraries I used.

Eventually, I started to fear that these bugs would not be found until after a device I had made was shipped to its customers. At that point, it would be hard to fix the situation without incurring significant financial losses and damage to reputation. This resulted in my search for a tool that could alleviate my concerns by showing that the system model behaves according to its requirements specification (or at least part of it).

After researching current microcontroller program verification utilities available, described in Chapter 4, I arrived to the conclusion that none of them fulfill the criteria I seek. Most program verification tools, due to working with a higher-level language description, do not support time-based checking. This is not acceptable for microcontroller programs as they are fundamentally real-time devices. The binary-level verification tools are few, not readily available, complex to use and limited to one or a few processor models without easy extensibility.

This is why I decided to create a new microcontroller binary-level verification tool to be readily accessible to writers of microcontroller programs and help them with fault-free program development. I compiled the following list of requirements:

- The tool must be free-to-use, simple to install and start using.
- The tool must perform deadline checking (checking whether an “action” occuring always results in a “reaction” occuring within a certain amount of time) soundly but may resort to oversimplification.
- The tool must support microcontroller ATmega328P, be easily extensible to support other AVR processors and only require a moderate amount of work to extend to other simple microcontroller architectures by converting the appropriate datasheets to a well-defined language.
- The tool should be able to perform the checking of programs in a reasonable amount of time (seconds to minutes).
- The tool should be easily usable by an inexperienced hobbyist, student or engineer without formal verification experience.
- The tool should provide the user with insight into the program execution through a way to visualise the state graph structure and found counterexamples. This may greatly increase the user’s understanding of the possible execution paths and consequently their understanding of the program and the microcontroller, paving the way for use in formal education or self-education.
- The functionality of the tool should be easy to extend with new model checking techniques, visualisation options, etc. This means the tool should be written with high cohesion and low coupling in mind to allow reworking or adding a part without introducing changes to other parts of the tool.

The choice of the processor is due to ATmega328P being arguably most popular simple microcontroller today due to being used in the “classic” development kits of the Arduino project. Furthermore, their instruction set and peripheral functions are very similar across the whole line of AVR devices, including the ATtiny range which includes low-cost devices used for simple tasks.

Due to the complexity of the task and limited scope of the diploma thesis, I restricted myself to model checking simple programs with these restrictions:

- Limited microprocessor peripheral and outside world support. Only digital pin input/output is supported; the outside world only exhibits itself via digital pin inputs and uninitialized values.
- No support of interrupts. Interrupt-based programs are fundamentally parallel in nature and extreme state space explosion may occur for all but the simplest programs unless advanced methods are used.

Chapter 6

Deadline checker usage

I created a deadline verification tool that fulfills the goals specified in Chapter 5. The tool is released it as free software under the MIT license. Before I delve into implementation details in Chapter 7, I aim to provide a high-level understanding of its usage and value to an embedded system programmer. In this chapter, I describe how it can be used by programmers to verify properties of microcontroller programs.

6.1 Usage for visualisation

In normal AVR microcontroller programming workflow, the programmer writes a program in C or assembly language and compiles it to an Intel HEX file. This file can be directly used for programming the microcontroller using a specialized software tool. In case the programmer is only interested in visualisation of program execution, no detailed knowledge of the processor and specification description language is needed. A simple command may be used to generate the program execution state space:

```
$ ./model_checker.exe -a atmega328p -g graph_output.txt
    basicbranch.hex
INFO Checking.
INFO Processor description loaded successfully.
INFO Processor description compilation took 0.0881747 seconds.
INFO Stepping took 0.0756552 seconds.
INFO Number of space nodes: 20
INFO Generation states: 20
INFO Specification states: 0
INFO Execution state size: 20
INFO Checker successfully terminating after 0.166336 seconds.
```

During the execution of this command, the model checker builds the state space of program in `basicbranch.hex` assuming it is executed on microcontroller ATMega328P. Default state space information is then output to `graph_output.txt`. It contains information about state graph nodes (states),

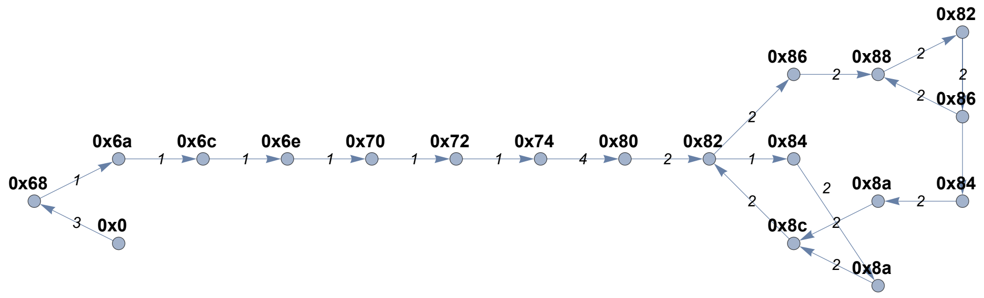

Figure 6.1: Example state space visualisation. Program counter values are displayed above nodes, transition cycle counts are displayed over edges. The data come from the checker. Wolfram Mathematica is used for visualisation.

edges (transitions), program counter values, and transition cycle counts. In Figure 6.1, the resulting state space information is visualised by Wolfram Mathematica.

To compare the state space to the program that was used for its generation (in addition to architecture information), more information about the example program is needed. The `basicbranch.hex` program is a basic program for ATmega328P that reads an input pin and sets an output pin to the same value. It was written in the C language and then compiled by the GCC C language compiler. Its shortened disassembly (with some comments added to aid understanding) is:

```
Disassembly of section .text:

00000000 <__vectors>:
   0:   0c 94 34 00     jmp     0x68    ; 0x68 <__ctors_end>
   4:   0c 94 3e 00     jmp     0x7c    ; 0x7c <__bad_interrupt>
(...)

00000068 <__ctors_end>:  ; performs C stack initialization
  68:   11 24           eor     r1, r1
  6a:   1f be           out     0x3f, r1        ; 63
  6c:   cf ef           ldi     r28, 0xFF       ; 255
  6e:   d8 e0           ldi     r29, 0x08       ; 8
  70:   de bf           out     0x3e, r29       ; 62
  72:   cd bf           out     0x3d, r28       ; 61
  74:   0e 94 40 00     call    0x80    ; 0x80 <main>
  78:   0c 94 47 00     jmp     0x8e    ; 0x8e <_exit>

(...)

00000080 <main>:
                        ; set pin 1 in port B as output
  80:   21 9a           sbi     0x04, 1 ; 4
                        ; skip next instruction if input value of
                            pin 0 in port B is 1
  82:    18 9b           sbis    0x03, 0 ; 3
  84:    02 c0           rjmp    .+4            ; 0x8a <main+0xa>
                         ; set output value of pin 1 in port B to 1
  86:    29 9a           sbi     0x05, 1 ; 5
  88:    fc cf           rjmp    .-8            ; 0x82 <main+0x2>
                         ; set output value of pin 1 in port B to 0
  8a:    29 98           cbi     0x05, 1 ; 5
  8c:    fa cf           rjmp    .-12           ; 0x82 <main+0x2>

(...)
```

While this example is very simple, there still are some interesting details in Figure 6.1. For example, the graph is somewhat asymmetric due to the nature of branching constructs in the AVR architecture and optimization done by compiler even though the branches are symmetrical in C code.

6.2 Verification of adherence to specification

While visualisation of execution state space provides some insight into program execution, actual verification of specifications with simple deadline rules is the key feature of the model checker. Writing verification specifications requires the programmer to become familiar with the description language, further detailed in Chapter 7. For now, let us assume that the programmer has written a specification of `basicbranch.hex` called `basic_branch_rules.txt` with 5 rules that they are interested in:

- The output pin is configured as output within 30 cycles.
- If the output pin is set to 1, it becomes 0 within 30 cycles.
- The input pin is never configured as output.
- The output pin is never set to 1.
- If the input pin is held at 1 for at least 20 cycles, the output pin is set to 1 at most 30 cycles after the input pin becomes 1.

With an additional command-line option, the execution state space is checked against the specification. A `-c` option is also used to output the result of checking to a separate file for visualisation.

```
$ ./model_checker.exe -a atmega328p -g graph_output.txt -s
    basic_branch_rules.txt -c check_output.txt basicbranch.hex
INFO Checking.
INFO Using value propagation.
INFO Processor description loaded successfully.
INFO Processor description compilation took 0.105436 seconds.
INFO Rules loaded successfully.
INFO Rules IR generated successfully.
INFO Stepping took 0.10558 seconds.
INFO Number of space nodes: 19
INFO Checking that the specification holds.
INFO --- checking rule 1 ---
INFO      rule passed, worst case satisfaction path length: 15,
    number of premise holding states: 19
INFO --- checking rule 2 ---
SEVERE    RULE FAILED
INFO --- checking rule 3 ---
INFO      rule passed, worst case satisfaction path length: 0,
    number of premise holding states: 0
INFO --- checking rule 4 ---
SEVERE    RULE FAILED
INFO --- checking rule 5 ---
INFO      rule passed, worst case satisfaction path length: 19,
    number of premise holding states: 17
INFO Rule checking took 0.0477221 seconds.
INFO Generated states: 19
INFO Execution state size: 19
INFO Peak working set size: 20 MB
SEVERE Checker terminating after 0.271605 seconds in a failure
    state.

$ echo $?
127
```

Rules 2 and 4 did not pass verification. This results in the checker ending in a failure state and returning a non-zero value code in addition to printing information about the failures. This enables the checker to be used in automated fashion, ideally between compilation and uploading to the microcontroller. This scheme could detect regressions in program behaviour before they could ever manifest on the hardware (potentially even damaging it, e.g. in case of microcontroller pin being mistakenly set as output when connected to another strong output with different potential).

6.3 Interpretation of counterexamples and worst-case paths

Since the checking result was also output for visualisation, the counterexamples for rules 2 and 4 can be used to pinpoint the source of rule failure. Figure 6.2 visualises the counterexample to rule 2. By studying the marked instructions, it is easy to realize that rule 2 cannot hold for the program as written. If the output pin is set to 1 and the input pin continues to stay at 1, the output pin will never be set to 0. Thus, it can be concluded that either the program or the rule is faulty.

