FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Alberte Thegler - alberte@thegler.dk

Towards formal verification of FDR4

Department of Computer Science

Advisors: Professor Brian Vinter and Kenneth Skovhede

August 2018

Abstract

Abstract

Contents

Chapter 1

Introduction

When we create programs, we wish to verify that it is also correct. There are several ways to do this, one commenly used is `testing` which require that the programmer creates several different scenarios and its expected output, or that the programmer programs a test-generator to create the scenarios and expected output. This, however, is not adequate for critical systems since it is never a 100% accurate. Therefore it is of high interest to create a formal verification of the system or program.

In this thesis we look at model checking, that is, verifying that a specific property will always hold for a piece of code.

Formal verification is the process of checking whether a program satisfies specific properties. Different methods have evolved, all having different advantages and disadvantages. FDR is sometimes referred to as a model checker however is it actually a refinement checker.

"Matematicians tend to reject proofs by exhaustive checking of all cases as being less satisfying than deductive proofs, and with good reason. First, they are not applicable for proving theorems about integers and real numbers, which are infinite domains so that the number of interpretations is infinite and they cannot be exhaustively checked. Second, they offer no insight into why a theorem is true. But computer scientists have more practical concerns. If they can check all computations of a program and show that they all satisfy a correctness property, we will be willing to forego elegance and be more than satisfied that our program has been proven correct." from "A primer on model checking af Ben-Ari" [?]

1.1 Motivation

1.1.1 Ariane 5 failure

The Ariane 5 space rocket[?] was designed to launch large payloads into Earths orbit, such as communications satelites, etc. Ariane 5 was the follow-up on the sucessful Ariane 4 launchers. On june 4th, 1996, the Ariane 5 rocket had its first test flight. The rocket, which was owned by The European Space Agency (ESA) and the French spatial agency Centre national d'Ã©tudes spatiales (CNES) was manufactured by Airbus Defence and Space.
The rocket was launched in French Guiana, and only 37 seconds after successful lift-off, the rocket flipped 90 degrees and two seconds later the forces of aerodynamics ripped the boosters appart from the core stage. This caused the self-destruct mechanism to trigger and the rocket self-destructed in a giant explotion shortly afterwards.
This giant disaster cost approximately 500 million dollars and it was a huge loss for ESA and CNES.
It turned out that the initial reason for the failure was a software error that could have easily been avoided. Luckily the rocket was unmanned, but this kind of error could have happen in any other space rocket. This failure launch is acknowledged as one of the most expensive software failures in history.
The failure was caused by a software bug in the Inertial Reference System (SRI). The SRI system is used to determine the orientation of the rocket, e.i if the rocket is pointing up or down. This is also known as the horizontal bias or the BH value. The error occured when a 64-bit floating point number, representing the horizontal velocity, was converted to a 16-bit signed integer, without any exception handling on that piece of code, which happened to be code written in Ada, a language that other Hardware Description Languages (HDLs) have later been based on.
As the rockets velocity increased the 64-bit floating point number became to big to fit into a 16-but signed integer and which caused an overflow of the variable. The SRI system misinterpreted this as true flight-data, and to counteract the "wrong" direction, the engines thrusted to change course and thus it was ripped apart by aerodynamics. As there should be, there was a backup system that should take over when errors occur in the main system, however the backup system was running the exact same code and the main system and therefore it had failed, for the same reasons, just before the main system failed, causing the self-destruct mechanism to activate.
This is, in itself, a horrible situation that should never have taken place. However, it becomes even more horrible when it was discovered that the BH value was not neseccary after launch. The code had been reused from the Ariane 4 rocket, which required the value after launch, but Ariane 5 did not. The code that could have handled these problems had been disabled due to performance issues on Ariane 4 and had not been reapplied on Ariane 5. Also, Ariane 4 was launched with a less steep trajectory than Ariane 5, and therefore it did not overflow the BH value because the value never became large enough. However, since Ariane 5 ascent to space faster, it was highly probable that the BH value would overflow. If anyone had taken a look at this code, taken the new rocket into account, this massive failure could have been avoided.

