UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's Thesis

Alberte Thegler - alberte@thegler.dk

Towards Automatic Program Specification From SME Models

Department of Computer Science

Supervisors: Brian Vinter and Kenneth Skovhede

Handed in: November 21, 2018

Abstract

This thesis introduce a method to simplify hardware modeling and verification thereof, in order for software programmers to, more easily, meet the demands of the growing embedded device industry. The system TAPS is presented, which is a system for transpiling from the new SME Implementation Language (SMEIL) into the machine-readable CSP language (CSP_M). The translated CSP_M code is augmented with assertion statements, derived from values observed by simulating SME models. Utilising the formal verification properties of the FDR4 refinement tool, the assertion statements of the translated CSP_M code can be formally verified. An extension to the initial version of TAPS is also introduced in the thesis. The extension provides the possibility of modeling a globally synchronous clock in CSP_M, providing the possibility of verifying the internal state of the CSP_M network separately for each clock cycle. A small example consisting of a seven segment display clock network is presented, to introduce how to verify the widths of channels in the network. A small cyclic network is presented, to show the possibilities of verifying separate clock cycles, using the extended version of TAPS.

TAPS source code at the time of hand in:
`https://github.com/althegler/TAPS/tree/faaf384b71d83faa821d294cd16b5c7943bd7e57`

Acknowledgements

I would like to thank my supervisors Brian Vinter and Kenneth Skovhede for their support, patience, and guidance throughout this project, and for encouraging me to write a paper based on this work. I'd like to give special thanks to Mads Ohm Larsen for his immense help in writing the paper and for always taking the time to discuss my ideas. Another special thanks goes to Truls Asheim for his kind support in introducing me to SMEIL, proofreading my writings, and for patiently answer all of my questions. I would also like to thank the employees in the eScience group at the Niels Bohr Institute, for their encouragement and for providing a welcomming environment. Last but not least, I would like to thank Lars and Asbjørn Thegler for taking their time to proofread my thesis and for their help towards making my writings clear and concise.

Contents

Chapter 1

Introduction

The Internet of Things, computerised medical implants, and the omnipresent growth in robotics, bring with them an increased demand for programmers, capable of developing software for these devices. While this observation may not in itself appear to present a new challenge, many other areas have previously presented a need for more programmers. The new challenge is that these new growth areas are all focused on small size, low power consumption, and high reliability. This means that traditional software engineering methods, and thus traditionally trained programmers, are often not sufficiently qualified to work with these technologies. In previous decades, such systems have been developed by electronic engineers that apply far more rigid development approaches. Especially for hardware solutions like Very-Large-Scale Integration (VLSI) and Field-Programmable Gate Array (FPGA), correctness has always been favored over productivity due to a more rigid environment, than traditional software developers are used to. While tools have obviously improved and methods refined, the VLSI process is still mostly the same as presented in [9]. The primary workflow from [9] is shown in Figure 1.1; note the focus on verification in each step.

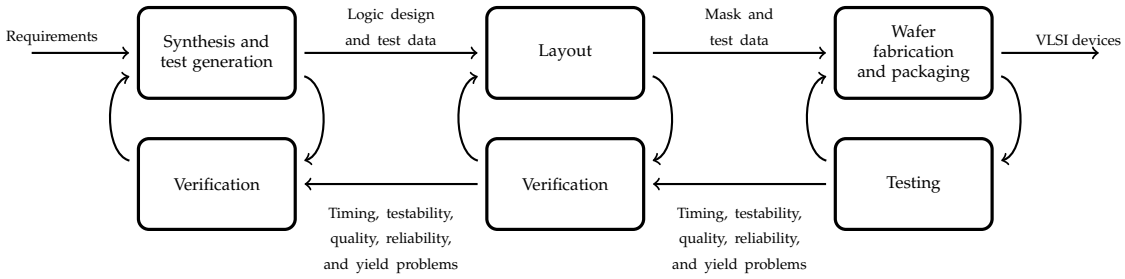

Figure 1.1: VLSI process workflow.

While the VLSI community is fundamentally following this 1980's design approach, more high-level tools and abstractions have been introduced. Philippe et al. [23] show a workflow (reproduced in Figure 1.2) where the important part is the verification, that has been partly automated by basing the development on a formal specification of the solution.

There is no denying that the subjectively slow and rigid development process in the VLSI world [34] is highly successful in producing correct and reliable circuits. At the same time, conventional software development is highly focused on productivity and time-to-market, for example, smartphone applications are often developed for continuous release, where bug patches and new features are rolled out daily. This is of course not possible with hardware.

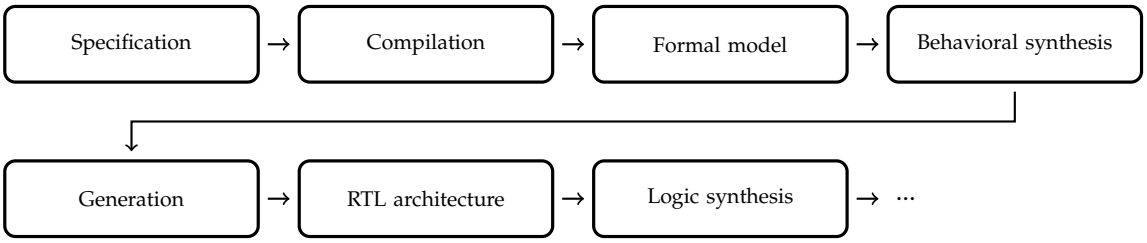

Figure 1.2: Reproduced workflow from Philippe et al. [23].

Thus, a growing chasm exists between the way most programmers are trained, and the competencies that are needed to support the growth in mission critical embedded devices.

In this thesis, I propose a tool to help bridge the gap between available programmer profiles and the required competencies for developing embedded devices. My approach is based on generating a specification from a software implementation and test-suite observations. The overarching goal is to reach a level where a conventional software programmer can write a solution in the new hardware design model, Synchronous Message Exchange (SME) [69, 70, 62], and develop a conventional test suite in the software engineering tradition. SME will be further introduced in Chapter 4. By combining this implementation with the *observed* values of internal states in an SME based system implementation, I can produce a formal specification of the system. These observed values will be introduced further in Chapter 4. This generated specification model can be fed into a formal verification tool and thus improve the correctness guarantees from what is covered by the individual test vectors to the entire space that is spawned by the set of test vectors.

I approach the task by transpiling[1] the new SME Implementation Language (SMEIL) [17] for SME into CSP$_M$ [56] and verify the formal properties of this version with a tool like FDR4 [28].

Limitations This thesis does not discuss low-level details of the FDR4 tool, but uses it as a verification tool to verify the generated CSP$_M$ code. Similarly, the SME model is a predefined model that is used to describe hardware structures. The SMEIL language is used as a basis for the input programs for the transpiler. This work utilises each of these projects and languages in their current form, to solve the problem of translating hardware models into specification models.

1.1 Motivation

As explained, hardware testing is not as simple and reusable as software testing, and it is not possible to deploy automatic bug patches for physical hardware. Therefore, it is cheaper for the hardware companies to have their newly developed hardware model tested as extensively as possible before production, than shipping physical hardware with a critical error. Although the testing can be extensive, it is difficult to achieve complete code coverage accounting for all corner cases using handwritten test cases. There are several examples throughout history where more verification could have saved both lives and resources.

1.1.1 Ariane 501 Failure

The Ariane 5 lifting rocket [33] was designed to launch large payloads into Earth orbit, such as communications satellites, etc. It was owned by the European Space Agency (ESA) and the

[1] Source-to-source compile.

French space agency Centre National d'Études Spatiales (CNES), and manufactured by Airbus Defence and Space.

Ariane 5 was the follow-up to the successful Ariane 4 launchers. On June 4th, 1996, the Ariane 5 rocket was first flight tested. At about 30 seconds after successful lift-off, the rocket exploded midair, resulting in an approximately $500 million loss for ESA and CNES. The failure could have been avoided if there had been more focus on verification or simulation within the systems of Ariane 5. Luckily the rocket was unmanned, but this type of failure could happen in any other critical system. This launch failure is acknowledged as one of the most expensive human errors in history.

The failure of Ariane 5 was partially caused by a bug in the Inertial Reference System (SRI) that measures the attitude of the launcher and its movements in space. The SRI comprises an internal computer which calculates angles and velocities. The failure occurred due to an unexpected high value of an internal alignment function result called Horizontal Bias (BH). The BH value is related to the horizontal velocity and was represented as a 64-bit floating-point number. This value was converted to a 16-bit signed integer within the SRI and sent to the On-Board Computer (OBC), which controls the direction of the nozzles of the solid boosters.

The BH value had also been calculated in the Ariane 4 systems, so it was not thought of as insecure. However, because the trajectory of Ariane 5 was considerably different than that of Ariane 4, this value was much higher. Due to this high BH value, the 64-bit floating-point number could not be represented by a 16-bit signed integer, so the conversion caused an overflow. The result of the overflow was interpreted as actual flight data by the OBC, and according to this misinterpreted flight data, the rocket was off course. The OBC counteracted what it thought was a wrong angle of the rocket by turning the nozzles to change course. Within a few seconds, the forces of aerodynamics ripped the solid boosters apart from the core stage. This caused the self-destruct mechanism to trigger, and the rocket self-destructed in a giant explosion shortly afterwards. The backup SRI, that should take over when errors occur in the active SRI, was executing the same code as the active SRI and had failed, for the same reasons, just before the active SRI.

On the Ariane 4 rocket, the BH value was necessary for a short while after lift-off, but this had been changed in the Ariane 5 where the data was only crucial to the rocket at lift-off. Afterwards, the data should not have had any influence on the rocket. Unfortunately, it had not been predicted that the conversion could overflow, so the functionality of the BH value had been kept without extensive testing or verification. If this functionality, and the consequences of a different trajectory, had been properly considered for the Ariane 5 rocket, this failure might 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 [39], which could provide two different kinds of radiation treatment. At that time, hospitals would typically have two different machines to be able to perform both of the treatments that a single Therac-25 machine could perform.

Radiation is used to kill cancer cells, so a patient is exposed to a beam of particles, or radiation, in specific doses that are designed to kill the specific type of cancer cells. Because cancer cells are more sensitive to radiation than normal cells, the radiation will kill the cancer cells but cause relatively minor damage to the normal tissue. However, radiation is damaging to normal tissue as well and it is, therefore, essential to specify the exact amount of radiation needed in a specific area to minimise the damage to the healthy cells.

The first type of treatment, provided by the Therac-25, was an electron-beam treatment, which kills shallow tissue, like skin cancer. The second treatment option of the Therac-25 was a beam of higher-energy X-ray photons, which travels further into the tissue and are therefore better suited for cancer in deeper-lying tissue. The Therac-25 was based on the previous Therac-20 and Therac-6, which had been very successful. The Therac-20 and Therac-6 both had hardware safety interlocks to avoid failures but unfortunately, this had been removed in the Therac-25 in favor of software-based security. AECL put more faith in software reliability than on hardware.

After the Therac-25 had been operational for a couple of years, a series of incidents happened, where patients were overexposed to radiation, leading to six cases of serious injury, resulting in death for some of them. Friz Hager, a physicist at East Texas Cancer Center, tested the Therac-25 rigorously to reproduce the errors they had experienced. He was able to demonstrate the error showing that if the user selected the X-Ray mode on the Therac-25, the machine began setting up for high-powered X-rays, which would take about 8 seconds. If the user switched to Electron-beam mode before the machine finished setting up for X-ray mode, i.e within 8 seconds, the turntable that controlled the amount of radiation, would not switch to the correct position causing an enormous amount of radiation to reach the patient.

After solving the problem and releasing a new version of the Therac-25, another problem emerged where a patient was overexposed. This time it turned out to be a counter overflow within the system and if a command was sent at the exact moment the counter overflowed, the machine would not set up properly, again resulting in an overexposure of radiation for the patient.

After this incident with the Therac-25, it was found that some of the same software problems existed in Therac-20, but due to the hardware precautions, the problems never occurred. This example shows how important it is for critical systems to be well-designed, as well as well-tested or verified.

1.1.3 The Patriot Missile Failure

During the Persian Gulf war, on February 25, 1991, an American Patriot missile failed to intercept an incoming Iraqi Scud missile, which caused the Scud to hit an American Army barracks injuring around 100 people and killing 28 soldiers [65].

The Patriot missile failed due to an error in converting an integer, representing time since last boot, to a floating-point number using a 24-bit register. As time since last boot increased, the limited 24-bit register did not represent enough precision, so the chopping error increased. At the time of the incident, the Patriot missile battery had been on for approximately 100 hours, which caused the chopping inaccuracy to be around 0.34 seconds. The Scud travels at around 1.676 meters per second, and therefore in the 0.34 seconds, it travels more than a half kilometer. The inaccuracy of the 24-bit representation caused the Patriot missile to perform inaccurate calculations, and thereby wrongly predict the location of the Scud. The consequence of this was that the Patriot missile missed its target Scud missile.

This example shows how it is impossible to test 100% of any system and how that can cause horrible failures. If the Patriot missile systems had been verified or had been subject to long-term testing, this accident might have been avoided. An example, similar to the Patriot missile failure, will be introduced in this thesis and will be verified to show potential failures.

Chapter 2

Related Work

In 1954 Martin Davis created the first computer-generated mathematical proof. It proved that the product of two even numbers is even. In spite of its simplicity, it was the beginning to theorem provers and automatic verification. In the late 1960s, first-order theorem provers were applied to verification problems in Pascal, Ada, and Java. One of these verification systems was the Stanford Pascal Verifier [40] which was the first system to automatically solve a mathematical problem given by the American Mathematical Society before any official solutions had been published. In 1972, Sir Robin Milner introduced an automated theorem prover, along with the ML programming language. The theorem prover was called Logic for Computable Functions (LCF) [41], and Milner's work in automated reasoning have been the foundation for a lot of other theorem provers, like the proof assistant Higher Order Logic (HOL) by Mike Gordon, which was originally developed for reasoning about hardware. The formal proof management system Coq [4] is also a descendant of LCF.

Automatic theorem provers have been very valuable in many different ways, but especially one limitation caused problems. The theorem provers could not provide a reason for a failure. If a problem was found a problem within a theorem, the theorem prover could not provide a reason for the failure. It was hard to solve the problems within the theorems when the reason for the error was unknown. A solution to this limitation was to build systems that could provide a counterexample or some other explanation for the failure of the theorem.

In 1967, the paper *Assigning meaning to programs* [25] by Robert W. Floyd was published, and in it, Floyd provided argumentation for formal definitions of the meaning of programs which could be used for proving correctness, equivalence, and termination of computer programs. 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 axiomatic basis for computer programming* [29] was published. The logic he presented there, which was later known as Hoare Logic, was built on Floyd's ideas and proposed the notation Partial Correctness Specification; $\{P\}C\{Q\}$ where 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. Hoare's method can be used to prove correctness properties of sequential programs. Hoare Logic has also been the basis of several different formal languages and has contributed to the continuous work on formal verification.

Since the original Hoare Logic was not developed to model concurrent programs, Leslie Lamport extended it, in the paper *The 'Hoare logic' of concurrent programs* [36] in 1980. In it, it was discussed why Hoare Logic, as proposed by C.A.R Hoare, did not adapt to concurrent pro-

grams and proposed a "generalized Hoare Logic" that is generalized to concurrency.

In 1978, Hoare's paper *Communicating Sequential Processes* [30] was published and with it, CSP was born. CSP was introduced as a model to describe patterns in concurrent systems and communication between sequential processes running in parallel. Today, CSP is a process algebra which describes a formal method for modelling concurrent systems. CSP has been widely used in many different types of work and has also been expanded since Hoare initially presented it in 1978, which has been described in the book [7] published for the 25th anniversary of CSP. The first version of CSP was a simple programming language that had a quite different syntax than today's CSP. In 1984, Brookes, Hoare, and Roscoe published their continued work on CSP with the paper *A Theory of Communicating Sequential Processes* [55] and introduced 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 books *The Theory and Practice of Concurrency* [53] and *Understanding Concurrent Systems* [54].

2.1 CSP Tools

A number of tools have been developed in order to analyse, verify and understand systems written in CSP. The original CSP process algebra is mostly a blackboard language which can be difficult to use on a large scale. Different types of machine-readable CSP syntaxes have been created over the years, in order to make it easier and more accessible for both the industry and academia. Most of today's CSP tools supports a version of machine-readable CSP called CSP_M which was created by Bryan Scattergood [56]. Scattergood created CSP_M as a combination of the standard CSP algebra and a functional programming language which provided a better baseline to create tools for CSP. Here is a subset of the different CSP tools:

- One of the most known CSP tools is the Failure-Divergence Refinement tool (FDR) [28], built by Formal Systems (Europe) Ltd. and is currently at version 4.2.3. FDR4 is a model-checking tool for state machines which is based on the theory of CSP. FDR4's method for evaluating whether a property holds, is to test the refinement of a system against a specified model. FDR is able to run in parallel as well as do state compression in order to avoid a very large state space.

- Process Behaviour Explorer (ProBE) [26] is a tool to visualise CSP programs in order to explore the state space of CSP processes in an interactive way. It provides the user with a hierarchical structure of the possible actions and states of the process. It is based on the same CSP_M version as FDR4 is and it was also created by Formal Systems (Europe) Ltd. ProBE is integrated into the current version of FDR [28].

- At Adelaide University, The Adelaide Refinement Checker (ARC) [44] is an automatic verification tool for CSP. It uses Ordered Binary Decision Diagrams (OBDDs) to represent the internal representation of data structures. This lessens the state explosion problem that other model checker tools have had.

- The ProB project [5, 38] was originally created as a visualisation and model checker tool for the B-Method [8] but it also supports other languages, CSP_M included. Newer versions of ProB can perform refinement checking of CSP_M scripts as well as visualise the process like the ProBE tool. ProB is an active project with continuous updates and improvements.

- Jun Sun, Yang Liu, J.Dong et al. presented the Process Analysis Toolkit (PAT) in their paper from 2009 [63]. PAT is a model and refinement checker for concurrent and real-time systems. Models in PAT are interpreted as labeled transition systems (LTS) and the system can perform Linear Temporal Logic (LTL) model checking, refinement checking and simulation of CSP processes.

2.2 Verification Tools

The programming language Occam [60], which was first released in 1983, is a concurrent programming language that is also based on the CSP process algebra. Occam's strong basis in CSP makes it a well-suited choice for formal proof of correctness. Occam have been continuously developed during the years, and at Kent University the Kent Retargetable occam Compiler (KRoC) team created the Occam-π [66] variant of the Occam programming language. Occam-π is an Occam variant that extends the ideas of CSP from the original Occam language but adding mobility features from pi-calculus [42].

In the paper *The symbiosis of concurrency and verification: teaching and case studies* [46] Jan Pedersen and Peter Welch use Occam-π along with CSP$_M$ to present a workflow methodology for the development and verification of concurrent systems. 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 understanding the verification of a program with FDR4.

Another type of verification tool is SPIN [35], that uses process interactions to prove the correctness of a system. The systems are described in the formal language PROcess MEta LAnguage (PROMELA) [31] and, in contrast to CSP$_M$, the correctness properties are not specified within the same language, in SPIN it is defined in Linear Temporal Logic (LTL) [48]. SPIN performs verification on concurrent software and can prove the correctness of process interactions which can be specified in several different ways, one being asynchronous message passing through buffered channels. 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* [32].

Another verification tool was developed as a collaboration between the Department of Information Technology at Uppsala University in Sweden and the Department of Computer Science at Aalborg University in Denmark. Kim G. Larsen et al. first proposed the idea for UPPAAL [37] in 1995, and it was further introduced by Bengtsson et al. in the paper *UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems* [19]. UPPAAL is a tool for modelling, validating and verifying real-time systems based on the theory of timed automata [10]. The systems that gain the most advantage of being verified with UPPAAL, are systems where timing aspects are critical. As many other model checkers, UPPAAL has a description language wherein the system behavior is described. It also consists of a simulator and a model-checker. The simulator provides the opportunity to examine the system and validate possible executions while the model-checker checks invariant and reachability properties by performing a reachability analysis of the state space. The model-checker can also provide a trace which can explain failures or successes, and can be visualised for further trace examination. The current version of UPPAAL is called UPPAAL2K and was released in 1999 [11].

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 [22]. It was capable of proving properties of programs as well as producing counterexamples. J. Burch, E. Clarke, K. McMillan et al. used, in 1992, Binary Decision Diagrams (BDDs) in *symbolic model checking* [20] to represent 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.

SPIN, as well as other model checker tools, have been built on the pioneering work of Clarke and Emerson [22], and Sifakis and Queille [49]. Their works have provided a research area that is still active today. Vardi and Wolper extended the work with an automata-theoretic approach to automatically verify programs [67].

2.3 Hardware Description Languages

Due to the increasing complexity of digital electronic circuits in the 1970s, a need arose for systems to be able to model the timing and data flow of a circuit with a certain amount of abstraction. This led to the introduction of Hardware Description Languages (HDLs). One of the most popular HDLs today is VHSIC Hardware Description Language (VHDL) [6] which was ordered by the United States Department of Defence in 1981. It is based on the Ada programming language and in 1987 the first IEEE standard was released. Initially, it was developed to be a pure specification language, simulators for VHDL were developed later.

Another very used HDL is the language Verilog which was published by Gateway Design Automation in 1985 and are, along with VHDL, the two main HDL's used for modelling circuits. Verilog, as well as VHDL, is used to write executable specifications for hardware, which can then be run in a simulation program in order to test the logic design before building it in hardware. An essential part of HDL design is the ability to simulate the programs. This allows the model to pass design criteria and validate the design functionality and intended purpose. The possibility of simulating the HDL programs also give the opportunity of exploring the possibilities within the language and within hardware modelling. The developer is able to test multiple versions of a solution and compare results before the hardware is produced. Thus, the possibility of simulating the HDL programs is essential for hardware design.

2.4 Translations from CSP$_M$

Even though the translation between a specification language and an HDL or other programming language is usually nontrivial, it is still an advantage over manual translation. CSP$_M$ can specify a model and FDR4 can verify it, but to manually translate a model to hardware can be error-prone. Therefore several different researchers have, over the years, presented systems for translating CSP$_M$ or subsets of CSP$_M$ into different HDLs:

In 2000 W. Zhou and G. S. Stiles presented the paper *The Automated Serialization of Concurrent CSP Scripts using Mathematica* [71] where they present a package of Mathematica based tools to translate concurrent CSP into equivalent sequential code. In relation to this, V. Raju, L. Rong, and G. S. Stiles presented the paper *Automatic Conversion of CSP to CTJ, JCSP, and CCSP* [50] in 2003. They present a tool that can generate executable C or Java code from a CSP$_M$ program. In both cases, the implementation presented in the papers are using a subset of the CSP algebra. Raju, Rong, and Stiles use the packages CTJ, JCSP, and CCSP to resemble CSP structure in their paper. The packages are all Java or C packages which are created with the purpose of adding CSP-like features to C and Java. Neither of these systems has, to my knowledge, been extended or used in the industry.

In the paper *An Automatic Translation of CSP to Handel-C* [47] Jonathan D. Phillips and G. S. Stiles describe how their translator, CSPtoHC, can translate a small subset of CSP$_M$ to Handel-C [21], which can then be compiled into files for programming an FPGA. Marcel Oliveria and Jim Woodcock present, in the paper *Automatic Generation of Verified Concurrent Hardware* [43], the system csp2hc. It is an automatic translator from CSP$_M$ to Handel-C which, according to Oliveria and Woodcock, have a very similar methodology to the `CSPtoHC` translator by Phillips and Stiles, however the subset of CSP$_M$ supported by the two solutions is different which makes comparisons impossible.

Another example of translation between specification languages and HDLs is the translation between CSP||B to Handel-C [59]. CSP||B [58] is an approach that combines the descriptions of events from CSP and the state from the B-method. The ProB system [38] directly supports simulating CSP||B models. In the paper, Schneider et al. present two different case studies which

introduce different aspects of translating from CSP||B to Handel-C. The translations, presented in the paper, are carried out by hand and are thus not automatically generated.

Finally, James Dibley and Karen Bradshaw present the CSPIDER tool in their paper *Deriving Reusable Go Components from Verified CSP Prototypes* [24]. The CSPIDER tool is a software tool for automatic deriving reusable Go [2] components from verified CSP_M prototypes. The tool is able to implement prototypes constructed from a subset of CSP_M including recursively-defined processes, alphabetised parallel composition, channel input/output, and external choice. The CSPIDER tool is, to my knowledge, the newest system working with the translation of CSP_M.

Chapter 3

Approach

As explained in Chapter 2 several attempts have been made to translate programs written in CSP into a hardware description language. But even a good implementation of this type of system would require that the developer manually models the specification of the CSP_M network, which can be very tedious especially for a novice. The complexity of CSP is most likely leading to fewer developers utilising the functionalities and advantages of the CSP process algebra.

What I aim to achieve is to create a translation reversed from what has previously been done. I wish to provide a solution where the developer can model the network first and then, using the system, automatically generate the specification for that exact network. On top of this, I want to be able to formally verify specific properties of this specification model. This can provide valuable insights into the possible pitfalls of the hardware model that a standard test bench cannot provide.

In this thesis, I introduce the system TAPS, a transpiler that provides translation from hardware models to specification models, while also introducing specific properties for verification within the generated specification model. An illustration of the system structure can be seen in Figure 3.1.

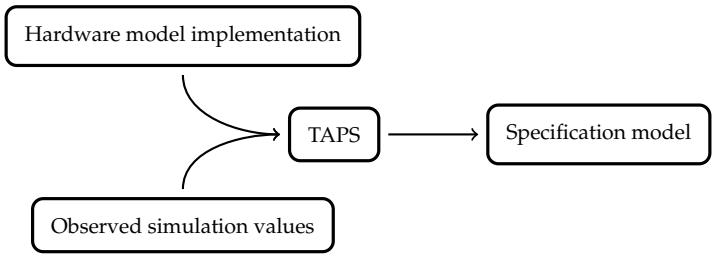

Figure 3.1: Overall system structure of TAPS.

The language used to model the hardware is the SME Implementation Language (SMEIL), which is based on the Synchronous Message Exchange (SME) model. SMEIL resembles a standard general-purpose programming language, while still providing all the necessary elements of hardware modelling. The programming structure of SMEIL enables the traditional developer to model hardware models in a simple but efficient manner. There are several different ways to use SMEIL, which will be introduced in Chapter 4. In this work, I focus on the independent SMEIL representation and thus I only present examples in pure SMEIL.

TAPS provides translation to the machine-readable version of the CSP process algebra, CSP_M. The generated CSP_M code will not only be equivalent to the SMEIL code, but it will also include assertion statements to formally verify properties within the hardware model. These properties can be verified with the CSP_M refinement checker tool FDR4, described in Chapter 2.

Chapter 4

Background Technologies

In this chapter, I will introduce the SME model and the SME Implementation Language, SMEIL. I will also introduce CSP, the machine-readable CSP language, CSP_M, and the FDR4 tool. The basic structures of both SMEIL and CSP_M are introduced in order to provide the reader with a basic knowledge of the functions and structures mentioned in this thesis.

4.1 Synchronous Message Exchange

SMEIL is based on the SME model and therefore I will give an introduction to SME to familiarise the reader with the SME model before introducing SMEIL.

The development of SME was mainly driven by the need to provide a simple framework for programming an FPGA. A lot of today's research focuses on General Purpose Graphical Processing Units, GPGPUs. The GPGPU has been extensively researched and different environments and tools have been implemented to give developers the possibility of utilising all of its capabilities [69]. However, FPGAs can be a better choice when it comes to energy sensitive applications, since FPGAs can, in some cases, achieve the same performance as a GPGPU but with lower energy consumption [69]. Previously, the developer needed to design an integrated circuit on the gate level for the FPGA, which could be difficult, and this is no longer common knowledge amongst developers. Some high-level methods for programming an FPGA have been developed, however, these are often tedious to work with, so when using SME it becomes more accessible to design and implement hardware models.

The primary purpose of SME is to give software developers a tool which provides the opportunity for the developer to program hardware but with an added abstraction layer which separates the developer from the hardware details, such that, the development resembles the structures and semantics that is known from software development. SME was first introduced in 2014 and after several iterations [69, 70, 62] now presents as a programming model, a simulation library, and VHDL code generators. The original idea was conceived following an attempt to create a hardware implementation of a vector processor, modeled in PyCSP [68], a CSP library for Python. The work was initially presented in the paper *BPU Simulator* [52] by M. Rehr, K. Skovhede, and B. Vinter, which introduced a high abstraction level simulation. The subject was explored more in detail in the master's thesis project *Generation of FPGA Hardware Specifications from PyCSP Networks* [61] by E. Skaarup and A. Frisch. The results of this master's thesis made it clear that PyCSP could be used to model hardware, however the need to enforce global synchrony to the circuit resulted in a state explosion. The number of channels, for controlling the progress and for simulating the clock, became immense. Even simple circuits became overwhelmingly large.

The design approach within the master's thesis project by Skaarup and Frisch was to implement a clock process that would drive the circuit. This meant that each process would have to read the clock signal in, to comply with the clock. In order to avoid race conditions, the system had to be implemented with a two-way clock, the so-called *tick*, *tock* signals. Since deadlocks can happen in CSP, it was essential to implement deadlock prevention, which was done by adding channels with a single buffer element. This way, no process would end up in a deadlock. Figure 4.1 shows how a simple CSP network would increase in complexity when modelled in the synchronous PyCSP model. It is clear how trying to use PyCSP for modelling synchronous hardware would result in extremely large and complex networks, which is not an ideal way to write hardware models.

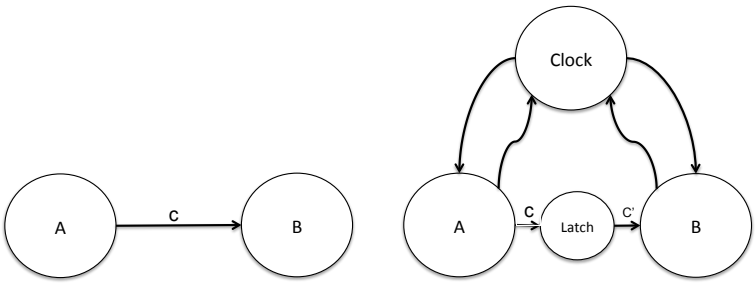

Figure 4.1: Enforcing global synchrony on a simple reader-writer network in CSP, resulting in a substantial increase in complexity. Figure from [69].