This reasoning may be used for any rule that does not hold. Care must be taken as the checker is not required to prove a rule that holds for the original model. It is therefore also possible that the counterexample is a false

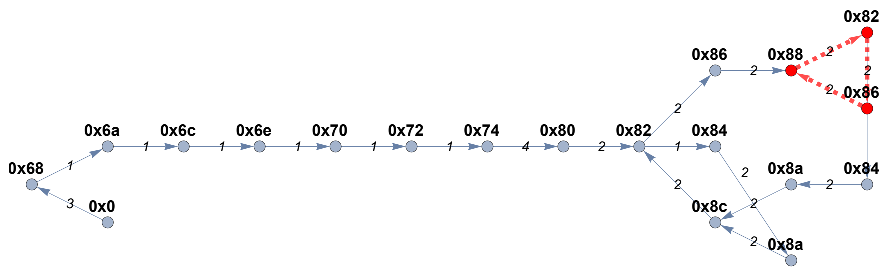

Figure 6.2: Counterexample to specification rule 2 visualised using Wolfram Mathematica, shown as the cycle marked in red with edges dashed.

counterexample, produced by an execution path that does not exist in the original model.

In my opinion, the visualisation of the counterexamples greatly aids in understanding where and how the rule fails. In cases where graph visualisation is unwieldy, the counterexample can be studied in textual format.

If a rule passes, the checker proved that the model fulfills it. A *worst-case satisfaction path* is also output. It is guaranteed that the rule will be satisfied even with a changed deadline that is equal or longer to the path length (as long as no other settings change). No guarantees are made about the possibility of the path being of the same length as it would be in actual execution (it may actually be a longer path that does not exist in the original system model). Despite this, it may give some insight into the “weak point” of the program. This can be seen in Figure 6.3. The corresponding rule states that if the input pin is held at 1 for at least 20 cycles, the output pin is set to 1 at most 30 cycles after the input pin becomes 1. The worst-case scenario occurs when the input is being held at 1 from the start of the program execution. In that case, it takes at most 19 clock cycles for the output to be set to 1 as well.

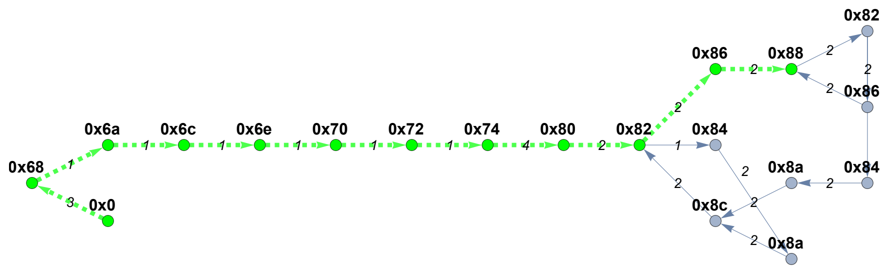

Figure 6.3: Worst case execution path to specification rule 5 visualised using Wolfram Mathematica. The path marked in green with edges dashed shows the worst-case satisfaction path output by the checker.

Chapter 7

Deadline checker implementation decisions

To properly fulfill the goals specified in Chapter 5 and create an easy-to-use deadline checker as described in 6, I had to make a multitude of fundamental decisions about the implementation.

7.1 Processor description

The most important decision concerning the checker was undoubtedly the choice of how to represent the microcontroller memory and behaviour in order to achieve easy extension to further microcontrollers and perhaps even other processing devices. I searched for a suitable solution to this problem in current binary-level model checkers described in Chapter 4.

I was inspired by [20] where model checker Arcade.µC was augmented by automated creation of "state space generators" from their hardware description in a custom language. While this scheme is certainly interesting, it cannot handle nondeterministic techniques well, requiring the processor description creator to identify variables that can be handled in a nondeterministic fashion. As the transcription tool and Arcade.µC are independent, usage of advanced verification techniques requiring access to a currently generated state space and the processor description at once is problematic.

Since I the fundamental notion of describing hardware by a custom language is a suitable solution to the problem of extensibility and I was not restricted by existing codebase, I decided to use a novel hybrid approach. The core functionality is that of a model checker, but the state variables description and state transition behaviour (processor stepping) is defined by an interpreted processor description. This allows advanced verification techniques to be performed on-demand during state space generation.

After the choice of this technique, it became necessary to choose a proper processor description language. A multitude of processor specification languages exists, mostly used for the purposes of simulation. Detailed summaries of some of these languages can be found in [20, p. 24-25] or [22, p. 3-5]. Unfortunately, I did not find a suitable language for my purposes. Specifications of most of them are not publicly available. One notable exception was the MADL language described in [23], which should have its specification available through `https://ptolemy.berkeley.edu/projects/embedded/mescal/forum/3.html`, but the link to the actual archive is dead at the time of writing and the page was last updated in 2005.

Therefore, I decided to create my own processor description language, aiming to provide a simple imperative language similar to the C language familiar to most embedded system programmers. To facilitate simple implementation of enhanced techniques, I also decided to convert the language into simple Intermediate Representation. The actual state manipulation is performed using this Intermediate Representation. It is reasonably easy to perform advanced transformations on it.

7.2 Specification description

I chose not to use a full-fledged temporal logic (discussed briefly in Section 2.2) for deadline specification. This is due to the fact that understanding temporal logics takes a large amount of time and effort, which is precisely the thing I was aiming to avoid. In my opinion, a tool should support describing specifications in an imperative language since it is the only paradigm most programmers are proficient in.

As I chose to implement my own processor description language, it became apparent that using the same language for writing specifications could greatly reduce implementation time and complexity. In case of deadline checking, a function must be written to determine whether a singular state is a premise, and another to determine whether it is a premise. Using the processor description language for this is extremely simple: the state under consideration is copied, the copy is transformed by the function, and the function return value is extracted from the resulting state.

The next important choice concerned the actual format and behaviour of deadline specifications. As the notion of a *deadline* is rather vague, I needed to come up with a well-defined meaning that would allow the implementation of a model checking strategy. First, I defined the input specification to be comprised of a set of individually named *rules*. This allows checking multiple statements about the system model using an already built state space.

My choices were determined by simple programs that would respond to some pin being "asserted", requiring some reaction to be made. The most basic example of this is the microcontroller acting as a buffer, repeatedly setting the value on an output pin to mirror the value on an input pin. (One of the possible implementations of such behaviour is the `basic_branch` example in Chapter 6.) To allow reasonable checking of this example, the rules are constructed using four pieces of information:

- Premise function. This function is called with state information present and returns a truth value about whether the state fulfills the premise.
- Premise duration. This is non-obvious but needed for the ability to prove reactions to held input pins. Typically, some work is done between the values of input pins being checked. A very temporary pulse setting the pin to the value where premise holds might not be detected, as the microcontroller might not check the input pin values during that period.
- Eventual consequence function. This function works similarly to the premise function.
- Deadline. If the premise holds for the premise duration or longer, the eventual consequence must hold in at least one state before or exactly at the deadline time.

An important consideration is that of time between states, as the state space is discrete, but real systems work in a continuous time environment. This is problematic when considering the time when a premise is fulfilled for the first time. It may be fulfilled between states. If the successor state was selected for consideration, a shorter hold time and deadline could be computed. This forces the usage of the predecessor state, which can only produce paths with longer hold length and whole path length, not excluding possible paths.

The need for consideration of time between states adds an important consideration for the processor description: there cannot be any nontrivial changes in outputs exposed to the world. For example, if two successive states had an output variable set to 0 and in the actual system, the output would actually change to 1 for a short time between states, checking could produce a false proof of the output being always 0. This is fortunately not a typical issue in simple microcontrollers such as ATmega328P.

7.3 State representation and nondeterminism handling

An important implementation decision is how exactly should the checker store states. Due to the nature of microcontroller memory and variable manipulation, retaining the definitely known values explicitly seems as the best choice. A question arises how to store non-deterministic values.

Handling non-deterministic values in binary-level model checkers is necessary not only for handling inputs, but also indeterminate values in processor memory (especially after start, but potentially also e.g. after a peripheral is shut down). For example, the used ATmega328P processor has 2048 bytes of RAM that is uninitialized after reset (in addition to other uninitialized registers) [24]. Creating a state for every possible RAM combination would entail the creation of 2^{16384} vertices. Clearly, this is not an ideal approach.

The approach taken by Arcade.μC is to use three-valued logic [11]. In three-valued logic (also described in Subsection 2.2.2), the third value means “unknown” (usually shortened as ’X’). Uninitialized memory locations and inputs can then be represented by “unknown” (’X’), reducing the number of states drastically.

I use an extension of this concept. In a typical binary computer, two bits are used for storage of one three-state bit representation to allow fast data manipulation. This prompted me to use the fourth value to represent an “uninitialized” (usually shortened as ’U’) value. This is a value that effectively is unknown (can be 0 or 1), but its manipulation is strongly undesirable in typical cases. This is comparable to the IEEE 1164 standard for nine-valued logic [25] used in VHDL designs for ease of simulations, which also defines these 4 values (and additionally, low-impedance counterparts, a high impedance state and a don’t care value, none of which are appropriate for this application).

I use the “unintialized” (’U’) value to detect usage of uninitialized memory. This is usually not a desirable behaviour for programs, typically occurring due to bugs, making such detection useful. My model checker is able to detect usage of uninitialised values and immediately return with error.

The behaviour of unknown variable manipulation is also extremely important. I made a choice to represent the digital input addresses as variables holding (and being updated at the start of each state by) unknown values. Immediate determinization of values on assignment is not feasible since it would immediately determinize all unknown values at the start of each state. Due to this problem, I chose to use aggressive nondeterminism as the “core” behaviour. It does not perform determinization for assignment and binary operations (bit negation, and, or, xor), where three-valued logic is used. Unfortunately, this technique results in a large amount of approximation, prompting me to use advanced techniques described in Chapter 9 to reduce it.