1.1.2 Therac-25 failure

In the 80's the company Atomic Energy of Canada Limited (AECL) manufactured a revolutionary radiation therapy machine, the Therac-25[?], which could provide two different kinds of treatment. At that time, hospitals would typically have two different machines for the two different treatments that the Therac-25 machine could provide in one machine. The Therac-25 could provide two treatments, the first being a beam of low-energy electrons which used scanning magnets to spread the electron beam, and secondly a beam of higher-energy X-Ray photons which worked by rotated four components into the beam. The Therac-25 was build based on the previous Therac-20 and Therac-6 and some of the software from the Therac-20 was reused in the Therac-25. Unfortunately, there was no independent protective circuits for monitoring the electron beam or any interlocks to ensure safety with the Therac-25, which had been in the previous versions. AECL put more faith on software reliability than on hardware.

After the Therac-25 had been operational for a couple of years on several different hospitals, a series of incidents happened with the Therac-25 where patients were exposed to too much radiation and that led to six patients being seriously injured or killed. Friz Hager, the staff physicist at East Texas Cancer Center, did some investigation on his own and tried to reproduce the errors they had experienced on the Therac-25, and ended up being successful. When the user selected the X-Ray mode on the Therac-25, the machine began setting up for high-powered X-rays, which took about 8 seconds. If the user switched to Electron mode before the machine finished setting up for X-ray mode, e.i within 8 seconds, the turntable would not switch to the correct position causing an enormous amount of radiation to reach the patient. After an investigation started on the Therac-25 system, it turned out the the system was very poorly written and very unprofessionel, and the testing had not been adequate for a critical system like this.

After solving the problem and releasing a new version of the Therac-25, another problem emerged where a patient was overdosed. This time it turned out to be a counter overflow. If a command was sent at the exact moment the counter overflowed, the machine would not set op properly and again, resulting in an overdose.

After the incident with the Therac-25, it was found that some of the same software problems was found in Therac-20, but due to the hardware precautions on the Therac-20, the problems never occurred.

This example shows how a critical system was placed on a system that was not designed correctly to handle that kind of system and how testing was done so poorly that several people lost their lives.

herac-20 has indepen- dent protective circuits for monitoring electron-beam scanning. p l us mechaniÂ cal interlocks for policing thc machine and ensuring safe operation

1.2 Learning goals

The learning goals accepted for this project are:

- Reflect on the set of problems that are verifiable with FDR4.

- Reason about efficient code transformation from an executable format to a verifiable format
- Reason about design choices and their consequenses for execution performance.
- Demonstrate efficient constraint transfer from SME to FDR4.
- Reason about SME program size and time to verification.

Chapter 2

Related work

The concepts of formal verification was first expressed in 1954 when Martin Davis created the first computer generated mathematical proof that the product of two even numbers, is even. First-order theorem provers were applied to verification problems in Pascal, Ada and Java, in the late 1960s. At Stanford, in 1972, Sir Robin Milner had success building the original LCF system for proof checking. His work in automated reasoning have been the foundation for a lot of other theorem provers, like the proof assistant HOL (Higher Order Logic) by Mike Gordon, which was originally developed for reasoning about hardware. The formal proof management system Coq is a descendent of LCF.
Also in 1972, Robert S. Boyer and J. Strother Moore was successful in building a machine-based prover, called Nqthm which became the basis for ACL2 which is a programming language and a theorem prover. Theorem provers have proved very valuable over the time, but one problem with them was, that if they found a problem in a theorem, they could not tell why it could not prove the theorem. It was not possible to create a counter example or any other explanation as to why it was not possible to prove this theorem.