The advantage of using PyCSP for processor simulation is the flexibility and simplicity of the language. It is easy to experiment with various ideas and structures, but with the increased complexity of the added clock process and clock-signal propagation, the advantages of PyCSP seemed to diminish. Therefore the conclusion of the master's thesis project was that using PyCSP alone for building synchronous processor simulators was not ideal since the global clock was forced onto the CSP model. It was also concluded that external choice, which is a powerful and essential part of CSP, was not utilised in the globally synchronous approach. The thesis did, however, conclude that several other CSP concepts were fitting well with the hardware concepts, such as shared-nothing in CSP matching the structure of hardware communication because the network state could only be changed by process communication. After this attempt, it became clear that the structure of CSP was poorly suited for modelling clocked systems, and therefore it was decided to create the Synchronous Message Exchange framework, based on the CSP algebra. The idea was to only use the subset of the CSP algebra that provided beneficial functionality to hardware modelling which, most importantly, meant that external choice was omitted while shared-nothing was retained.

In SME, a network is a combination of processes that are connected through buses. The processes communicate through a collection of signals in a bus, instead of CSP's synchronous rendezvous model, but retains the shared-nothing trait of CSP. SME uses the term `bus` instead of `channel` to enforce the semantic correlation between the SME bus and a physical hardware signal bus. The process communication is handled by a hidden clock which eliminates the complexity that arose from adding synchronicity to a CSP network. The combination of the hidden clock and the synchronous message passing between processes means that the SME model provides hardware-like signal propagation. An SME clock cycle consists of three phases: it reads, computes, and writes as can be seen in Figure 4.2. The process is activated on the rising clock edge where it reads from the bus and then it computes and writes to the bus, all in one clock cycle. Just before the rising edge of the clock, all signals are propagated on all buses which means that all communication happens simultaneously. Because of this structure, if a value is written by a process in cycle i, it is read by the receiving process in cycle $i + 1$.

SME is able to detect read/write conflicts where multiple writes are performed to a single bus within the same clock cycle, as well as reads from a signal that has not been written to in the previous clock cycle.

All data that is written to a bus, can be logged for each clock cycle and saved as Comma-Separated Values, CSV, which means that they can be used for validating the VHDL implementation with a VHDL tool. This eliminates the need to write a separate VHDL test bench which will improve developer productivity immensely. A test bench is a piece of software used for testing hardware models. It provides input data to the hardware model and verifies its output.

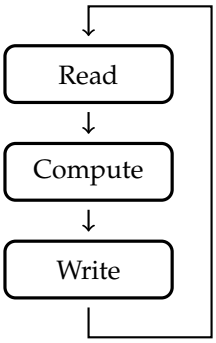

Figure 4.2: SME process flow for one clock cycle.

Since SME is based on CSP, all SME models have a corresponding CSP model, and because of this property, it is possible to create a transpiler translating SME models to CSP_M. The SME model is currently implemented as libraries for the general-purpose languages C# [62], C++ [15], and Python [18] as well as the language SMEIL which will be introduced further below. The Python and C# libraries both have code generators for VHDL as well.

4.1.1 SMEIL

With the different SME implementations, a need arose for a common intermediate language. SMEIL was developed as a Domain Specific Language (DSL) for SME, usable both as an Intermediate Language (IL), and as an independent implementation language. It is accompanied by the implementation, LIBSME [13]. In Figure 4.3 the SMEIL transpiler structure can be seen.

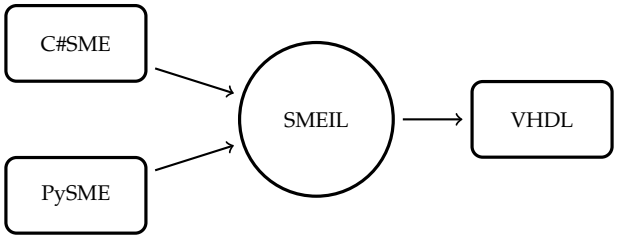

Figure 4.3: SMEIL transpiler structure.

SMEIL has a C-like syntax with a specialised type system that makes hardware modelling simple. In spite of its simplicity, SMEIL still provides hardware-specific functionality that is more difficult to express with general-purpose languages.

Programs written in SMEIL are run by using the LIBSME library, either by using the command line utility or through the provided API. The different methods of using SMEIL are:

- **Pure SMEIL simulation:** A pure SMEIL program is a network which does not depend on external inputs. This means that the network generates data itself and the only communication are between the processes defined in the network.
- **Co-simulation of SMEIL:** The original intent with SMEIL was to create an intermediate language which could be used together with the general-purpose language implementations of SME, which is called co-simulation. With co-simulation, a test bench can be generated as well as VHDL code.
- **Direct code generation:** This is an SMEIL program which does not need to simulate before generating VHDL. In this case, all types are constrained and all information needed for the code generation is in place. Because the simulation step is not executed, a VHDL test bench is also not automatically generated, since this depends on input from the simulation step.

An SMEIL module consists of the actual SMEIL program as well as import statements. SMEIL programs can be separated into modules which can then be imported into other SMEIL programs, creating a library-like structure and reusable components.

The two fundamental components of an SMEIL program are `process` and `network`. The `process` consists of variable and bus definitions, as well as the statements that are evaluated once for each clock cycle. The purpose of the `network` declaration is to define the relations between each entity in the program. A small example of the `process` and `network` syntax can be seen in Listing 1 and a further introduction to the language can be found in Chapter 5.

The SME model supports both synchronous and asynchronous processes. A synchronous process is run during every clock cycle and an asynchronous process is only run when receiving a signal on the input bus. Unfortunately, the SMEIL implementation does not currently support asynchronous processes.

```
1  proc double (in inbus)
2      bus outbus {
3          val: uint;
4      };
5  {
6      outbus.val = inbus.val * 2;
7  }
8
9      ⋮
10
11 network double_net() {
12     instance a of double(b.outbus);
13     instance b of ..
14     ⋮
15 }
```

Listing 1: Small example of process and network syntax in SMEIL.

4.1.1.1 SMEIL Type System

The SMEIL language is strongly, statically typed and has a simple type system that is checked at compile-time. The purpose of SMEIL is to be able to model hardware, and since hardware is static, it was important to create a type system that was capable of enforcing as many static invariants as possible. Because of this, the only type coercion possible in SMEIL is between signed and unsigned integers. Consequently, only booleans can be used in conditionals. The SMEIL type system differs from standard general-purpose languages mainly in the support for bit-precise types. General-purpose languages are typically targeting a CPU which consists of fixed-width registers. Therefore it can not work with data smaller than a byte. However, an important part of modelling hardware is to be able to define the exact widths of the buses, in order to optimise the hardware implementation. SMEIL supports unlimited size integers as well as integers constrained to a specific bit size. SMEIL also supports booleans, double and single precision floating-point numbers as well as strings. Currently, floating-point numbers are supported in the SMEIL simulator but have not been extensively tested.

In SMEIL buses can contain more than one channel and since processes or networks can accept buses passed as parameters, it is important to ensure that no process or network reads or writes to a channel that is not contained within the expected bus. In the same way, it is essential to ensure that the directionality of the bus is enforced. When buses are passed as parameters, they are explicitly declared as either an input or an output bus. However, when defining a bus within a process, it is not possible to define the directionality of the bus. Therefore the bus is defined as input or output based on their first use. The type system of SMEIL enforces a set of rules that define how two buses are unified and the directionality of the bus, which ensures that unification and directionality problems, does not happen. All types of declarations in SMEIL are private, except bus declarations. As mentioned above, buses are used for establishing communication between processes and therefore needs to be a part of the public interface of a process or network.

4.1.1.2 Simulating SMEIL programs

It is not possible to express unlimited size integers in hardware, and therefore, when translating SMEIL to a hardware model, it is necessary to have all types in the program, constrained to a specific width. However, as mentioned before, SMEIL does support unlimited size integers. Since it can be difficult to define an optimal width of channels when writing the SME model, it was necessary to provide both the support of the unlimited size integers while still being able to translate to hardware descriptions. Therefore LIBSME provides a way to re-type the SME network based on the values observed during the simulation of the program. During the simulation, the observed minimum and maximum value for all variables and channel are captured and saved along with the current value. SMEIL has been extended to include lower bounds for the purpose of this thesis. After the simulation, these minimum and maximum values are then converted into SMEIL types large enough to hold the range. The new SMEIL program, with the updated types and observed ranges, are then passed through the type checker in order to ensure that all constraints from the original program are still respected. An example of this can be seen in Listing 4 where an unlimited size integer is re-typed to an integer of fixed size after the simulation.

```
proc double (in inbus)
    bus outbus {
        val: uint;
    };
{
    outbus.val = inbus.val * 2;
}
```

Listing 2: Example of unconstrained channel type before simulation.

```
proc double (in inbus)
    bus outbus {
        val: u5 range 0 to 25;
    };
{
    outbus.val = inbus.val * 2;
}
```

Listing 3: Example of constrained channel type and range after simulation.

Listing 4: Example of channel types before and after simulation.

4.1.1.3 Co-simulation

Often when modelling hardware in HDLs like VHDL or Verilog, code for testing and verifying are often written in the same language as the design itself. Unfortunately, writing test codes in HDLs are often tedious. Using general-purpose languages for testing hardware models are useful since the range of available libraries is much larger. Therefore the SMEIL simulator provides a simple language-independent API which enables SME implementations written for general-purpose languages to communicate with SME networks written in SMEIL, so-called co-simulation. The big advantage of the SMEIL approach to co-simulation is that SME is used on both sides of the co-simulation and therefore can act as a single entity. The PySME [14] library has been extended to support co-simulation with SMEIL, thereby providing the possibility of writing networks in PySME which can interact with SME networks written in SMEIL. Currently, only PySME has been extended with this, but is expected to be implemented in the other SME implementations as well. When simulating the SMEIL network, the compiler can record a trace of all communication between processes in the network. This trace file can then be used as a source for the VHDL test bench, which can be used to verify the generated VHDL code.

4.2 CSP

CSP is a process algebra that provides a method to express interaction between processes of concurrent systems. CSP had a lot of influence in the design of several programming languages such as Occam as well as the Go programming language. As described in Chapter 2, it was first introduced in 1976 by C.A.R Hoare but, not until 1984, did it develop into a process algebra. CSP is still a large research subject, and due to the tools available today, it is becoming increasingly more accessible for the general users.

CSP provides the possibility of describing systems in terms of message passing communication between independently operating processes. It can be argued that the *sequential* part of the CSP name is not providing a full understanding of the current version of CSP. This is because the current version of CSP allows not only sequential processes, but also processes consisting of parallel compositions of other processes. By using message passing between processes the language avoids certain problems that arise with the use of e.g shared variables. It is possible to describe complex concurrent and parallel structures with a few simple elements of the CSP algebra.

In CSP, a process is mainly defined by the way it can communicate with its external environment. An essential part of this communication is the *alphabet* of possible communication events. This is the set of events that the process, and related processes, can use for communication. Events in CSP can only occur when both the process and the environment allow, but then they are instantaneous.

Here I give a brief introduction to the essential parts of CSP and structures that are used in TAPS, in order to give the reader a basic understanding to build on with CSP$_M$ and FDR4. To learn more in-depth about CSP the reader is encouraged to seek out the books *The Theory and Practice of Concurrency* [53] or *Understanding Concurrent Systems* [54] by A. W. Roscoe.

- **Prefix:** Given an event a in the alphabet Σ, and a process P, $a \rightarrow P$ is the process which will communicate a and then behaves like the process P. The process will wait indefinitely to communicate a. This is known as prefixing. The simplest CSP process possible is the *STOP* process. It never does anything and never communicates and it is also known as the deadlocking process. The process $left \rightarrow right \rightarrow STOP$ is the process which communicates a left and a right before stopping. Another essential process is the *SKIP* process which is the process that terminates successfully.

- **Recursion:** The process described above ends with *STOP* and never resumes, but if I wish to have a process that performs the communication indefinitely, then recursion is used. $P = left \rightarrow right \rightarrow P$ is the process that performs left and right in an infinite loop.

- **Events:** As described above, the only possible communication between processes are events within the defined alphabet. If $A \in \Sigma$ then the process $?x : A \rightarrow P(x)$ is the process which accepts an element x of the set A and then behaves like the process $P(x)$. It can often be valuable to have a "channel" to communicate values between processes. Therefore if c is a channel, then $c?x : A \rightarrow P(x)$ is the process which receives an element x on the channel c. This can also be written simply as $c?x \rightarrow P(x)$ where A is the type of c. The "outputs" can be written in two ways, $c.x$ and $c!x$, which are equivalent. An example of a simple process receiving an element x on the channel c and outputs it again on the channel c are $P = c?x \rightarrow c!x \rightarrow P$.

- **Guarded Alternative:** The guarded alternative provides the environment a choice between different actions. A simple choice statement could be $a \rightarrow P \mid b \rightarrow Q$ which gives the environment the possibility of choosing either to communicate a and then behave as P or communicate b and then behave as Q.

- **Deterministic Choice:** Also called external choice is the operator which offers the environment the choice between two different processes. $(a \rightarrow P) \Box (b \rightarrow Q)$ is the process where the environment chooses to either perform event a and then behave as the process P or perform event b and then behave as the process Q. This is completely the same as the guarded alternative example given above, and usually it is equivalent, however, some advanced uses of external choice might be hard to express simply with guarded alternative.

- **Nondeterministic Choice:** Also called internal choice, defines the choice between two processes but is not affected by the environment to make the decision. The process $(a \rightarrow P) \sqcap (b \rightarrow Q)$ can perform event a or b and then behave like the corresponding process, but it does not have to accept either. It is only required to accept one if the environment offers both of them.

- **Conditional Choice:** Conditional choices are also possible in CSP. Conditionals can be represented in different ways, which will be explained further in the CSP$_M$ section below. Hoares syntax for conditionals are $P \lhd b \rhd Q$ which is the same as if b then P else Q.

- **Interleaving:** This operator represents concurrent activity between two independent processes. $P \mid\mid\mid Q$ defines a process behaving like P and Q simultaneously.

- **Generalised Parallel:** This operator represents concurrent activity between two independent processes but where the processes are required to syncronise on the set of events, defined in the operator. $P \underset{a}{\parallel} Q$ defines the process where P and Q must syncronise on event a before the event can occur. All events not defined within the synchronisation can happen at any given time.
- **Alphabetised Parallel:** Imagine two processes that run in parallel. Some of the communication is with each other, but others are not. In this case, it is not possible to use the generalised parallel operator, since the processes also have to be able to communicate outside of the synchronisation. $P \ {}_X\|_Y \ Q$ is the alphabetised parallel operator where P is allowed to communicate to the set X and Q is allowed to communicate with the set Y. The two processes will have to synchronise on all communications within $X \cap Y$.
- **Hide:** This operator represents a process which performs any event from the defined set, but the event is hidden and becoming an internal event, a *tau* (τ).

4.2.1 CSP$_M$

In 1998 Bryan Scattergood introduced a machine-readable version of CSP, called CSP$_M$. This language is an adaption and extension of the process algebra that Brookes, Hoare, and Roscoe introduced in [55]. The process algebra notation is extended with a functional programming language to be able to express CSP in a functional way, resulting in a language that can be used with tools and therefore providing a broader usability of CSP.

CSP$_M$ use the existing theory of CSP and in spite of being combined with a functional programming language, CSP$_M$ retains the idioms of CSP. The main purpose of creating CSP$_M$ was not so it could be executed, like most programming languages. The main reason for creating CSP$_M$ was to provide an approach to describe parallel systems in a structure which can be automatically manipulated, with tools like FDR4.

4.2.1.1 Essentials of CSP$_M$

The basics of a CSP$_M$ program are to define the processes, their functionality, and the communication. CSP$_M$ supports sequences, sets, booleans, tuples, user-defined types, local definitions, pattern matching, and lambda terms. CSP$_M$ does not enforce restrictions on the use of upper/lower-case letters in identifiers, although some standard conventions can be followed if needed. Some conventions define processes either all in capital letters or with an initial capital letter. Channels and function naming are typically all in lowercase lettering.

CSP$_M$ also supports integer arithmetics like addition, subtraction, product, quotient, remainder/modulus, and unary minus. Integer arithmetics are supported within a signed or twos-complement 32-bit representation. Floating-point numbers are not supported in CSP$_M$.

CSP$_M$ provides several different built-in functions, like `concat(s)` or `elem(x,s)`, which both takes sequences as parameters, or `card(a)` and `union(a, b)` which are set functions. All of these functions are common in functional programming languages. CSP$_M$ support boolean `and`, `or`, `not` operations, equality operations as well as ordering operations and conditional expressions. In CSP$_M$, all types except functions and processes can be compared with the equality operators. Ordering is defined for all but booleans and user-defined types.

Local definitions can be defined by wrapping them inside a `let within` statement. For example `P(n) = let (x,y) = n within x + y` , where `x` and `y` are local definitions.
Channel and datatype definitions can only be defined at top-level and therefore not in a `let within` statement. CSP$_M$ provides support for lambda terms which have the syntax $\setminus x_1, ..., x_n @ x$.

An example could be `f = \x, y @ x % y`. It is simple and ideal to use pattern matching in a lot of cases in CSP$_M$, similar to most functional programming languages. Case statements can be defined via pattern matching as well as comprehensions, communications etc. There are several rules that belong to pattern matching within CSP$_M$, that will not be introduced here. For more information see the CSP$_M$ Reference Manual [57].

In CSP$_M$ a channel is defined with the keyword `channel` and an identifier as a tag. Channels can be simple structures like `channel clock` or more complex like `channel c : {0..10}` which represents a channel `c`, that can communicate integers in the range of 0 through 10. A channel becomes an event when all necessary values have been provided. With channel `c`, the value `c.5` is of type `Event`. All events are members of the built-in data type `Events`. It is possible to define channels to be infinite, but when using CSP$_M$ in FDR4, it is not common use, since FDR4 would run out of space due to the state space becoming too large.

4.2.1.2 CSP$_M$ Processes

In CSP$_M$, processes are defined much like described in section 4.2, above. `SKIP` and `STOP` are defined the same, but external choice has a new syntax adapted to the functional language. Communications are also performed with the `?` and `!` operator. For example, `Proc(c) = c?x -> c!x -> Proc(c)`, is the process which receives an element on the channel `c` and outputs it on the same channel again. Sequential composition is defined with a semicolon, for example, `P;Q` is first behaving like `P` until it terminates and then it behaves like `Q`. External choice is defined with `P [] Q` and internal choice with `P |~| Q`. A boolean guard, `b & P`, is the same as `if b then P else STOP`. Generalised parallel is defined by `P [| A |] Q`, similar to how it is defined in the section above. It is worth noting that `P [|{}|] Q` is the same as using the interleave operator.
It is also possible to define alphabetised parallel, `P [A || B] Q`, which means that the set of events, `A`, are the events that `P` is allowed to perform and the set of events, `B` are the events allowed by `Q`.

4.2.2 FDR4

Note that the information represented in this section is based on the information published, which unfortunately does not include a lot of details about the internal workings of FDR4. I have assumed that the internal workings of FDR4 are similar to FDR3, which has been described in [28].

Today, there exists several tools for formal verification. One of the currently most favored tools is the Failures-Divergences Refinement tool (FDR4). FDR4 is a CSP refinement checker that can analyse programs written in CSP$_M$. It is an obvious choice as the verification tool for the generated CSP$_M$ programs from TAPS, as it has the ability to perform parallel refinement-checking, which drastically increase the number of states it is able to check. FDR4 is also an actively maintained tool, so it is less likely to become obsolete.

Using the denotational models of CSP, *traces*, *failures*, and *failures-divergences*, FDR4 is able to check if an implemented process refines a specification process. FDR4 also has the functionality of checking more properties than these models, like deadlock freedom or livelock freedom.

FDR was first released in 1991 and has since then been used both in academia and in the industry. The FDR4 tool incorporates features like a CSP$_M$ type checker, a debugger as well as a built-in version of ProBE, the CSP process animator mentioned in Chapter 2. FDR4 provides a parallel refinement-checking engine that can scale up linearly with the number of cores. This means that it can handle processes with a very large number of states in a reasonable time, which, along with compression, provides the possibility of verifying even larger problems. FDR4 also provide a cluster version, also providing verification of larger problems.

An assertion in FDR4 contains a specification process as well as an implemented process to verify. For instance, `assert P [T= Q` means to assert if `Q` refines `P` according to the trace model. FDR4 converts the two processes to a Labelled Transition Systems, LTS. A transition system consists of states and transitions between states which gives the opportunity of exploring the state space of the process. FDR4 specifies the LTS further by using Generalised LTS (GLTSs). A GLTS can have individual states being labeled with different properties, depending on the semantic model used for the refinement. For instance, if the trace model was used, then the states would be labeled with the traces in a GLTS. FDR4 transforms both the specification process and the implementation process into GLTSs, which can then be checked for refinement. In order to be able to perform the verification, the specification GLTS is normalised, which has been extensively introduced in [53], but in short, the GLTS is transformed into a deterministic GLTS without any internal transitions or τ transitions. This can be an extensive operation, but since specification processes often are quite simple, this has proven to be the best approach. After this, FDR4 is ready to check if the implemented process, transformed to a GLTS, refines the normalised specification GLTS, which is further introduced in section 4.2.2.1. Since it is only necessary to normalise the specification process, the automatic refinement check becomes much more feasible than if it was necessary to normalise both.

FDR4 supports other compression functionalities besides normalisation, which can reduce the state space without changing the overall semantics of the process. This helps improve the runtime of the refinement checks. The reduction in state space is a crucial element of FDR4, since the state space quickly becomes overwhelmingly large, making the refinement check infeasible. The compression algorithms provide the possibility of verifying otherwise absurdly large problems. These compression algorithms will not be further introduced in this thesis, but more information on these can be found in [54]. FDR4 also supports refinement of Timed CSP [51, 12] which is used within time-critical systems.

4.2.2.1 Refinement Checking

Refinement checking consists of searching through each of the GLTS, checking that all reachable states within the verification GLTS are compatible with every state within the normalised specification GLTS. It performs a breadth-first search, which has the advantage that when an error is found, the counterexample provided is the smallest possible. In each state, properties are checked according to the semantic model used. The refinement algorithm stores a set of states in order to keep count of what has been checked, what is currently being checked and what is left to be checked. FDR4 uses B-Trees to store data during refinement, and since version 3 of FDR, FDR3, the refinement can also be performed in parallel. I will now introduce the three basic semantic models of CSP.

Traces model and refinement A trace is the sequence of communications between the process and the environment. Each element in the trace is the record of a single communication with the process, and a trace can be used to verify the correctness of a process in CSP. In the CSP algebra, refinement is written with the symbol $\sqsubseteq$, and usually with a specific semantic model defined as well. Trace-refinement is specified by:

$$P \sqsubseteq_{\mathrm{T}} Q \text{ if and only if } traces(P) \supseteq traces(Q)$$

This means that Q *trace-refines* P, or in other words, that every finite trace of Q is a trace of P. Some example are:
$traces(STOP) = \{\langle\rangle\}$
$traces(a \rightarrow STOP) = \{\langle\rangle, \langle a\rangle\}$
$traces((a \rightarrow STOP) \,\Box\, (b \rightarrow STOP)) = \{\langle\rangle, \langle a\rangle, \langle b\rangle\}$.

Failures model and refinement Traces are great at telling what the processes do, but they do have limitations. For example, $a \rightarrow P \sqcap STOP$ and $a \rightarrow P$ have the exact same trace, even though one has the possibility of *STOP* and the other does not. This is why it, in some cases, is necessary to look at not only what the process can do, but also what it can refuse to do. A refusal is a set of all the events that a process can fail to accept, no matter how long it is offered. A failure is a pair (s, X) where s is a trace and X is the set of events that the process can refuse after the trace s.

failure(P) is the set of all failures within the process P. The GLTS is traversed and all routes through the graph are collected while ignoring τ actions. This results in a trace and if the node at the end of each trace is stable, which means that there are no τ actions going out of it, this will give a failure since the node must refuse all actions that do not lead out of it. However, if the node has a τ action leading out from it, also called an unstable node, then it does not give a failure since the internal events will eventually happen.

An example of four different graphs can be seen in Figure 4.4. It is assumed that the alphabet is $\{a, b, c\}$ and as can be seen, all stable notes have a subset of the alphabet next to it. These are the set of possible refusals for that node.

- P_1 is a deterministic process, since it contains no τ events. A subset of the failures for this process are ($\langle\rangle$, $\emptyset$), ($\langle\rangle$, $\{c\}$), ($\langle a\rangle$, $\{a, c\}$) and ($\langle a, b\rangle$, $\{a, b, c\}$).
- P_2, P_3 and P_4 are nondeterministic due to the internal events.

I know that

$$P \sqsubseteq_{\mathrm{F}} Q \text{ if and only if } traces(P) \supseteq traces(Q) \text{ and } failures(P) \supseteq failures(Q)$$

When looking at Figure 4.4 it is clear that both P_2 and P_3 trace refines P_1, and they also have equivalent traces. In this example, only P_2 failure-refines P_3.

Failure-Divergences model and refinement Traces and failures are some very powerful models, however they do not account for divergence, which is addressed by the failure-divergence model. Divergence is a situation where a process performs an endless series of τ events. If a stable node in a transition graph, like the one in Figure 4.4, performed endless internal actions, it would be a process that diverges. In the failure-divergences model, a process is represented by $(failures\perp(P), divergences(P))$. $divergences(P)$ represents the set of traces where P can actually diverge afterwards. $failures\perp(P)$ represents the set of failures of P extended by all the failures from the traces in $divergences(P)$, that is $failures(P) \cup \{(s, X) | s \in divergences(P), X \subseteq \Sigma\}$. In a transition graph, divergences occur where a process can reach itself with a number of τ events, i.e a τ-loop.

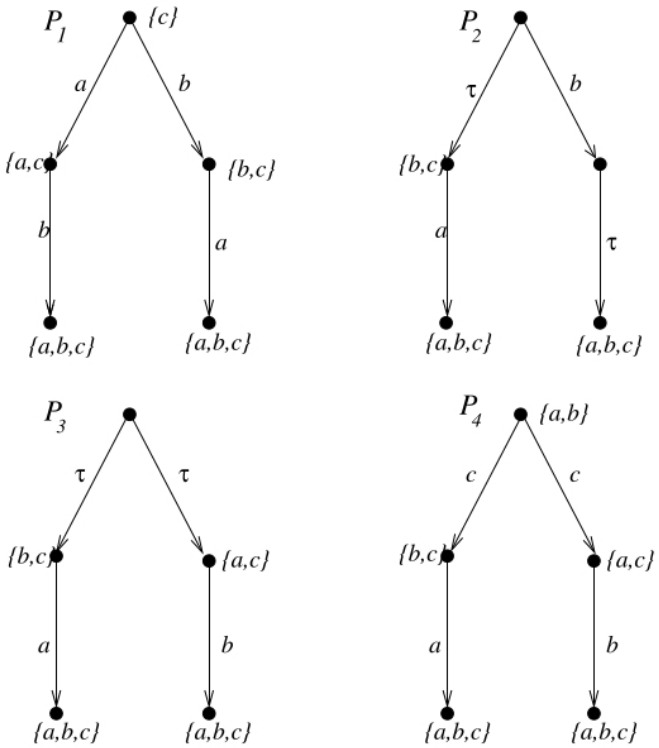

Figure 4.4: Four different transition graphs and their refusal sets. Figure from [54].

Chapter 5

Analysis

In this section, I will analyse the differences between the SMEIL model and the CSP_M language. I will also introduce important parts of the SMEIL structure and grammar to provide an understanding of the different challenges that lie in translating a hardware model to a specification model.

5.1 Verification

The reason for translating SMEIL to CSP_M is to use the refinement checking properties of the CSP language. By loading a CSP_M program into FDR4, it is possible to utilise these properties on the SMEIL network. Before I can design the transpiler and generate the CSP_M code, it is essential to examine what kind of properties would be beneficial to verify in hardware. It is also relevant to explore how these properties can be verified with FDR4. As mentioned in Chapter 4, FDR4 provides refinement checking with different models, like the traces and failures models. FDR4 is often used for deadlock checking, but since the SME model guarantees deadlock freedom, it is unnecessary to use this property within FDR4.

In hardware, it is typically relevant to verify that the communication on a bus does not exceed a certain range, or that the sum of multiple signals does not exceed a specific value. A bus might be able to carry unexpected data, and being able to formally verify that this never happens, is of great value. Therefore it would be interesting to verify that certain values are never communicated on a specific channel, or that they are the only values communicated on the channel. When designing hardware, each component is designed separately, and usually, several components are combined to create a larger network with more functionality. When modelling a small component, it can be difficult to predict the possible input values for the component, especially when parallel structures are involved. It is therefore interesting to be able to verify that a specific range of input values will not cause failures in the parallel model.

To provide an understanding of the type of problem that this kind of verification would be able to solve, an example is introduced in Section 5.1.1, which will be referred to several times throughout this thesis.

5.1.1 Seven Segment Display Example

A seven segment display is an electronic display device which is used in digital clocks or other devices that display numerals. An example of a typical digital clock display can be seen in Figure 5.1. As the name states, each display consists of seven segments which can be lit up in patterns to display symbols, like numerals. When a digit has been determined for a seven segment display, it is encoded to a bitstream that represents the digit in the correctly activated

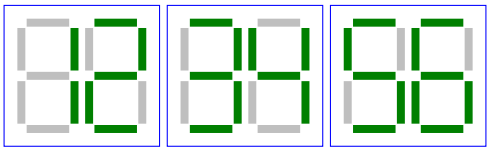

Figure 5.1: Digital clock with six seven segment displays, displaying 12:34:56.

display segments. In this example, I wish to model a typical digital clock that is able to calculate and display the current time in hours, minutes, and seconds. Listing 5 shows a similar example written in Python. Like `time_since_midnight` in Listing 5, the network must have some input of data. The input value represents seconds since midnight, and in order to calculate hours, minutes, and seconds, three different processes are modelled, called the `time` processes in this example.