7.4 Programming language and external dependencies selection

I had to make a choice of the programming language and external dependencies before implementing the checker. As speed and memory consumption are important factors for a model checker, I decided to use the C++ language. While I believe that the state space explosion problem is best countered by advanced techniques rather than brute force and therefore have decided to focus mainly on them rather than on best state space generation performance, using C++ provides high performance and allows for aggressive optimization should it ever become necessary. Compared to the C language, the language facilities allow for speedier development and better modularity.

Ultimately, I chose not to use external dependencies for the core tool functionality. This largely stems from the fact that binary-level checkers are *unique* in the world of compilers, static analyzers, simulators, verification tools, etc. They require a fusion of state space generation, nondeterminism abstractions, higher-level techniques such as path reduction, and so on; this makes the other related tools and frameworks unusable. Furthermore, as my tool is being created since no current binary-level checkers are suitable for me, using them is also not possible.

Tools such as `lex` or `yacc` would have been usable for the minor task of creating lexer and parser for the processor description language, but writing a recursive-descent parser for it proved to be easy enough to obviate the need for them.

In order to avoid unnecessary work in non-core areas, I elected to use the `g3log` library for logging and the `TCLAP` library for command line arguments parsing. The libraries are available under permissive licenses and thus do not interfere with my goal of releasing the software under the MIT license.

Chapter 8

Deadline checker implementation structure

In this section, the various top-level processing modules of the checker are examined in more detail. The modules are designed to be highly cohesive to minimise the amount of them impacted by advanced model checking techniques. Some advanced model checking techniques are already implemented in the model checker and are further discussed in Chapter 9.

Figure 8.1 shows a top-level view of the checker tool functionality. From user's point of view, the tool takes the filename of a program in its binary form along with name of the processor it is supposed to be run on. Optionally, a deadline specification and other flags may also be input: if no deadline specification is used, the checker will merely produce a state space structure for visualisation purposes. The tool then works in the following way:

1. Processor description corresponding to the provided processor name is loaded from the checker's processor library.
2. Program instructions are loaded from a `.hex` file.
3. Deadline specification (a list of rules) is loaded if specified.
4. The first state is created using a memory initialization function in the processor description.
5. Stepping is performed using the processor description and the program instructions. New states are only added if they do not correspond to states already found. The complete state space structure is available once no states remain to be processed.
6. The state space structure is output for examination / visualisation if the user indicated the need.

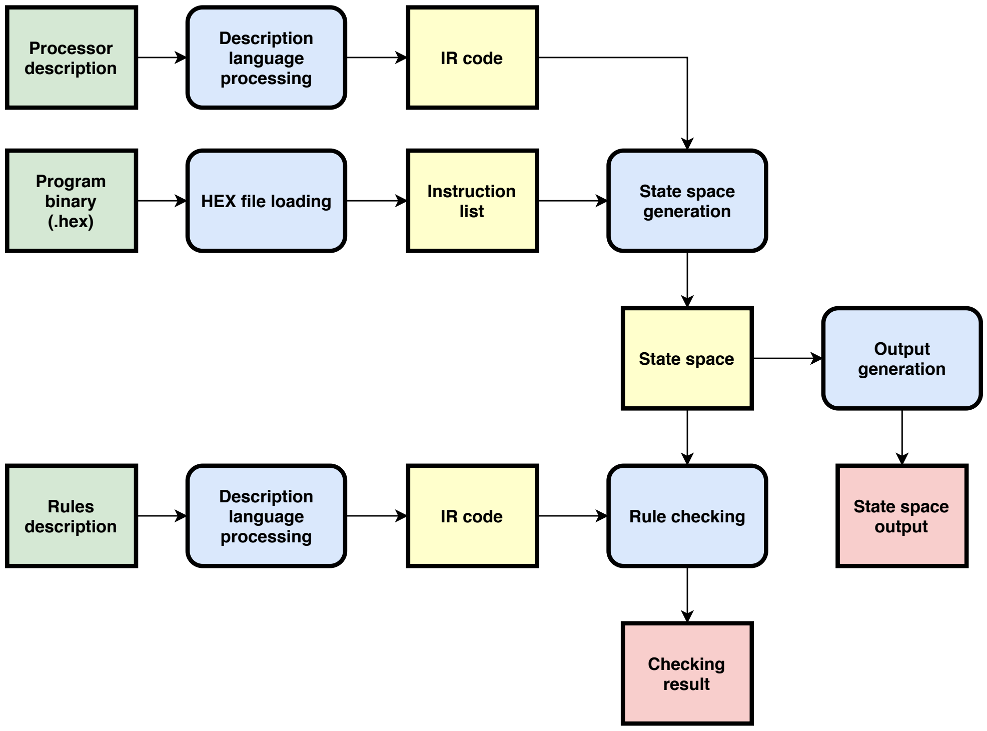

Figure 8.1: Top-level view of implementation inputs, outputs and core modules. Source boxes (green) represent inputs, sink boxes (source) represent outputs. Intermediate yellow boxes without rounded edges represent internal data structures and blue boxes with rounded edges represent manipulation.

7. If a deadline specification was loaded, deadline verification is performed. Concise results are output to console. Further details (e.g. counterexamples) are output for examination / visualisation if the user indicated the need.

8. The program returns zero (non-failure return) only if no error occured. If deadline checking was performed, the failure to prove at least one rule also results in returning a non-zero value.

8.1 Description language processing

In order to create a model checker easy to use with different processors and architectures, I placed a great emphasis on the ability to describe processors in a language that mirrors their usual datasheet or programmer's guide terminology. Furthermore, it should be simple enough to enable using enhanced model checking techniques in a fully automated way, without extra input from the processor description designer or user. Current industry-standard languages are not very suitable for this purpose as they are highly complex, often not fully specified (having undefined states and behaviours), and not offering constructs that are typically used in datasheets and programmer's guides.

For this reason, I designed a new processor description and deadline specification language for binary-level model checking use. It is an imperative language inspired by datasheets of simple microcontrollers in order to facilitate their easy transcription. It is also inspired by languages C and VHDL (Very High Speed Integrated Circuit Hardware Description Language) to provide a gentle learning curve for programmers familiar with at least one of these. This is how the language is used for describing processors (only a small part shown):

```
void memory() {
(...)
    Uint16 PC = 0;
(...)
    alias Uint8 DATA[0x0900];

    // --- General Purpose Registers ---
    Uint8 R[32];
    alias DATA[0x0000..0x001F] = R;
(...)
    // --- IO registers ---

    alias Uint8 IO[64];
    alias DATA[0x0020..0x005F] = IO;
(...)
    // Status Register
    Uint8 SREG = 0x00;
    alias IO[0x3F] = SREG;
(...)
}

(...)

// for instructions AND, EOR, OR
// R: destination register after being set
void set_status_logical(Uint8 R) {
    // Z - zero flag
    SREG[[1]] = ~R[[7]] & ~R[[6]] & ~R[[5]] & ~R[[4]] & ~R[[3]] &
        ~R[[2]] & ~R[[1]] & ~R[[0]];

    // N - negative flag
    SREG[[2]] = R[[7]];

    // V - two's complement overflow flag
    SREG[[3]] = '0';

    // S - sign flag
    SREG[[4]] = SREG[[2]] ^ SREG[[3]];
}
```

```
(...)

void step() {
(...)
Uint16 instruction = fetchInstruction(PC);
PC = PC + 1;
(...)
    masked_case (instruction) {
    (...)
        // AND
        "0010_00rd_dddd_rrrr":
                // logical and
                R[d] = R[d] & R[r];
                set_status_logical(R[d]);
(...)
        }
}
```

The processor description sample shows most of the description language syntax. While the syntax is supposed to be similar to the C language most microcontroller programmers are familiar with, it also features some constructs not found in the C language, yet highly helpful for transcription of microcontroller datasheets into their description.

8.1.1 Array slicing

Unlike the C language, the description language is able to manipulate arrays not only by indexing, but also by *slicing*, taking a part of the array for further manipulation. It is performed using a two-dots syntax akin to the Pascal language declarations. For example, `A[5..8]` returns a 4-element array, the first element of it corresponding to `A[5]` and its last to `A[8]`. Slices may be used as L-values, that is, `A[5..8] = B[0..3]` sets `A[5]` to `B[0]`, `A[6]` to `B[1]`, and so on. This is not very helpful on its own (at least for describing a simple processor), but allows for implementation of two constructs critical for easy construction of a processor description, *alias variables* and *bit slicing.*

8.1.2 Alias variables

In the AVR architecture (and many others), processor registers are named differently depending on the type of access. For example, the status register can be accessed by I/O register manipulating instructions using I/O register offset `0x3F`, but is called `SREG` when it fulfills the role of status register in instruction (and status flags are set, such as in the `ADD` instruction).

To allow simple descriptions, I devised an *alias variable* construct. It allows the same register to be described in multiple ways, making instruction set transcription straightforward and drastically reducing the possibility of an error being introduced. The alias variables construct is nicely paired with array slice assignment, making it possible, for example, to map the 32 data space adresses from 0x0000 to 0x001F to the 32 general purpose working registers of the microcontroller. After this is done, any read or write to DATA[a] with $0 \leq a < 32$ is applied to the actual variable R[a] instead.

For some registers, this is still not enough to easily ensure a sound processor description. For example, on ATmega328P, the PINx addresses return digital inputs value on read, but writing to them toggles the corresponding PORTx values by performing a bit XOR on them with the written value. This is further complicated by the fact that the PINx addresses may also be interacted with using their IO port address or their data memory address. I devised another construct, *alias functions*.

Instead of assigning to the variable, the alias may be assigned functions that execute when reading and writing it. For the aforementioned PINx adresses, this solves issue of soundness. Special variables are created for the actual pin inputs. When reading from PINx, these variables are read. When writing to PINx, the write function instead performs a bit XOR with the corresponding PORTx. As the alias variable itself may be assigned to other aliases (IO registers and data memory), all interaction types are neatly handled.

This construct is fundamental to future extensibility of the checker as it separates description of the core processor and its instruction set from interaction with peripherals in address space. Crucially, it does this while keeping the same imperative paradigm, not creating further complexity.