In 1967, when Robert W. Floyd was published with the paper *Assigning meaning to programs*[**?**]. Floyd provided a basis for the formal definitions of the meaning of programs which can be used for proving correctness, equivalence and termination. By using flowcharts, he argued that when a command is reached, all previous commands will have been true as well.
C.A.R Hoare was inspired by Floyd and in 1969 his paper *An axiomtic basis for computer programming*[**?**] was published. The logic he presented there (later known as *Hoare logic*), was build on Floyd's ideas and proposed the notation *Partial correctness specification*; $\{P\}C\{Q\}$. Here, C is a command and P and Q are conditions on the program variables in C. Hoare showed that whenever C is executed in a state that satisfies the condition P, and if the execution terminates, then the state that C terminates in, will satisfy Q. Hoares logic have been the basis of a lot of different formal languages and have contributed to the continuous work on formal verification.
Since the original Hoares logic was not originially thought as to model concurrent programs, L. Lamport extended Hoare's logic in his paper *The 'Hoare logic' of concurrent programs*[**?**] in 1980. Here, he discuss why Hoare's logic, as proposed by C.A.R Hoare, does not work for concurrent programs and proposes

a "generalized Hoare's logic" that takes concurrency into account.

In 1978 Hoares paper *Communicating Sequential Processes* was published and with it, CSP was born. It have been widely used in many different types of work and have also been expanded since Hoare initially described it in 1978[?]. The first version of CSP was a simple programming language that had quite a different syntax than todays CSP. In 1984, Brookes, Hoare and Roscoe published their continued work on CSP with the paper *A Theory of Communicating Sequential Processes*[?], and created the modern process algebra it is today. Only a few minor changes have been made to CSP since then, and they are described in Roscoe's *The Theory and Practice of Concurrency*[?].
A number of tools have been created in order to analyse, verify and understand systems written in CSP. Since CSP was mostly a blackboard language and difficult to use on larger scale, different types of machine-readble CSP syntaxes have been created over the years in order to make it easier to use CSP on a larger scale. Most of todays CSP tools use a version of machine-readble CSP called CSP$_M$ which was created by Scattergood[?]. Scattergood created a combination of the standard CSP and a functional programming language which created a better baseline for tools to work with CSP.
Here is a subset of the different CSP tools:

- One of the most known CSP tool is the Failure-Divergence Refinement (FDR), build by Formal Systems (Europe) Ltd., which is currently at version 4.2.3[?]. FDR is a refinement checker and the newer version of FDR is able to run in parallel as well as do state compression in order to avoid a very large state space. FDR only work on finite-state processes.

- ProBE (Process Behaviour Explorer)[?] is a tool to animate CSP in order to explore the state space of CSP processes. It can handle infinite state and is based on the same CSP$_M$ version as FDR is. ProBE was also been created by Formal Systems (Europe) Ltd that created FDR and ProBE is integrated into the current version of FDR.

- At Adelaide University, The Adelaide Refinement Checker (ARC)[?] was created. It is a automatic verification tool for CSP that uses Ordered Binary Decision Diagrams (OBDDs) to represent the internal representation of data structures. This lessen the state explosion problem that other model checker tools have had.

- The ProB project[?][?] was originally created as an animation and model checker tool for the B-Method[?] but it also supports other languages like Z and CSP$_M$. Newer versions of ProB can do refinement checking of CSP$_M$ scripts but does not have the full functionality that FDR does.

- J. Sun, Y.Liu, J.Dong et al. presented the Process Analysis Toolkit (PAT) in their 2009 paper[?]. PAT is a CSP analysis tool that can perform Linear Temporal Logic (LTL) model checking, refinement checking and simulation of CSP processes.

- CSP-Prover[?] is a theorem prover which works on CSP and based on the theorem prover Isabelle. It is an entirely different way to check programs than model checking. It attempts to prove some general results based on specific theory. It is better at proving general results where model checkers are better at proving combinatorial problems.

The programming language Occam[?], which was first released in 1983, is a concurrent programming language that builds on the CSP process algebra. Occam was continuouly in development during the years and the Kent Retargetable occam Compiler (KRoC) team at Kent University created the Occam-π[?] variant of the Occam programming language. It is a version that extends the ideas of CSP in the original Occam language but adding mobility features from pi-calculus. In the paper *The symbiosis of concurrency and verification: teaching and case studies*[?] Pedersen and Welch uses Occam-π along with CSP$_M$ in order to reason about the logic behind CSP$_M$ and FDR. By using an executable language like Occam-π which is based on the concurrency model of CSP it becomes easier to understand the logic of CSP$_M$ and thereby verify the program with FDR.