When writing hardware models in pure SMEIL, one of the possible ways to generate input for the network is to create a data generator process. This process, called the `clock` process in this example, is instantiated with a start time and is incremented by 1 for each simulation cycle, representing a one second increase. The result of the `clock` process is communicated on the process output bus where the three `time` processes are listening. These `time` processes receive the number and by the use of simple integer arithmetic, calculate the hours, minutes, and seconds since midnight respectively. It is obvious that at some point in time, each `time` process will calculate a two-digit result, for example at 12 hours or 42 seconds. However, a single seven segment display can only show one digit between 0 and 9. Therefore it is necessary to have two seven segment displays for each `time` process, in order to show the correct time in a 24-hour interval. Each `time` process has an output bus with two individual channels that represent the communication to each different display. The number representing either hours, minutes, or seconds are separated into first and second digit, by $\lfloor \frac{x}{10} \rfloor$ and $(x \bmod 10)$. These six different results are then communicated onto the six different channels which represent the six different seven segment displays. The outline of this network can be seen in Figure 5.2 where the network consists of four processes. The data generator process, I, which creates the input that is broadcast out on the network. The three `time` processes, hours (H), minutes (M), and seconds (S) are the processes described above, which calculate each part of the current time. The outputs are communicated on the six outgoing channels.

```python
from math import floor

def time(time_since_midnight):
    hours   = floor(time_since_midnight / 3600)
    minutes = floor((time_since_midnight - hours * 3600) / 60)
    seconds = time_since_midnight - hours * 3600 - minutes * 60
    return [hours, minutes, seconds]

print(time( 57100)) # =>  [15,51,40] or 15:51:40
print(time(  3601)) # =>  [01,00,01] or 01:00:01
print(time( 66666)) # =>  [18,31,06] or 18:31:06
```

Listing 5: A Python implementation of the seven segment display example.

In this example, the properties I wish to assert with FDR4, are the width of the channels. That is, I want to prove that certain values will never be communicated on certain channels. One could imagine that 4 bits can be communicated between the `time` processes and the seven

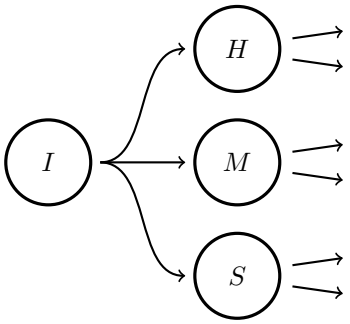

Figure 5.2: SMEIL network for a seven segment display clock. Each SMEIL process is represented by a cicle with a letter corresponding to the processes `input`, `hours`, `minutes` and `seconds` respectively.

segment displays. But 4 bits can represent the numbers 0 through 15, and the seven segment displays can only display the numbers 0 through 9. Therefore I wish to assert that even though the channels can carry 4 bits, the actual communication on the six output channels does not exceed 9. In general, the displays will always be able to display 0 through 9, but since the example is a clock showing a 24-hour interval, the displays will of course not be able to show minutes and seconds above 59 and hours above 23. The full SMEIL code for this example can be seen in Listing 45 in Appendix B.

5.2 Transpiling from SMEIL to CSP$_M$

Since I am translating code from SMEIL to CSP$_M$, the challenge is to find CSP$_M$ structures that correspond to the SMEIL structures. The ultimate goal is to find methods for transpiling, that can be generalised to most problems. From the introduction to both SMEIL and CSP$_M$, it is clear that the main intention of each language is different and therefore the transpiling of a process in SMEIL to a CSP$_M$ process might not be completely trivial.

When analysing the general structure of SMEIL programs, there are three structures that are particularly interesting in terms of translating from SMEIL to CSP$_M$. These three structures are:

1. **Behavioral:** The general behaviour of each process.
2. **Structural:** How the circuit is connected, i.e which buses connect to which processes.
3. **Verification data:** All data from the SMEIL network that could be useful for the verification of the hardware model.

5.2.1 Behavioral

The processes in an SMEIL program is describing the basic behaviour of the SMEIL program. It is necessary to analyse the different process structures in SMEIL in order to understand how it can be translated into CSP$_M$.

5.2.1.1 Processes

An SMEIL process is defined by the `proc` keyword and consists of an identifier, process parameters, bus, and variable declarations. The body of the process, the process statements, consists of sequential statements such as communications and calculations that are to be evaluated once for each clock cycle. A small example of an SMEIL process has been presented in Listing 1 in Chapter 4.

A process is initiated in the network of an SMEIL program, which will be explained further in Section 5.2.2. A process can be instantiated with a set of parameters. These parameters can be a mix of input and output buses and constants, and similar parameters can be defined for a CSP$_M$ process. In CSP$_M$ the process behaviour also represents functionality, but the behaviour is often more focused on communicative behaviour between processes.

A typical process in CSP$_M$ could look like the one in Listing 6 which represents a philosopher process from the dining philosophers problem. The process behaviour shows the actions of a philosopher process when communicating with other processes. As can be seen, the process also conduct some simple calculations, but the main structure consists of communication, which is the main purpose of CSP$_M$, to be able to describe communication between processes.

```
1 PHIL(i) = thinks.i -> sits!i -> picks!i!i -> picks!i!((i+1)%N) ->
2           eats!i -> putsdown!i!((i+1)%N) -> putsdown!i!i -> getsup!i -> PHIL(i)
```

Listing 6: A dining philosopher process from the dining philosophers problem example file provided at the FDR4 webpage [1].

In the SMEIL grammar, it is possible to declare a process to be synchronous or asynchronous by the keywords `sync` and `async`. A `sync` process is run during each clock cycle and an `async` process is only run when it receives a signal on the input bus. The current implementation of SMEIL does not support `async` processes, so all SMEIL process are synchronised and behave like described in Chapter 4, where they read, compute and write once in each clock cycle. The synchronicity of the processes is an essential part of the SMEIL network and therefore it is crucial that the generated code can model this correctly. In CSP$_M$ a process does not communicate until it receives an input, which matches the asynchronous process of SMEIL. However, since only the synchronous processes are supported in SMEIL, it is necessary to introduce a structure to the CSP$_M$ network that simulates the synchronous structure of the SMEIL process.

In an SMEIL process, the declarative part of the process can consist of different variables and constants as well as bus definitions. Next, variables and constants are introduced, whereas buses are described in section 5.2.2.

Variables and Constants It is not possible to declare variables, constants, or communication buses inside the process statements of an SMEIL process. Therefore all of these must be declared in the declarative part of the process, which should simplify the translation to CSP$_M$ since it will only be necessary to search the declarative part of the process to find all variable and bus definitions.

In SMEIL, constants consist of an identifier, a type, and an expression. The expression could be an integer assigned to the constant, for instance:

```
1 const hours: uint = 24;
```

Variables are very similar to constants but simply hold mutable values within the process. In SMEIL processes, variables preserve their values between clock cycles, which mean that it is possible for a process to save a value or result, and reuse it in the next clock cycle. The difference between variable and constants in SMEIL are that variables can contain an optional expression and an optional range of values, for example:

```
1 var minutes: u6 = 0 range 0 to 59;
```

The assignment `u6 = 0` is providing an initial value to the variable, but both initial values and ranges are optional. The range declaration defines the range of expected values for the variable.

In CSP$_M$, constants can be declared as we know it from many other languages and variables can be used without declaring them first, for example in a process declaration within a local definition.

```
1 N = 5 -- Constant declaration
2
3 Proc(x) =
4     let
5         m = x % 60 -- Variable without declaration
6     within
7         channel ! m -> SKIP
```

A variable is also defined when communicating specific values over channels in CSP$_M$. For example in the example below, the received value is assigned to the variable x which is then written to the `output` channel.

```
1 P = input ? x -> output ! x -> STOP
```

In SMEIL, the type of a variable, a constant or a bus channel, must always be defined. A type can represent either unlimited size integers, `int`, and `uint`, or be restricted to a specific bit-size. The prefixes `i` and `u` refer to signed and unsigned integers followed by a number determining the bit-length. For instance, `i4` represents a signed 4-bit integer. The unlimited size integers are not practical to have in CSP$_M$ because when verifying a program, FDR4 will look at the entire possible state space and with unlimited integers the state space will be unbounded. FDR4 restricts its integer types to signed 32-bit [27]. The LIBSME compiler provides, because of its type system, a warning if the values of a constant or variable are above or below the range of the type bit size. In the example below the number 59 cannot be represented by 4 bits and therefore it would cause a warning from the compiler.

```
1 var minutes_wrong_type: u4 = 0 range 0 to 59;
```

Translating constants and variables from SMEIL to CSP$_M$ will not prove to be difficult since the structures are quite similar. A potential challenge lies in deciding how to define a variable that has been declared with an initial value in SMEIL. The CSP$_M$ translation will have to define the variable before it is used within the calculations, which might increase the complexity of the translation.

The second part of an SMEIL process is the statements where the actual behaviour of the process is defined. The semantics of a statement in SMEIL corresponds to what we typically see in C-like languages, some are mentioned here below.

Assignments An assignment in SMEIL consists of a name and an expression. It can be used in two different ways within the SMEIL process statements: assigning to a variable and assigning to a bus channel. In SMEIL, the compiler will always be able to recognise what is being assigned by looking at the type of the object and therefore SMEIL does not differentiate between these two assignments. This property will cause a challenge when translating to CSP$_M$ since communication and variable assignments can be two different things in a CSP$_M$ program. Thus TAPS must recognise the type of the assignment, in order to create the correct translation.

if-statements `if` statements in SMEIL are structured similarly to most other programming languages with the keywords `if`, `then`, `elif` and `else`. `if` statements can also be defined in CSP_M, but CSP_M does not support the `elif` expressions, so the translation must restructure the `elif` expression to a nested `if then` expression. These `if` statements can quickly become quite complex, but it should not be difficult to automatically generate.

To translate processes from SMEIL to CSP_M is a challenge because they cannot be translated directly. The challenges lie in structuring the CSP_M processes so that the CSP_M code provides the same functionality as the SMEIL model while still keeping the CSP_M communication behaviour and allowing assertion properties.

5.2.1.2 Generating data

It will always be necessary to generate input for pure SMEIL networks. If the program was not written as a pure SMEIL program, the input for the network would be provided by the surrounding code, but in the case of pure SMEIL, the input data must be generated, which can be done in a few ways. One way of initialising the data in the SMEIL network is by instantiating the process with a constant given as a parameter or hardcode internal values into the process. Another way is to have a separate process that generates data for the network. I call this type of process a data generator process. Examples of both can be seen in Listing 7 and in Listing 8.

Listing 7 shows an example of the network `addone_network` with two instantiated processes. The `add` process reads an input value and a constant, add the two values together, and writes it out onto the output bus. The `id` process reads an input and writes it to the output bus immediately. Networks in SMEIL are introduced further in Section 5.2.2, but the network in this example is defining the input bus of the `add` process to be the output bus of the `id` process, as well as defining that the constant value in the `add` process is 1. The `addone` example will be introduced further in Chapter 8.

Listing 8 shows another way to instantiate data into the SMEIL network. Here the process `clock` is a data generator process. It does not read data from any input bus, so it can only generate data to write to the network. The example shows the `minutes` process calculating the number of minutes that have passed since the simulation started by dividing the input with 60.

The data generator process has the same structure as any other process in SMEIL and it is therefore crucial that TAPS can recognise a data generator process from other processes. An SMEIL process that does not read any input is obviously a data generation process, however, in SMEIL it is possible to communicate to or from buses directly in the process body without defining them as input parameters. This will be explained further in Section 5.2.2.2.

An SMEIL network can consist of different permutations of data generator processes and processes with initial values. This might increase the complexity of the translation but it should be feasible as long as TAPS can separate the data generator processes from other processes.

5.2.2 Structural

The backbone of an SMEIL network is the communication between the processes. Analysing the different communication structures in SMEIL will provide the insight necessary for designing the translation structures to create equivalent communication in the generated CSP_M network.

5.2.2.1 Network

A network in an SMEIL program is a crucial part that connects all processes together with communication. In an SMEIL network, processes are instantiated using the `instance` keyword

```
proc add (in inbus, const constant)
    bus outbus {
        val: uint;
    };
{
    outbus.val = inbus.val + constant;
}

proc id (in inbus)
    bus outbus {
        val: uint;
    }
    var from_add : uint = 0;
{
    from_add = inbus.val;
    outbus.val = from_add;
}
network add_network() {
    instance a of add(i.outbus, constant: 1);
    instance i of id(a.outbus)
}
```

Listing 7: The SMEIL network `add_network` with two processes. The `add` process is instantiated with a value which is constant and used once for each clock cycle. The example is similar to the Addone example in [17].

within the network declaration. This instance declaration instantiates the process with a set of parameters defined for the specific process. Defining the network from process instances also has the advantage that one process can be instantiated with different parameters several times within the same network, providing the possibility of reusing the processes for different purposes. An example of an SMEIL network can be seen in Listing 9, where the `add` process from Listing 7 is instantiated twice with different constant values.

In CSP_M there is no network structure, but CSP_M does have a structure for declaring parallel communication which resembles the instance declarations in SMEIL. The parallel operators define communication between processes in CSP_M, which is described in Chapter 4. While it is possible to model equivalent communication by using these operators it quickly becomes complex because the operators only syncronise two processes at a time. It is a challenge to ensure that an SMEIL network is translated to an equivalent CSP_M structure, and it requires careful planning to provide a complete translation of all the relevant information. However, it is possible due to the simple network structure in SMEIL.

5.2.2.2 Buses and Channels

As previously explained, an SMEIL bus defines a collection of channels which are used for all communication between the processes. Each channel has a type describing the communicated data and can be initialised with an initial value. The syntax of a read or write in SMEIL is `bus.channel` where `bus` specifies the bus and `channel` specifies the channel within the bus. An example of this can be seen in the `add` process in Listing 7 where `inbus.val` refers to the `inbus` bus and `val` refers to the channel within the bus. A bus in itself does not have a type or values, but all channels within a bus have an identifier, types, and values, however the same types do not have to apply to all channels in a single bus. An SMEIL bus does have an identifier which is used for referencing the bus. All channels within a bus are connected to the process at the same

```
proc clock()
    bus outbus {
        val: uint;
    };
    var i: uint = 0;
{
    i = i + 1;
    outbus.val = i;
}

proc minutes (in inbus)
    bus outbus {
        val: uint;
    };
    from_clock : uint;
{
    from_clock = inbus.val / 60;
    outbus.val = from_clock
}


network minutes_net() {
    instance c of clock();
    instance m of minutes(c.outbus)
}
```

Listing 8: The SMEIL network `minutes_net`, with a data generator process and a calculation process that calculates minutes since simulation start.

```
network add_net() {
    instance a1 of add(i.outbus, constant: 1);
    instance a2 of add(i.outbus, constant: 2);
    instance i of id(a.outbus)
}
```

Listing 9: Example of two instantiations of the `add` process from Listing 7.

time, and it is up to the developer to call the correct channel within the bus for either a read or a write. An example of a bus definition in SMEIL can be seen in Listing 10. The bus is identified with the name `day` and it consists of three channels, `hours`, `minutes`, and `seconds`. Each channel is defined with a type, `u5` or `u6` and a range, 0 through 23 or 0 through 59. A single SMEIL bus channel corresponds to a single CSP$_M$ channel and since the channel is a more general structure than the bus structure, it should be relatively simple to translate.

SMEIL Input Bus There are two different methods for accessing an input bus in an SMEIL process. The name of the input bus can be given as a process parameter, which the process can use to read from the actual bus channel, as can be seen in the `add` and `id` processes in Listing 7. In this example, the processes reads the data from the channel `val` in the bus `inbus` and writes the data to the channel `val` on the `outbus` bus.

It is also possible to use the formal identifier of the bus in the process body, thereby not accessing it through the input parameter. An example of this can be seen in Listing 11. Here the

```
bus day {
    hours:   u5 range 0 to 23;
    minutes: u6 range 0 to 59;
    seconds: u6 range 0 to 59;
};
```

Listing 10: Example of an SMEIL bus channel with three channels.

```
proc add ()
    bus outbus {
        val: uint;
    }
{
    outbus.val = id.outbus.val;
}

proc id ()
    bus outbus {
        val: uint;
    }
{
    outbus.val = add.outbus.val;
}

network add_id_net ()
{ instance _ of add();
  instance _ of id();
}
```

Listing 11: Example showing how processes can read from a channel using its formal identifier.

`add` process reads the input by accessing the formal name of the bus channel `val` within the bus `outbus` defined in process `id`. As a result of this, the process can be instantiated several times in the network but it will always read from the channel `id.outbus.val`.

SMEIL Output Bus As with the input bus in SMEIL, there are several methods for defining and accessing the output bus in SMEIL. As seen in Listing 11, the output bus can be defined inside the process itself using the keyword `bus` along with channel definitions. As with input buses, it is also possible to reference the formal name of the channel directly inside the process body in the same way as can be seen in Listing 11. Lastly, it is possible to give the output bus as a process parameter. An example of this can be seen in Listing 12 where the process `add` reads from its input bus and writes to its output bus, both defined as parameters. The input bus for the `add` process is the `outbus` defined in the `constant` process. The output bus of the `add` process is a bus defined inside the network called `netbus`. In this example, the network calculates `1 + 1` in each clock cycles.

SMEIL was designed to be very flexible. Therefore, each process can have any number and combinations of input buses, output buses, and constants as parameters as well as communication directly inside the process body. TAPS will have to be able to translate all these different methods of communication.

```
proc add (in inbus, out outbus)
{
    outbus.val = inbus.val + 1;
}

proc constant ()
    bus outbus {
        val: uint;
    }
{
    outbus.val = 1;
}

network add_const_net() {
    bus netbus {
        val: uint;
    }
    instance const of constant();
    instance add of add(const.outbus, netbus);

}
```

Listing 12: SMEIL example of a process `add` having both input and output bus as parameters.

CSP$_M$ Channels Like in SMEIL, it is possible to communicate either by using a process parameter name or by communicating directly using the channels global name. These similarities between SMEIL and CSP$_M$ should provide a relatively simple translation of both communication methods in TAPS.

An example of a CSP$_M$ process, communicating both via its process parameter and via a global name, can be seen in Listing 13. The process `P` reads a value from the `input` channel, which is the channel `d`. The process then writes the value on the `c` channel and terminates.

```
channel c : {0..10}
channel d : {0..10}

P(input) = input ? x -> c ! x -> SKIP

Network = P(d) -> SKIP
```

Listing 13: Example of different communication methods in CSP$_M$.

5.2.3 Verification data

As described in Section 5.1, one of the goals of this thesis is to verify data communicated in an SMEIL network. For example, in the seven segment display example, it would be highly valuable to verify that the values communicated to the displays are never outside the range of values defined. For TAPS to create the assertions in CSP$_M$, it is essential to examine which values should be allowed to be communicated, and which should not. As introduced in Chapter 4 during the simulation of an SMEIL program, all observed values on each channel are tracked and turned into a range of the maximum and minimum values for that specific channel. During the simulation, the type representing the values of the channel will also be restricted to the

shortest representation possible. For example, if a channel was originally set to be `int` (unbounded), and the observed values from the simulation showed that it did not take on other values than the range -110 to 110, the type could be changed to an `i8` instead, which represents a signed 8-bit integer with a range of -128 to 127.

In Listing 14 an example of the simulated `seconds` process from the seven segment display example can be seen. As explained previously, the seven segment displays can only display digits from 0 through 9, but in this example, the simulation of the system results in a more restricted range of values. As explained in Section 5.1, the `seconds` process cannot write values over 5 for the first digit and 9 for the second. This is also the result from the simulation, as can be seen in Listing 14. It is crucial that TAPS gather the type and range of observed values for each channel in the SMEIL program, in order to create the proper assertions for FDR4.

The main challenge lies in constructing CSP_M structures that can provide simple assertions of the communicated values while still being applicable to the CSP refinement models.

```
proc seconds (in seconds_in)
    bus out {
        first_digit: u3 range 0 to 5;
        second_digit: u4 range 0 to 9;
    };
    var seconds: u6 range 1 to 59;
    var seconds_first_temp: u3 range 0 to 5;
    var seconds_second_temp: u4 range 0 to 9;
{
    seconds = seconds_in.val % 60;
    seconds_first_temp = seconds / 10;
    seconds_second_temp = seconds % 10;
    out.first_digit = seconds_first_temp;
    out.second_digit = seconds_second_temp;
}
```

Listing 14: Example of the simulated `seconds` process from the SMEIL seven segment display example. See the full code in Listing 45 in Appendix B.

Chapter 6

Designing TAPS

The goal of automatic translation is to create a general solution that can fit different types of problems. It is therefore necessary to generalise the different aspects of the translation to find a solution that fits all problems.

To achieve this I have designed the system TAPS which is a transpiler from SMEIL to CSP$_M$. A transpiler is a source to source compiler which takes source code, written in one language, and outputs equivalent source code in another language.

TAPS consists of two main parts, the parser, and the code generator which will be introduced in the following chapters. Based on an SMEIL program, TAPS will generate an equivalent CSP$_M$ program. This can be loaded into FDR4, which can provide refinement checks on the properties specified by TAPS. An overview of the system can be seen in Figure 6.1.

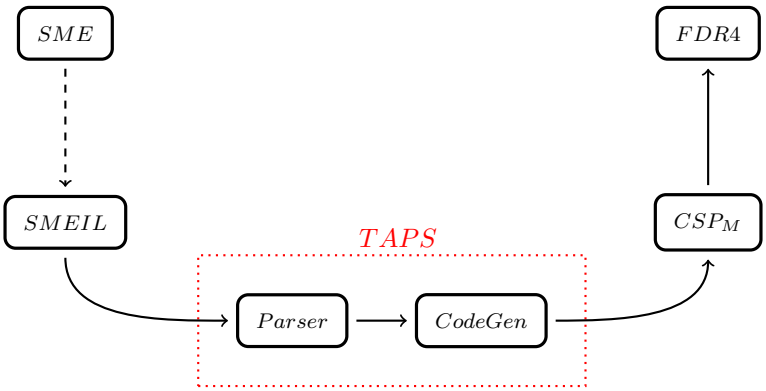

Figure 6.1: TAPS system structure.

It can be difficult to generate code while still retaining the clean structure and organisation that a skilled developer can produce, but this is not the point of automatically generated code. TAPS comply with a set of predefined structures that is used to transform the specific program into a specific solution, using general structures. Because the code is generated using these standard structures, the resulting source code will likely be large and awkward. This is one of the disadvantages of automatically generated code. However, these standard structures are also why automatically generated code is perfect for translating from SMEIL to CSP$_M$. The work that lies in manually translating a hardware model to a specification model can be substantial. The code generated by TAPS will mostly be used by FDR4 to run refinement checks, so the complexity and efficiency or the code structure are of less importance.

In this chapter, I describe the main design decisions behind TAPS and address the challenges that lie in translating SMEIL to CSP$_M$.

6.1 Translating Processes

In order to translate an SMEIL process into a CSP$_M$ process, it is necessary to create a general process structure in CSP$_M$. The SME model enforces that each process reads, computes, and writes, in that order, for each clock cycle, so the CSP$_M$ structure must support this. To reduce complexity, it is preferable to have one CSP$_M$ process per SMEIL process, since it would simplify the translation. A simpler solution typically results in a less erroneous solution.

What first comes to mind in CSP$_M$ when a process must read and then write, is the simplest possible process structure using prefix and communication operators.

```
1 Proc = c ? x -> d ! x-> SKIP
```

This is one of the simplest CSP$_M$ processes that match the SME model, however with this structure, it is not possible to include all the possible computations that SMEIL support. By instead using the `let within` structure in CSP$_M$, it is possible to keep the communication together with the computations in one CSP$_M$ process, while keeping a simple structure. All computations are performed inside the `let` section which can be referenced to inside the `within` section where each write is performed. In Listing 15, an example of the `let within` structure can be seen. This structure will work as a general translation structure from SMEIL processes to CSP$_M$ processes. Since all reads and computations must be performed before writing, this structure should always work for a well-structured SMEIL process.

```
1 channel c : {0..100}
2
3 P(input) =
4     let
5         x = input * 5
6         y = input * 10
7     within
8         c ! x -> c ! y -> SKIP
```

Listing 15: Example of the `let within` structure used to create the general process structure within CSP$_M$.

6.2 Translating Data Generation

It is important to make sure that TAPS can translate all the different methods of generating data in SMEIL, and it is therefore essential that TAPS can recognise a data generator process in SMEIL. As previously explained, a data generator process in SMEIL does not read any input value, neither through a process parameter nor by using the channels formal name. Therefore TAPS must examine the process body, and if the process does not contain input communication, then TAPS will conclude that the process is a data generator process.

In CSP$_M$, it is possible to create a process with the same functionality as the SMEIL data generator process. However, in that case, it would be necessary to synchronise the data generator process with all the processes reading the data. Otherwise, FDR4 would evaluate all values in

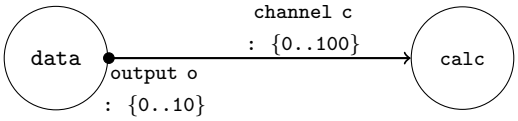

Figure 6.2: A CSP$_M$ network with two processes. The output `o` of the process `data` is within the range 0 through 10, and the channel `c` is defined for the range 0 through 100.

the defined range of the channel, instead of what was actually communicated on the channel. This would make the data generator process redundant. This extra synchronisation and process added to the CSP$_M$ network will increase the complexity, and it might also increase the runtime of the verification. A figure to visualise this concept can be seen in Figure 6.2. The channel `c` between the `data` process and the `calc` process is defined for the range {`0..100`}. If the two processes are not synchronised on this channel, the two processes do not have to agree on communication and therefore the search space for FDR4 includes all the values from 0 through 100, even though the `data` process only outputs 0 through 10. So, if the processes are not synchronised on the channel, the result would be the same as if the `data` process did not exist.

However, if the processes are synchronised, FDR4 will still allocate all 100 possible communications, but it only continues the search on the values actually communicated on the channel. Adding a data generator process, the data generated for the network could be made more complex which, in some cases, could be an advantage. However, the current goals are to verify that the hardware model handles all communicated data as expected. For this kind of verification, it is necessary to have FDR4 search a larger value space to find potential problems. It is important to find a balance between verifying input values outside of the expected range space and verification runtime. The larger the state space in FDR4, the longer the verification time.

```
1 channel clock : {0..100}
2 channel minutes_outbus_val : {0..1}
3
4 Minutes(input) =
5 let
6     from_clock = input / 60
7 within
8     minutes_outbus_val ! from_clock ->
9     SKIP
```

Listing 16: Example of the translated `Minutes` process defined in Listing 8 in Chapter 5.

To summarise, when generating data in CSP$_M$ from an SMEIL data generator process, the two possibilities are either to define the data in CSP$_M$ by one or more data generator channels or by a data generator process and synchronisation. Currently TAPS only support generating a data generator channel from the SMEIL network, but translating the SMEIL data generator process to a CSP$_M$ data generator process would be the same as translating other SMEIL processes to CSP$_M$. Listing 16 shows the translated code of the SMEIL network in Listing 8 in Chapter 5. Here it can be seen that the `clock` process from the SMEIL network is translated into a data generator channel `clock` in CSP$_M$. The translation of the range values defined for each channel is described later in this section.

In the case where the SMEIL network does not have a data generator process but is instantiated with constants as parameters or internal values, the CSP$_M$ processes would also be instantiated with these values as parameters.

If there are several data generation processes in the SMEIL program, TAPS will not handle the processes any differently than if there was only one data generator process. Each data generator process will be translated into a CSP$_M$ channel, and the communication to the rest of the network will be kept intact because of the communication specified in the SMEIL network structure.

6.3 Translating Buses and Channels

As has previously been introduced, the SMEIL bus channel and the CSP$_M$ channel are equivalent, and can therefore be translated almost directly. However, when the bus is defined as a process parameter within the SMEIL process, the translation becomes more complex, because the parameter name, used inside the process, is a placeholder for the formal bus channel name. It is therefore not possible to translate the channel without either knowing the formal name of the bus channel or changing the structure of the SMEIL process. When translating the SMEIL process to CSP$_M$, it is important to design a well-formed structure to handle these processes parameters.

If a process contains an input bus parameter, there are several different methods for translating this input bus parameter to CSP$_M$, each with its own advantages and disadvantages. Listing 17 shows a simple SMEIL process which takes an input bus as parameter. The three other examples in Listing 21 show different translations.

```
proc double (in inbus)
    bus outbus {
        val: uint;
    };
{
    outbus.val = inbus.val * 2;
}
```

Listing 17: An SMEIL process with an input bus as parameter.

```
Double(input_val) =
    let
        result = input_val + 1
    within
        double_outbus_val ! result ->
        SKIP
```

Listing 18: A CSP$_M$ process with input value as parameter.

```
Double() =
    other_outbus_val ? value ->
    let
        result = value + 1
    within
        double_outbus_val ! result ->
        SKIP
```

Listing 19: A CSP$_M$ process with no input parameter. `other_outbus_val` is the formal name of the input bus, defined in the process `Other`.

```
Double(inbus_channel) =
inbus_channel ? value ->
    let
      result = value + 1
    within
      double_outbus_val ! result ->
      SKIP
```

Listing 20: A CSP$_M$ process with input channel as parameter.

Listing 21: Three different methods of translating the input bus in an SMEIL process to a CSP$_M$ process input.

In Listing 18 the value itself is the parameter, which means that the value can be used directly inside the `let` section. This translation results in a simpler translation process within TAPS, because all the information needed to translate lies within the SMEIL process itself. However, when using this structure, the channel read must happen outside of the process, which does not match the original SME model structure.

The CSP$_M$ process in Listing 19 does not have an input parameter. The input parameter from the SMEIL process has been translated directly to its formal channel name corresponding to the bus channel input seen in Listing 17. This solution fits within the original SME model structure because the process itself reads a value before computing and writing. Translating the channel name directly leads to a more complex translation, and TAPS must search the SMEIL network declaration to find the formal channel name. Another downside with this translation is that it eliminates the reusability of the process. The generated code can become unnecessarily complex because similar processes must be defined several times instead of reusing the process and declaring it with different parameters in the network.

In Listing 20, the channel name is given as input parameter to the CSP$_M$ process. This translation is the best match to the original SMEIL process structure. The channel name is provided as a parameter, so the process reads the value itself, which means that the process can be reused, as opposed to the solution in Listing 19. This solution also has the advantage that TAPS does not have to search for information in the SMEIL network to translate the process. The downside to this solution is that the internal structure of the process must be changed during the translation, as opposed to the solution in Listing 18. In the original SMEIL program in Listing 17, the `input` value is declared in the process parameter and used directly in the process body. In this solution, TAPS would have to change one of these and put in an intermediate step where the process reads a value and then uses the value in the process body. The process in Listing 20 is implemented with this intermediate step.

Out of these three solutions, it is clear that even though the solution in Listing 19 is the simplest version, the translation would become quite complex, which is a big disadvantage. In order to create the simplest translation possible, I decided to use the solution in Listing 18. Even though it does not exactly match the SME model, this will not be a problem, since the read is performed in close relation to the process. This will be explained in Section 6.5. Using this solution results in a simpler translation for each process, and thus most of the translations can be performed easily without needing detailed information about the rest of the system.

If the SMEIL process contains an output bus parameter, the translation does not become as complex as translating an input bus parameter. The process must always write the value itself and therefore the method for translating the output bus parameter into CSP$_M$ is similar to the translation in Listing 20, where the process receives the bus name as parameter. However, when using this method with output bus parameters, the process structure can be translated directly without an intermediate step, thus providing a less complex translation.