8.1.3 Individual bit indexing and slicing

Another construct highly useful for simple transcription and its inspection is the possibility of working with individual bits of integer variables. This construct uses doubled brackets in order to be recognizable from regular array indexing or slicing. Representation using bits is similar to that of the programmer's level references. In the code example of processor description, the AND instruction is taken almost directly from the AVR instruction set reference [26, p. 35], just without aliasing the SREG bits (which could be technically done, but I chose not to in order to reduce the number of single-letter variables).

8.1.4 Control flow

A masked case construct is supported, intended to be used especially for branching on instruction code. The selected branch has its '0' and '1' characters corresponding to bits set to zero or one. The letters in the branch name correspond to variables which are to be created with their values. This corresponds to the traditional way of describing instruction opcodes, used in the AVR instruction set reference [26].

The language also contains if branches based on value of a single-bit variable (not shown in the code listing). It does not contain any looping constructs. Recursion is prohibited. This guarantees that the description language functions always halt, simplifying higher-level analysis.

8.2 Language processing

The language is read and transformed into Abstract Syntax Tree (AST) representation using a classic lexer and recursive descent parser combination. The AST is not used directly, but generates an Intermediate Representation which contains simpler constructs than AST and therefore is more easily analysed with advanced techniques.

8.3 Description language intermediate representation

The processor description and specification rules are ultimately converted to Intermediate Representation (IR) that can be directly used to transform a (general) state into another state.

The IR is intentionally designed to be as simple as possible to facilitate easy high-level analysis. It is of Static Single Assignment form and only contains a few types of simple instructions. They can be split into two types, assignment and control flow.

Assignment instructions always assign to a single variable. They differ mainly in the right side behaviour:

- Constant assignment: a constant value is assigned.
- Unary operation assignment: the result of an unary operation is assigned. Simple assignment is considered to be an identity unary operation.
- Binary operation assignment: the result of a binary operation is assigned.
- Assignment with right indexing: right-side variable is indexed or sliced according to indexing variables and the result is assigned.
- Assignment with left indexing: right-side variable is assigned to a slice or an index of the left-side variable according to indexing variables.
- External function call assignment: an external function call (specially implemented within the checker tool) is performed with specified variables as parameters and the result is assigned.

There is only one control flow instruction:

- Branch: contains multiple branches, each is taken based on its single-bit variable. Also contains a default branch, taken when no normal branch is taken.

No simple block instruction is needed as the IR is automatically generated and thus is not subject to scope problems. A loop instruction might be added in the future if it becomes needed, at the price of more problematic checking.

While internal functions are supported by the description language, they are always inlined in the IR representation to facilitate better high-level analysis. This is not a memory problem as the auxiliary functions are usually quite simple (e.g. setting eight status flags). As the description language does not support loop constructs nor recursion, IR control flow is extremely simple and suitable for use of enhanced techniques.

8.4 Program binary loading

The program loader loads the binary version of the program from a file. Currently, this is an Intel HEX file that is used for programming AVR devices. The contents of program memory of the microprocessor are replicated to be used for state space generation.

8.5 State space generation

The state space structure is explicitly generated via classic breadth-first search as described in [27, p. 156]. Since the whole structure is always generated (checking is only performed afterwards), depth-first search (DFS) and breadth-first search (BFS) result in the same structure. The main difference lies in the assignment of identifiers to states. If sequentially-increasing identifiers are assigned to states as they are generated, this results in BFS assigning sequential integers to successor states of any state. This makes it easy to follow the flow in the state space graph if it is visualised with such identifiers as labels. DFS, on the other hand, usually results in very disproportionate identifiers of such sibling states.

Alternatively, checking could be performed on-the-run together with state generation to leverage early termination in case of a counterexample. I did not choose this approach as it has a disadvantage in inability to perform their meaningful separation resulting in higher implementation complexity. It is also incompatible with the focus on visualisation: the whole state space structure should be visualised and the counterexample marked to provide the insight into program execution. Furthermore, this does not help at all with cases where no counterexample is found. This led me to separate these functionalities altogether.

8.6 State space representation

Program execution state space is represented explicitly in a structure similar to a Kripke structure. Unlike the traditional space spaces based on Kripke structures that consist of a directed graph where only vertices are stateful, both vertices and edges are stateful in my implementation. This is chiefly due to the need for a variable holding the number of cycles taken during a transition between two processor states (i.e. vertices in state space graph). The alternative approach would be to store this information directly in target vertex (it cannot be stored in source vertex as the source vertex may have multiple successors with different transition times), but that presents a problem with visualisation and simplification since multiple vertices with same processor states but different transition times would not be equivalent. Using stateful edges is clearly a more natural choice as the transition time is related to edge rather than vertex.

I use the same state representation for vertex states and edge states. This leads to high generality as the edges are not restricted to cycle counts, enabling them to carry more information. It also drastically simplifies the processing of the processor description. The only added complexity is in determining which variables belong to edge and which belong to vertex when stepping. I solved this problem in the following manner:

1. The first vertex is generated by executing the IR of function `memory()` on an empty state. The variables in the function scope are retained to form the first vertex.
2. When generating a new state, function `edge()` is executed on an empty state. The variables in the function scope are retained to form the basis for the new edge state.
3. The parent vertex state and edge state are merged. Function `step()` is executed on this state, resulting in the merged final state of new vertex end edge.
4. The variables are split and assigned to their respective locations (vertex/edge).

This ensures that the `step()` function can manipulate both processor registers and cycle count. Furthermore, this can be easily used for cleanly “ignoring” some vertices, not putting them in the state graph but retaining the cycle count elapsed. This is used in path reduction improvement in Section 9.4.

8.7 Output generation

If specified by user, the tool generates output that can be loaded into Wolfram Mathematica [28]. In addition to the state graph, it outputs program counter value for each vertex and cycle count value for each edge. This information may be used for visualisation as was done in Figure 6.1. This allows one to obtain a concrete idea about the program execution.

Other visualisation formats may be added in future. Due to the amounts of data that may be output, a binary program may be a highly reasonable choice. Considering that the state space may become highly complex for a human to understand, it seems that a visualiser that could reduce the complexity by e. g. temporarily merging states differing only by unnecessary variables might be useful.

8.8 Deadline specification

The deadline specifications are written in the same language as the processor model. This has a distinct advantage of being able to use all of the internal processor registers and aliases.

Each rule has four functions which control its behaviour: premise, premise duration, eventual consequence and eventual consequence deadline. The returned value is specified in variable named `result`. The premise duration and eventual consequence deadline are supplied with an empty previous state and thus effectively return a constant value.

The deadline is specified in this manner: on every possible execution path from each starting state, if the premise holds for at least the amount of cycles specified by premise duration, the eventual consequence must hold at least in one state before deadline is reached, that is, the path between the starting state and the state fulfilling the eventual consequence takes lesser or equal amount of cycles than the eventual consequence deadline.

I use such a definition since it allows practical checking of responses to digital inputs. In case the input changes to a new value and the program should respond to that, the input must stay at this value for a specified amount of time to give the program time to finish its previous work, fetch the digital input state and decide how to react. If the premise duration would be zero in this case, a counterexample would be generated in which the input would flip to its old value before the digital input state would be fetched.

An example follows with some of the rules for checking a program which should set port B bit 1 as a digital output in its initialization and then loop, checking the input value of port B bit 0 and setting the output value of port B bit 1 accordingly.

```
void rule_1_premise() {
        // always true
        Bit result = '1';
}

void rule_1_premise_duration() {
        // instanteneous
        Uint16 result = 0;
}

void rule_1_eventual_consequence() {
        // DDRB bit 1 becomes 1
        // should always happen -> rule should pass
        Uint8 ddrB = IO[0x04];
        Bit result = ddrB[[1]];
}

void rule_1_eventual_consequence_deadline() {
        // set a reasonable deadline
        Uint16 result = 30;
}

(...)

void rule_5_premise() {
        // PINB bit 0 becomes 1, this could happen
        Uint8 pinB = IO[0x03];
        Bit result = pinB[[0]];
}

void rule_5_premise_duration() {
        // set a reasonable duration
        Uint16 result = 25;
}

void rule_5_eventual_consequence() {
        // PORTB bit 1 eventually becomes 1
        // this should pass, but will produce a false
            counterexample
        // if the checker does not support unknown value
            propagation
        Uint8 portB = IO[0x05];
        Bit result = portB[[1]];
}

void rule_5_eventual_consequence_deadline() {
        // set a reasonable deadline
        Uint16 result = 30;
}
```

This could be extended in future by allowing the language to describe the path in a more imperative fashion. Some kind of temporal logic might also be used, but this would require the users to be familiar with its syntax and semantics, which is the thing that I would like to avoid.

8.9 Deadline checker

For each deadline specification rule, the deadline checker checks each state.

If the premise may be fulfilled (i. e. it is true or unknown, not false), it creates a path starting with it and pushes it to queue. If not, it searches for successor states that fulfill the premise; for each one found, a path is created containing the current state followed by the successor. This is done to add paths where the premise may be fulfilled between the states.

The deadline checker then takes paths from queue until it is empty (using depth-first search for fast counterexample generation) and for each path:

1. If the premise is definitely not fulfilled in some path state (it is *broken*) and the length of the premise holding is smaller than required premise duration, the path cannot be a valid counterexample (and so cannot be other paths starting with it). Its processing ends.

2. The path is now definitely valid (there are no other constraints on its validity). If it is longer than the current worst-case path, it becomes the worst-case path.

3. If the current path is at longer than the deadline, it is a counterexample.

4. If the current path ends with a state that has been on the path previously (forming a cycle), it is a counterexample.

5. If the eventual consequence is definitely fulfilled, the path may not be a counterexample (and so cannot be other paths starting with it). Its processing ends.

6. If the current path is exactly as long as the deadline, it is a counterexample.

7. The path is valid but not a counterexample; this means paths starting with it may form a counterexample. Generate longer paths by adding all possible transitions from the last state of the original path as the new last node.

The implemented deadline checking algorithm supports both of the “classic” strategies of breaking on a counterexample, bounded (by reaching a maximum path length) and unbounded (by cycle detection).

