1 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. One of these verification systems was the Stanfort Pascal Verifier[30] which was developed by David Luckham at Stanford University. Also 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 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*[11]. 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*[14] 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 work with concurrent programs, L. Lamport extended Hoare's logic in his paper *The 'Hoare logic' of concurrent programs*[20] 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[1]. The first version of CSP was a simple language but in 1984, Brookes, Hoare and Roscoe published their continued work on CSP with the paper *A Theory of Communicating Sequential Processes*[8], 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*[27]. Now, several different variations of CSP exists today which all specialize in dif-

ferent areas of formal descriptions.
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[28]. 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[32]. 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.

- ProBE (Process Behaviour Explorer)[23] 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)[24] 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[12][22] was originally created as an animation and model checker tool for the B-Method[2] 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[31]. PAT is a CSP analysis tool that can perform Linear Temporal Logic (LTL) model checking, refinement checking and simulation of CSP processes.

- CSP-Prover[18] 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[29], 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[33] 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.

SPIN[19] 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)[15] and the correctness properties are spcified in Linear Temporal Logic (LTL)[25]. In the paper *Reasoning About Infinite Computations*[35], 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*[16]. SPIN, as well as other model checker tools, has been build on the pioneering work on logic model checking by Clarke and Emerson[10], as well as Sifakis and Queille[26]. Vardi and Wolper extended their work with an automata-theoretic approach to automatically verify programs[34].

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[21] in 1995 and further introduced it in the paper *UPPAAL - a Tool Suite for Automatic Verifcation of Real-Time Systems*[7]. UPPAAL is a verification tool for modelling, simulating and verifying real-time systems. It is based on the theory of timed automata[17][5] 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[13] [21]. 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[6].

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[10]. 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.[9] 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)

The VHSIC Hardware Description Language VHDL was initially ordered by the United States Department of Defence in 1981 to help with the growing problem of hardware life cycles.

VHDL: (VHSIC - Very High Speed Integrated Circuit) HDL Initially sponsored by DoD as a hardware documentation standard in early 80s Transformed to IEEE and ratified it as IEEE standard 1176 in 19887 (Known as VHDL-87) Major modification in 93 (Known as VHDL-93) Continuously revised It is based on the Ada programming language.

Verilog: Introduced by Gateway Design Automation in 1985. Cadence Design Systems got the rights to Verilog-XL, the HDL simulator that would become the de-facto standard og Verilog simulator. Due to a request from the U.S Department of Defence, the development of VHDL came to be.

However, VHDL and Verilog share many of the same limitations: neither is suitable for analog or mixed-signal circuit simulation; neither possesses language constructs to describe recursively-generated logic structures. Specialized HDLs (such as Confluence) were introduced with the explicit goal of fixing specific limitations of Verilog and VHDL, though none were ever intended to replace them. (From WIKI) (From WIKI): Essential to HDL design is the ability to simulate HDL programs. Simulation allows an HDL description of a design (called a model) to pass design verification, an important milestone that validates the design's intended function (specification) against the code implementation in the HDL description. It also permits architectural exploration. The engineer can experiment with design choices by writing multiple variations of a base design, then comparing their behaviour in simulation. Thus, simulation is critical for successful HDL design.

Look at functional verification

Look at Property Specification Language also look at SVA (two property languages that are derived from LTL) (used for Hardware)

HDL include explicit notation for expressing concurrency as well as a notion of time. HDLs are used to write executable specifications for hardware. Because HDLs can be executed it gives the illusion of programming languages even though it is more of a specification language or modelling language. First HDLs in late 60's. C.Gordon Bell and Allan Newells text "Computer Structures" in 1971 - first to give a hdl with lasting effect.

(from http://www.techdesignforums.com/practice/guides/formal-verification-guide/) "Equivalence checking has been used for more than a decade to check that RTL and gate-level descriptions of a design represent the same design"

Take a look at Temporal logic model checking (As it is mentioned in the formal verification - evolution article) - Clarke et. al. CMU 1981 - Sifakis et. al. Grenoble 1982 and also look at Symbolic model checking McMillan 1991 SMV

WRIGHT[4][3] is an architecture description language which was developed 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/

Both theorem provers and model checkers have been, and are still, widely used for both software and hardware. There is a third form of formal verification that is also being used more often now. This is equivalence checking, which compares two models of a design and produces an outcome that either shows that they are equal or provides a counter-example to show when they disagree. It is beginning to become common practice for hardware designers to

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 generating CSP$_M$

use equvalence checking to compare the design of an optimized digital design and an unoptimized digital design. This way it is possible for the designer to check that the optimizations did not change the functionality of the design.

References

[1] A. E. Abdallah, C. B. Jones, and J. W. Sanders. *Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers.* 2005.

[2] J. R. Abrial. The B tool (Abstract). In R. E. Bloomfield, L. S. Marshall, and R. B. Jones, editors, *VDM '88 VDM — The Way Ahead*, pages 86–87, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[4] R. J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, 1997.

[5] R. Alur and D. Dill. Automata for modeling real-time systems. In *Automata, Languages and Programming*, pages 322–335, 1990.

[6] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, and Others. Uppaal-now, next, and future. *Modeling and verification of parallel processes*, 41:99–124, 2001.

[7] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a Tool Suite for Automatic Verifcation of Real-Time Systems. *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, (1066):232–243, 1995.

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

[9] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10ˆ20 States and beyond. *Information and Computation*, 98(2):142–170, 1992.

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

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

[12] Heinrich-Heine-University. ProB, 2017.

[13] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, pages 193–244, 1994.

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

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

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

[17] J. E. Hopcroft, R. Motwani, and J. D.Ullman. *Introduction to Automata Theory, Formal Languages and Computation.* Addison-Wesley, 2001.

[18] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings*, 3440:108–123, 2005.

[19] B. Labs. SPIN.

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

[21] K. G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In H. Reichel, editor, *Fundamentals of Computation Theory*, pages 62–88, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[22] M. Leuschel and M. Butler. ProB: A Model Checker for B. *FME 2003 Formal Methods*, 2805:855–874, 2003.

[23] F. S. E. Ltd. Process Behaviour Explorer - ProBE User Manual.

[24] A. N. Parashkevov and J. Yantchev. ARC-a tool for efficient refinement and equivalence checking for CSP. *Algorithms amp; Architectures for Parallel Processing, 1996. ICAPP 96. 1996 IEEE Second International Conference on*, (July):68–75, 1996.

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

[26] 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.

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

[28] B. Scattergood. *The semantics and implementation of machine-readable CSP.* PhD thesis, 1998.

[29] SGS-THOMSON Microelectronics Limited. occam 2.1 reference manual. 1995.

[30] Stanford Verification Group. Stanford Pascal Verifier User Manual. 1979.

[31] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. pages 709–714, 2009.

[32] A. B. A. W. R. Thomas Gibson-Robinson, Philip Armstrong. FDR3 — A Modern Refinement Checker for CSP. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[33] Univeristy of Kent. occam-pi: blending the best of CSP and the pi-calculus.

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

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