In the case where the process does not have any parameters but is communicating via the bus channels formal names, TAPS will be able to translate the processes directly because the formal name of the bus channel is defined within the process itself.

6.4 Verification in CSP$_M$

In order to create the assertions for the refinement checking, TAPS will create separate assert functions to keep the code structure clean. It is entirely possible to include the assertions within the processes, but to keep the separation of concerns, I have decided to add this separately. This will increase the complexity of the generated CSP$_M$ network, but it is still preferable to more

complex process structures. For each CSP$_M$ channel there must be an assertion, except for data generator channels. Consequently, TAPS creates a *monitor* process for each channel. This monitor process will listen in on the channel communication, and assert the values communicated there. The monitor process is a process that is added specifically for asserting legal communication values in FDR4, and it does not affect the original SME network. The monitor process checks the values communicated on the channel, and if all values are within the expected range, the process behaves as the `SKIP` process to indicate successful termination. If all values are not within the expected range, the monitor process behaves as the `STOP` process to indicate failure.

For security and simplicity reasons, all channels, except data generator channels, are always verified. If only a subset of channels were verified, the verification might not catch all potential failures. In Figure 6.3, the outline of the monitor process structure can be seen. I expect that this structure can be used for several different types of problems and thereby ensure a cleaner code structure.

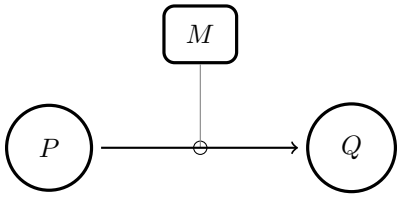

Figure 6.3: The monitor process M listens in on the communication between P and Q in order to assert the communicated values.

For FDR4 to actually perform any refinement checking, an assertion must be added to the CSP$_M$ program by using the `assert` keyword. To assert that the entire network terminates successfully the `SKIP` process is used as the specification process and the network as the implementation process to verify. An example of this can be seen below.

```
1 assert SKIP [F= Network \ Events
```

By hiding all events, they become internal events which still complies with the rules and structures of the network, but the refinement check does not include them. This also means that the refinement check is between `SKIP` and the resulting process of the network. If the network terminates successfully, its resulting process is the `SKIP` process. Therefore the refinement check passes because the `SKIP` process failure-refines the `SKIP` process.

The assertion uses the *failures* model described in Chapter 4, since it is not possible to use the *traces* model in this case. The *traces* model cannot be used because `SKIP` is defined as $\checkmark \rightarrow STOP$. Failures within the network are defined using `STOP`, so a failure in the network would also be accepted when using the *traces* model because the traces of `STOP` and $\checkmark \rightarrow STOP$ are the same. By using the *failures* model, the refinement check fails if a monitor process does not terminate successfully. The failures of a `SKIP` process includes $\checkmark$, which is not included in the failures of the `STOP` process. Therefore, even though their traces are equal, the failures are not, thus providing the result needed for the verifications.

6.5 Translating Network

When translating a network in SMEIL to CSP$_M$, it is crucial that the composition of the network is kept intact. As much as the network is a crucial part of the SMEIL program, it is also one of the most tricky structures to translate correctly. A generated network can quickly become very

complex, making it harder to ensure correct translation.

An SMEIL network is separated into instances that each instantiates a single process and parameters. As mentioned in Chapter 5, the closest equivalence to the SMEIL network in CSP$_M$, are the parallel operators. To create a similar network in CSP$_M$, TAPS will synchronise the two processes with the channels they are communicating on. This is the only way in CSP$_M$ to enforce synchronised communication on channels, which is an essential part of the synchronised network in SMEIL. An example of a simple network in CSP$_M$ can be seen in Listing 22, where the process `P` is synchronised with the process `Q` over the channel `c`. This example uses the generalised parallel operator because both processes communicate only on the channel `c`.

```
channel c : {0..10}

P = c ! 42 -> SKIP
Q = c ? x -> SKIP

Network = P [|{| c |}|] Q
```

Listing 22: Example of synchronisation using the generalised parallel operator.

Two processes that must synchronise on the same channels but also must communicate on other channels, can be synchronised using the alphabetised parallel operator. An example can be seen in Listing 23, where process `P` communicates both on channel `c` and `d`. Process `Q` does not communicate on channel `d`, and therefore if the generalised parallel operator was used in this case, then process `Q` would also have to agree on communication on channel `d`. In this example, the alphabetised parallel operator expresses that process `P` is allowed to perform events from the set on the left-hand side of the operator, which contains both channel `c` and `d`, and process `B` is only allowed to perform events on channel `c`, defined on the right-hand side.

```
channel c : {0..10}
channel d : {0..100}

P = c ! 4 -> d ! 50 SKIP
Q = c ? x -> SKIP

Network = P [{| c, d |} || {| c |}] Q
```

Listing 23: Example of synchronisation using the alphabetised parallel operator.

The challenge when translating the network from SMEIL to CSP$_M$, is to ensure that all processes are connected with the other processes on the correct channels. Since it is only possible to synchronise two processes at a time in CSP$_M$, it is a challenge to generate an entire network. The synchronisation of two processes becomes a new process which can then be synchronised with another process. This results in a lot of nested synchronisations, one for each process in the network, which quickly becomes very complex. However, since the CSP$_M$ code is generated automatically, this is not a problem.

Not only is it important that the network is generated to be equivalent to the original SMEIL network, but the monitor processes generated for each of the CSP$_M$ processes must also be included in the network. Since the monitor processes are only listening in on one specific channel,

TAPS starts out by synchronising the monitor with the process by using the generalised parallel operator on the channel. The monitor process is synchronised with the writing process, since a potential erroneous value will emerge at the write end of the channel.

If there are several output channels from a process, the synchronisation of the monitor processes will be nested together. A structure like this can be seen in the example below.

```
Process_P_monitor_network = (P [|{| c1 |}|] monitor1) [|{| c2 |}|] monitor2
```

As mentioned above, if the SMEIL process contains an input parameter, the value must be read and added to the process as an input parameter. This is also included in this process monitor network before the processes are synchronised. This can be seen in the example below.

```
Process_P_monitor_network =
    channel ? x -> (P(x) [|{| c1 |}|] monitor1) [|{| c2 |}|] monitor2
```

This design ensures that the monitor processes are included correctly in the network, and also that each read for the processes is defined without too much complexity. This process monitor network can be considered a shell around process `P`, since it only contains process relevant information. After this small network has been generated, it can be synchronised with other similar networks that share communication.

As mentioned in Chapter 5, it is possible to instantiate one SMEIL process several times within the SMEIL network. Since TAPS is creating the network in CSP$_M$ by synchronising processes on channels, it can synchronise the same process several times with different channels or parameters, creating the same functionality as the instances declaration in SMEIL. An example of this can be seen below.

```
Process_P_monitor_network =
    (P(1) [|{| c |}|] P(2)) [|{| c |}|] P(3)
```

Translating the SMEIL network into an equivalent CSP$_M$ network is difficult and will quickly become very complex. Adding extra monitor processes increase the complexity of the network, and it is crucial that it does not affect the functionality of the network, but by using the structure introduced above, TAPS can include the monitor processes without affecting the original SMEIL network.

6.6 Translating the SME Clock Structure

CSP was not initially developed for hardware modelling, and therefore it is not evident how to handle the clock cycle in the translation. However, clock cycles is an essential part of hardware modelling and therefore it is crucial to design a solution that represents the simulated clock cycles from the SMEIL program. The observed values of the SMEIL simulation represent all values communicated in all the simulated clock cycles. When using these observed values in the generated CSP$_M$ code, FDR4 will create a state space of all possible combinations of these values. Therefore, even though the translation is between an SME model, where the clock is crucial, and a CSP$_M$ model, which do not have a clock structure, the translation will still represent the input for all simulated clock cycles.

Chapter 7

Implementation

In this chapter I introduce some of the main challenges in translating from SMEIL to CSP$_M$, and how they have been solved in TAPS.

The requirements for TAPS are considerable because of the flexibility of SMEIL. Translating the different functionalities of SMEIL have been prioritised according to the requirements of the examples used within this thesis, and due to time restrictions, the current version of TAPS does not support the entire SMEIL grammar. TAPS expects a well-formed legal SMEIL program as input and does not perform any type-checking. The best way to ensure an accurate translation to CSP$_M$ is to simulate the SMEIL program before using TAPS, in order to ensure that the program complies with all the rules of the LIBSME compiler.

7.1 Transpiling SMEIL Statements

SMEIL statements are defining the functionality of the process and can consist of simple arithmetic operations, conditionals, or communication. The statements are translated into the `let within` structure as introduced in Chapter 6.

All assignments in SMEIL are translated directly into CSP$_M$ without much change, however if the assignment is consisting of communication either in or out, TAPS must handle these assignments differently, which will be explained later in this section.

When translating a constant from SMEIL to CSP$_M$, the constant is added to the CSP$_M$ program separately from the process it was defined within. The SMEIL programs must be well-formed, and thus the constant will only be used by the process it was defined within. It is therefore not a problem to have the constant defined outside of the CSP$_M$ process. If a constant is defined as a process parameter in the SMEIL program, will be translated as an initial value for the process and not as a separate CSP$_M$ constant.

Variables in SMEIL can be instantiated with an initial value, and therefore TAPS must search through all the variable declarations in the SMEIL process. If a variable is defined with an initial value, this information is gathered so that the initial value of the variable can be added to the generated CSP$_M$ process. Variables with no initial value will be ignored by TAPS because they do not need instantiation in CSP$_M$, and will be translated along with the other statements. With the current version of TAPS, only values communicated on channels are verified, and therefore the range and type of a variable have no purpose within the generated CSP$_M$ code. It would, however, be simple to gather this information if needed.

The `trace` statement in SMEIL is not affecting the behaviour of the process. It prints out the string and arguments that are given, like a printf in C. It is possible to add a `print` statement in CSP$_M$, but it has a very different functionality than the `trace` statement in SMEIL. A `print` statement in a CSP$_M$ program will appear in the right-hand side of the FDR4 session window, where it can be activated for easy evaluation. It does not support printing out text, but rather calculations and thus the `trace` statement in SMEIL is not compatible with the `print` statement in CSP$_M$.

The `assert` statement is used internally in SMEIL and evaluated during program execution. If the `assert` statement is not valid, then the execution is halted and the error message, defined in the `assert` statement, is printed. An example can be seen below.

```
1 assert(hour < 23, "hours must be less than 23");
```

Even though the purpose of this project is to assert properties in FDR4, the properties that FDR4 can verify are not similar to the properties that the `assert` statement in SMEIL can assert. As was introduced in Section 4.2.2, FDR4 is a refinement checker and the `assert` statements in FDR4 are asserting the refinement checks. In the CSP$_M$ language, assertions like `assert 4 + 4 == 8` are possible, however FDR4 does not support this. The two types of `assert` statements are therefore not equivalent.

Neither the SMEIL `trace` statement nor the `assert` statement provides any functionality to the translation or the network model, so there are two options for how TAPS should handle these. Either TAPS throws them away, or it keeps them as comments for reference in the generated code. Currently, TAPS throw them away, but it would be possible to change this and add them as comments in the generated CSP$_M$ code.

Expressions in SMEIL and their precedence rules are similar to C-like languages, and they are used for defining operations and naming in SMEIL. All of the standard binary operators behave the same in SMEIL as in CSP$_M$, as well as relational operators and logical conjunction and disjunction. The unary `!`(not) operator does not have the same precedence in the SMEIL grammar as in CSP$_M$. Also, the equality operators and comparison operators have the same precedence in CSP$_M$, but not in the SMEIL grammar. To avoid erroneous computation, TAPS adds parentheses around all nested expressions. The SMEIL grammar also consists of bitwise operations which FDR4 does not support, because CSP$_M$ only supports integers. It is likely possible to model the bitwise operations using the already supported operators, but this has not been a priority to implement and the current version of TAPS does not support it.

As mentioned in Section 4.1.1, SMEIL supports floating point numbers in the simulator, but this has not been extensively tested. Floating point numbers are not supported by CSP$_M$ because it is not possible to assert if two floating point numbers are exactly equal and therefore it is not possible to create refinement checks with floating point numbers. It is possible to create models in CSP$_M$ that define a floating point number, but the type itself is not supported in CSP$_M$ or FDR4. TAPS does not provide any type checking and therefore it does not provide a warning if floating point numbers occur within an SMEIL program.

7.2 Transpiling Channels

The read and write syntax in SMEIL is used as a variable in the SMEIL process. In Listing 24, the syntax `inbus.val` indicates a read to the channel `val` within the bus `inbus`. As can be seen, the read and write structures are used as if they are simple variables representing a value.

```
1 proc double (in inbus)
2     bus outbus {
3         val: uint;
4     };
5 {
6     outbus.val = inbus.val * 2;
7 }
```

Listing 24: Example of a read and a write in SMEIL.

SMEIL does not separate the communication and calculation in a process statement as can also be seen in Listing 24. This is of great advantage in SMEIL, but the translation becomes more complex. TAPS must be able to recognise the communication in an assignment which is possible because both the bus and the channel must be specified in a read or write in SMEIL. If one of the elements in an assignment, right or left-hand side, contains a dot, TAPS can assume that it is communication. The original grammar of an assignment like the one in Listing 24 can be seen in Listing 25. A communication is the *hierarchical accessor* alternative in the *name* rule. The communication can either be the right-hand side of the assignment, which is a write, or part of the left-hand side expression, which is a read. A write is simple to recognise since it is not combined with other parts of the grammar. TAPS can search the *name* of a right-hand side assignment and see if it contains a dot. A read can be used as an internal variable in the expressions and therefore TAPS must search all names within the nested expression to find a potential read.

⟨*statement*⟩ ::= ⟨*name*⟩ '=' ⟨*expression*⟩ ';' (assignment)

⟨*name*⟩ ::= ⟨*ident*⟩
| ⟨*name*⟩ '.' ⟨*name*⟩ (hierarchical accessor)
| ⟨*name*⟩ '[' ⟨*array-index*⟩ ']' (array element access)

⟨*expression*⟩ ::= ⟨*name*⟩
| ⟨*literal*⟩
| ⟨*expression*⟩ ⟨*bin-op*⟩ ⟨*expression*⟩
| ⟨*un-op*⟩ ⟨*expression*⟩
| ⟨*name*⟩ '(' { ⟨*expression*⟩ } ')' (function call)
| '(' ⟨*expression*⟩ ')'

Listing 25: The original assignment, name and expression grammars defined in [16].

7.2.1 Naming Channels

When translating an SMEIL bus channel to a CSP$_M$ channel, TAPS must define CSP$_M$ channel names that are unique, the same way that the formal name of a bus channel in SMEIL is unique.

The simplest way of translating the SMEIL channels into CSP$_M$ channels is to use the already defined formal name of the SMEIL bus channel. By concatenating the channel name, bus name, and process name into a single channel name in CSP$_M$, the naming becomes unique. This means that all reads and writes using the syntax `outbus.val` from the process `double` in SMEIL, will be translated to `double_outbus_val` in CSP$_M$. It is also possible to use randomly generated strings instead of using the original SMEIL names, but to ensure the readability of the generated code, it is necessary to have recognisable names for the CSP$_M$ channels.

```
network net() {
    instance c of clock();
    instance a of A(c.outbus, val: 1);
    instance _ of A(c.outbus, val: 2);
    instance s of src(a.outbus);
}
```

Listing 26: Example of a network with four instances, where two are instances of the same process.

The example in Listing 26 shows a network consisting of four instances, where two of them are instantiating the same process with different constant parameters. All but one instance declaration also includes an instance name which can be used when referencing the process in the instance parameters. SMEIL enforces the rule that two instances of the same process cannot both have the anonymous instance name, defined as _.

With two instances of the same process in SMEIL, the CSP$_M$ channels of those processes, would have identical names. It is, of course, crucial to avoid duplicate channel names, and therefore TAPS looks through the instances in the SMEIL network, and appends the instance name to the CSP$_M$ channel name. If the instance name is anonymous, a _ is appended to the CSP$_M$ channel for simplicity. The restrictions against identical process instances in SMEIL ensures that there will never be two identical named CSP$_M$ channels in the generated code. It is also possible to have bus declarations within the network declaration in SMEIL. These bus channels are named in the same manner as other channels in CSP$_M$, however without an instance name appended. The naming of each channel quickly becomes long and chaotic, which is one of the compromises of generated code.

7.2.2 CSP$_M$ Channel Range

When defining channels in CSP$_M$ it is important to define a restricted range of acceptable values for each channel. If the channel is defined with an unbounded range of integers, FDR4 would search through all possible integers which would result in a state space explosion and FDR4 would run out of space. These restricted ranges must, of course, represent relevant values.

As explained in Chapter 5, all simulated SMEIL channels and variables are defined with a range of observed values and a restricted type. Because FDR4 only supports integers, only signed and unsigned integer types are supported in TAPS. The integer following the type defines the bit size of the observed values. This bit size is used to generate a restricted range for the CSP$_M$ channel, and thereby avoid having a seemingly endless runtime in FDR4.

In Listing 14 in Chapter 5, the simulated `seconds` process from the seven segment display example can be seen. The types of the two channels are `u3` and `u4` and the observed values are between 0 and 5 for the first digit and 0 and 9 for the second. TAPS converts the bit size of the type `u3` to the range {`0..7`} which is then used as the restricted range for the CSP$_M$ channel representing the first digit. The second digit channel type `u4` is converted to the range {`0..15`}. In Listing 27, the converted ranges are defining the values of the CSP$_M$ channels.

It might seem redundant to create CSP$_M$ channels with a limited range, when TAPS is also asserting the widths of the channels. FDR4 will always only perform refinement checks using the values defined for each channel in CSP$_M$, and therefore there seems to be no point in asserting if the values go beyond this range. The method for performing usable refinement checks is

```
channel seconds_out_first_digit : {0..7}
channel seconds_out_second_digit : {0..15}

Seconds(seconds_in) =
let
    seconds = seconds_in % 60
    seconds_first_temp = seconds / 10
    seconds_second_temp = seconds % 10
within
    seconds_out_first_digit ! seconds_first_temp ->
    seconds_out_second_digit ! seconds_second_temp ->
    SKIP
```

Listing 27: Example of the `Seconds` process from the generated CSP$_M$ code in the seven segment display example, translated from the example in Listing 14 in Chapter 5. See the full code in Listing 46 in Appendix B.

to use both the type and the range of observed values, defined for each simulated SMEIL channel. The range of observed values are used within the monitor process to ensure that no values outside the range are communicated on the channel, and because the range of the bit size can only be equal or larger than the range of the observed values, the assertions become valuable.

As explained previously, an SMEIL data generator process will be translated into a CSP$_M$ channel. In this case, it is important to use the bit size range for the CSP$_M$ channel, because otherwise it is not possible to guarantee the precise input values in the network. If TAPS were to use the range of observed values, the monitor process assertions will always pass because the input values are the exact values used for the simulation. These input values are the base of all the observed values for all SMEIL channels, so the assertions would never fail. It is, therefore, necessary to have a range of input that goes beyond the range of observed values, in order to verify that the model can handle a larger range of input without failures.

7.3 Generating Monitor Processes

It is clear that all channels except data generator channels must be verified with FDR4 and thus TAPS must generate a monitor process for all channels except the data generator channels.

As explained above, the observed values for each SMEIL channel are used within the monitor processes to verify the values communicated on the channels. For TAPS to be able to generate the monitor process, it is essential that all the channels are defined with observed values. In the original SMEIL grammar, defined in [16], it is optional to include a range in a bus channel definition as can be seen in Listing 28. In order to restrict the SMEIL network to include ranges for each SMEIL bus channel before translating with TAPS, I have changed the original grammar of SMEIL. The grammar shown in Listing 29 is slightly different compared to the grammar defined in Listing 28. The difference is that a range is no longer optional and that the keyword `exposed` is no longer allowed. The keyword `exposed` can be defined for a bus in SMEIL to indicate buses that communicate with another SME implementation using co-simulation, but since TAPS does not support co-simulation, this keyword is not needed. A consequence of this change in the grammar is that the output bus channels within the data generator process must be defined with a range, even though the range is not needed for the translation. Because the data generator processes are not defined differently in SMEIL, the same grammar rules apply to them.

⟨*bus-decl*⟩ ::= ['exposed'] 'bus' ⟨*ident*⟩ '{' ⟨*bus-signal-decls*⟩ '}' ';'

⟨*bus-signal-decls*⟩ ::= ⟨*bus-signal-decl*⟩ { ⟨*bus-signal-decl*⟩ }

⟨*bus-signal-decl*⟩ ::= ⟨*ident*⟩ ':' ⟨*type*⟩ ['=' ⟨*expression*⟩] [⟨*range*⟩] ';'

Listing 28: The bus grammar defined in [16]. The square brackets indicate an optional nonterminal and curly brackets indicate zero or more nonterminals.

⟨*bus-decl*⟩ ::= 'bus' ⟨*ident*⟩ '{' ⟨*bus-signal-decls*⟩ '}' ';'

⟨*bus-signal-decls*⟩ ::= ⟨*bus-signal-decl*⟩ { ⟨*bus-signal-decl*⟩ }

⟨*bus-signal-decl*⟩ ::= ⟨*ident*⟩ ':' ⟨*type*⟩ ['=' ⟨*expression*⟩] ⟨*range*⟩ ';'

Listing 29: The bus grammar defined in [16] changed to match the demands of the translation.

The monitor process asserts that the observed values from the SMEIL simulation are the only values communicated on the channels. The two monitors of the `Seconds` process from the seven segment display example can be seen in Listing 30. The values used for these monitors are 0 through 5 for the first digit monitor and 0 through 9 for the second digit monitor. These values have been defined for each channel in the simulated SMEIL program in Listing 14 in Chapter 5. Each monitor asserts values on one channel, and thus the name of the monitor

```
1 Seconds_out_first_digit_monitor(c) =
2     c ? x ->
3     (0 <= x and x <= 5) & SKIP
4 Seconds_out_second_digit_monitor(c) =
5     c ? x ->
6     (0 <= x and x <= 9) & SKIP
```

Listing 30: Example of the two generated `Seconds` monitor processes from the seven segment display example. See the full code in Listing 46 in Appendix B.

process is the channel name with `_monitor` appended. This means that TAPS can easily find the name of the monitor process from the channel name, and vice versa. Since no CSP$_M$ channels can have identical names, this ensures that no monitor processes will have colliding names.

Using this monitor process structure in the seven segment display example also expose a limitation to the structure. In the `Hour` process, the observed values are $\{0..2\}$ for the first digit, and $\{0..9\}$ for the second digit. This combination of values can represent 24 hours, as expected, but it will also accept values from 25 to 29 because the digits are evaluated separately. In order to solve this, it would be necessary to assert values across more than one channel in a monitor process. This has been mentioned in Chapter 11 as future work, since it is not yet implemented.

7.4 Generating a CSP$_M$ Network

The process monitor network consists of three parts. The first part consists of reading input values for the process. As explained previously, all reads will be performed outside the process itself. From the instance parameters, defined in SMEIL, TAPS will look up the name of the SMEIL process where the input is defined. The read within the SMEIL process defines the

specific channel to read from. This information is gathered to clarify which CSP$_M$ channel the process monitor network should read from. Each read is added using prefixing, so it would also work if there is more than one input bus for the process. An example of the first part of the network can be seen in Listing 31. The second part is to add the process name with the expected parameters, both input values, output channels as well as constants and internal values. The example in Listing 32 is a continuation of the example in Listing 31.

```
1 N_seconds = clock_out_val ? variable -> ...
```

Listing 31: First part of the network generation where an input value is read from the channel `clock_out_val`.

```
1 N_seconds = clock_out_val ? variable -> Seconds(variable) ...
```

Listing 32: The second part added to the network generation in CSP$_M$.

In the last part, TAPS will synchronise the process with the monitor processes over the channels they are asserting. For each monitor process, TAPS builds a new layer of synchronisation around the network. This is straightforward to automatically generate, since TAPS can loop over the list of writes for the process.

Listing 33 shows the network generated from the translated `Seconds` process in Listing 27 and the corresponding monitor processes in Listing 30. Adding parentheses around each synchronisation, the `first digit` monitor process is synchronised on the `first digit` channel and the `second digit` monitor process is synchronised on the `second digit` channel which completes the process monitor network.

Lastly the CSP$_M$ `assert` operator is specifying the refinement check for FDR4. The assertion defines a refinement check from the specification process `SKIP` and the implementation process `N_seconds`. The assert statement also indicates that the refinement check should be performed using the failures model.

```
1 N_seconds = clock_out_val ? variable ->
2             (Seconds(variable)
3             [| {| seconds_out_first_digit|} |]
4             Seconds_out_first_digit_monitor(seconds_out_first_digit))
5             [| {| seconds_out_second_digit|} |]
6             Seconds_out_second_digit_monitor(seconds_out_second_digit)
7
8 assert SKIP [F= N_seconds \ Events
```

Listing 33: Example of the `Seconds` network processes from the generated CSP$_M$ code in the seven segment display example. See the full code in Listing 46 in Appendix B.

In SMEIL it is possible to define more than one network within the same SMEIL program. This is not a problem for TAPS to translate as long as the program is still a well-formed SMEIL program. Each instance defines a process and its parameters. The communication will always refer to a specific bus, defined in either a process or a network. This will not change no matter how many networks are defined within the program, because communication cannot span

across networks. In theory, it is possible to create nested networks where a network can instantiate other networks, and thereby allowing communication across networks, but this has not yet been implemented in SMEIL and therefore TAPS does not support this. The only situation where several networks can become a problem for TAPS is if the networks have instance names in common. As mentioned above, it is necessary to include the instance name in the name of a generated CSP_M channel in order to tell the difference between two CSP_M channels originating from the same process. If there is more than one network using the same instance name for a specific process, the generated code would have identical channel names in one CSP_M program. To solve this, TAPS could include the network name in the generated CSP_M channel name to add another layer of separation. This is a corner case, and the current version of TAPS does not support this solution, but since it is a potential problem it is important to identify.

7.5 The Technologies of TAPS

The two main parts of TAPS are the parser and the code generator. The parser is created with ANTLR4 [45], which is described further below. The data transformation and code generation have been developed in Python 2.7. The templating language Jinja2 [3] has been used to define and generate the standard CSP_M structures, that the SMEIL program will be translated into.

The SMEIL program is parsed using ANTLR4 and the resulting parse tree is traversed, while TAPS collects all relevant data from the SMEIL program. The data is then transformed to match the requirements of the CSP_M structures used for the code generation. Together with the transformed data, TAPS uses the templates to generate the CSP_M code. The generated code can then can be verified in FDR4.

Often when developing any form of translation either by a compiler or interpreter, the use of a symbol table is often necessary in order to keep information about variable names, types, values etc. During the translation when reaching a symbol, the system will use the symbol table to look up the symbol and to retrieve its context. The symbol table is usually generated in the analysis section and is used for lookups throughout the compilation or interpretation. TAPS expects the SMEIL program to be well-formed and does not provide any checking of the SMEIL code, and therefore it is not necessary for TAPS to include a symbol table to check if variables have been properly declared in the process declaration, before being used in the body of the process.

7.5.1 ANTLR4

For the transpiling from SMEIL to CSP_M, I have decided to use a parser generator tool to generate a parser for the SMEIL language.

I have chosen a parser generator library because it will simplify the parsing process so I can focus more on the development of correct code generation. Parser generators are tools that provide an automatically generated parser from a specified grammar. Using a parser generator is an obvious choice since the SMEIL grammar is already in a well defined Extended Backus-Naur Form (EBNF) format. It is also possible to create a parser from scratch but there seems to be little reason to spend the time writing and debugging a bespoke parser when I can generate it directly.

I have decided to use the ANTLR4 tool after looking at several other parser generator tools. ANTLR4 is a Java-based parser generator library that, based on a grammar, can generate parsers in Java or another target language, like Python 2.7. ANTLR is a well-used tool which has recently been updated to the new and improved ANTLR4 version. Companies like Twitter and Oracle are using ANTLR [45] for different parse tasks. ANTLR4 also has an active user base

and there is plenty of documentation and guides for the new ANTLR4 version online. A lot of parser generator tools exists today and some of the most known parser generators are Yacc or Bison but unfortunately, a lot of these tools seemed to be a bit outdated and not very user-friendly in terms of error messages or simplicity in the grammar structure. ANTLR4 seems to be the right choice for this simple grammar.

By using ANTLR4 it is simple to transform the SMEIL grammar into a parser and lexer using the already defined SMEIL grammar. ANTLR4 supports EBNF grammars which must be defined in `.g4` file format, which is quite similar to other standard grammars. ANTLR4 also provides the possibility of generating a tree walker, which can be used to traverse the parse tree. There are three options for tree traversal, either using the listener or visitor method provided by ANTLR4 or manually traversing the parse tree. The listener methods are called directly by the walker object, that triggers parser events for the listener to respond to. The listener is analogous to XML document handler objects that respond to SAX events triggered by XML parsers. The visitor method in ANTLR4 is similar to the standard visitor patterns seen in other parser structures. In ANTLR4 each visitor must call `visit()` on its children to walk the subtrees. ANTLR4 also provides the possibility of adding labels to alternatives in grammar rules, which creates a visitor method for each labeled alternative. This ensures a more manageable parsing and less complex visitor functions.

Both the visitor method and the listener method can provide the same results. In the first version of TAPS I used the listener method, but after realising that it is not necessary to traverse the entire parse tree, TAPS has been rewritten to use the visitor method that only visits the necessary subtrees.

After TAPS has walked the parse tree and collected all relevant information from the SMEIL program, the data is transformed to match the requirements of the CSP_M code. For example, the name of a bus and its channels in SMEIL is transformed into combined names for the CSP_M channels. Because TAPS only collects data during the tree walk and does not perform direct translation, it does not matter how the SMEIL code is structured. With this solution, the SMEIL network declaration can be defined first or last in the program without any difference to the translation.

Chapter 8

Extending TAPS for Clocked Systems

After developing the initial version of TAPS, I realised that the solution was not broad enough in terms of what type of network it could accurately translate. The type of network that is possible to translate with the initial version of TAPS, is a network that does not keep internal states between clock cycles. In the seven segment display example, no internal results have any influence on the result from other clock cycles, and therefore it can be correctly translated using the structures described in the previous chapters. However, the original TAPS system was unable to generate semantically equivalent CSP$_M$ code from networks where the results depend on the internal state of the network. Examples of such networks include the Aaddone example from [17].