The reason for checking deadline length twice is as follows. If the path is longer than deadline, fulfilling the consequence is always too late. However, non-positive transition lengths are prohibited in the implementation when checking deadlines (producing an error if such transition is detected), since they may lead to dangerously counter-intuitive behaviour. This means that the path can be also considered a counterexample when it is exactly as long as the deadline and the consequence is not definitely fulfilled, since any longer path would fail (and at least one path must exist). This is desired especially due to the special case of zero-length deadlines which intuitively mean that if the premise holds, the eventual consequence must hold in the same state. In such case, the counterexample path length is reduced.

Concerning the counterexamples found by cycle detection, the fact that their path forms a cycle means that it could be added to the end of the path infinitely many times. Such a cycle could not break path validity. Either the required premise duration was already fulfilled, in which case the path validity cannot be broken anymore, or all path states possibly fulfill the premise and the cycle states, being their subset, must possibly fulfill it as well, being unable to break the path validity. Furthermore, the cycle cannot definitely fulfill the eventual consequence, since that would require some cycle state to definitely fulfill it, but that is impossible since it already is on path.

If desired by the user, the checker results are output in format suitable for Wolfram Mathematica [28] for visualisation purposes. If the rule holds for all paths, the worst-case path before the eventual consequence was fulfilled (on paths where premise was valid) is output. If a counterexample is found, it is output.

Unfortunately, the naïve approach used suggests a $O(n^2)$ processing time based on the number of states, due to paths starting from each state and being bounded by the state space size at most (since cycles cannot occur). This assumption is shown to be true in Chapter 10, severely reducing checking effectiveness. A more advanced checking approach may be implemented in the future to reduce the severity of this problem.

Chapter 9

Advanced techniques implemented

In addition to the core deadline checker modules, I implemented some advanced techniques for evaluation and use. New techniques may be added to the checker with a moderate amount of work.

9.1 Decision value propagation

In Subsection 8.6, an *aggressive nondeterminism* technique used by the deadline checker is described. Simply put, nondeterministic bits are propagated when assignment and bit operations are used according to three-valued logic. When value determinization is performed, the relationships to previous variables are lost. This is especially problematic because the Intermediate Representation interpreted typically has computations with intermediate temporary values inserted, especially when alias variables are used.

To counter this behaviour, I implemented a scheme where after each determinization is performed and multiple states created, each state has the actual value propagated backwards through assignment, unary negation and indexing operations in the current IR block. The corresponding variable values are set. Most importantly, this allows the conditional control flow instructions that depend on a bit in I/O space being set to propagate this bit back to the proper I/O address.

This technique has a similar concept to delayed nondeterminism introduced in [29], but is different in some aspects. The delayed nondeterminism technique is concerned with successive states, while decision value propagation currently only supports propagation within a single state. Delayed nondeterminism also does not propagate the determinized value backwards.

9.2 First relationship instantiation

The overapproximation created by nondeterminism is present due to the fact that nondeterministic values are copied [29, p. 187]. The relationship between the values in different variables (they must be equal to the same value in the same execution) is fundamentally lost.

Temporary variables in IR are not "interesting" to us as their value is not preserved in the state space. Constant values are also not interesting since they do not infer a "relationship" between two variables. This is why I devised a *first relationship instantiation* strategy to allow the checker to work very much like an *immediate instantiation* strategy checker but retain the advantages of IR-based processor description.

When data are taken from an "interesting" variable (named variable other than a formal function parameter), they have a "determinize" boolean flag set. This flag travels with the data through all uninteresting variables. When another interesting variable is found, determinization is performed immediately. This works very well when combined with decision value propagation, the program working almost as a normal immediate instantiation checker.

9.3 Automatic control flow computation

In classic static analysis and processor model checking, many high-level program transformation and simplification techniques use the control flow of the program, considering only the program variables that interact with it in order to build a "high-level" understanding of the program. Using the control flow, advanced manipulation and simplification techniques can be used.

While the compilation of a program to binary code removes a high amount of information about it, control flow usually can be easily reconstructed if there is not a high amount of nondeterministic manipulation (depending on data instead of instructions). Direct jumps, calls and returns do not depend on data. Conditional branch and skip instructions only depend on one data bit. In contrast, indirect branching, indirect function calls, etc. usually have an extremely high number of potential paths, but only a few are ever taken. The same problem may occur when the program counter (PC) or the program stack is manipulated. It is then extremely hard to generate a useful state space without combinatorial explosion occuring. Fortunately, simple programs on microcontrollers usually do not feature these constructs, making the control flow computation easy.

The control flow graph generation is implemented by using a separate processor description which eliminates variables other than those on which PC value depends. Since the PC is loaded by value from stack if the `RET` (return) instruction is encountered, this requires a further modification. I solved this problem by the processor description using a separate "virtual" stack when path reduction is used. In control flow graph generation, the "virtual" stack is used to push stack values whenever the `CALL` instruction is used and pop them whenever the `RET` instruction is used. This generates the expected control flow graph. The "virtual" stack is then also used when building the actual state space, a check being done whenever the `RET` instruction is used. If the popped stack value to be used for setting PC does not correspond to the "virtual" stack value, the checker is aborted as using this technique would produce wrong results.

9.4 Simple cycle path reduction

During the planning and implementation of the checker, I was especially concerned about time-wasting delay constructs. These constructs drastically increase the size of checked state space while usually not providing any useful information (besides the delay length). This is especially unacceptable when rule checking with superlinear complexity is performed. Furthermore, visualisation of the full state space becomes infeasible with any but shortest delays.

To counteract this undesired behaviour, I implemented a variant of *path reduction* originally introduced in [30] and discussed for use in binary-code checking in [31, p. 29-31]. As my ultimate aim in the future is full computation of behaviour of uninteresting cycles in constant time by high-level loop analysis, I restricted myself to replace the cycles introduced by time-wasting delays without impacting the other states. I also decided against eager path reduction since reducing too much could hamper visualisation and understanding of the program function. This is in contrast to previous approaches, which aim to reduce as much as possible.

Time-wasting delays only use a few registers which are typically preset to some values and then are changed within a simple loop until the ending condition is fulfilled. This is also true on the AVR architecture. The first task to be done is to identify these simple loops. This is done by building a control flow graph and searching for cycles which only have a single state with two successors; other states must have only one successor. The state with two successors form a "breaking point" as used in [30]. The question of whether the other states also form a "breaking point" (in which case, the technique could not be used on that loop) is rather simplified as no interrupts are allowed and no non-deterministic assignments are performed (this could change in the future as the checker is enhanced to support programs with interrupts and more complex peripherials, however).

The usage of path reduction for such a cycle would be prohibited if a variable used for deadline checking was manipulated. Guaranteeing this is not trivial if the checker is not supposed to know about the internal workings of the processor. To solve this problem, I devised a technique of IR *tailoring.* Each control flow state belonging to the cycle is an overapproximation of the corresponding execution state where only a few variables are known (namely PC and stack). This means it can be interpreted using the execution processor description in the same way an execution state would be (after appropriate variable remappings are done). When a branch instruction is encountered and the branch taken is well-known, it means that the other branches cannot be ever taken from the control flow state. This allows the IR to be tailored to the instruction, removing all other branches. This tailored IR only contains a small subset of variables used. All values that are assigned to are then recovered. If any one of these variables is used in the specification, the cycle is not enabled.

9.5 Simple cycle tailored IR stepping

As the path tailoring considerably simplifies the IR, especially removing the computation of the proper instruction that should be executed after the instruction data are fetched, I devised a technique where this IR is used instead of normal IR when stepping simple cycles.

Obviously, it would be even better to compute the cycle values in constant time by combining all of the tailored IR and performing advanced static analysis on it. That falls outside the scope of this thesis, but may be a very suitable enhancement in the future.

Chapter 10

Testing the deadline checker

As the verification tool may be used to verify the behaviour of real-world programs, proper testing is crucial. I used a processor description developed for the ATmega328P microcontroller in conjunction with custom-written example programs to test the checker functionality and the impact of different advanced techniques implemented.

10.1 Testing methodology

The checker is subject to two fundamental types of requirements: qualitative and quantitative. Both are interrelated: a wrong answer requiring little computational resources is not useful (and, worse, is misleading). Similarly, a perfect answer that requires more computing resources than available is also not useful.

In formal verification, the checker must be, above all, sound, i.e. never allow specification to pass the check if the actual system does not fulfill it. While this largely relies on the properties of algorithms used, the implementation itself may contain bugs. As formal verification of the deadline checker is infeasible in the scope of this thesis, thorough testing is needed for a sufficiently high degree of trust in the quality of the checker. To facilitate this, I test scenarios where failure is expected. It is paramount that these scenarios end in failure.

The other qualitative aspect is the prevalence of false negatives. While a checker that always returns a failure is trivially sound, it is not useful at all. I did not seek to eradicate all false negatives, instead focusing on the ones that may occur with specifications of simple, real-world programs. I also elected to return a negative checking result in other potentially dangerous cases not usually covered by specifications, namely when uninitialized memory is used. For these reasons, I aim to test the prevalence of false negatives on programs that might very well be used in real-world applications. Ideally, the deadline checker should not produce a false negative on most reasonable specification rules in such a program.

The quantitative aspect of checking is also important. The programs should be checked in seconds to minutes in order to support the rapid development cycle common for simple microcontroller programming. The most problematic aspects of non-parallel programs are nondeterministic inputs and busy delays. To determine their impact on resources used, programs for testing these behaviours are analysed in a quantitative manner.

I used a standard personal computer with an AMD Ryzen 5 1500X processor to compute the checking results. The result therefore should match the experience of standard users of the deadline checker. Full optimizations were used and a proper log level was set in order to eliminate the possible dependence on I/O operations.

10.2 Programs tested

The restriction of peripherals to general digital inputs and outputs (GPIO) and restriction to disabled interrupts severely limits the amount of useful real-world programs that could be used by students and hobbyists in real world. Some programs do not need other peripherals, however. The digital inputs allow for various buttons, switches and keypads to be used as inputs. To provide some real-world-like programs for testing, I was inspired by actual real-world microcontroller programs.

To have proper control over the program behaviour, I implemented these programs in assembly language. This is in contrast to the C programming language used for the basic branch program shown in Chapter 6. The basic branch program was technically also used for testing, but its trivial nature is not particularly interesting for further consideration.