SPIN[?] is a verification tool that uses process interactions to prove correctness for a system. The systems are described in the formal language `PROMELA`(PROcess MEta LAnguage)[?] and the correctness properties are spcified in Linear Temporal Logic (LTL)[?]. In the paper *Reasoning About Infinite Computations*[?], Vardi and Wolper showed that all LTL formulas can be translated into a Büchi automata which SPIN makes use of and thus converting the given LTL into a Büchi automaton. Spin performs verification on concurrent software and does not perform verification on hardware circuits.
Spin was developed at Bell Labs, starting in 1980. Gerard J. Holzmann gives an introduction to the theoretical foundations, the design and structure and examples of applications in the paper *The model checker SPIN*[?]. SPIN, as well as other model checker tools, has been build on the pioneering work on logic model checking by Clarke and Emerson[?], as well as Sifakis and Queille[?]. Vardi and Wolper extended their work with an automata-theoretic approach to automatically verify programs[?].

Another verification tool was developed as a collaboration between the Department of Information Technology at Uppsala University (UPP) in Sweden and the Department of Computer Science at Aalborg University (AAL) in Denmark. Larsen et al. first proposed the ideas for UPPAAL[?] in 1995 and further introduced it in the paper *UPPAAL - a Tool Suite for Automatic Verifcation of Real-Time Systems*[?]. UPPAAL is a verification tool for modelling, simulating and verifying real-time systems. It is based on the theory of timed automata[?][?] and typical systems to gain advantage of UPPAAL are systems where timing aspects are critical that communicate through channels or shared variables. As other model checkers, UPPAAL have a modelling language, wherein the system is specified, and a query language that is used to specify the properties to check against the system. The query language is a subset of CTL (computational tree logic) that work for real-time systems[?] [?]. The model checking is done by checking the state-space by making a reachability analysis. The current version of UPPAAL is called UPPAAL2K and was first released in 1999[?].

In 1981, Edmund M. Clarke and E. Allen Emerson managed to combine temporal logic with the state-space exploration in order to provide the first automated model checking algorithm[**?**]. It was capable of proving properties of programs as well as producing counter examples. In the mid 1980s it was shown how model checking could be applied to hardware verification. However, it quickly became clear that model checking on hardware was very limited due to the state-space explosion that occurs especially on hardware.

Randall Bryant from the CMU electrical engineering department invented ordered Binary decision diagrams (OBDDs). Later on, J. Burch, E. Clarke, K. McMillan et al.[**?**] used OBDDs and created *symbolic model checking* which represents the state space symbolically. The symbolic model checking can verify systems with an extremely large number of states and thus creating a solution to the problems of state space explosion.

Because of the state-space explosion problem and the increasing complexity of digital electronic circuits, there was a need to be able to model the timing and data flow of a ciruit with a certain amount of abstraction. This became Hardware Description Languages (HDL)

VHDL (VHSIC Hardware Description Language) was initially ordered by the United States Department of Defence in 1981 to help with the growing problem of hardware life cycles. It is based on the Ada programming language and have been the base Hardware Description language that was used to model hardware. In 1987 it became an IEEE standard, known as VHDL-87. After a major modification in 1993 it was known as VHDL-93. VHDL ...

write something more

Verilog was published by Gateway Design Automation in 1985 and along side VHDL are the two main HDL's used for modelling circuits. Cadence Design Systems received the rights to Verilog-XL which is the HDL simulator that would end up being the de-facto standard Verilog simulator.

Chapter 3

Analysis

3.1 SME

3.2 SMEIL

3.3 CSP

Today, Communicating Sequential Processes (CSP) is a process algebra that provides a way to express concurrent systems. By using message passing between processes the language avoids certain problems that arise with the use of e.g shared variables. An essential part of CSP is message passing and the syntax for input is `X?c`. This represents an input from channel`X` and an assignment of the input value to the variable `c`. The output syntax is `X!c` where the value of the variable `c` is sent over the output channel `X`. At first Hoare had defined the message syntax to use the process names, but later on when CSP was developed into a proper process algebra, the syntax changed into using channels in order to be able to have several processes connected via the same channels. **do not write too much here. just short explain csp**

3.4 CSP$_M$

CSP$_M$ is a formal language that combines CSP with a functional programming language in order to make it easier for the programmer to model the systems and then use the code on tools that can animate, verify or similar.