The `addone` network is a simple cyclic network that consists of two processes communicating with each other. The SMEIL code for this example can be seen in Listing 34. The `add` process receives a value and increments it by a constant parameter. The `id` process receives the value and passes it along on its output bus. An illustration of the network can be seen in Figure 8.1.

As presented in Chapter 6, all possible communication combinations are represented in the generated CSP$_M$ code, however, the generated CSP$_M$ processes are not recursing, so only one cycle of the loop is simulated in the `addone` example. This means that if the network was provided with an input that, after several clock cycles, resulted in a failure, this would not be caught by FDR4 because the CSP$_M$ code only represents one clock cycle. It was, therefore, necessary to extend TAPS so that network with internal states can be verified.

The purpose of the initial design of TAPS was to model a synchronous network in CSP$_M$ without having to model a global synchronous clock. As described in Chapter 4, the results from the master's thesis *Generation of FPGA Hardware Specifications from PyCSP Networks* [61] by

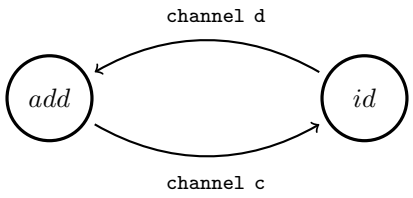

Figure 8.1: The `addone` network consists of two proceses that communicate with each other on channel `c` and `d`.

```
proc add (in inbus, const constant)
    bus outbus {
        val: u4 = 0 range 0 to 10;
    };
{
    outbus.val = inbus.val + constant;
}

proc id (in inbus)
    bus outbus {
        val: u4 = 0 range 0 to 10;
    };
    var from_add: u4 range 0 to 10;
{
    from_add = inbus.val;
    outbus.val = from_add;
}


network addone_network ()
{
    instance id of id(add.outbus);
    instance add of add(id.outbus, constant: 1);
}
```

Listing 34: The simulated SMEIL network `addone_network` with two processes. The example is similar to the addone example in [17].

E. Skaarup and A. Frisch, established how much the complexity of the network would increase when trying to model this in CSP. However, the `addone` example shows that it is necessary to extend TAPS to model a global synchronous structure in CSP_M. As Skaarup and Frisch already learned, enforcing a global synchronous model onto CSP is not trivial, and even simple networks quickly become very complex. The reason for implementing this structure in spite of the results from [61], is that TAPS will automatically generate the CSP_M code and therefore the complexity and size of the corresponding CSP_M network are less of an issue. However, the extra complexity might become a problem when verifying with FDR4. It is possible that the added complexity of the network increases the requirements for FDR4, and that the size of problems verifiable with FDR4, becomes smaller with this solution. In this chapter, I will introduce the approach for extending TAPS to translate clocked systems.

8.1 Global Synchronisation

In order to verify networks for more than one clock cycle, it is necessary to translate the SMEIL processes to recursive CSP_M processes. As explained in the CSP background section in Chapter 4, these types of processes do not terminate with `SKIP` but will behave as itself forever, unless the network informs otherwise. An example of this can be seen in Listing 35. An actual SME process will never terminate, but it is of course not possible to simulate endless runtime, so when simulating the SMEIL network the developer indicates the number of clock cycles to simulate. The results of the simulation can be seen as a snapshot of the process trace. The SMEIL simulation consists of a finite number of clock cycles and it is not possible to verify an infinite state space with FDR4, therefore, it is necessary to introduce process termination to the translation, even though this is not the reality of actual hardware.

In Listing 35, process `Inc` performs an endless loop with no option to terminate. As previously mentioned, it is important to limit the range of values for FDR4 to verify, in order to avoid running out of space. If the example in Listing 35 was verified with FDR4, it would eventually run out of space. It is, therefore, crucial to model a structure that can drive the network synchronisation, and that can ensure that the processes terminate at a specified time. This structure is represented by a `Clock` process added to the generated network in CSP$_M$. The `Clock` process drives the network for a specific number of clock cycles and then terminates, causing all other processes to do the same.

```
1 Init = d ! 1 -> Inc(1)
2 Inc(x) = d ! (x+1) -> Inc(x+1)
```

Listing 35: Example of the a recursive CSP$_M$ process that is initialised by the `Init` process.

It is, of course, still necessary for the new version of TAPS to model a CSP$_M$ network that reflects the SME model, and therefore it must adhere to the SME model structure. To model the global synchronicity in CSP$_M$, it is necessary to enforce a synchronising event. This synchronicity can be emulated by having a `sync` channel, where all processes synchronise and thus they can emulate rising and falling clock signals. All clocked processes in the network will synchronise on the `sync` channel, and as previously introduced, when two processes are synchronised on a channel, they must agree on communication before any process can continue. The same `sync` channel is used for syncronising read as well as write, thus all processes syncronise on the `sync` channel before they read and before they write. The `Clock` process is also synchronising on the `sync` channel, so when the specified number of clock cycles has passed, the `Clock` process terminates. This means that none of the other processes will be able to synchronise on the `sync` channel because all processes must synchronise together. They will instead behave as `SKIP`, so the system terminates successfully.

It is possible to design the `Clock` process to send values to each process, so that the process itself decides to terminate or continue, based on the value received. However, this would increase the complexity of the processes. The `Clock` process is instead designed to perform the `sync` event twice, one for read and one for write, for each clock cycle, and then recurse. In Listing 36, the CSP$_M$ code for a `Clock` process can be seen. The `Clock` process is instantiated with a start value and the desired number of clock cycles, so for each recursion, the internal value is incremented by one. By using pattern matching, the internal value of the `Clock` process is checked against the number of desired clock cycles, and if it is equal, the process terminates by `SKIP`. When the `Clock` process has terminated, all other processes will terminate as well.

```
1 Clock(10) = SKIP
2 Clock(n) =  sync -> sync -> Clock(n+1)
```

Listing 36: Example of a `Clock` process that runs for 10 clock cycles before terminating.

8.2 Clocked Processes

All clocked processes must synchronise on the same `sync` channel as the `Clock` process described above. Each process must synchronise on this channel twice in each clock cycle, and then recurse. To ensure the SME model structure is kept intact, the clocked processes are still defined with the `let within` structure, but the read must happen inside the process itself. A

read can be performed in every clock cycle and therefore the read cannot happen in the surrounding network as in the initial version of TAPS. As explained in Chapter 6, the initial version of TAPS performs reads for each process outside of the process itself, which simplifies the translation, but it is not completely consistent with the SME model. Listing 21 in Chapter 6 introduces three different methods for translating the input buses from SMEIL to CSP$_M$. The method that matches the SME model best, is the method in Listing 20, where the process parameter is a placeholder for the channel name, so the process can read the value directly from the placeholder. Because a clocked process can read an input in each clock cycle, the read must be within the process recursion. Therefore the solution used in the initial version of TAPS cannot be used in the clocked version, so the method used in the clocked version of TAPS is the method from Listing 20.

To ensure the synchronicity of the network, each process must synchronise before a read and before a write. A simplified process structure looks like this:

```
1 P = sync -> read_channel ? val -> sync -> write_channel ! val -> P
```

This is, of course, not including computation, so after the second synchronisation the process can include a `let within` structure with all computations and writes.

```
1 P = sync ->
2     read_channel ? val ->
3     sync ->
4     let
5         compute
6     within
7         write_channel ! val -> P
```

This `let within` structure is only necessary if the process is actually performing a computation. The `id` process in Listing 34 does not compute but only reads and writes, and therefore TAPS must not include the `let within` structure, because FDR4 will fail if an empty `let` is included.

Simple calculations in SMEIL can be included in-line with the read and the write. An example of this can be seen in the `add` process in Listing 34 where `constant` is added to the input value and written to the output bus on the same line. Listing 39 shows two examples of dividing simple computation in CSP$_M$. As seen in Listing 37, the simple computation of the `add` process can trivially be included directly to the read or write. However, to simplify its implementation, TAPS does not distinguish the generated code based on the complexity of expressions. Therefore, these types of SMEIL structures will be divided into a separate read and write by TAPS. This means that the example in Listing 37 will be fitted to the general solution that can be seen in Listing 38, even though it is unnecessarily complex. As previously explained, this is one of the disadvantages of automatically generated code.

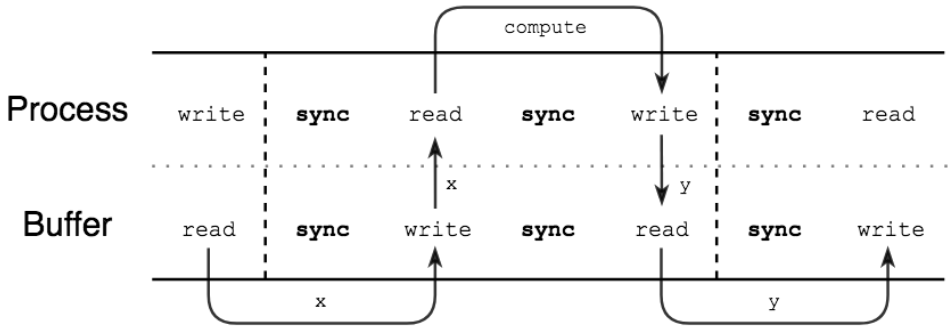

Figure 8.2: Illustration of the relationship between a process and a buffer. In one clock cycle the buffer writes a value which the process reads, and after the computation the process writes the value to the buffer which holds it for the next clock cycle.

```
1 P = read_channel ? val ->
2     write_channel ! val + 1 -> P
3
4
5
```

Listing 37: An example of an addition computation added directly to the CSP_M write.

```
1 P = read_channel ? val ->
2     let
3         result = val + 1
4     within
5         write_channel ! result -> P
```

Listing 38: An example of an addition computation and a write separated by the `let within` structure.

Listing 39: Examples of how a simple computation must adhere to the general translation structures in CSP_M.

8.3 Introducing Buffers

As can be seen in Figure 8.1, both the `id` process and the `add` process are both reading and writing. A problem occurs because they are both trying to read and write at the same time. It is clear that both processes cannot begin by reading, because no process has yet written anything for the other to read. To solve this problem, TAPS defines a buffer structure for each SMEIL channel. A buffer is a process that stores data between clock cycles. The buffer first writes a value to a channel and then reads a value from a channel, all in one clock cycle. Thus, the buffer structure is the reverse of a 'normal' process, because it will write first and then read in a clock cycle. Figure 8.2 illustrates the relationship between a process and a buffer.

The buffer solves the problem of initialising the network. Each buffer is instantiated with a value that the buffer writes to a channel from which a process can read. However, the initial value is only used to initialise the network. I refer to the initial value as a *dummy value* which is determined as a value outside of the channel range so that the process is able to distinguish between the initial value and the actual network values. It is, of course, not ideal to represent the dummy value as an integer. The dummy value must be outside the channel range, and therefore the channel range cannot represent the entire signed 32-bit representation that FDR4 support. Even though it is highly unlikely for a channel range to be this large, it is an unneces-

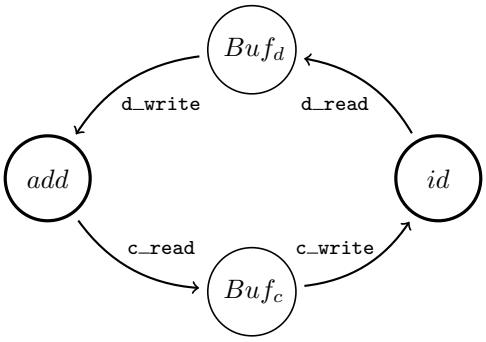

Figure 8.3: The clocked addone network has two proceses and two buffers, which ensure the global synchronicity.

sary restriction. It is possible to define a data structure in CSP$_M$ that removes this restriction, for example by having the dummy value being represented by a boolean instead of an integer. This extended structure will increase the complexity of the communication structures, but it is simple to automatically generate. In this thesis, the dummy value is represented by an integer value to simplify the examples.

The add process and the id process are both instantiated with a value, which is used instead of the dummy value from the buffer process. When the process encounters the dummy value, the process ignores it and continues with the value that the process was instantiated with instead. After the first clock cycle, the process loop will continue, and the communication will hold according to the SME model.

A clocked version of the addone network can be seen in Figure 8.3. The clocked addone network consists of four channels, two buffer read channels and two buffer write channels. A buffer process is defined for each original SMEIL channel in the network.

8.4 Buffer Structure

The buffers are designed to adhere to different kinds of process behaviour, and therefore the structure becomes complex. However, the buffer structures are standardised, so that they can be used for all clocked SMEIL channels. Therefore the complexity is not an issue when automatically generating the buffer. A buffer must still comply with the synchronisation of the network, so a buffer is divided into a read and a write structure to simplify the buffer structure, and to comply with some of the requirements that the SME model sets for the processes, which will be explained below. The buffer structure can be seen in Listing 40.

When a network is initialised, all processes will synchronise first and then the Write_buf process will write its initial value to the corresponding channel. The write can be seen in Listing 40 in the Writes_buf process. The reason for creating a separate process for the actual write event is that the SME model specifies that several processes can read the same value from the buffer in the same clock cycle. It is therefore necessary to define a recursive structure that enables several buffer writes in the same clock cycle. The Writes_buf process will write a value to the channel and then either write again or behave as the Read_buf process.

If no processes are reading from the buffer, the buffer will not write, and instead behaves as the Read_buf process. This can be seen as the external choice between the Writes_buf process and the Read_buf process, after the initial synchronisation in the Write_buf process.

The last action before a process terminates will always be a write, which means that the buffer will read the value and then terminate. The buffer must have the choice to terminate after a read, otherwise, it would wait forever and FDR4 would fail. Therefore the `Write_buf` includes an external choice with SKIP.

It is defined in the SME model, that a buffer can never be written to twice in the same clock cycle. In the current version of SMEIL, this is not implemented, as several processes can write to the same bus within the same clock cycle. However, this is not an accurate solution and a check must be added in SMEIL to ensure that this never happens. To verify that two reads cannot happen in the same clock cycle, an extra assertion have been added to the buffer structure. If more than one read can be performed by the `Read_buf` process, the process behaves as the `STOP` process which indicates failure, and thus FDR4 would catch the failure when verifying the network.

The SME model requires that there are no values to be read from the buffer if no value was written to it. And therefore the buffer does not perform a write in a clock cycle if it did not read a value in the previous clock cycle. This can be seen in the `Read_buf` process as the external choice between reading or synchronising and then recursing to the `Read_buf` process again.

In the unlikely case that there are no writes to the buffer at all, the buffer must still be able to terminate along with the rest of the processes and therefore the `Read_buf` process also includes an external choice with `SKIP`.

```
1 Write_buf(x) = sync -> (Writes_buf(x) [] Read_buf) [] SKIP
2
3 Writes_buf(x) = w ! x -> (Writes_buf(x)
4                           [] Read_buf)
5
6 Read_buf = sync -> ((r ? x -> (r ? x -> STOP [] Write_buf(x))
7                 [] sync -> Read_buf) [] SKIP)
```

Listing 40: The synchronised buffer structure.

8.5 The Out Of Bounds Problem

Attempts to verify the CSP_M translations of `addone` in FDR4 failed with a compilation error message, indicating that the system was trying to communicate a value that was not a part of the set of values defined for the channel.

The reason for this error is that the `add` process in the `addone` example has to read, compute and write in all clock cycles. Because of the standard structure of a process, the last action it can perform before terminating is to write a value. In the `add` process, this value is incremented with a defined constant.

A simplified example can be seen in Listing 41, where the channels are defined with the range $\{0..5\}$, the `Clock` process is running for 11 clock cycles, and the `add` process is incrementing by 1. The result of this network is that the `add` process would write 6 to the channel just before terminating, but since the channel is defined only for the range $\{0..5\}$, FDR4 fails and provides an error message declaring that this is not possible.

```
channel input, output : {0..5}

Add = sync ->
      input ? x ->
      sync ->
      let
        result = x + 1
      within
        output ! result -> Add

Clock(11) = SKIP
Clock(n) =  sync -> sync -> Clock(n+1)
```

Listing 41: A simplified example of the `Add` process in the `Addone` CSP$_M$ network. See the full code in Listing 48 in Appendix C.

Even if the maximum value was increased so much that the actual communicated values would never be above, FDR4 would still provide the error. For example, if the network was the same as described above, but the channels were defined for a range $\{0..500\}$, the values the `add` process would communicate after 11 clock cycles would not be near the maximum value of 500. But in this case, FDR4 still raised an error because the problem does not lie in the values communicated on the network, but the structure of the network.

The reason that FDR4 fails in both cases lies in the internal structure of FDR4 and the verification method. For all networks, FDR4 must allocate all possible states, even though not all of them are visited during the refinement check. This means that it allocates states where the process writes values larger than the maximum channel value, even though those states would never be visited.

The solution to this problem is to add a guard before the write. A CSP$_M$ guard is essentially `if b then P else STOP`. When adding this before the write, FDR4 recognises that there is a check to avoid writing a value that is potentially too high, and therefore it does not raise the error. The conditional in the guard is testing whether the value to write is less than or equal to the maximum value defined for the channel. If the channel is defined for the range $\{0..500\}$ the guard would be as below.

```
(value <= 500) & channel ! value -- if (value <= 500) then channel ! value else STOP
```

This guard adds an extra layer to the generated process structure. This will increase the complexity of the clocked processes, but it is also a structure that is easy to automatically generate. It is not necessary to know the values actually communicated on the channels in order to create the bound. The value used in the bound is the maximum channel value, which is already defined in the SMEIL input.

In order to generalise the translation structures, it is necessary to add this guard to all writes in the network, even though it might not be necessary for all writes. It is not possible to know beforehand the data communicated in the network, and therefore it is not possible to know where this type of problem could occur. In the `Addone` case, it is simple to understand the failure and why it occurs, but TAPS must be able to verify different types of problems, and therefore the translation structures must be general structures that can be used without knowledge of the data communicated within the network.

In the `Addone` network, only the upper channel range limit is affected, but the same problem could also occur with the lower limit of a channel range. For example, if the network was decreasing a value instead of increasing it, the problem would be reversed. It is therefore necessary to check both upper and lower bound of the channel range in order to create a general solution for TAPS. A bound for all writes in a network could therefore look as in the example below.

```
1 (0 <= value and value <= 500) & channel ! value
```

8.6 Generating Data for Clocked Networks

As in the initial version of TAPS, the clocked version must translate an SMEIL data generator process into a CSP_M data generator channel, instead of a separate process. In a clocked network, the number of clock cycles makes it possible to represent the various internal states in the circuit. A data generator channel in CSP_M represents the entire range defined for the original SMEIL channel. As with the initial version of TAPS, FDR4 will check all possible inputs to the network. In the clocked version of TAPS, a network is run for a specified number of clock cycles, and in each clock cycle, the entire input range is verified by FDR4.

In a clocked version of the seven segment display example, there is no reason to run the network for more than one clock cycle because all possible states have been verified after one clock cycle, since the example does not keep internal state. This, however, is not the case with networks that keep internal state between clock cycles, like the `addone` network.

In the `addone` network, each clock cycle represents a new internal state. For each clock cycle or internal state, FDR4 verifies all possible inputs, which results in the network reading unexpected values. The internal structure of the network must be able to handle corner cases of unexpected values, so by providing this type of verification, these cases are explored.

It would of course not be possible to know when all possible state combinations have been verified in a network, and it will be up to the user to define how many clock cycles to verify with FDR4.

8.7 Verifying Clocked Networks

It is, of course, also necessary to add verification structures to the clocked CSP_M network and even though the communication structure has changed, there is no reason for changing the verification structure. The clocked version of TAPS model the monitor processes as described in Chapter 6 with one small change. The values to verify do not change, but since the initial value of the buffer processes is a dummy value, this value must be defined as an acceptable value in the monitor process.

The monitor process will read the same value as the buffer process, and therefore it is possible to add the monitor verification inside the buffer processes, but there are several reasons why this is not a feasible solution. It is necessary to keep separation of concerns, and even though it would reduce the number of processes to have the monitor processes also perform the verification, the complexity would increase. The most important reason why it is not a feasible solution is that it is not a requirement that a channel has a buffer. If no process reads from a channel, the channel does not need a buffer to keep the value. All channels must have a monitor process, but not all channels must have a buffer process, and therefore the monitors and buffers are

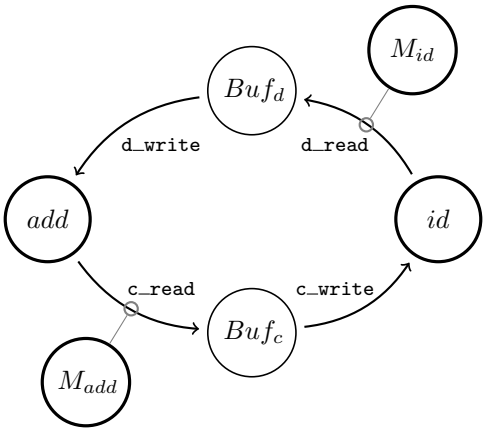

Figure 8.4: The clocked `addone` network as in Figure 8.3. The two monitor processes are connected to the `read` channels.

separated. Also, a channel is not required to have both a write and a read end, but the values communicated on the channel should still be verified. Therefore the monitor must be in the write end of the channel to ensure that the monitor will be added, no matter if there is a process to read from it.

The monitor processes are not part of the clocked network and do not synchronise together with the other processes. It would not be a problem to have them clocked, but there is no reason for it. The process reads a value when one is written to the channel, so it will verify all values whether it is clocked or not. An illustration of the clocked `addone` network with monitor processes can be seen in Figure 8.4.

8.8 Extensions for TAPS

For TAPS to be able to verify clocked networks, it must be extended with the structures described in this chapter. The buffer structure is a general structure which, does not need to be adapted much. For each SMEIL channel, two CSP$_M$ channels must be created, one for the buffer write and one for the read. The dummy value is currently chosen as an integer but should be updated so it does not restrict the channel range.

The process structure is updated to include the synchronisations on the `sync` channel and the process read is included inside the process. Each process will be augmented with the input channel, in the network structure. To ensure that no problems arise with the out of bounds problem, the process structure will also be augmented to include a CSP$_M$ bound on all writes.

The `Clock` process must be defined with the number of clock cycles to verify, which the user should provide via the command line as an argument for TAPS. It is also possible to create a structure that will allow the developer to define the number of clock cycles in a comment within the SMEIL program, but as the number of clock cycles to verify is not necessary the same as the number of clock cycles to simulate, this information might not belong within the SMEIL program itself.

The network structure will also be including the `Clock` process as well as the buffer processes. Lastly, all processes will be expanded with a `[] SKIP` around the entire body of the process, to ensure that the process can either synchronise for a new clock cycle or terminate.

Chapter 9

Experiments and Results

In this chapter, I first present examples of verification for the seven segment display example as well as the addone example. Secondly, an experiment has been conducted to gain further insight into program size and validation time. The experimental setup and results are introduced with the three properties number of visited states, verification time, and maximum resident set size.

9.1 Seven Segment Display Example Validation

The seven segment display example has been presented in different sections throughout this thesis. An illustration of the entire translated unclocked seven segment display network can be seen in Figure 9.1. The SMEIL representation of the same network can be seen in Chapter 5, in Figure 5.2. The unclocked CSP$_M$ network consists of 12 different processes, all created so that not only the network is simulated correctly, but also so the monitor processes are placed correctly. The input is represented by a triangle, since it transpiles from an SME process to a CSP$_M$ channel, it is not represented as a process in this network. Each of the dotted squares represents the network of synchronisations for each `time` process, which in itself is a process in CSP$_M$. For each network, there is the `time` process and two monitor processes, for example, H, M_{H_1} and M_{H_2}.

In order to show that the verification is accurate, the example in Listing 42 contains an error that results in FDR4 failing the verification. In Listing 42, the example is only able to handle an input that is less than 24 hours. This is because the calculation in the `Hours` process does not perform the wrap-around at the 24$^{\text{th}}$ hour. An example of such could be the input `131071`, which represents 36 hours, 24 minutes and 31 seconds. When trying to assert the code from Listing 42 in FDR4, the assertion fails. The counterexample, provided by FDR4, shows that the number 3 is communicated on `hours_out_first_digit`, which is not allowed according to the monitor process in Listing 42.

This example of failure shows how verifying the solution with a tool like FDR4 actually catches errors that the developer might have overlooked. In this case, the error is simply corrected by adding `% 24` on the end of line 9 in Listing 42 and can be seen corrected in Listing 46 in Appendix B, at line 15. When trying to verify the example in FDR4 now, it passes. Using modulo on the result, ensures that I still get the accurate time of day, no matter how many full days the input represents. The full SMEIL and CSP$_M$ code for the unclocked seven segment display example can be seen in Listing 45 and in Listing 46 in Appendix B.

```
channel clock_out_val : {0..131071}

channel hours_out_first_digit : {0..3}
channel hours_out_second_digit : {0..15}
    ⋮

Hours(hours_in) =
    let
        hours = hours_in / 3600
        hours_first_temp = hours / 10
        hours_second_temp = hours % 10
    within
        hours_out_first_digit ! hours_first_temp ->
        hours_out_first_digit ! hours_second_temp ->
        SKIP

Hours_out_first_digit_monitor(c) =
    c ? x ->
    (0 <= x and x <= 2) & SKIP
Hours_out_second_digit_monitor(c) =
    c ? x ->
    (0 <= x and x <= 9) & SKIP
```

Listing 42: Example of an erroneous version of the `Hours` process from the CSP$_M$ seven segment display example. A corrected an full version of the code can be seen in Listing 46 in Appendix B.

9.1.1 Clocked Seven Segment Display Example

The internal state does not change between clock cycles in the seven segment display example, so it can be verified with the unclocked version of TAPS. However, it is interesting to see how the network can be translated into a clocked structure. The full CSP$_M$ code of the clocked seven segment display example can be seen in Listing 47 in Appendix B. The example is very similar to the unclocked version, however, the processes all synchronise on the `sync` process as described in Chapter 8. Each `time` process synchronises, then reads a value from the input range, then synchronise again before performing the computation. Then, the value is written to the output channel, that the monitor process reads from, just like in the unclocked seven segment display example. This clocked example is a bit different than the `addone` example introduced in Chapter 8, because there are no buffers in the clocked seven segment display network. This is because there is only one synchronised process communicating, and therefore there is no need for buffers. This means that the network is not as complex as it would have been if buffers were added. Because the seven segment display network does not persist a state between each clock cycle, there is no need to verify more than one clock cycle. When the `Clock` process terminates after one clock cycle, the network will behave the same as the unclocked seven segment display network. By looking through the FDR4 traces and the ProBE visualisations, it is clear that the two networks are equivalent in terms of verification and failures.

9.2 Addone Example Validation

The `addone` example was introduced in Chapter 8, and an illustration of the clocked network with its monitor processes can be seen in Figure 8.4 in the same chapter. As explained, the `addone` example does not translate well in the initial version of TAPS and therefore the clocked

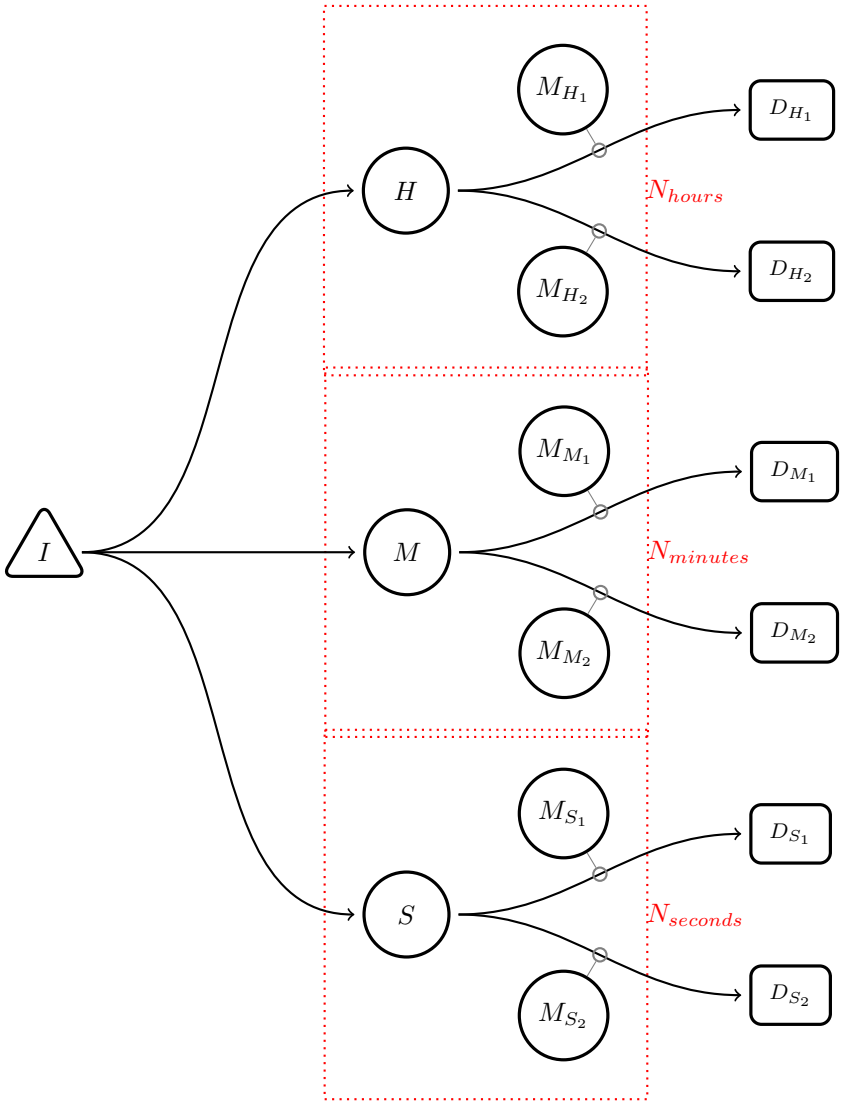

Figure 9.1: A seven segment display clock network in CSP$_M$. I represents the input channel. N_{hours}, $N_{minutes}$ and $N_{seconds}$ represent the network processes with H, M and S as the `time` processes. The results from the `time` processes are communicated to the displays D. The displays are represented by a square, since they are not actual CSP$_M$ processes. Each display communication also has a monitor process M, which assert the legal communication values.

version was created. The difference between verifying the `addone` network with a clocked and unclocked network, is that FDR4 is only able to verify one internal state in the unclocked version, whereas it is able to verify multiple internal states in the clocked version. The possibility of verifying different internal states suits this cyclic network perfectly.