10.2.1 Expected-failure programs

To obtain at least some information about soundness of the checker, it is important to know that illegal operations fail. This yields some confidence in the belief that the checker does not permit ill-formed programs or programs that use unimplemented constructs to pass checking.

I created simple programs that implement some behaviours that should result in a checking failure:

- Jumping outside program instructions.
- Processing an instruction that is unimplemented in the processor description.
- Enabling the global interrupt flag. Interrupts are currently unimplemented.

- Setting a value using an undefined value. This occurs when uninitialized memory is used. This may help with finding of bugs.
- Getting a value from a nonaliased location (i. e. a part of memory that is not described).
- Setting a value from a nonaliased location.
- Setting a value to a bit restricted for writing. AVR architecture typically has unused bits of registers in 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.

The checker outputs a corresponding failure when the state space for the programs is being built. This is a welcome behaviour, paving the way for the possibility that no false positives are generated by the checker for such violations.

10.2.2 Blink

To test the checker for behaviours not dependent on nondeterminism, a classic "first introduction" program is used. It sets a pin to output mode and then periodically toggles it, using a busy delay loop to slow the toggle rate to allow the human eye to recognize the toggling as discrete.

In addition to its ubiquity as an example program, another reason for the inclusion of this program is its applicability to measurement of quantitative dependencies on delay length. Its serial nature also allows easy crosschecking of results with a processor simulator.

For this program, this specification is checked by the deadline checker (the pin used for output is shortened to O):

- O is never in internal pull-up mode.
- O is set to output mode in a specified amount of cycles.
- If O output value is set to 0, it must become 1 within a specified amount of cycles (toggle period).
- If O output value is set to 1, it must become 0 within a specified amount of cycles (toggle period).

The deadline checker is unable to check if the actual toggle period is lower than that specified one (in another words, to check hold time). It is also unable to check that once O is set to output mode, it never goes back to input mode. This is a limitation of the current deadline checking algorithm and the specification format, not an inherent problem of the checker. In future, the checker may be extended to support more powerful specifications.

An important implementation decision in this program was to use pin toggling by writing a corresponding bit on PINx to 1. This is a corner case that could not be resolved soundly without introducing alias functions. This program therefore also tests this functionality.

10.2.3 Gate array

This program makes the microcontroller emulate a discrete gate IC with five classic logic gates (buffer, inverter, OR, AND, XOR). For each gate, it checks the values of input pins, computes the gate output and sets the output pin to that value.

This testing program is chosen to thoroughly test the ability of the checker to work with unknown values. Since delayed nondeterminism is not implemented, the program is implemented by branching on concrete bits in data registers. This keeps state explosion at a reasonable level for evaluation.

The specification is considerably more complex than the one for the Blink program. First, for every pin O used for output:

- O is never in internal pull-up mode.
- O is set to output mode in a specified amount of cycles.

Similarly, for every pin I not used for output:

- I is always in input mode.

That forms the input direction specification. Fortunately, only 2 rules are needed in practice, one that describes the requirements that should be true globally (outputs never in internal pull-up mode, others always in input mode) and another which checks that all outputs are set (at the same time) within a specified deadline.

Then, for every gate and combination of its values:

- If the gate inputs are in the proper combination of values to produce a logic 1 a specified amount of cycles, the corresponding output value must become 1 in a specified amount of cycles.
- If the gate inputs are in the proper combination of values to produce a logic 0 a specified amount of cycles, the corresponding output value must become 0 in a specified amount of cycles.

10.2.4 Switch with momentary selection

This program is directly inspired by microcontroller-assisted relay switching schemes used e.g. for guitar pedals. A momentary switch input controls the output while a separate alternate action switch determines the control type. One control type results in the momentary switch input value being copied to output, the other results in the input toggling every time the momentary switch is pressed.

In real systems, switches usually must be *debounced*, since their value usually does fluctuate for a few milliseconds after being toggled. For simplicity, the mode selection switch is implemented without debouncing. 10 millisecond debouncing is used for finding out if the action switch was pressed since the last time a check was done.

However, the time before the action switch value is copied to output when in momentary mode is required to be 5 ms or less. The specification contains two rules for checking this: if both the momentary mode is selected and the action switch is held at a certain value, this value must become present at output in 5 ms or less.

The debouncing behaviour is intentionally implemented by waiting in a busy loop for 10 ms when waiting for the button to debounce. The action switch is then checked once more to see if the logic 1 detected was not just spurious. This breaks the specification, since if both the mode selection switch is set to momentary and action switch value is changed, the system may fail to adhere to its specification. The checker therefore should return a failure, not proving that the specification holds.

10.2.5 Independent nondeterminism

As an objective measurement of possible worst-case generation speed and program memory usage behaviour is needed, I devised a program which uses single-bit branches to set different register bits in succession, forcing the determinization. This allows somewhat precise control of the number of states and does not result in sequential state processing, as with the Blink program.

Using instructions loading whole bytes together with first relationship instatiantion, described in Section 9.2, is not desirable in this case since the precise amount of determinized bits would not be controllable.

10.3 Qualitative testing

As the expected-failure programs are supposed to quickly fail (and they do), they are not discussed in this section. Rather, the actual usefulness of the checker for the user is discussed.

10.3.1 Blink

The Blink program does not contain non-determinism and its simplicity ensures that almost no false counterexamples to the specification are generated. The returned worst case path for transition from 0 to 1 is one cycle longer than in actual case. This is presumably due to the conservative path handling

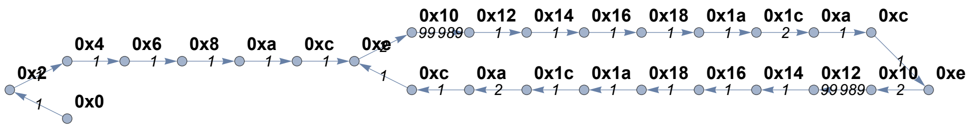
0x10 0x12 0x14 0x16 0x18 0x1a 0x1c 0xa 0xc
99989 1 1 1 1 1 2 1
0x4 0x6 0x8 0xa 0xc 0xe
0x2 1 1 1 1 1 2
1 0xc 0xa 0x1c 0x1a 0x18 0x16 0x14 0x12 0x10 0xe
1 2 1 1 1 1 1 99989 2
1 0x0

Figure 10.1: Execution state graph generated for the Blink program when simple path reduction is used. Values in bold are program counter values in corresponding states, values in italic are transition cycle counts. While it does not exactly describe the execution state space, it describes it extremely concisely and akin to actual programmer's view: the exact execution of delay cycles present is not important for understanding and only muddles it.

described in Section 7.2. The checker also properly signals when the specification is violated, e. g. by the toggling being slower than required. The deadline behaviour was crosschecked using an official processor simulator for ATmega328P present in Atmel Studio 7 [32].

This program is studied in bigger depth in the quantitative section, but using the simple cycle path reduction technique (discussed in Section 9.4) presents a big qualitative difference when studying or visualising the output. When simple cycle path reduction is used, the simplified state space is extremely simple and understandable, as seen in Figure 10.1.

10.3.2 Gate array

The Gate array test program is an interesting test of deadline checker usefulness. In Table 10.1, it is shown that the checker is not able to prove any rules concerning the gates without the decision value propagation technique described in Section 9.1. This makes the decision value propagation technique extremely useful, almost completely necessary. The state space does not grow significantly, but the added information about state of input pins is sufficient to allow proving the full specification.

Enabling the first relationship instantiation technique described in Section 9.2 did not have a noticeable effect on the output. This is expected as the program was written in a way that uses branching on bit value, which always results in deterministic value instantiation.

The speed of checking is problematic. Even though the program is written in a way that aims to keep the generated state space small, the number of states still increases rapidly with each new gate and checking becomes much slower.

I consider the evaluation of this test program to be a success. I am especially motivated by Table 10.2. Determining the contained information using a simulator or even through stepping by hand would be impractical and prone to error due to the large amount of execution paths. By using the deadline checker, guarantees are made automatically and easily.

techniques/criterion	global	gates	state#	step	check
no techniques	all pass	all fail	3799	16.19 s	1.33 s
decision value prop.	all pass	all pass	4229	15.48 s	487.13 s
d.v.p., first rel. inst.	all pass	all pass	4229	14.83 s	494.98 s

Table 10.1: Comparison of results of techniques used when the Gate array program is checked. For each combination of technique, the result of proving the "global" rules (setting pin as inputs and outputs), rules concerning each gate, number of states generated, time elapsed when generating the state space (stepping) and time elapsed when checking are shown. Decision value propagation allows the specification to be fully proven. Not using it results in a fast failure to prove. First relationship instantiation does not help further as there is no further nondeterminism present after decision value propagation.

Gate	Buffer	Invertor	And	Or	Xor
Worst-case path length	55	53	47	37	36

Table 10.2: Worst-case lengths of paths before satisfaction of rule eventual consequences for individual gates in the Gate array program. The paths are measured in processor cycles. The worst path length was selected from each set of rules for a gate.

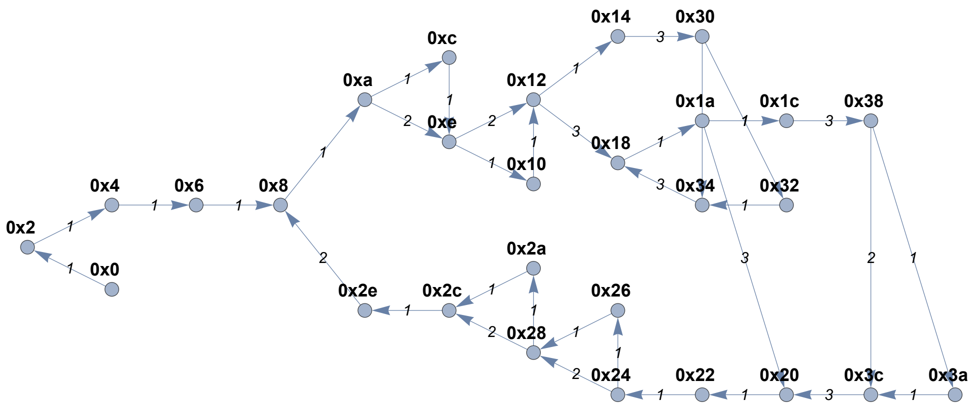

Figure 10.2: Control flow graph generated for the Gate array program using automatic control flow computation.