3.5 FDR

In the paper *A primer on model checking*[?] Mordechai Ben-Ari explains a concurrent problem that he had used for many years, to teach his students about concurrency. ... **write this when I have read the article again**

Chapter 4

Generalize specification from program and traces

4.1 Target solution

4.2 Manual translation

The first step in translating from SMEIL to CSP$_M$ is to create a small example and create a manual translation. This ensures that we have a suitable example to test the automatic gode generation, but also gives a good understanding of how the translation could be created and what kind of challenges there will arise from translating from SMEIL to CSP$_M$. The example *Seven segment example* is an example of modelling a digital clock that consists of 6 different 7-segment displays. A 7-segment display is a display device for displaying decimal numerals. It consist of 7 identical segments which can be lit in different combinations in order to show the Arabic numerals 0 to 9. In the example we model a circuit that receives an input in the form; "seconds after midnight" and from this, calculates and displays the correct hours, minutes and seconds on the displays. Since only one digit can be shown on each 7-segment display it is necessary to separate the actual number into two e.g if the hour is 12 it will be shown as 1 and 2 on two separate displays.

The SME example in figure 4.2 inputs a numeral from a source process that is incremented by one for each run. It then sends the input out on the output bus where three different calculating processes receives from. Each process calculates respectively hours, minutes and seconds and separates the number in order to output the result to the two output channels.

The CSP$_M$ code in figure 4.2 is the handmade translation from the SME file. The translation in CSP$_M$ is equivalent to the SME version where each process calculates either the hours, minutes or seconds and separate the result into two digits, one for each output channel.

What we need to assert in this example is the number that is sent to the 7-segment displays. A 7 segment display can only represent 0-9 but 4 bits can represent 0-15. This means that we are interested in figuring out if this model can result in an ouput less that 10, in which case, the assertion fails and the model needs to be changed. The assertion used in CSP$_M$ checks if the processes

Figure 4.1: Seven segments example in SMEIL

Figure 4.2: Seven segments example in CSP$_M$

refines the `SKIP` process i.e if the process terminates. This works because of the `if-then-else` statement that ensures that the process never stops (`STOP`) if one of the outputs are larger than 10.

One problem that arises when trying to translate SMEIL to CSP$_M$, is that in SMEIL the input has to be generated by a process, i.e there is no input from file or stdout. Therefore we create a source process, in this case the `clock()` process, that does not have any input but, in this case, generates a variable which is saved and incremented by one for each run. Now, FDR checks all possible inputs and therefore we have to find a solution so that we can figure out the entire input range automatic. The solution lies in the SMIEL simulator. When simulating the sme program, it creates a range of all observed inputs on all channels in order to change the general `int` to e.g `i4` if all observed inputs are within i4. Then we can observe and generalize that a process with no input bus will be the source process that generates the input for the circuit. Due to this generalization, we can translate the output bus of the source process in sme to be the the input channel in CSP$_M$. That is, the range observed on the output bus in sme will be translated into the range of the input channel in CSP$_M$. So in reality the source process is not created as an actual process in CSP$_M$ and therefore it is also crucial that the source process in sme does not have hidden channels or do calculations that we wish to assert on.
In CSP$_M$, one always note the input ranges of channel if the channel carries numerals. However if we simply write `channel input : int`, then FDR will check for all integers, which will last forever, therefore we wish to create the proper range on the input channel by using the source process in sme. However for the rest of the circuit, we might need to assert on the input or output of the busses and therefore we do not wish to use the ranges from the sme simulator on other busses than the output bus from the source process. Since it is necessary to give a range to all channels in CSP$_M$, as mentioned above, the challenge lies in figuring out the correct ranges for these channels.

4.3 Automated translation

It is important to mention that the FDR version of the SMEIL program are represented as one clock cycle and therefore we do not have to handle implicit clock cycle issues. we can just translate one-to-one, because FDR models one clock cycle and the input represents all possible input in one clock cycle.

Chapter 5

Experiments and results

Chapter 6

Discussion

Chapter 7

Conclusion

7.1 Future work