The `addone` example differs from the seven segment display example in that it does not require an input range, because the cyclic network is instantiated with initial values instead of a data generator process. The cyclic structure of the `addone` example causes the values to circulate and increase indefinitely if not restricted. It is not possible to represent an indefinite amount of values on hardware buses and therefore it must be restricted to a specific bit size. If the network is not restricted to specific values, the verification will be based on the values from the simulation of the SMEIL network, even though the network does not fail if the values increase

further. This can cause an unnecessary failure in the verification. If an SMEIL simulation of an unrestricted `addone` network results in the internal values reaching 10 and the FDR4 verification verifies more clock cycles than the SMEIL simulation, this would cause the values of the FDR4 verification to exceed the observed values, which would cause FDR4 to fail the verification, even though it would not cause a failure in the SMEIL network. It is, therefore, necessary that the user makes an informed choice as to the number of clock cycles to simulate in SMEIL but also to verify in FDR4. The number of simulated clock cycles and FDR4 verified clock cycles do not have to be equal, but in most cases, it might be the obvious choice.

As mentioned, the `addone` example should be restricted, so a `% 11` was added to the value-incrementing expression in the `add` process to avoid the value becoming larger than 10. Listing 43 shows the simulated `addone` network with the restriction added in the `add` process. Besides the restriction, this example is identical to the example in Listing 34 in Chapter 8. With this enhanced example, the SMEIL simulation provides reasonable observed values which can be used for the verification in FDR4. In Listing 44, a subset of the translated `addone` example can be seen which includes the restriction. The full CSP_M example can be seen in Listing 48 in Appendix C.

After 21 clock cycles, of the example in Listing 43, the value becomes larger than 10, but the restriction ensures that the verification will succeed even if verified for more than 21 clock cycles. The value will simply wrap around and continue at 0. If the restriction is removed and FDR4 verifies more than 21 clock cycles, the verification fails as expected.

This example is somewhat awkward to verify with FDR4, since it does not take advantage of the state space exploration that FDR4 provides. The lack of an input range for the system means that for each clock cycle a new state machine is verified, but only the internal values change. In this example, FDR4 only verifies the relation between internal values for each clock cycle with no external influence. The advantage of the clocked structure is that values, which might cause failures after a certain amount of clock cycles, are now possible to verify, which was not possible with the initial version of TAPS.

It is easy to see that this example never fails with the added restriction, but the example clearly introduces the clocked version of TAPS and how it is possible to verify clocked networks in FDR4 and therefore it still provides value to this thesis.

9.3 Problem Size Experiments

The examples presented in this thesis provide a suitable introduction to the translation and the verification in FDR4. They consist of straightforward structures that are simple to grasp, so the functionality of TAPS can be proved without confusion. The challenge with verification via exploration of a state space, is to keep the verification time to a minimum. This does not only apply for FDR4, but for model checking tools in general. FDR4 performs different kinds of internal optimisations on the networks, to minimise the state space before the refinement check. FDR4 also provides several compression algorithms, to provide further compression of larger problems.

I have performed some experiments on the seven segment display example, to examine the behaviour in FDR4. Both experiments have been run on an AMD Ryzen Threadripper 1950X processor with 16 cores at 3.4 GHz, with 128 GB 2933 MHz DDR4 ram. The experiments consist of measuring three different properties from the FDR4 verification. The first property is number of visited states which is a piece of information provided by FDR4. As previously presented, FDR4 performs compression to minimise the state space. This property provides an insight into the amount of work FDR4 performs when verifying a network, so it is interesting to learn

```
proc add (in inbus, const constant)
    bus outbus {
        val: u4 = 0 range 0 to 10;
    };
{
    outbus.val = (inbus.val + constant) % 11; //upper limit + 1
}

proc id (in inbus)
    bus outbus {
        val: u4 = 0 range 0 to 10;
    };
    var from_add: u4 range 0 to 10;
{
    from_add = inbus.val;
    outbus.val = from_add;
}

network addone_network ()
{
    instance id of id(add.outbus);
    instance add of add(id.outbus, constant: 1);
}
```

Listing 43: The restricted SMEIL network `addone_network` similar to the example in Listing 34.

how the number of visited states corresponds with the verification time. This will also give an insight into the inner workings of FDR4 and how the state space compression behaves. Because the seven segment display example is divided into three different assertions, one for each `time` process, FDR4 provides a separate number of visited states for each verified `time` network.

The second property is verification time which is measured by the `time` command. Even though all experiments have been performed on the same machine, to ensure uniformity, the GNU `time` command was used instead of the built-in `time` command that shells such as `bash` and `zsh` provide. This property will provide insight into the size of feasible inputs for a CSP_M network.

The last property is maximum resident set size, which is also provided by the GNU `time` command. The maximum resident set size describes the amount of memory the process holds. It will provide an insight into how much memory FDR4 requires to verify the network, and how the memory usage behaves as the input size increase. If FDR4 requires too much memory, it is not feasible to verify larger problems unless compression algorithms can reduce the state space.

The experiment has been designed to keep the internal system fixed and only increase the size of the input range for the system. This means that FDR4 will verify increasingly more values, but the network in itself stays the same. The lower bound of the input range will be fixed at 0 and the upper bound will be increased in steps of 500, up to 15000. The input range $\{0..15000\}$ represents 4 hours and 10 seconds. All three property values are gathered after FDR4 finishes the verification.

```
channel sync
channel d_read, c_read : { -1..15} -- u4 and initial value
channel d_write, c_write : { -1..15} -- u4 and initial value

DUM_VAL = -1 -- initial value

Add(i, inbus_channel) =
    (sync ->
     inbus_channel ? x ->
     sync ->
        if (x == DUM_VAL) -- initial value
            then (
                let
                    var = i
                within
                    var <= 15 & -- upper limit
                        c_read ! var -> Add(i, inbus_channel))
            else (
                let
                    var = (x + 1) % 11 -- (observed value + 1) restriction
                within
                    var <= 15 & -- upper limit
                        c_read ! var -> Add(i, inbus_channel))
    )
    [] SKIP


c_read_monitor(c) =
    (c ? x ->
    (0 <= x and x <= 10 or x == -1) & -- observed values and initial value
        c_read_monitor(c)
    ) [] SKIP
```

Listing 44: Sections of the translated addone network. The Add process has a restriction included to ensure no values above 10. The monitor process defines this range along with the acceptance of the dummy value -1. This example has been manually translated due to implementation limitations in the clocked version of TAPS. See the full code in Listing 48 in Appendix C.

9.3.1 Unclocked Experiment

The full code for the unclocked seven segment display example can be seen in Listing 46 in Appendix B. The unclocked seven segment display example consists of three `time` processes with associated monitor processes. Each `assert` function verifies the process monitor network described in Chapter 6.

Number of visited states For each verification, three assertions are performed within the seven segment display example, one for each `time` process network. With the unclocked example, all three verifications contain the same number of visited states, so this property will not be divided into three different results.

Figure 9.2 presents the results of the number of visited states property. From this graph, it is very clear how the state space increases linearly with the input range. This result means that FDR4 is not able to compress the state space further with the increase of input. A reason for FDR4 not providing any additional state space compression could be that because no input is

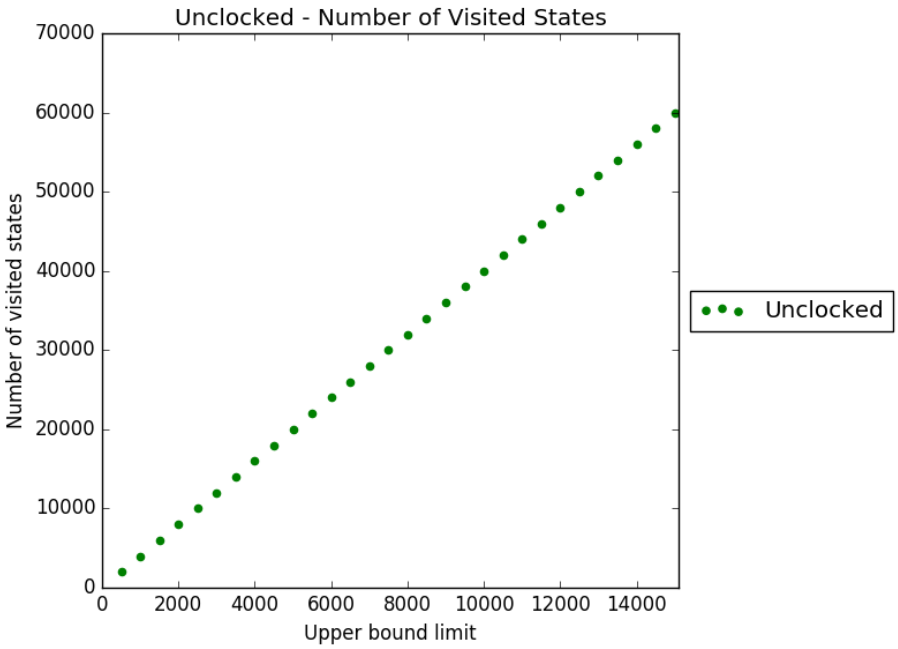

Figure 9.2: Graph of the number of visited states property from the unclocked seven segment display experiment.

repeated, it is not possible for FDR4 to provide further compression, so the number of visited states remains the same. This does, however, show how a problem quickly can become very large within FDR4, which is something a user must consider when choosing the data to verify.

Verification time In Figure 9.3 the verification time results can be seen. The graph represents the verification time in seconds for each increase in the input range. As can be seen, the verification time increases exponentially with the input values. Since the number of visited states is increasing linearly, it can seem odd that the verification time does not follow that same pattern. However, besides the refinement checking of the Generalised LTS (GLTS), which will increase with the number of visited states, FDR4 must compile the network and generate the GLTS. It is reasonable to expect that the larger the state space, the more effort for FDR4 to complete all the steps of the verification. Therefore these results are consistent with what could be expected.

Maximum resident set size The result from this property can be seen in Figure 9.4. These results are not fitted to a simple function as well as the other two experiment properties. It is clear that the amount of memory used for the verification grows with an increase in the input range. It is also somewhat consistent until around 10000 in upper bound limit. This fluctuation could be caused by some internal structure in FDR4 or it could be a result of other processes running on the machine that is running the verification. Unfortunately, FDR4 does not provide a lot of information about the internal workings, so it can be very difficult to examine these results further. However, the results are overall consistent with the results from both number of visited states and verification time.

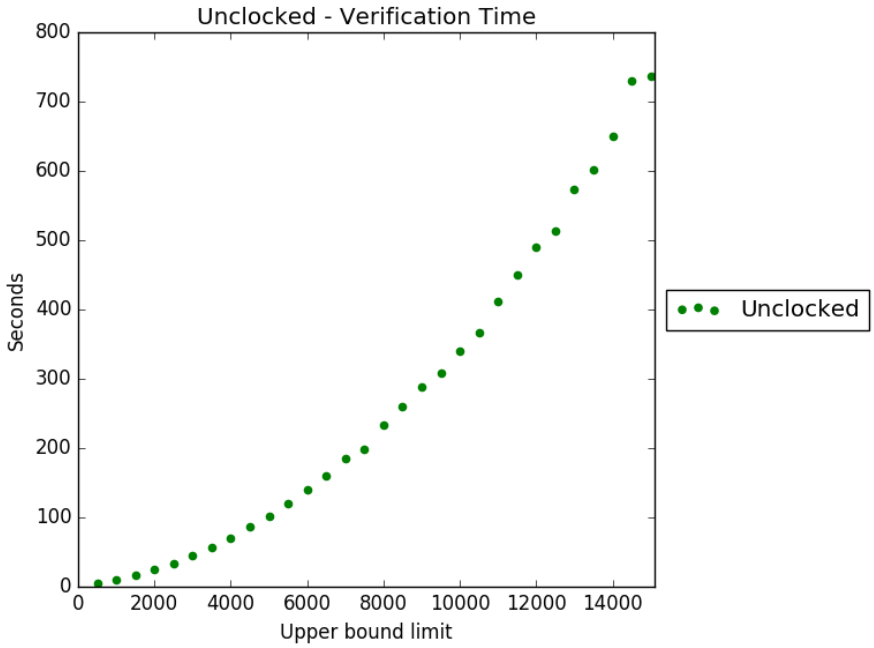

Figure 9.3: Graph of the verification time property from the unclocked seven segment display experiment.

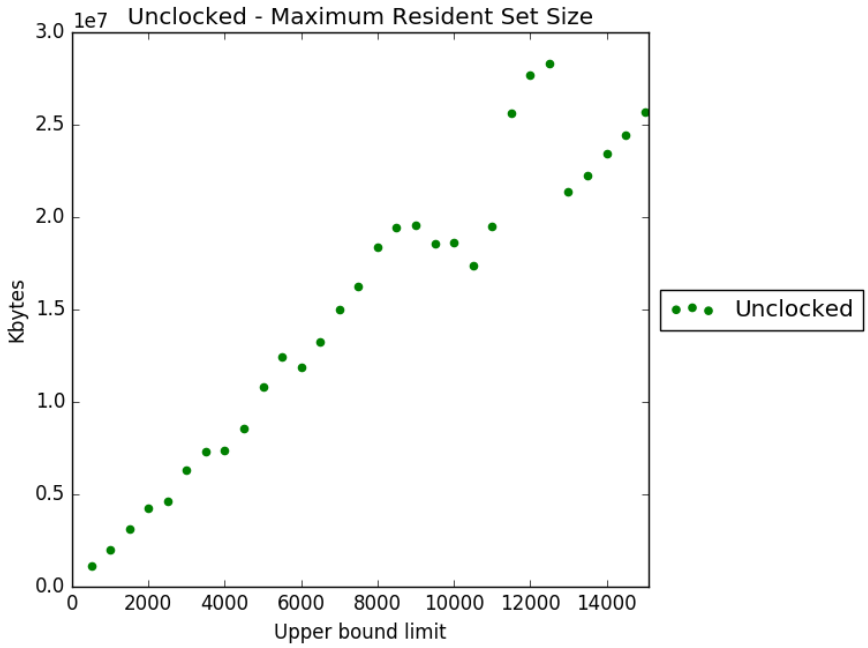

Figure 9.4: Graph of the maximum resident set size property from the unclocked seven segment display experiment.

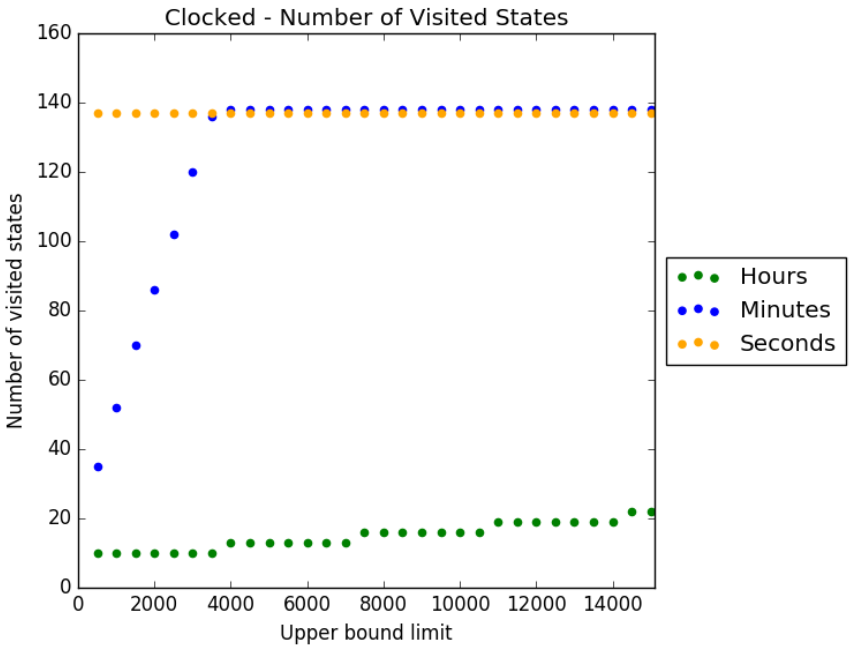

Figure 9.5: Graphs of the three number of visited states properties from the clocked seven segment display experiment.

9.3.2 Clocked Experiment

The full code for the clocked seven segment display example can be seen in Listing 47 in Appendix B. As in the unclocked seven segment display example, the clocked example also consists of three `time` processes with associated monitor processes. In order to make the clocked seven segment display experiment equivalent with the unclocked seven segment display example, the `Clock` process in the clocked network only verifies one clock cycle and therefore the two networks should be equivalent in the verification.

Number of visited states In Figure 9.5, the results of the clocked number of visited states property can be seen for each `time` verification. As can be seen, they differ quite a lot from each other. The results of the experiment, show an increase in the number of visited states for `Hours` in input values between 3500 and 4000. Further investigation shows that the number of visited states increase precisely every 3600 increase. This corresponds to the number of seconds in an hour, meaning that the number of visited states increases linearly with the number of hours. The number of visited states for `Minutes` increase linearly until between 3500 and 4000. Also, in this case, further investigation shows that the number of visited states increase until exactly 3539 where it levels out and stays at 134 for the rest of the verifications. It is very clear from this result that the number of visited states reaches its maximum when the input range represents the maximum amount of minutes. The input range $\{0..3539\}$ represents 59 minutes which is maximum. The number of visited states for `Seconds` is constant at 134, however if the input range was decreased to $\{0..59\}$ the same result could be seen as with the `Minutes` graph. These results make it clear that FDR4 is able to decrease the number of states to the number of `Hours`, `Minutes` and `Seconds` represented by the input.

Verification time In Figure 9.6, the clocked verification time results can be seen. This graph is very similar to the verification time results for the unclocked experiment, but with a slightly

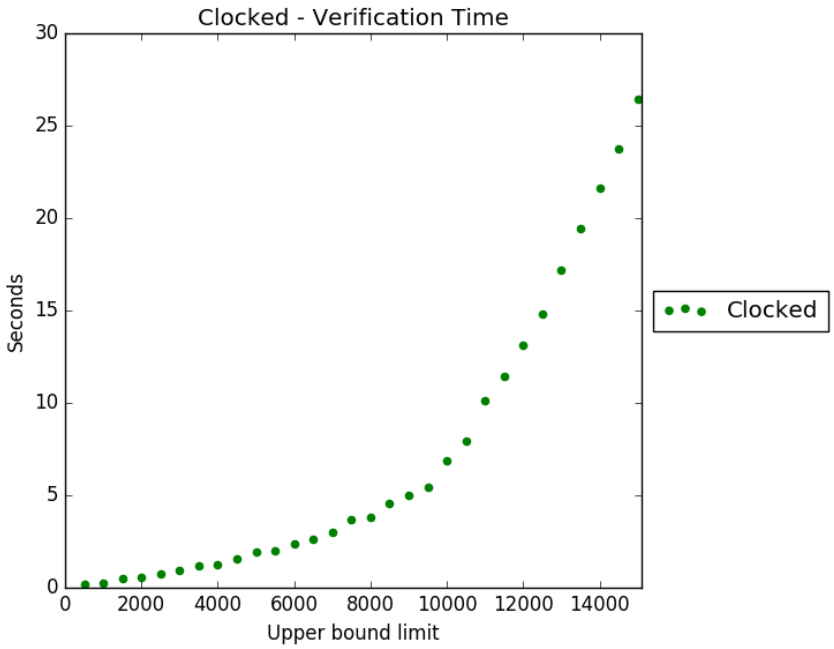

Figure 9.6: Graph of the verification time property from the clocked seven segment display experiment.

different increase. However, the values also increase exponentially with the input which, as explained above, is to be expected. What is important to notice is that even though the graph is similar to the unclocked verification time graph, the number of seconds is very different than in the first experiment. The reason for this is probably the decrease in number of visited states which will be discussed further below.

Maximum resident set size Figure 9.7 shows the clocked maximum resident set size. This graph shows the same situation as the verification time graph. The graph is similar to that of the unclocked maximum resident set size with a slightly more flattened increase and not nearly as much fluctuation. The big difference with this graph is, as with the verification time property, the low values compared to the unclocked maximum resident set size values. This is also discussed further below.

9.3.3 Results

As explained above, the number of visited states property from the unclocked seven segment display experiment showed to be equal for all three `time` verifications whereas, for the clocked experiment, the number of visited states varied between the three verifications. Figure 9.10 represents all four graphs, but the unclocked values are several order of magnitude larger than the clocked values, making the clocked values hard to discern in the graph.

In Figure 9.11 the combined verification time graph can be seen. Here, the difference also shows very clearly. The clocked experiment verifies much faster than the unclocked. Again it is difficult to see the actual values of the clocked version in the graph because they are significantly lower. The same case can be seen in Figure 9.12 which represents the combined maximum resident set size.

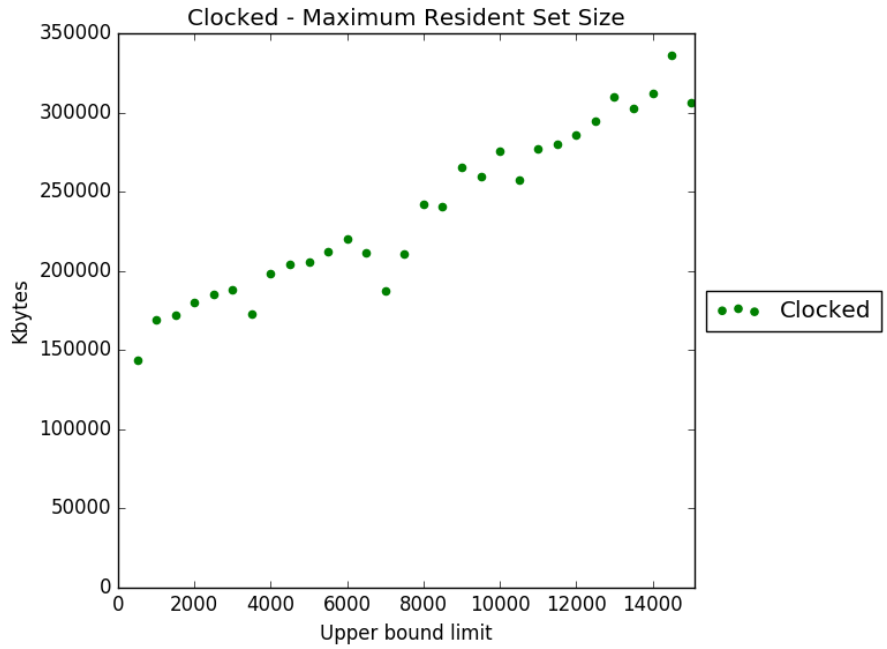

Figure 9.7: Graph of the maximum resident set size property from the clocked seven segment display experiment.

Both verification time and resident set size seem to be affected by the number of visited states property. This is evident because the number of visited states is the only internal FDR4 property of the experiment. Unfortunately, it can be quite difficult to investigate why the clocked experiment performs so much better than the unclocked. When looking at the trace of both networks within FDR4, both verify the full input range, as expected. The visualiser ProBE also shows equivalent traces and the two networks behave identically on failures. The only difference between the two networks is that in the clocked system all processes are recursive and synchronise together before continuing. The unclocked processes will always terminate after one clock cycle. However, since the clocked network only simulates one clock cycle, the two networks should still be equivalent. These differences do not affect the actual values verified, or the result of the verification.

As previously stated when introducing the clocked version of TAPS in Chapter 8, the increase in complexity might increase the verification time in FDR4, but in this case, it is the opposite. As the two networks seem equivalent in both input range and verification, the reason for the difference in performance might lie in what the number of visited states property shows. It might be that FDR4 is able to compress the state space in the clocked experiment while not being able to perform the same compression in the unclocked experiment.

The FDR4 tool provides a machine structure viewer which can provide information about how FDR4 represents the processes and how effective the compression is. When looking at the two different results from FDR4, it becomes clear why the clocked experiment performs so much better than the unclocked experiment. In Figure 9.8 and Figure 9.9, the output of the machine structure viewer can be seen. In the unclocked result in Figure 9.8, the unclocked network `N_hours` is represented by a high-level machine with 12 formats, 77 rules, and 34 leaf machines. A high-level machine means a process with subprocesses and the only information needed about formats and rules are that the more there are, the more complex the machine.

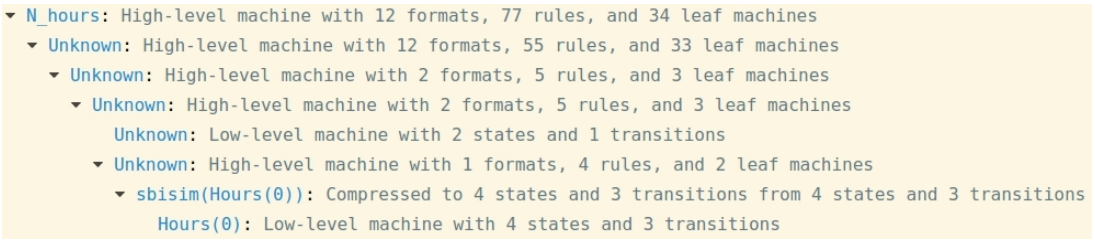

Figure 9.8: Screen dump of the results of the unclocked seven segment display network in the FDR4 machine structure viewer.

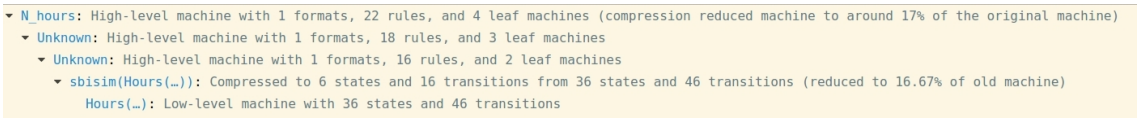

Figure 9.9: Screen dump of the results of the clocked seven segment display network in the FDR4 machine structure viewer.

Because all events are hidden in the network, as explained in Chapter 6, the subprocesses are `Unknown`. The second to last line shows that the compression algorithm `sbisim` has been applied to `Hours(0)` but it also shows that it was not able to reduce the number of states or transitions.

Figure 9.9 represents the clocked network and, as can be seen, the `N_hours` process includes compression information. It provides an estimate that the reduced machine is around 17% the size of the original machine. On the second to last line, it can be seen that the `sbisim` compression algorithm is not applied to `Hours(0)` as in the unclocked experiment, but to `Hours(..)`, which indicates that FDR4 represents the two networks differently. The `sbisim` compression algorithm is able to compress the state space from 36 states to 6 states. It clearly shows that because FDR4 is able to perform compression on the clocked system, the performance also increases dramatically.

It is quite surprising that the clocked experiment has such a drastic difference and that it actually performs better than the unclocked version. As the network increases in complexity, the state space should also increase, and it is because of the compression that the clocked version performs so well. In this aspect, FDR4 is a black box, and it is therefore not possible to learn exactly why FDR4 is able to perform compression on the clocked network and not on the unclocked. It is, of course, interesting to experiment further to find patterns in the CSP$_M$ structures that suit the compression algorithm best.

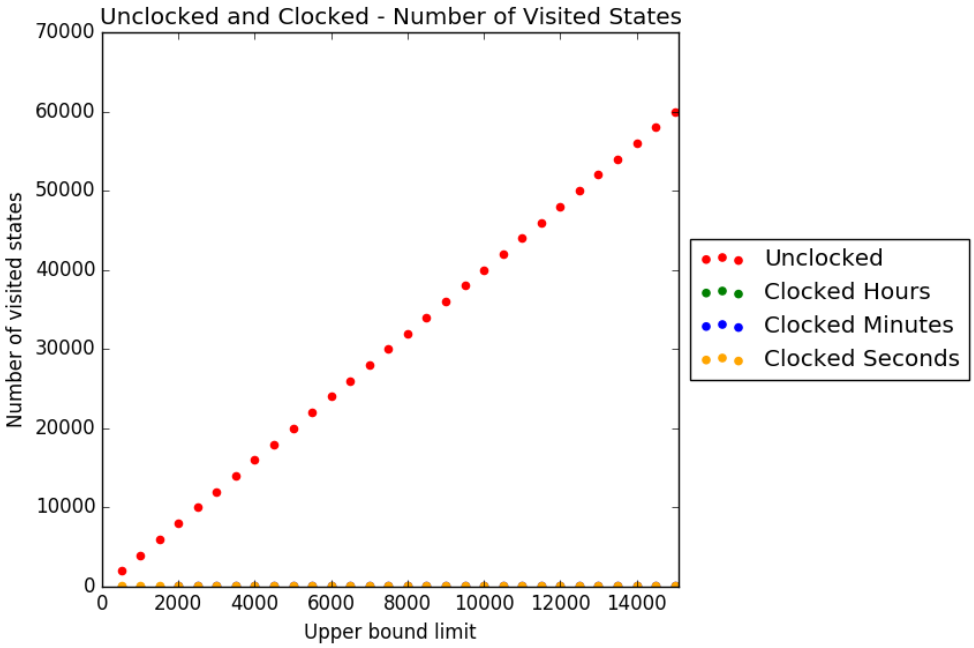

Figure 9.10: Graphs of the number of visited states properties combined from both the unclocked and clocked seven segment display experiment.

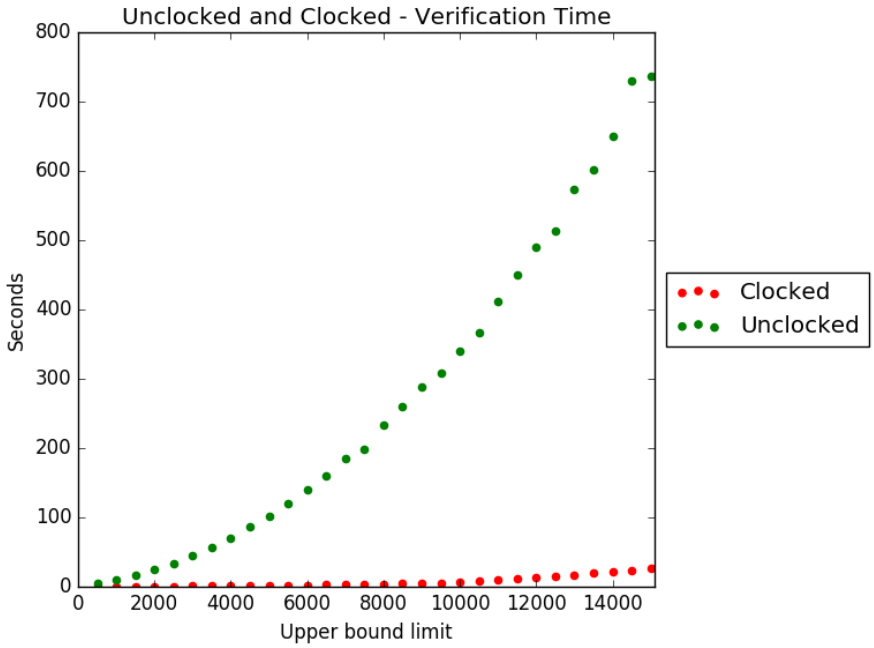

Figure 9.11: Graphs of the verification time properties combined from both the unclocked and clocked seven segment display experiment.

