FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Alberte Thegler - alberte@thegler.dk

Towards formal verification of FDR4

Department of Computer Science

Professor Brian Vinter

August 2018

Abstract

Bla bla bla bla

Contents

Todo list

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 (word for important systens). Therefore it is of high interest to create a verification of the system or program.
Talk about how verification was first created and how it became to be used for concurrent systems. Then write about how it works and then write about the different systems and formal languages that is used for it.

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]

1.1 Motivation

Intels-division bug
Toyota bremse-fejl
Adriane 5 haeldning
Terac-25

1.2 Learning goals

This is where the learning goals go.

Chapter 2

Related work

The concepts of formal verification began in 1967, when Robert W. Floyd was published with the paper *Assigning meaning to programs*[4]. 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 argues 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*[5] was published. The logic he presented (later known as *Hoare logic*), was build on Floyd's ideas and he proposed that a program could be viewed as a partial correctness relation between a precondition and a postcondition predicate. This means that if the state, the program starts in, satisfies the precondition and it terminates, then the final state satisfies the postcondition. Hoares logic have been the basis of a lot of different formal languages and have contributed to the continuous work on formally verifying programs.

Maybe add more here about what uses Hoare logic today

Since the original Hoares logic was not originially thought as to work with concurrent programs, L. Lamport extended Hoare's logic in his paper *The 'Hoare logic' of concurrent programs*[8] 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.

Figure out if Hoare used this information/update in his work with CSP. I am not sure if CSP work on Hoare logic?

In 1972 Hoare's paper *Towards a Theory of Parallel Programming*was published and in 1978 his paper *Communicating Sequential Processes* was published. With the 1978 paper, CSP was born and have been widely used in many different works and have also been expanded since Hoare initially described it in 1978. The first version of CSP was mostly a concurrent programming language but in 1984, Brookes, Hoare and Roscoe published their continued work on CSP with the paper *A Theory of Communicating Sequential Processes*[2], 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*[11]. Now, several different variations of CSP exists today which all specialize in different areas of formal descriptions.

download file and make citation - I have not been able to find a free version of this paper

A number of tools have been created to analyse, verify and understand CSP written systems. In order to use these tools along with CSP, different types

Why is

of machine-readble CSP syntaxes have been created over the years, but most of todays CSP tools use a version of machine-readble CSP calles CSP$_M$which was created by Scattergood[?]. Since CSP was mostly a blackboard language, Scattergood created a combination of the standard CSP and a functional programming language which created a better baseline for tools to work with CSP. 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 which differs from a lot of other CSP tools that are merely model checkers. FDR only work on finite-state processes..

Figure out the precise difference between refinement and model checking

ProBE (Process Behaviour Explorer)[?] is a tool to animate CSP in order to explore the state space of CSP processes, and can even handle infinite state. ProBE is based on the same CSPM version as FDR and ProBE have also been created by Formal Systems (Europe) Ltd that also created FDR.

Why can't I find any more information about ProBE?

This might not be relevant since it does not actually verify anything

The Adelaide Refinement Checker (ARC)[?] is a automatic verification tool for untimed CSP. It represents the internal representation by using Ordered Binary Decision Diagrams (OBDDs). This lessen the state explosion problem that other model checkers have with LTS representations.

The ProB project[?] is originally a constraint solver and model checker for the B-Method but it also supports other languages like Z and CSP$_M$. ProB can also be used for automated refinement checking and LTL model checking. ProB can work with some CSP$_M$on its own or it can be used to verify combined CSP$_M$and B specifications.

J. Sun, Y.Liu, J.Dong et al. present the Process Analysis Toolkit (PAT)in their 2009 paper[?]. PAT is a CSP analysis tool that can perform LTL model checking, refinement checking and simulation of CSP and Timed CSP processes. ***http://www.cs.ox.ac.uk/ucs/CSPtools.html* claims that Pat uses a liberal version of CSP and not according to the original semantics. PAT apparently supports shared variables, which the original CSP does**

It might use another type of CSP

SSG is a parallel refinement checker based on CSP. It can do refinement checks, deadlock checks and divergence checks. It can do parallel checking and therefore the time for verification is a lot smaller than with fx. FDR.