While simple cycle path reduction does not help with evaluation of the gate array since no valid simple cycles are found, control flow graph generation can be used for visualisation since the generated state space is too big and complex to be helpful to humans. Figure 10.2 shows the control flow graph generated.

10.3.3 Switch with momentary selection

The rules for guaranteed momentary selection fail as expected. A simple cycle (delay) is found if simple cycle path reduction is used, but it is not used since variables representing inputs present in the specification are manipulated by being set to a constant unknown value. This is a problem since the state space exhibits growth both due to the nondeterminism present and to the delay used. Otherwise, this program behaves as similarly to Blink and Gate array programs.

10.4 Quantitative testing

Quantitative testing was motivated by the desire for a lesser memory usage and (especially) processing time by using advanced techniques. It is shown that processing time is the main bottleneck in practice.

10.4.1 Blink

The Blink program is ideal for testing the dependence of execution space size and time elapsed on introduced delays. Usage of various techniques for handling nondeterminism does not have a significant impact, since no non-determinism (apart from undefined values in unused processor registers and memory) is present. Therefore, I further interpret the results depending on the usage of the simple cycle path reduction technique and varying delay length.

In Figure 10.3, time spent by state space generation and deadline checking is shown depending on varying delay length with simple cycle path reduction disabled. It is apparent that the deadline checking algorithm used has a quadratic dependency on the number of states. This is unacceptable if the amount of states is not extremely small. Unfortunately, busy delays increase the amount of states checked in a linear fashion, making the Blink program infeasible to check if the toggle delay is on the order of milliseconds. This means the checker cannot check the Blink program if it actually performs blinking (requiring delays on the order of hundreds of milliseconds) without simple cycle reduction.

Simple cycle path reduction helps immensely, as shown in Figure 10.4. The deadline checking time is negligible in all cases since the amount of states checked becomes constant. The states still are generated one by one, resulting in linear generation time. This enables the checker to check the Blink program in a reasonable amount of time. The time needed is further improved by a factor of 3 when IR tailoring is used.

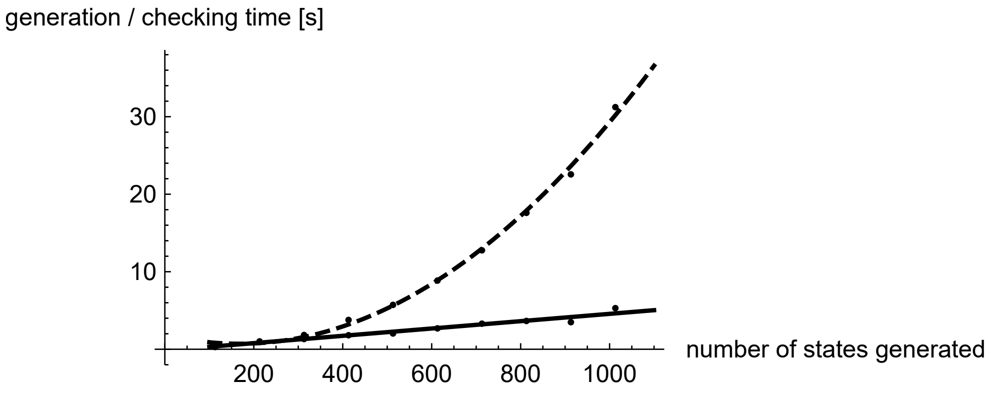

Figure 10.3: Dependence of state space generation time and deadline checking time on number of states for the Blink program with simple cycle path reduction disabled. The state space generation is solid black and shows linear dependency. Deadline checking time is shows quadratic dependency. Deadline checking time quickly dominates and becomes unacceptable. One millisecond of delay corresponds to 1000 states.

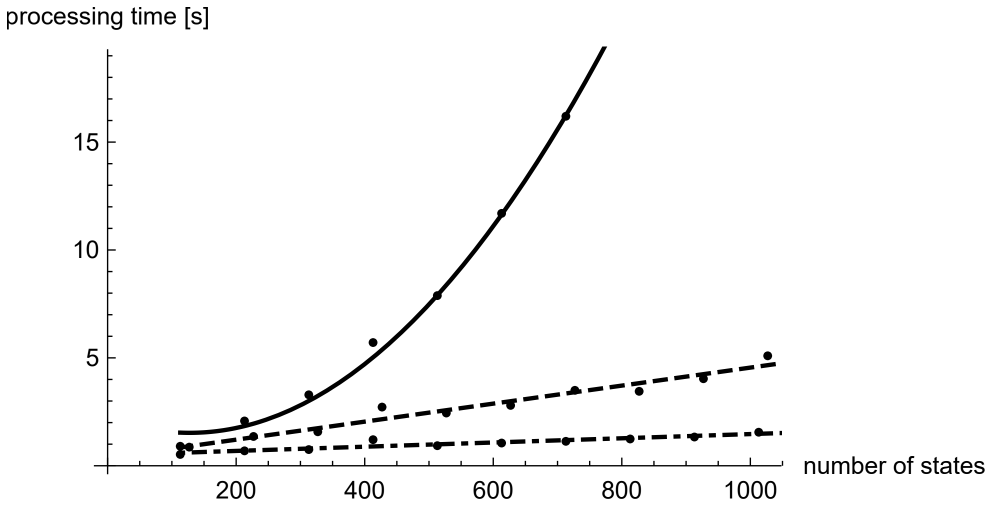

Figure 10.4: Dependence of full deadline checker processing time on the number of states when checking the Blink program. The case without using simple cycle path reduction is solid black. The dashed line represents simple cycle path reduction and the dot-dashed line represents simple cycle path reduction with IR tailoring.

10.4.2 Independent nondeterminism

I used the independent nondeterminism program to evaluate time and memory requirements when multiple states are generated from one. As expected, the amount of states generated roughly doubled with each nondeterministic bit introduced. Generation time remained strongly linearly dependent on the number of states, as shown in 10.5.

Unlike the previous test programs where maximum memory consumption increased insignificantly on the order of megabytes to low tens of megabytes, the memory consumption in this case exhibited a strong linear dependence on the number of states, as shown in Figure 10.6. This would correspond to memory consumption of each state being approximately 11 kB, which is larger than expected given the fact that the all processor memory used in description sums to approximately 2.1 kB and is expected to be doubled in the state representation. Despite this, memory consumption is clearly not the main target for future improvements.

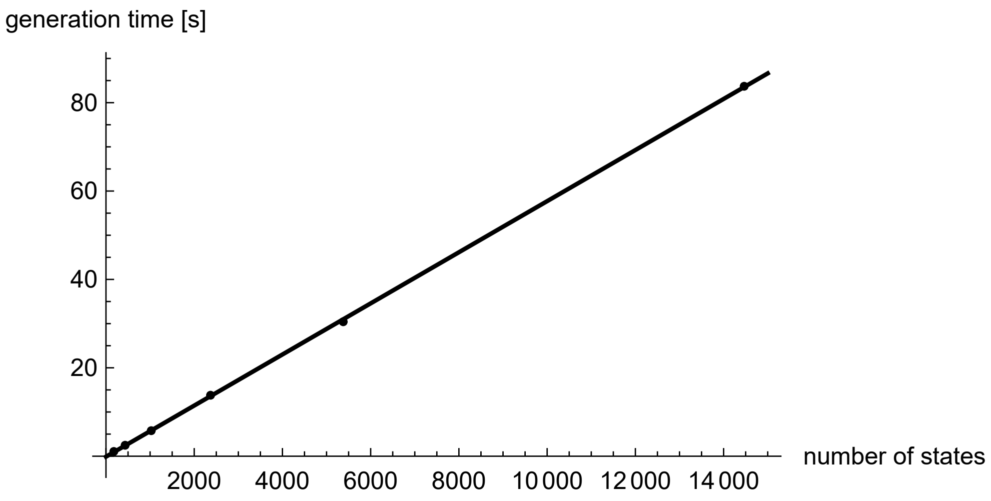

Figure 10.5: Dependence of state space generation time on number of states when the independent nondeterminism program is used. The time needed to generate one state remains constant.

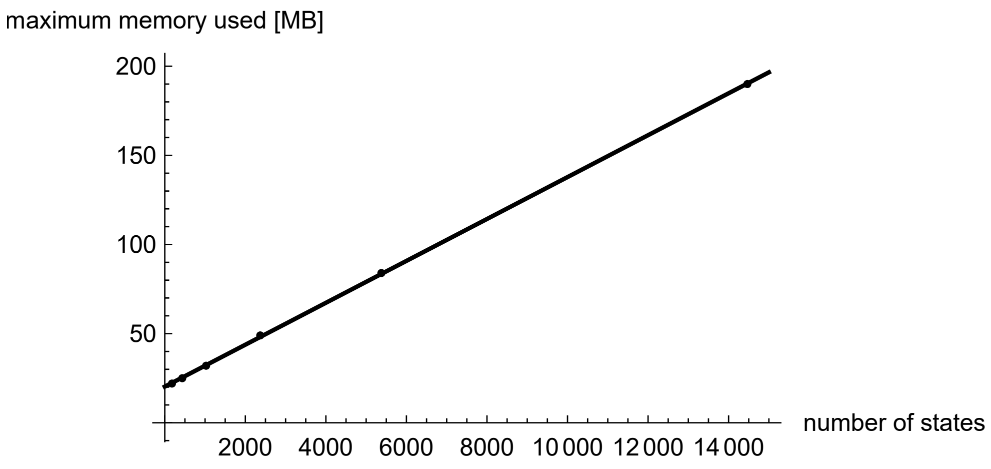

Figure 10.6: Dependence of maximum memory used on number of states when the independent nondeterminism program is used. The nature of successive states generating more states requires many states to be stored before they are processed. This is not the case with the other programs where the branching is not as extensive.

10.5 Interpretation of results

The checker can be reasonably used to check simple microcontroller programs. While the limited amount of testing done is not enough to assure typical users of its soundness, this could not be reasonably done in the scope of this thesis. I consider the checker in its current state to be a good first step towards a full-fledged, usable model checker.