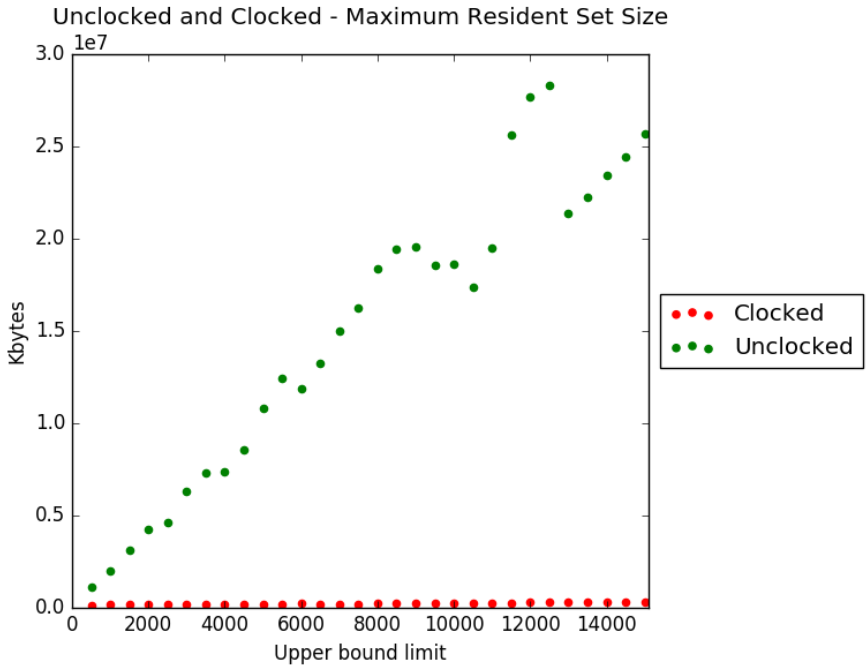

Figure 9.12: Graphs of the maximum resident size set properties combined from both the unclocked and clocked seven segment display experiment.

Chapter 10

Discussion

In this chapter, I will discuss the usability of TAPS and how the clocked version increases the set of verifiable problems. I will also discuss how TAPS currently only allows translation of pure SMEIL networks, and how TAPS can be extended for co-simulation. Lastly, I will discuss the advantages and limitations of the FDR4 tool.

10.1 Usability of TAPS

TAPS have proven to be a capable tool for automatic translation from SMEIL to CSP$_M$. It is still in early stages of development and more examples must be developed and verified, in order to demonstrate how it handles more aspects of the SMEIL language.

To my knowledge, transpiling from a hardware description language to a specification language like CSP has not been done successfully before, and the advantages of such a system are many. When using TAPS, it is not necessary to develop a test bench for the system, and it is not necessary to develop the specification model separately from the hardware model. This means that the workflow from developing the hardware model to verifying it, has been simplified dramatically with TAPS. I believe that being able to perform verification with no extra development steps will increase the attractiveness of SME for hardware development and further its advantage over traditional HDLs.

There are limitations to TAPS and it is still possible for failures to happen with systems verified with TAPS. The bottlenecks of the verification lie both in the observed values as well as in the number of verified clock cycles. The values verified in FDR4 are based on the observed values from the simulation, so if the SMEIL simulation does not represent all possible values, the verification will be incomplete. It is difficult to find a balance between how long to simulate the SMEIL network and how critical failures can be. In some systems it will not be possible to know when all corner cases have been found in the simulation, however, FDR4 might provide insight into some corner cases missing in the simulation.

In the clocked version of TAPS, additional information must be provided to the system. The user must define the number of clock cycles to verify, which bring more uncertainty to the verification. If the user chooses too few clock cycles to verify, potential failures might not be caught by FDR4. The number of clock cycles to verify is a balance between the input range for the system, the time requirement of the verification, as well as the complexity of the system. The user must ultimately determine the balance between verification and verification time, based on the program at hand.

The reason that it is possible to create the relatively simple translations from SMEIL to CSP$_M$,

lies in the fact that SME is defined from the CSP model and therefore, as previously mentioned, all SME models will have an equivalent CSP model. Having the same basis on both sides of the translation results in a much smoother translation.

10.2 Clocked or Unclocked CSP$_M$ Systems

The initial idea for the design of TAPS was to build a system which did not have to enforce a global synchronous clock structure, since it would increase the complexity drastically. As the implementation of the initial version of TAPS progressed, it was clear that the design was too simple. The set of verifiable problems in the initial version of TAPS are too small to have an actual impact. The clocked version of TAPS is still able to verify the set of problems verifiable with the initial version of TAPS. By only verifying one clock cycle, the results are the same as for the unclocked version of TAPS. It is, of course, an advantage that the new design only adds to the set of verifiable problems, and doesn't reduce it.

It is a huge advantage that the clocked version of TAPS is able to reuse such a large part of the initial design. It was possible to extend the initial version of TAPS instead of rewriting the entire system, which also made it feasible to design within the timeframe of this thesis. It is also clear that the set of verifiable problems with the clocked TAPS system are far bigger than the initial version of TAPS, so there is no doubt that this extension has been an advantage, in spite of the increase in complexity. The clocked CSP$_M$ network provides comparability with the original SMEIL network, which I believe is a huge advantage for the further development of TAPS. When developing and testing new features of TAPS, it is a big advantage to be able to compare the behavior of the SMEIL simulation to that of the FDR4 verification.

From the experiments presented previously, it is clear that the clocked structure performs a lot better than the unclocked system, which increases the usability of TAPS. It is, of course, necessary to perform further experiments to see if the compression is possible because of the general clocked structure of the network, or because of the structure of that specific clocked network. The implementation of the clocked version of TAPS is still in progress, and there are bound to be challenges to the translation structures, which will require some ingenuity to ensure proper code generation.

10.3 From Pure SMEIL to Co-Simulation

Co-simulation is a major feature of SMEIL and yet I only focused on pure SMEIL support in this thesis. The reason for this decision was to have a simple base to start from, to ensure the accuracy of the translations. TAPS should definitely be extended to support translation of co-simulated networks. TAPS will not have any real value to the industry or academia until this happens, because pure SMEIL does not allow for a lot of functionality. However, I do believe that the extension to support co-simulated programs is possible.

The external communication between SMEIL and another SME implementation can be represented in trace files, generated from the simulation. TAPS can use these data as the observed values for the translation. The external channels are defined with the keyword `exposed`, so TAPS can recognise these, and read the data directly from the corresponding trace file. This way, TAPS would be able to generate a channel range similar to the internal observed values. It will, of course, be a challenge to handle the data and communication that spans across the SME implementations in co-simulation. However, since it is very precisely defined which buses communicate across the two implementations, and the data communicated can be saved in trace files, it should be possible to implement.

Focusing first on pure SMEIL programs have definitely led to a more manageable set of problems, but extending this to co-simulated problems seems like the perfect next step.

10.4 Verification with FDR4

There is no doubt that there lies a huge advantage in the possibility of hardware verification. Failures in critical systems can be disastrous and testing simply does not provide the security needed for these types of systems. FDR4 is a solid system and the addition of the ProBE visualiser within FDR4 is a huge advantage to provide an understanding of the network, but FDR4 does have its limitations. Even though it is relatively simple to verify a network in FDR4, the counterexamples can be challenging to understand, depending on the network. Understanding the different aspects and functionality of FDR4, requires the user to know a substantial amount about the structures of CSP. This means that the requirement of understanding CSP is not entirely obsolete when using TAPS. The developer does not need to model the specification manually, but verifying it requires some CSP knowledge. The need to have a basic knowledge of CSP occurs for example in the debug viewer. Here, a process that terminates successfully ends the trace with the symbol ✓. Processes that deadlocks or behaves as the `STOP` process are defined with the symbol $\{\}$. These two symbols are standard syntax in CSP, but not for CSP_M. This means that if the user does not have any knowledge in CSP, it can be difficult to understand the debug viewer since it uses different symbols than used in the CSP_M program. Therefore the user must have a basic knowledge of CSP to utilise all the different possibilities of FDR4.

With FDR4, as with every other model checking system, the size of the state space is a bottleneck that can hinder the verification of larger problems. FDR4 performs state space compression and it also provides several compression algorithm functions to apply directly within the CSP_M network. However, it is essential that the verification time is manageable, and for networks like the unclocked seven segment display network, this is not the case. As it is clear from the results in Chapter 9, a network can quickly grow to a point where verification is infeasible due to time constraints. This is, however, a general problem with model checking and it is an active research area that is continuing to improve.

Chapter 11

Future Work

SMEIL is not a complete implementation, and as other SME implementations are improved, so should the SMEIL implementation. As the SMEIL language becomes more comprehensive and supports more features, TAPS should be kept in line with these advancements.
The automatic verification provided by TAPS lightens a major workload in testing and verifying hardware models. Therefore, it might also be relevant to structure the SMEIL implementation towards better FDR4 results, while still maintaining the basic SME model structure.

The extended version of TAPS provides verification of the different internal states within a hardware model. It was introduced rather late in the project, and therefore the implementation is not as extensive as the initial version of TAPS. Substantial groundworks have been laid in providing the design structures for the extended version, making further development more straightforward. It is, however, obvious that future work should include providing a full implementation of the extended TAPS system.

Besides further development of the extended version of TAPS, more advanced and in-depth examples should be developed in SMEIL, in order to understand the limitations of the translation as well as FDR4. This will also provide a better understanding of the compression FDR4 provides. It is important to learn if the compression could be performed in the experiments because of the general clocked network structure, or if it is only performed in that specific network.

FDR4 also provides the possibility for integration with other tools using the FDR4 API, which is currently available for C++, Java, and Python. The FDR4 API is currently not used in TAPS, but it is an obvious choice to extend TAPS with the FDR4 API to provide a cleaner workflow. Because TAPS has been developed for FDR4 specifically, no other verification tool will match the current translation structures, and therefore the current version of TAPS would benefit from using FDR4 directly via the API.

As described in Chapter 4, SMEIL was mainly developed to provide an intermediate language between the existing SME implementations. To provide support for the entire SMEIL functionality, TAPS should, of course, be augmented to support co-simulation as well. As mentioned in Chapter 10, it might be possible to make use of external communication trace files, to generate channel ranges for the generated CSP_M network.

Another point for future work is to extend the different assertion possibilities within TAPS. Currently, only channel communication can be verified, but as described in Chapter 7 these monitor processes do have their limitations, as it is not currently possible to verify a combination of values. Therefore it would be an advantage to extend TAPS to define more advanced assertions to verify values over multiple channels.

It would also be an advantage to extend TAPS to support software-hardware co-design. The idea behind software-hardware co-design is that hardware and software are designed in parallel, so that both can be implemented on either hardware or software depending on what is most suited. If SMEIL and TAPS were extended to support this, then the communication between software and hardware would be possible to verify with FDR4.

When verifying a system in FDR4, it can be crucial for the developer to know what values and states have been verified. It is therefore desirable to have TAPS generate a human-readable report on the ranges and communications that are verified with TAPS. This would also give the developer a possibility of better evaluating the number of clock cycles to verify in FDR4. This report could become a standard addition to the documentation of the hardware model, which would give a developer an easy overview of a complicated system, and would also allow for reasoning about the system.

Chapter 12

Conclusion

In this thesis, I have presented TAPS, a transpiler that translates programs written in the new SME intermediate language (SMEIL) into the machine-readable CSP language (CSP_M). In TAPS, the translated CSP_M code is augmented with assertion properties which can be verified with the Failure-Divergences Refinement tool (FDR4).

I have presented an initial system, which is able to translate SMEIL networks to CSP_M, and verify a large range of defined input values. TAPS can assert that the observed values of a channel in a simulated SMEIL program are in fact the only possible values communicated on that specific channel.

An extension to the initial version of TAPS has also been presented in this thesis. This extension provides TAPS with the support to model a global synchronous clock in CSP_M. It has previously been shown [61] that enforcing a global synchronous clock onto a PyCSP network results in code complexity explosion. To avoid having to model networks with an enormous amount of states, the SME model was developed, based on the CSP algebra. With the extended version of TAPS, the state explosion still occurs, but because TAPS automatically generates the CSP_M code, the state explosion is not a hindrance to the translation.

The translated CSP_M network can be verified with the FDR4 tool, which can perform refinement checks on CSP_M networks. I present examples of failures and solution to these with an example of a seven segment display network and a cyclic "addone" network, showing the advantages of both the initial version of TAPS, and the clocked version.

This system makes it more accessible for software programmers to program hardware, thereby bridging a gap between software programmers and the needs of the industry. Instead of having to create advanced test-benches, TAPS provides a simple way to verify the hardware model via the assertion functionalities of FDR4. TAPS does not yet provide support for the entire SMEIL language. In spite of this, I am pleased with the possibilities that TAPS present, and I have shown the current capabilities of TAPS are valuable.

Part of this work has been published in the paper *Towards Automatic Program Specification Using SME Models* [64], which is included in full length in Appendix D. The extended clocked version of TAPS was designed subsequently.

Bibliography

[1] FDR4 Manual Example Files. `https://www.cs.ox.ac.uk/projects/fdr/manual/examples/index.html`. [Online; accessed October 2018].

[2] Go Project. The Go programming language. `https://golang.org`. [Online; accessed October 2018].

[3] Jinja2 Documentation. `http://jinja.pocoo.org/docs/2.10/`. [Online; accessed November 2018].

[4] The Coq Proof Assistant. `https://coq.inria.fr/`. [Online; accessed October 2018].

[5] The ProB project. `https://www3.hhu.de/stups/prob/`. [Online; accessed October 2018].

[6] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009.

[7] A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors. *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*, volume 3525. Springer Science & Business Media, 2005.

[8] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[9] V. Agrawal and S. H. C. Poon. VLSI Design Process. In *Proceedings of the 1985 ACM Thirteenth Annual Conference on Computer Science*, CSC '85, pages 74–78. ACM, 1985.

[10] R. Alur and D. Dill. Automata for modeling real-time systems. In M. S. Paterson, editor, *Automata, Languages and Programming*, pages 322–335, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[11] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise, and W. Yi. *UPPAAL - Now, Next, and Future*, pages 99–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[12] P. J. Armstrong, G. Lowe, J. Ouaknine, and B. Roscoe. Model checking Timed CSP. In *HOWARD-60*, pages 13–33, 2014.

[13] T. Asheim. LibSME source code. `https://github.com/truls/libsme`. [Online; accessed November 2018].

[14] T. Asheim. PySME source code. `https://github.com/truls/pysme`. [Online; accessed November 2018].

[15] T. Asheim. Implementing high performance synchronous message exchange. Bachelor's thesis, University of Copenhagen, Niels Bohr Institute, 2015.

[16] T. Asheim. A domain specific language for synchronous message exchange networks. Master's thesis, University of Copenhagen, Niels Bohr Institute, 2018.

[17] T. Asheim. SMEIL: A Domain-Specific Language for Synchronous Message Exchange Networks. In K. Chalmers, J. Pedersen, M. Smith, K. Skovhede, and P. Welch, editors, *Communicating Process Architectures 2018*. IOS Press, Amsterdam, The Netherlands, August 2018.

[18] T. Asheim, K. Skovhede, and B. Vinter. VHDL Generation From Python Synchronous Message Exchange Networks. In K. Chalmers, J. Pedersen, J. Broenink, B. Vinter, and P. Welch, editors, *Communicating Process Architectures 2016*, pages 161 – 184. IOS Press, Amsterdam, The Netherlands, August 2016.

[19] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a tool suite for automatic verification of real-time systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, pages 232–243, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[20] 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.

[21] Celoxica, Handel-C. Language reference manual. *Document number: RM-1003-4.2*, 2002.

[22] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[23] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[24] J. Dibley and K. Bradshaw. Deriving Reusable Go Components From Verified CSP Prototypes. In K. Chalmers, J. Pedersen, M. Smith, K. Skovhede, and P. Welch, editors, *Communicating Process Architectures 2018*. IOS Press, Amsterdam, The Netherlands, August 2018.

[25] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[26] Formal Systems (Europe) Ltd. *Process Behaviour Explorer - ProBE User Manual*, 2003.

[27] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. *Failures Divergences Refinement (FDR) Version 3*, 2013.

[28] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. 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.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[30] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

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

[32] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[33] J.L. Lions. ARIANE 5 flight 501 failure, report by the inquiry board. `http://sunnyday.mit.edu/nasa-class/Ariane5-report.html`, 1996. [Online; accessed November 2018].

[34] J. Kepner. HPC productivity: An overarching view. *The International Journal of High Performance Computing Applications*, 18(4):393–397, 2004.

[35] B. Labs. The SPIN model checker. `http://spinroot.com/spin/whatispin.html`. [Online; accessed October 2018].

[36] L. Lamport. The "hoare logic" of concurrent programs. *Acta Informatica*, 14(1):21–37, 1980.

[37] 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.

[38] M. Leuschel and M. Butler. ProB: A Model Checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, pages 855–874, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[39] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *IEEE computer*, 26(7):18–41, 1993.

[40] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier User Manual. Technical report, Stanford, CA, USA, 1979.

[41] R. Milner. Logic For Computable Functions Description of a Machine Implementation. Technical report, 1972.

[42] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, 1992.

[43] M. Oliveira and J. Woodcock. Automatic Generation of Verified Concurrent Hardware. In M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, editors, *Formal Methods and Software Engineering*, pages 286–306, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[44] A. N. Parashkevov and J. Yantchev. ARC-a tool for efficient refinement and equivalence checking for CSP. In *Proceedings of 1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing, ICA/sup 3/PP '96*, pages 68–75, June 1996.

[45] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[46] J. B. Pedersen and P. H. Welch. The symbiosis of concurrency and verification: teaching and case studies. *Formal Aspects of Computing*, 30(2):239–277, 2018.

[47] J. D. Phillips and G. S. Stiles. An Automatic Translation of CSP to Handel-C. In I. R. East, D. Duce, M. Green, J. M. R. Martin, and P. H. Welch, editors, *Communicating Process Architectures 2004*, pages 19–38, sep 2004.

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

[49] 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.

[50] V. Raju, L. Rong, and G. S. Stiles. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In J. F. Broenink and G. H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 63–81, sep 2003.

[51] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1):249–261, 1988.

[52] M. Rehr, K. Skovhede, and B. Vinter. BPU Simulator. In P. H. Welch, F. R. M. Barnes, J. F. Broenink, K. Chalmers, J. B. Pedersen, and A. T. Sampson, editors, *Communicating Process Architectures 2013*, pages 233–248, nov 2013.

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

[54] A. W. Roscoe. *Understanding Concurrent Systems*. Springer Science & Business Media, 2010.

[55] A. W. Roscoe, S. Brookes, and C. Hoare. A theory of communicating sequential processes. *Journal of the ACM*, (3):560–599, July 1984.

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

[57] B. Scattergood and P. Armstrong. CSPM: A Reference Manual. 2011.

[58] S. Schneider and H. Treharne. Communicating B Machines. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002:Formal Specification and Development in Z and B*, pages 416–435, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[59] S. Schneider, H. Treharne, A. A. McEwan, and W. Ifill. Experiments in Translating CSP||B to Handel-C. In P. H. Welch, S. Stepney, F. Polack, F. R. M. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson, editors, *Communicating Process Architectures 2008*, pages 115–133, sep 2008.

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

[61] E. Skaarup and A. Frisch. Generation of FPGA Hardware Specifications from PyCSP Networks. Master's thesis, University of Copenhagen, Niels Bohr Institute, 2014.

[62] K. Skovhede and B. Vinter. Building hardware from c# models. In *FSP 2016; Third International Workshop on FPGAs for Software Programmers*, pages 1–9, Aug 2016.

[63] J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, pages 709–714, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[64] A. Thegler, M. Larsen, K. Skovhede, and B. Vinter. Towards Automatic Program Specification Using SME Models. In K. Chalmers, J. Pedersen, M. Smith, K. Skovhede, and P. Welch, editors, *Communicating Process Architectures 2018*. IOS Press, Amsterdam, The Netherlands, August 2018.

[65] United States of America, General Accounting Office. LIMTEC-92-26 Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia. Technical report, 1992.

[66] Univeristy of Kent. occam-pi: blending the best of CSP and the pi-calculus. `https://www.cs.kent.ac.uk/projects/ofa/kroc/`. [Online; accessed October 2018].

[67] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society, 1986.

[68] B. Vinter, J. M. Bjørndalen, and R. M. Friborg. PyCSP Revisited. In P. H. Welch, H. Roebbers, J. F. Broenink, F. R. M. Barnes, C. G. Ritson, A. T. Sampson, G. S. Stiles, and B. Vinter, editors, *Communicating Process Architectures 2009*, pages 263–276, November 2009.

[69] B. Vinter and K. Skovhede. Synchronous Message Exchange for Hardware Designs. In P. Welch, F. Barnes, J. Broenink, K. Chalmers, T. Gibson-Robinson, R. Ivimey-Cook, A. McEwan, J. Pedersen, A. Sampson, and M. Smith, editors, *Communicating Process Architectures 2014*, pages 169 – 180. Open Channel Publishing Ltd., Bicester, UK, August 2014.

[70] B. Vinter and K. Skovhede. Bus Centric Synchronous Message Exchange for Hardware Designs. In K. Chalmers, J. Pedersen, F. Barnes, J. Broenink, R. Ivimey-Cook, A. Sampson, and P. Welch, editors, *Communicating Process Architectures 2015*, pages 257 – 268. IOS Press, Amsterdam, The Netherlands, August 2015.

[71] W. Zhou and G. S. Stiles. The Automated Serialization of Concurrent CSP Scripts using Mathematica. In P. H. Welch and A. W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 15–32, sep 2000.

Appendix A

How To Use TAPS

The TAPS source code can be found at `https://github.com/althegler/TAPS`

The required dependencies for generating the ANTLR4 parser, running TAPS and verification with FDR4 are listed below:

- ANTLR4
- Python 2.7
- ANTLR Python runtime
- FDR4

ANTLR4 and FDR4 can both be downloaded from their websites, where installation instructions are also provided. The ANTLR4 project can be found at `https://www.antlr.org/` and the FDR4 Project at `https://www.cs.ox.ac.uk/projects/fdr/`. It is also necessary to download the ANTLR4 Python runtime from `https://pypi.org/project/antlr4-python2-runtime/`. Assuming that an ANTLR4 alias has been created as the ANTLR4 installation suggest, the parser and lexer can be generated with ANTLR4 using the command:
`antlr4 -Dlanguage=Python2 - visitor -no-listener Smeil.g4`. This will create all the required parser and lexer files, as well as the visitor methods. This step is only necessary if the .g4 grammar file has been modified.

When the ANTLR4 files have been generated, TAPS can be used directly with a well-formed SMEIL program with the command: `python taps.py input.sme output.csp`. The system requires both an input file and an output file. If the output file does not exist, it will be created by TAPS.

The resulting CSP_M file can be verified in FDR4, either by the command line tool or by the FDR4 tool, which is a graphical tool. The command line tool can be used by the command `refines output.csp`. There are several options to adjust the output of the command. The FDR4 command line tool is mostly used to quickly check if a network passes the verification, because it is difficult to navigate the counterexamples. The FDR4 graphical tool provides a better visualisation of counterexamples, and the ProBE visualiser can be called directly from the FDR4 graphical tool.

Appendix B

Seven Segment Display Example Full Code

Seven Segment Display Example SMEIL Code

```
proc clock ()
    bus out {
        val: u17 range 1 to 86401;
    };
    var i: u17 = 0 range 0 to 86401;
{
    i = i + 1;
    out.val = i;
}


proc hours (in hours_in)
    bus out {
        first_digit: u2 range 0 to 2;
        second_digit: u4 range 0 to 9;
    };
    var hours: u5 range 0 to 23;
    var hours_first_temp: u2 range 0 to 2;
    var hours_second_temp: u4 range 0 to 9;
{
    hours = hours_in.val / 3600 % 24;
    hours_first_temp = hours / 10;
    hours_second_temp = hours % 10;
    out.first_digit = hours_first_temp;
    out.second_digit = hours_second_temp;
}


proc minutes (in minutes_in)
    bus out {
        first_digit: u3 range 0 to 5;
        second_digit: u4 range 0 to 9;
    };
    var minutes: u6 range 0 to 59;
    var minutes_first_temp: u3 range 0 to 5;
    var minutes_second_temp: u4 range 0 to 9;
```

```
37
38  {
39      minutes = minutes_in.val / 60 % 60;
40      minutes_first_temp = minutes / 10;
41      minutes_second_temp = minutes % 10;
42      out.first_digit = minutes_first_temp;
43      out.second_digit = minutes_second_temp;
44  }
45
46
47  proc seconds (in seconds_in)
48      bus out {
49          first_digit: u3 range 0 to 5;
50          second_digit: u4 range 0 to 9;
51      };
52      var seconds: u6 range 0 to 59;
53      var seconds_first_temp: u3 range 0 to 5;
54      var seconds_second_temp: u4 range 0 to 9;
55  {
56      seconds = seconds_in.val % 60;
57      seconds_first_temp = seconds / 10;
58      seconds_second_temp = seconds % 10;
59      out.first_digit = seconds_first_temp;
60      out.second_digit = seconds_second_temp;
61  }
62
63
64  network clock_network ()
65  {
66      instance g of clock();
67      instance h of hours(g.out);
68      instance m of minutes(g.out);
69      instance s of seconds(g.out);
70  }
```

Listing 45: The full SMEIL code used for transpiling in the seven segment display example.

Unclocked Seven Segment Display Example CSP$_M$ Code

```
1   channel clock_out_val : {0..131071}
2
3   channel hours_out_first_digit : {0..3}
4   channel hours_out_second_digit : {0..15}
5
6   channel minutes_out_first_digit : {0..7}
7   channel minutes_out_second_digit : {0..15}
8
9   channel seconds_out_first_digit : {0..7}
10  channel seconds_out_second_digit : {0..15}
11
12
13  Hours(hours_in) =
14  let
15      hours = hours_in / 3600  % 24
16      hours_first_temp = hours / 10
```

```
17      hours_second_temp = hours % 10
18  within
19      hours_out_first_digit ! hours_first_temp ->
20      hours_out_second_digit ! hours_second_temp ->
21      SKIP
22
23  Hours_out_first_digit_monitor(c) =
24      c ? x ->
25      (0 <= x and x <= 2) & SKIP
26  Hours_out_second_digit_monitor(c) =
27      c ? x ->
28      (0 <= x and x <= 9) & SKIP
29
30
31  Minutes(minutes_in) =
32  let
33      minutes = (minutes_in / 60)  % 60
34      minutes_first_temp = minutes / 10
35      minutes_second_temp = minutes % 10
36  within
37      minutes_out_first_digit ! minutes_first_temp ->
38      minutes_out_second_digit ! minutes_second_temp ->
39      SKIP
40
41  Minutes_out_first_digit_monitor(c) =
42      c ? x ->
43      (0 <= x and x <= 5) & SKIP
44  Minutes_out_second_digit_monitor(c) =
45      c ? x ->
46      (0 <= x and x <= 9) & SKIP
47
48
49  Seconds(seconds_in) =
50  let
51      seconds = seconds_in % 60
52      seconds_first_temp = seconds / 10
53      seconds_second_temp = seconds % 10
54  within
55      seconds_out_first_digit ! seconds_first_temp ->
56      seconds_out_second_digit ! seconds_second_temp ->
57      SKIP
58
59  Seconds_out_first_digit_monitor(c) =
60      c ? x ->
61      (0 <= x and x <= 5) & SKIP
62  Seconds_out_second_digit_monitor(c) =
63      c ? x ->
64      (0 <= x and x <= 9) & SKIP
65
66
67  N_hours = clock_out_val ? variable ->
68            (Hours(variable)
69            [| {| hours_out_first_digit|} |]
70            Hours_out_first_digit_monitor(hours_out_first_digit))
71            [| {| hours_out_second_digit|} |]
72            Hours_out_second_digit_monitor(hours_out_second_digit)
73
74  assert SKIP [F= N_hours \ Events
```

```
75
76
77  N_minutes = clock_out_val ? variable ->
78              (Minutes(variable)
79              [| {| minutes_out_first_digit|} |]
80              Minutes_out_first_digit_monitor(minutes_out_first_digit))
81              [| {| minutes_out_second_digit|} |]
82              Minutes_out_second_digit_monitor(minutes_out_second_digit)
83
84  assert SKIP [F= N_minutes \ Events
85
86
87  N_seconds = clock_out_val ? variable ->
88              (Seconds(variable)
89              [| {| seconds_out_first_digit|} |]
90              Seconds_out_first_digit_monitor(seconds_out_first_digit))
91              [| {| seconds_out_second_digit|} |]
92              Seconds_out_second_digit_monitor(seconds_out_second_digit)
93
94  assert SKIP [F= N_seconds \ Events
```

Listing 46: The full unclocked CSP$_M$ code after transpiling the seven segment display example, as seen in Listing 45 in Appendix B.