Cite the webside

I have had a hard time finding papers on this - are there no papers on it?

SyncStitch is a refinement checker also based on CSP. It can perform refinement, deadlock checks and livelock checks. In SyncStitch it is possible to model, simulate and check concurrent systems.

Find more about this as well https://www.principia-m.com/syncstitch/ - I can't seem to find any papers on this.

CSP-Prover[?] (https://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html) 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 FDR is better at proving combinatorial problems (Not sure if relevant)

The programming language Occam, which was first released in 1983, is a concurrent programming language that builds on the Communicating Sequential Processes process algebra. Occam developed over 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 idas of CSP in the original occam language but adding mobility features from pi-calculus. On the KRoC webpage they describe the reason to include functionality from pi-calculus; *"Specifically, we want to allow networks of processes to evolve, to change their topologies, to cope with growth and decay without losing semantic or structural integrity. We want to address the mobility of processes, channels and data and understand the relationships between these ideas. We want to retain the ability to reason about such systems, preserving the concept of refinement."*[1]

SPIN[?] is a verification tool that uses process interactions to prove correctness for a system. The systems to be verified are described in the formal language `PROMELA`(PROcess MEta LAnguage)[6] and the correctness properties are spcified in Linear Temporal Logic (LTL)[9]. In the paper *Reasoning About Infinite Computations*[13], 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 verfication 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*[7]. SPIN was build on the pioneering work on logic model checking by Clarke and Emerson[3], as well as Sifakis and Queille[10]. Vardi and Wolper extended their work with an automata-theoretic approach to automatically verify programs[12].

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 [?]. UPPAAL[?] is a verification tool for modeling, 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[?].
There are several extensions available today, all with different focus.

VHDL

WRIGHT[?][?] is an architecture description language which was devel-

It would be worth to read more about this! They have done a bit of the same that I am to do in my thesis with auto

[1] https://www.cs.kent.ac.uk/projects/ofa/kroc/ access date: 3/4/18

oped at Carnegie Mellon University. They can auto generate CSP$_M$code from WRIGHT and from there they can confirm certain properties by using FDR. http://www.cs.cmu.edu/ able/wright/

Chapter 3

Theory

This is where the theory go. fx. SME and the correlation between that and CSP. CSPm was devised by Bryan Scattergood as a machine-readable dialect of CSP - se the paper *The Semantics and Implementation of Machine-Readable CSP*

" FDR2 is often described as a model checker, but is technically a refinement checker, in that it converts two CSP process expressions into Labelled Transition Systems (LTSs), and then determines whether one of the processes is a refinement of the other within some specified semantic model (traces, failures, or failures/divergence)" (from Wikipedia - se paper *Model-checking CSP - af Roscoe*

3.1 Hoare's logic

Chapter 4

Method

This is the method section that describes what I did, how and why.

Chapter 5

Results and tests (Experiment?)

Does it work? why, why not

Chapter 6

Discussion

Chapter 7

Conclusion

7.1 Future work

Bibliography

[1] M. M. Ben-ari. A Primer on Model Checking. 1(1):40–47, 2010.

[2] S. D. Brookes, C. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, 1984.

[3] E. Clarke and A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic, 1981.

[4] R. W. Floyd. Assigning Meanings to Programs. pages 19–32, 1967.

[5] C. A. R. Hoare. An axiomatic basis for computer programming, 1969.

[6] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[7] G. J. Holzmann. The Model Checker SPIN. 23(5):279–295, 1997.

[8] L. Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14(1):21–37, 1980.

[9] A. Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[10] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[11] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[12] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification, 1986.

[13] M. Y. Vardi and P. Wolper. Reasoning about Infinite Computations, 1994.

7.2 Appendix