The biggest problem encountered is the quadratic dependence of the rule checking routine on the number of states. This makes checking any advanced programs infeasible. This would require a new algorithm for deadline checking, but not necessarily requiring a change in other components. Pure processing speed should be also optimized.

Usage of nondeterministic overapproximation techniques severely limits the deadline rules that can be checked, but allows checking basic rules in simple programs. That may be useful e. g. for ensuring that microcontroller pins are always in proper modes, which may prevent possible damage caused by an inadvertent short circuit between two outputs. The technique of decision value propagation extremely helped in reducing overapproximation, making proving of normal deadline rules posssible. The first-relationship determinization technique was not very useful, either not improving upon decision value propagation or aggressively performing determinization without regard for state space size (avoided in the examples discussed by using single-bit nondeterminism). More advanced techniques are needed to properly balance the amount of overapproximation as required.

Simple cycle path reduction is a must for checking programs with any form of busy delays due to the shortcoming of the naïve deadline checking algorithm used. Simple cycle path reduction massively reduces the amount of states that must be checked, allowing checking of simple programs using busy delays in a reasonable amount of time. Currently, the states still must be generated in linear time, but the symbolic nature of the processor description in the implementation offers a highly attractive avenue of their computation in constant time. An improved handling of the question when a state may be used is also needed.

10.6 Comparison with other binary-level model checkers

While a comparison of the checker performance with other binary-level model checkers would be interesting, one needs to be very careful with proper interpretation, as all of the current present (including this one) are unique in many regards.

The State Exploring Assembly Model Checker (StEAM) discussed in Section 4.1 does not have almost anything in all common with the implemented deadline checker, as its purpose is not model checking of microcontroller programs, but of traditional computer programs. The binary-level approach is only used as a tool. Due to this, it is not sensible to compare it with the implemented deadline checker.

The Estes model checker discussed in Section 4.2 is more similar to the implemented deadline checker, model checking programs for Motorola 68hc11 processor by using a modified debugger `gdb`. The Estes model checker is reasonably fast, being able to check a program with 4443 assembly language lines and 31772 states in 45 seconds [17]. A comparison with the implemented deadline checker is problematic, but given the testing results, it is obvious that the deadline checker suffers from a fatal flaw in the quadratic complexity of checking. This does not seem to be present for Estes.

The same could be said for Arcade.µC discussed in Section 4.3. I consider the implemented deadline checker to at least have an advantage in the fact that it supports manipulation of time, allowing for checking deadlines, something not possible using Arcade.µC.

Conclusion

In this thesis, I presented the current state of research concerning verification of microcontroller programs with the focus on deadline verification. I identified the weaknesses of the current tools for this task, most notably the difficulty of enhancing them to support different processors and lack of availability to the general public.

Using this knowledge, I formulated the goals for a new model checker, mainly: it should be freely available, simple to use and easily extended to support new microcontrollers and verification techniques. This greatly increased the complexity of implementing it, but was worthwhile. I created a free, MIT-licensed model checker that allows easy transcription of processor descriptions from official datasheets to an imperative description language I devised. The language is then transformed to a simple Intermediate Representation language. This allows various verification techniques to be used with little to no dependence on the actual microcontroller.

In addition to the core model checking capabilities, I implemented some advanced techniques to enhance the capabilities of the checker. The technique of *simple cycle path reduction* is especially useful, allowing checking of simple programs that contain busy delays in reasonable amounts of time. Appropriately simplified state space of such programs may also be output to be visualised. This allows further understanding of program behaviour and may be a useful learning aid.

While the checker in its current state can be used to check simple programs, real-life programs are usually more complex, using advanced peripherals and interrupts. I intend to continue developing the created model checker in the future to further its usefulness. Currently, the field of binary-level verification is severely underdeveloped compared to source-level and Register Transfer Level verification, despite being able to guarantee program deadlines with cycle accuracy. This is crucial for developing more reliable, safe, and secure real-time systems. In my opinion, further research and development in this area is an objective need.

Bibliography

[1] Claus Beisbart. “What is Validation of Computer Simulations? Toward a Clarification of the Concept of Validation and of Related Notions”. In: *Computer Simulation Validation: Fundamental Concepts, Methodological Frameworks, and Philosophical Perspectives*. Ed. by Claus Beisbart and Nicole J. Saam. Cham: Springer International Publishing, 2019, pp. 35–67. ISBN: 978-3-319-70766-2. DOI: 10.1007/978-3-319-70766-2_2. URL: https://doi.org/10.1007/978-3-319-70766-2_2.

[2] Oxford University Press. *Oxford Advanced Learner’s Dictionary*. Accessed 2020-05-24. URL: https://www.oxfordlearnersdictionaries.com/.

[3] Real Time Engineers Ltd. *FreeRTOS*. URL: https://www.freertos.org/.

[4] Edsger W Dijkstra. “The humble programmer”. In: *Communications of the ACM* 15.10 (1972), pp. 859–866.

[5] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. “Introduction to Model Checking”. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 1–26. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_1. URL: https://doi.org/10.1007/978-3-319-10575-8_1.

[6] Robert W Floyd. “Assigning meanings to programs”. In: *Program Verification*. Springer, 1993, pp. 65–81.

[7] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[8] Nir Piterman and Amir Pnueli. "Temporal Logic and Fair Discrete Systems". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 27–73. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_2`. URL: `https://doi.org/10.1007/978-3-319-10575-8_2`.

[9] E.M. Clarke et al. *Handbook of Model Checking*. Cham: Springer International Publishing, May 2018, pp. 1–1210. DOI: `10.1007/978-3-319-10575-8`.

[10] Tom Melham. "Symbolic Trajectory Evaluation". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 831–870. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_25`. URL: `https://doi.org/10.1007/978-3-319-10575-8_25`.

[11] T. Reinbacher, M. Horauer, and B. Schlich. "Using 3-valued memory representation for state space reduction in embedded assembly code model checking". In: *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems*. 2009, pp. 114–119.

[12] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115. URL: `http://xavierleroy.org/publi/compcert-CACM.pdf`.

[13] Ken Thompson. "Reflections on trusting trust". In: *Communications of the ACM* 27.8 (1984), pp. 761–763.

[14] Y. N. Srikant and Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. 2nd. USA: CRC Press, Inc., 2007. ISBN: 142004382X.

[15] Robert P. Kurshan. "Transfer of Model Checking to Industrial Practice". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 763–793. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_23`. URL: `https://doi.org/10.1007/978-3-319-10575-8_23`.

[16] Tilman Mehler. "Challenges and Applications of Assembly-Level Software Model Checking". PhD thesis. University of Dortmund, 2006.

[17] E. G. Mercer and M. Jones. "Model Checking Machine Code with the GNU Debugger". In: *12th International SPIN Workshop*. Vol. 3639. Lecture Notes in Computer Science. San Francisco, USA: Springer, Aug. 2005, pp. 251–265.

[18] B. Schlich and S. Kowalewski. "[mc]square: A Model Checker for Microcontroller Code". In: *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*. 2006, pp. 466–473.

[19] Bastian Schlich. “Model Checking of Software for Microcontrollers”. In: *ACM Trans. Embed. Comput. Syst.* 9.4 (Apr. 2010). ISSN: 1539-9087. DOI: `10.1145/1721695.1721702`. URL: `https://doi.org/10.1145/1721695.1721702`.

[20] Dominique Gückel. “Synthesis of State Space Generators for Model Checking Microcontroller Code”. Dissertation. RWTH Aachen, Nov. 2014. URL: `http://aib.informatik.rwth-aachen.de/2014/2014-15.pdf`.

[21] Yajun Wu and Satoshi Yamane. “Model Checking of Real-Time Properties for Embedded Assembly Program Using Real-Time Temporal Logic RTCTL and Its Application to Real Microcontroller Software”. In: *IEICE Transactions on Information and Systems* E103.D.4 (2020), pp. 800–812. DOI: `10.1587/transinf.2019EDP7172`.

[22] Peter Grun et al. *EXPRESSION: An ADL for System Level Design Exploration.* Tech. rep. 1998.

[23] A. Smeda, M. Oussalah, and T. Khammaci. “MADL: Meta Architecture Description Language”. In: *Third ACIS Int’l Conference on Software Engineering Research, Management and Applications (SERA’05).* 2005, pp. 152–159.

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

[25] “IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)”. In: *IEEE Std 1164-1993* (1993), pp. 1–24.

[26] *AVR Instruction Set Manual.* 0856. Rev. L. Atmel Corporation. 2016. URL: `http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf`.

[27] Gerard J. Holzmann. “Explicit-State Model Checking”. In: *Handbook of Model Checking.* Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 153–171. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_5`. URL: `https://doi.org/10.1007/978-3-319-10575-8_5`.

[28] Wolfram Research Inc. *Mathematica, Version 11.3.* Champaign, IL, 2018.

[29] Thomas Noll and Bastian Schlich. “Delayed Nondeterminism in Model Checking Embedded Systems Assembly Code”. In: *Hardware and Software: Verification and Testing.* Ed. by Karen Yorav. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 185–201.

[30] Karen Yorav and Orna Grumberg. “Static Analysis for State-Space Reductions Preserving Temporal Logics”. In: *Formal Methods in System Design* 25.1 (July 2004), pp. 67–96. ISSN: 1572-8102. DOI: `10.1023/B:FORM.0000033963.55470.9e`. URL: `https://doi.org/10.1023/B:FORM.0000033963.55470.9e`.

[31] Bastian Schlich, Jann Löll, and Stefan Kowalewski. “Application of Static Analyses for State Space Reduction to Microcontroller Assembly Code”. In: *Formal Methods for Industrial Critical Systems*. Ed. by Stefan Leue and Pedro Merino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 21–37. ISBN: 978-3-540-79707-4.

[32] Microchip Technology Inc. *Atmel Studio 7.0.2389*. Champaign, IL, 2018.

Appendix **A**

Contents of enclosed SD card

```
 readme.txt ................................... contents description file
├─ exe ................. directory containing a deadline checker executable
├─ src ............................... source code of the deadline checker
└─ text .............................. directory containing the thesis text
   └─ thesis.pdf ......................... the thesis text in PDF format
```