Clocked Seven Segment Display Example CSP$_M$ Code

```
1   channel clock_out_val : { 0..1000}
2   channel sync
3
4   channel hours_out_first_digit : {0..3}
5   channel hours_out_second_digit : {0..15}
6
7   channel minutes_out_first_digit : {0..7}
8   channel minutes_out_second_digit : {0..15}
9
10  channel seconds_out_first_digit : {0..7}
11  channel seconds_out_second_digit : {0..15}
12
13
14
15  Clock(1) = SKIP
16  Clock(n) =  sync -> sync -> Clock(n+1)
17
18  Hours(input_channel) =
19      (sync ->
20       input_channel ? hours_in ->
21       sync ->
22          let
23              hours = ( hours_in / 3600 ) % 24
24              hours_first_temp = hours / 10
25              hours_second_temp = hours % 10
26          within
27              (hours_first_temp <= 3) &
28                  (hours_out_first_digit ! hours_first_temp ->
29                  (hours_second_temp <= 15) &
```

```
                    (hours_out_second_digit ! hours_second_temp ->
                    Hours(input_channel)
                    )
                )
    ) [] SKIP

Hours_out_first_digit_monitor(c) =
    (c ? x ->
    (0 <= x and x <= 2 or x == -1) &
        Hours_out_first_digit_monitor(c)
    ) [] SKIP

Hours_out_second_digit_monitor(c) =
    (c ? x ->
    (0 <= x and x <= 9 or x == -1) &
        Hours_out_second_digit_monitor(c)
    ) [] SKIP


N_hours =
        (
            (
                Hours(clock_out_val)
                [|{| hours_out_first_digit |}|]
                Hours_out_first_digit_monitor(hours_out_first_digit)
            )
            [|{| hours_out_second_digit |}|]
            Hours_out_second_digit_monitor(hours_out_second_digit)
        )
        [|{| sync |}|]
        Clock(0)

assert SKIP [F= N_hours \ Events


Minutes(input_channel) =
    (sync ->
     input_channel ? min_in ->
     sync ->
        let
            minutes = (minutes_in / 60)  % 60
            minutes_first_temp = minutes / 10
            minutes_second_temp = minutes % 10
        within
            (minutes_first_temp <= 7) &
                (minutes_out_first_digit ! minutes_first_temp ->
                (minutes_second_temp <= 15) &
                    (minutes_out_second_digit ! minutes_second_temp ->
                    Minutes(input_channel)
                    )
                )
    ) [] SKIP

Minutes_out_first_digit_monitor(c) =
    (c ? x ->
    (0 <= x and x <= 5 or x == -1) &
        Minutes_out_first_digit_monitor(c)
    ) [] SKIP
```

```
88
89  Minutes_out_second_digit_monitor(c) =
90      (c ? x ->
91      (0 <= x and x <= 9 or x == -1) &
92          Minutes_out_second_digit_monitor(c)
93      ) [] SKIP
94
95  N_minutes =
96          (
97              (
98                  Minutes(clock_out_val)
99                  [|{| minutes_out_first_digit |}|]
100                 Minutes_out_first_digit_monitor(minutes_out_first_digit)
101             )
102             [|{| minutes_out_second_digit |}|]
103             Minutes_out_second_digit_monitor(minutes_out_second_digit)
104         )
105         [|{| sync |}|]
106         Clock(0)
107
108 assert SKIP [F= N_minutes \ Events
109
110
111 Seconds(input_channel) =
112     (sync ->
113      input_channel ? sec_in ->
114      sync ->
115         let
116             seconds = seconds_in % 60
117             seconds_first_temp = seconds / 10
118             seconds_second_temp = seconds % 10
119         within
120             (seconds_first_temp <= 7) &
121                 (seconds_out_first_digit ! seconds_first_temp ->
122                 (seconds_second_temp <= 15) &
123                     (seconds_out_second_digit ! seconds_second_temp ->
124                     Seconds(input_channel)
125                     )
126                 )
127     ) [] SKIP
128
129
130 Seconds_out_first_digit_monitor(c) =
131     (c ? x ->
132     (0 <= x and x <= 5) &
133         Seconds_out_first_digit_monitor(c)
134     ) [] SKIP
135
136 Seconds_out_second_digit_monitor(c) =
137     (c ? x ->
138     (0 <= x and x <= 9) &
139         Seconds_out_second_digit_monitor(c)
140     ) [] SKIP
141
142 N_seconds =
143         (
144             (
145                 Seconds(clock_out_val)
```

```
146                 [|{| seconds_out_first_digit |}|]
147                 Seconds_out_first_digit_monitor(seconds_out_first_digit)
148             )
149             [|{| seconds_out_second_digit |}|]
150             Seconds_out_second_digit_monitor(seconds_out_second_digit)
151         )
152         [|{| sync |}|]
153         Clock(0)
154
155 assert SKIP [F= N_seconds \ Events
```

Listing 47: The full clocked CSP$_M$ code after transpiling the seven segment display example, as seen in Listing 45 in Appendix B.This example has been manually translated.

Appendix C

Addone Example Full CSP$_M$ Code

Addone Example CSP$_M$ Code

```
channel sync
channel d_read, c_read : { -1..15} -- u4 and initial value
channel d_write, c_write : { -1..15} -- u4 and initial value

DUM_VAL = -1 -- initial value


Add(i, input_channel) =
    (sync ->
     input_channel ? x ->
     sync ->
        if (x == DUM_VAL) -- initial value
            then (
                let
                    var = i
                within
                    var <= 15 & -- upper limit
                        c_read ! var -> Add(i, input_channel))
            else (
                let
                    var = (x + 1) % 11 -- observed value + 1 restriction
                within
                    var <= 15 & -- upper limit
                        c_read ! var -> Add(i, input_channel))
    )
    [] SKIP


Id(i, input_channel) =
    (sync ->
     input_channel ? x ->
     sync ->
        if (x == DUM_VAL) -- initial value
            then (
                i <= 15 & -- upper limit
                    d_read ! i -> Id(i, input_channel))
            else (
                x <= 15 & -- upper limit
                    d_read ! x -> Id(i, input_channel))
```

```
    )
    [] SKIP


c_read_monitor(c) =
    (c ? x ->
    (0 <= x and x <= 10 or x == -1) & -- observed values + initial value
        c_read_monitor(c)
    ) [] SKIP

d_read_monitor(c) =
    (c ? x ->
    (0 <= x and x <= 10 or x == -1) & -- observed values + initial value
        d_read_monitor(c)
    ) [] SKIP


Buf_d_write(x) = sync -> (Writes_d_write(x) [] Buf_d_read) [] SKIP

Writes_d_write(x) = d_write ! x -> (Writes_d_write(x)
                                    [] Buf_d_read)

Buf_d_read = sync -> ((d_read ? x -> (d_read ? x -> STOP [] Buf_d_write(x))
                    [] sync -> Buf_d_read) [] SKIP)



Buf_c_write(x) = sync -> (Writes_c_write(x) [] Buf_c_read) [] SKIP

Writes_c_write(x) = c_write ! x -> (Writes_c_write(x)
                                    [] Buf_c_read)

Buf_c_read = sync -> ((c_read ? x -> (c_read ? x -> STOP [] Buf_c_write(x))
                    [] sync -> Buf_c_read) [] SKIP)


Clock(21) = SKIP
Clock(n) =  sync -> sync -> Clock(n+1)


System =
        (
            (
                (
                    Add(0, d_write)
                    [{| sync, c_read, d_write |} || {| c_read |}]
                    c_read_monitor(c_read)
                )
                [{| sync, c_read, d_write |} || {| sync, d_read, d_write |}]
                Buf_d_write(DUM_VAL)
            )
            [{| sync, c_read, d_read, d_write |} || {| sync, c_read, c_write, d_read |}]
            (
                (
                    Id(0, c_write)
                    [{| sync, d_read, c_write |} || {| d_read |}]
                    d_read_monitor(d_read)
                )
```

```
98                  [{| sync, c_write, d_read |} || {| sync, c_read, c_write |}]
99                  Buf_c_write(DUM_VAL)
100             )
101         )
102         [|{| sync |}|]
103         Clock(0)
104
105 assert SKIP [F= System \ Events
```

Listing 48: The full CSP$_M$ code after transpiling the Addone example, as seen in Listing 43 in Chapter 9. This example has been manually translated.

Appendix D

Published Paper

A paper, based on this thesis, has been published as

> A. Thegler, M. Larsen, K. Skovhede, and B. Vinter. Towards Automatic Program Specification Using SME Models. In: In K. Chalmers, J. Pedersen, M. Smith, K. Skovhede, and P. Welch, editors, *Proceedings of Communicating Process Architectures 2018*. IOS Press, Amsterdam, The Netherlands, August 2018.

The following pages contain this paper.

Communicating Process Architectures 2018
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2018

Towards Automatic Program Specification Using SME Models

Alberte THEGLER [1], Mads Ohm LARSEN, Kenneth SKOVHEDE, and Brian VINTER

Niels Bohr Institute, University of Copenhagen, Denmark

Abstract. This paper introduces a method to simplify hardware modeling and verification thereof in order for software programmers to, more easily, meet the demands of the growing embedded device industry. We describe a simple method for transpiling from the new SME Implementation Language into CSP$_M$ and using formal verification to verify properties within the generated program. We present a small example consisting of a seven segment display clock network and introduce how to verify the widths of the channels in the network.

Keywords. CSP$_M$, SME, transpiling

Introduction

The Internet of Things, computerized medical implants, and the omnipresent growth in robotics, brings with them an increased demand for programmers to develop software for those devices. While this observation may not in itself appear to present a new challenge, many other areas have previously presented a need for more programmers. The new challenge is that these new growth areas are all focused on small size, low power consumption, and high reliability. This means that traditional software engineering methods, and thus traditionally trained programmers, are often not sufficiently qualified to develop these technologies. In previous decades such systems have been developed by electronic engineers that apply far more rigid development approaches. Especially for hardware solutions like VLSI[2] and FPGA[3], correctness has always been favored over productivity. While tools have obviously improved and methods refined, the VLSI process is still mostly the same as presented in [1]. The primary workflow from [1] is shown in Figure 1; note the focus on verification in each step.

While the VLSI community is fundamentally following this 1980's design approach, more high-level tools and abstractions have been introduced. Philippe et al. [2] show a workflow (reproduced in Figure 2) where the important part is the verification that has been partly automated by basing the development on a formal specification of the solution.

There is no denying that the subjectively slow and rigid development process in the VLSI world [3] is highly successful in producing correct and reliable circuits. At the same time, conventional software development is highly focused on productivity and time-to-market, for example, smartphone applications are often developed for continuous release, where bug patches and new features are rolled out daily. This is of course not possible with hardware.

[1]Corresponding Author: *Alberte Thegler, Blegdamsvej 17, 2100 Copenhagen OE*. E-mail: tpq587@alumni.ku.dk.

[2]Very-large-scale integration.

[3]Field-Programmable Gate Array.

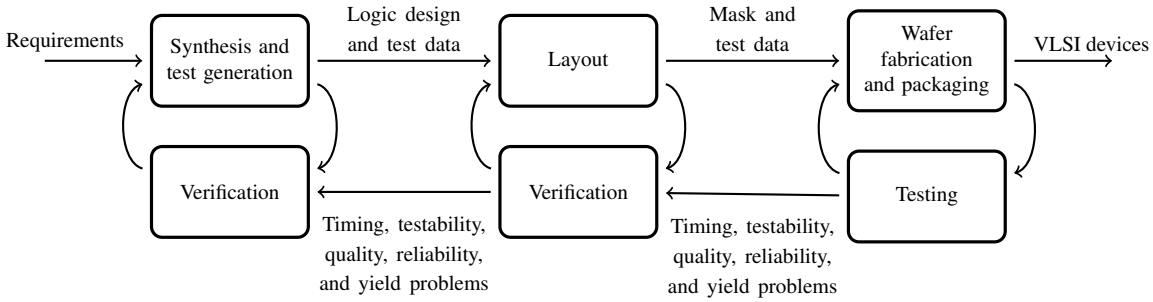

Figure 1. VLSI process workflow.

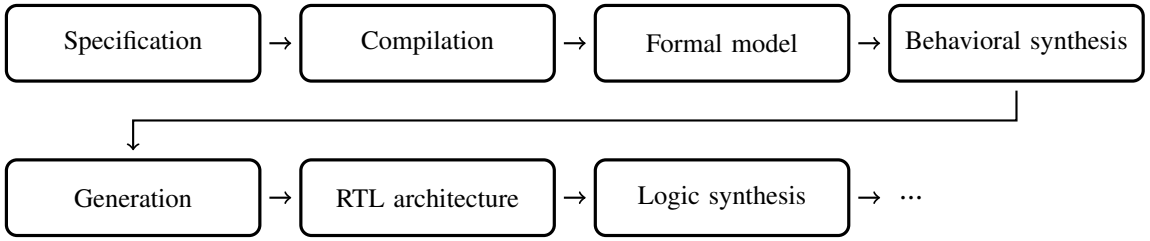

Figure 2. Reproduced workflow from Philippe et al. [2].

Thus, the authors argue that there is a growing chasm between the way most programmers are trained and the competencies that are needed to support the growth in mission critical embedded devices.

In this work, we propose a tool to help bridge the gap between available programmer profiles and the required competencies for embedded devices. Our approach is based on building a specification from a software implementation and test-suite observations. The overarching goal is to reach a level where a conventional software programmer can write a solution in Synchronous Message Exchange (SME) [4,5], and develop a conventional test suite in the software engineering tradition. By combining the implementation with the *observed* values of internal states in an SME based system implementation, we can produce a formal specification of the system. This specification can be fed into a formal verification tool and thus improve the correctness guarantees from only what is covered by the individual test vectors to the entire space that is spawned by the set of test vectors. We approach the task by transpiling[4] the new SME Implementation Language (SMEIL) [6] for SME into CSP$_M$ [7] and verify the formal properties of this version with a tool like FDR4 [8].

This paper builds on the SME model, which have been covered in papers [4,5,9]. In this paper we only include a brief description of the elements required to understand the setup we have developed, and encourage readers to seek out more information in the mentioned papers.

[4]Source-to-source compile.

1. Background

1.1. Synchronous Message Exchange

SMEIL is based on the SME model and therefore we give a brief introduction to SME.

SME was first introduced in 2014 and after several iterations [4,5,9] now presents as a programming model, a simulation library, and VHDL code generators [10]. The original idea was conceived following an attempt to create hardware descriptions from a vector processor model, modeled in PyCSP [11], a Communicating Sequential Processes (CSP) [12] library for Python. After this attempt, it became clear that the structure of CSP was poorly suited for modeling clocked systems, and therefore it was decided to create the SME model, based on the CSP algebra. The idea was to only use the subset of the CSP algebra that provided beneficial functionality to hardware modeling which, most importantly, meant that external choice was omitted. However, the shared-nothing property of CSP showed to be very useful, since the network state could only be changed by process communication.

In SME, a network is a combination of processes that are connected through buses. The processes communicate through a collection of signals in a bus, instead of CSP's synchronous rendezvous model, but retains the shared-nothing trait of CSP. SME uses the term `bus` instead of `channel` to enforce the semantic correlation between the SME bus and a physical hardware signal bus. The process communication is handled by a hidden clock which eliminates the complexity that arose from adding synchronicity to a CSP network. The combination of the hidden clock and the synchronous message passing between processes means that the SME model provides hardware-like signal propagation.

An SME clock cycle consists of three phases: it reads, executes, and writes as can be seen in Figure 3. The process is activated on the rising clock edge where it reads from the bus and it reads, executes and writes to the bus in one clock cycle. Just before the rising edge of the clock, all signals are propagated on all buses which means, that all communication happens simultaneously. Because of this structure, if a value is written by a process in cycle i, it is read by the receiving process in cycle $i + 1$.

SME is able to detect read/write conflicts where multiple writes are performed to a single bus within the same clock cycle as well as reads from a signal that has not been written to in the previous clock-cycle.

Since SME is based on CSP, all SME models have a corresponding CSP model, and because

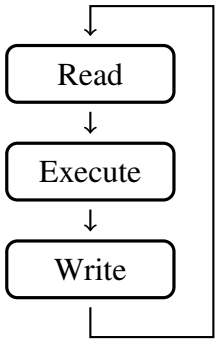

Figure 3. SME process flow for one clock cycle.

of this property, we are able to create a transpiler translating SME models to CSP_M. The SME model is currently implemented as libraries for the general-purpose languages C# [9], C++ [13], and Python [14]. The Python and C# libraries both have code generators for VHDL as well.

```
proc addone (in inbus)
    bus outbus {
        val: int;
    };
{
    outbus.val = inbus.val + 1;
}

    ⋮

network net() {
    instance a of addone(b.outbus);
    instance b of ..
    ⋮
}
```

Listing 1. Small example of process and network syntax in SMEIL.

1.2. SMEIL

With the different SME implementations, a need arose for a common intermediate language. SMEIL was developed as a Domain Specific Language (DSL) for SME, usable both as an IL and as an independent implementation language. It has a C-like syntax with a type system that makes hardware modeling simple. In spite of its simplicity, SMEIL still provides hardware-specific functionality that is more difficult to create with general-purpose languages. Often when modeling hardware in Hardware Description Languages (HDLs) like VHDL or Verilog, code for testing and verifying are often written in the same language as the design itself. Unfortunately, the HDLs often does not have the functionality for generating proper simulation input. Using general-purpose languages for testing hardware models are useful since the range of available libraries are much larger. Therefore the SMEIL simulator provides a simple language-independent API which enables SME implementations written for general-purpose languages to communicate with SME networks written in SMEIL, so-called co-simulation.

The two fundamental components of an SMEIL program is `process` and `network`. The process consists of variable and bus definitions, as well as the statements that are evaluated once for each clock cycle. The purpose of the `network` declaration is to define the relations between each entity in the program. A small example of process and network syntax can be seen in Listing 1.

There are several different ways to use SMEIL, one being co-simulation as described above. However, in this work, we focus on the independent SMEIL representation and thus we only present examples in pure SMEIL. These pure SMEIL programs must contain a process which generates input for the network since the network cannot receive input elsewhere. The program is simulated using the command line tool. Simulation is done in order to test the design of the system.

During the simulation, ranges for all observed values are captured so the observed values and types can be used to constrain the original defined types and ranges. This property is of great value when translating into CSP_M, and when creating assertions, since we can use these values to actually assert the network. The number of clock cycles, that the simulation is run for, is specified by the programmer via the command line tool. If the simulation is not passing through enough clock cycles, the verification might be inadequate. Since the verification builds on the observed values, the simulation needs to be long enough such that the whole possible range of input values is exhausted.

In Figure 4 the SMEIL transpiler structure can be seen.

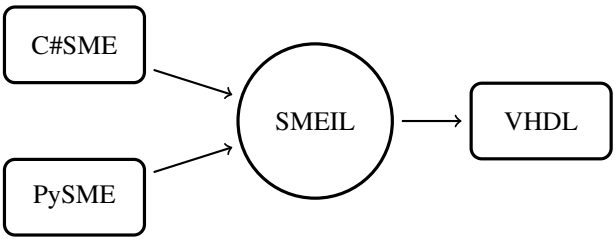

Figure 4. SMEIL transpiler structure.

2. Seven Segment Display Clock in SMEIL

In order to explain how we can transpile programs from SMEIL to CSP$_M$, we have designed an example using a seven segment display clock. In this section, the seven segment display example will be explained as well as the SMEIL implementation of the network.

A seven segment display is an electronic display device which is used in displays such as digital clocks or other types of devices that display numerals. An example of a typical digital clock display can be seen in Figure 5. When a digit has been determined for a seven segment display, it is encoded to a bitstream that represents the digit in the correctly activated display segments. In this example, we wish to model a typical digital clock that is able to calculate

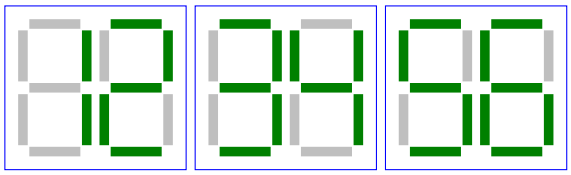

Figure 5. Digital clock with six seven segment displays, displaying 12:34:56.

and display the current time in hours, minutes, and seconds. Listing 2 shows this example written in Python. When creating this model in SMEIL some input must be added to the network, just like `time_since_midnight` in Listing 2. The input value represents seconds since midnight, and in order to calculate hours, minutes, and seconds we model three different processes, called the `time` processes in this example.

When writing hardware models in pure SMEIL, the only way to generate input for the network is to create a data generator process. This process, called the `clock` process in our example, is instantiated with the start time and is incremented by 1 for each simulation cycle, representing a one second increase. The result is communicated on the process output bus, where the three `time` processes are listening. These `time` processes receive the number and by the use of simple integer arithmetic, calculate the hours, minutes, and seconds since midnight respectively. It is obvious that at some point in time, each `time` process will calculate a two-digit result, for example at 12 hours or 42 seconds. However, a single seven segment display can only show one digit between 0 and 9. Therefore we need two seven segment displays for each `time` process in order to show the correct time in a 24-hour interval. Each `time` process has an output bus with two individual channels that represent the communication to each different display. The number representing either hours, minutes, or seconds are separated into first and second digit, by $\lfloor \frac{x}{10} \rfloor$ and $(x \bmod 10)$. These six different results are then communicated onto the six different channels which represent the six different seven segment displays. The outline of this network can be seen in Figure 6.

In Figure 6 the network consists of four processes, the data generator process, *I*, which creates the input that is broadcasted out on the network. The three `time` processes, hours (*H*),

```python
from math import floor

def time(time_since_midnight):
    hours   = floor(time_since_midnight / 3600)
    minutes = floor((time_since_midnight - hours * 3600) / 60)
    seconds = time_since_midnight - hours * 3600 - minutes * 60
    return [hours, minutes, seconds]

print(time( 57100)) # =>  15:51:40
print(time(  3601)) # =>  01:00:01
print(time( 66666)) # =>  18:31:06
```

Listing 2. A Python implementation of the seven segment display example.

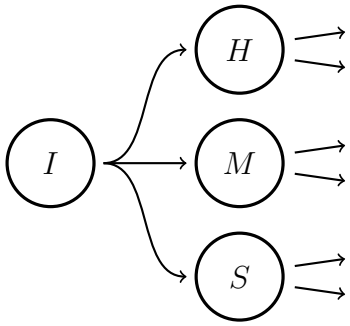

Figure 6. SMEIL network for a seven segment display clock. Each SMEIL process is represented by a cicle with a letter corresponding to the processes Input, Hours, Minutes and Seconds respectively.

minutes (M), and seconds (S) are the processes described above, which calculate each part of the current time. The outputs are communicated on the six outgoing channels.

The full SMEIL code for this example can be seen in Listing 7 in the appendix.

3. Supporting Technologies

3.1. FDR4

We not only want to transpile SMEIL to CSP$_M$, we also want to be able to verify different properties in CSP$_M$ in order to prove correctness. Today, there exists several tools for formal verification, both in academia and in the industry. One of the currently most favored tools is the Failures-Divergences Refinement tool (FDR4). This tool is a CSP refinement checker that can analyze programs written in the machine-readable version of CSP; CSP$_M$. It provides a parallel refinement-checking engine that can scale up linearly with the number of cores. This means that it can handle processes with a large number of states in a reasonable time. FDR4 can handle several different types of assertions, deadlocks being the most used. However, due to the structure of SMEIL, we use FDR4 in a different way than is typical. Since the SME model cannot have cyclic-wait we have no need to verify the system in this manner.

For our current implementation of the transpiler, we can assert the ranges of the channel inputs, for example, we can automatically assert that the observed ranges, provided by the SMEIL simulation, and the possible input on the CSP$_M$ channels are not conflicting. In hardware, we would typically want to verify that the communication on a bus does not exceed a certain range or that the sum of multiple signals does not exceed a specific value. A bus might be able to carry other data than needed, and being able to model a circuit that can assert that the bus never carries other data than expected, is of great value.

CSP was not initially developed for hardware modeling, and therefore it is not evident how to handle the clock cycle, which is an essential part of hardware modeling. When we transpile the SME network into CSP_M the SMEIL simulation have provided the ranges of all values from the simulation and therefore all clock cycles. This means that when FDR4 asserts a property it asserts on all possible communication combinations for all the simulated clock cycles. Therefore, even though we are transpiling from an SME model, where the clock is crucial, we can simply translate "one-to-one" from the SMEIL program and still get an accurate assertion on the properties.

3.2. Transpiling SMEIL to CSP_M

When transpiling from SMEIL to CSP_M one of the difficult components was to find a generalized method for transpiling, that could be generalized to most problems. We have worked on separation of concerns in order to simplify, but also have a greater chance of being able to match more SMEIL programs.

An SMEIL process consists of bus and variable declarations, the statements to be run per clock cycle as well as the outgoing communication from the process. Channels within an SMEIL bus can be translated directly to CSP_M channels. It is, however, important to give channel names that will be unique since a CSP_M channel is global as opposed to the local channel within each SMEIL bus. An example of an SMEIL process, where the process structure is evident, can be seen in Listing 3 and the corresponding CSP_M code in Listing 4.

In order to keep the outwards communication and the arithmetic statements together within each process in CSP_M, we generate CSP_M processes with a `let within` statement. The arithmetic statements go into the `let` section and the communications go into the `within` section. This gives us the possibility of separating the outwards communication and arithmetic statements while still keeping them within the same CSP_M process. In Listing 4, an example of the `let within` statement can be seen in lines 7-14. This structure will work as a general translation structure from SMEIL processes to CSP_M processes.

The network in an SMEIL program is the crucial part which ties all the processes and communication together. We can standardize the network generation by creating a two-step communication part. Instead of having the actual processes receive the incoming data, they receive the data by their process parameter. The process parameter is then set by the network process which receives the communication from the channels and provides the process with the communicated value. This ensures that we can generate the processes easily without having to traverse the network in the SMEIL program beforehand to find out which channel provides input for which process. An example of this is shown in Listing 8 in the appendix on lines 61 to 66.

4. Seven Segment Display Clock Transpiling

In the following we use a classic hardware design to illustrate each of the steps in the transpiling, and how the types, constraints, and assertions are carried from the original SMEIL program into the CSP_M program.

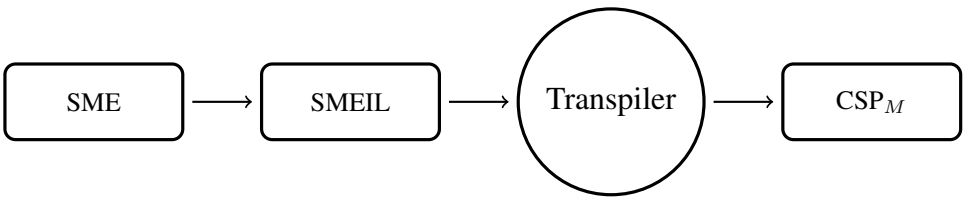

Figure 7. SME to CSP_M transpiler.

We wish to model the network presented in Section 2 in SMEIL in order to transpile it to CSP_M so that we may verify properties in FDR4. In Figure 7 the workflow of this system can be seen.

Even though SME buses can contain a series of channels, every single channel is translated into a CSP_M channel. The properties we will assert with FDR4, are the width of the CSP_M channels. That is, we want to prove that certain values will never be communicated on certain channels. It is easy to imagine that 4 bits can be communicated between the `time` processes and the seven segment displays. But 4 bits can represent the numbers 0 through 15, and our seven segment displays can only display the numbers 0 through 9. Therefore we wish to assert that even though the channels can carry 4 bits, the actual communication on the six output channels does not exceed 9. In general, the displays will be able to display 0 through 9, but since the example is a clock showing a 24-hour interval, the displays will of course not be able to show minutes and seconds above 59 and hours above 23.

We know that a program in pure SMEIL must have a data generation process, but this is not the case in a CSP network. Since we are only transpiling from pure SMEIL networks, we can be certain that there will always be a process which just contributes an initial value to the rest of the network. We also know that a process must either have communication in or out or both. Therefore, we can assume that all SMEIL processes with no input bus will be a data generator process of some kind, and therefore must have some outwards communication. So when transpiling to CSP_M, we do not translate the SMEIL process to a CSP_M process, but simply create a CSP_M channel that represents the values communicated out of this SMEIL process.

We assume that the SMEIL programs we transpile only contains channels with types and range annotations. During the simulation, the type will be restricted to the lowest representation possible. For example, if a channel was originally set to be `int` (unbounded), but the observed values from the simulation show that it could be changed to an `i8` (signed 8-bit integer with a range of -128 to 127), then the simulated output would be `i8`.

When creating channels in CSP_M, we need to define its range of possible values. If a channel is only defined by having the integer type, FDR4 would try to verify for all possible integers, which results in a seemingly unbounded runtime. As explained in Section 1.2, all simulated SMEIL programs will include the observed range and restricted types for all channels and variables. The types represent the observed width of the channels in bits, and by calculating the possible range from these types, we can create the corresponding channels in CSP_M, and thereby avoid having a seemingly endless runtime in FDR4.

Since the assertion we wish to make is to verify the widths of the channels, it might seem redundant to create CSP_M channels with a limited range. FDR4 would always only check the values in the defined channel range and therefore there is no point in asserting if the values go beyond this range. After simulating the SME network, SMEIL provides us with both a type and a range of observed values. The type is used to create the CSP_M channel range and the observed values are used for the assertion. The type will always represent equal or more values than the range of observed values, and by using these values the assertions becomes valuable.

When it comes to transpiling the data generator process into a CSP_M channel, we also use the types of the SMEIL simulation to define it. We use this instead of the observed values because we cannot guarantee the precise input values of the system. If we used the observed values, the assertions will pass every time, since it will test the values already used to generate the rest of the observed values.

An example of simulated SMEIL code can be seen in Listing 3. Notice on lines 2 and 3 that the two channels are defined both with a type `u3` and `u4` and with a range 0 to 5 and

```
1  proc seconds (in seconds_in)
2      bus seconds_out {first_digit: u3 range 0 to 5;
3                       second_digit: u4 range 0 to 9;};
4      var seconds: u6 range 1 to 59;
5      var seconds_first_temp: u3 range 0 to 5;
6      var seconds_second_temp: u4 range 0 to 9;
7  {
8      seconds = seconds_in.val % 60;
9      seconds_first_temp = seconds / 10;
10     seconds_second_temp = seconds % 10;
11     seconds_out.first_digit = seconds_first_temp;
12     seconds_out.second_digit = seconds_second_temp;
13 }
```

Listing 3. Example of the `seconds` process from the SMEIL seven segment display example. See full example in Listing 7 in the appendix.

```
1  channel seconds_out_first_digit : {0..7}
2  channel seconds_out_second_digit : {0..15}
3
4      ⋮
5
6  Seconds(seconds_in) =
7  let
8      seconds = seconds_in % 60
9      seconds_first_temp = seconds / 10
10     seconds_second_temp = seconds % 10
11 within
12     seconds_out_first_digit ! seconds_first_temp ->
13     seconds_out_second_digit ! seconds_second_temp ->
14     SKIP
```

Listing 4. Example of the `Seconds` process from the generated CSP_M code in the seven segment display example. See full example in Listing 8 in the appendix.

0 to 9. These are the observed types and value ranges the simulation tracked for the specific channel. In order to create the CSP_M channels based on the types, we need to convert `u3` and `u4` into its corresponding range, which for `u3` is 0 through 7 and for `u4` is 0 through 15. In Listing 4 on lines 1 and 2, the calculated ranges are used to define the CSP_M channels.

When creating the assertions, we decided to create separate assert functions to keep the code structure clean. We know that for each CSP_M channel there must be an assertion, except for the input channel. Consequently, we create a *monitor* process for each channel and its only job is to listen in on the channel communication and assert the values communicated there. The monitor process is a process that we add specifically for asserting legal communication values in FDR4 and it does not affect the original SME network. In Figure 8 the outline of this kind of structure can be seen and we expect that this structure can be used for several different types of problems and thereby ensure a cleaner code structure.

The monitor process asserts the observed values of the CSP_M channels and in Listing 5 the two monitor processes for the Seconds `time` process can be seen. The values used for these statements are the observed values from the SMEIL simulation, as can be seen at the end of lines 2 and 3 in Listing 3. In Listing 5 the ranges are used to assert that the only values communicated on the channels are within 0 and 5, and 0 and 9 respectively.

After translating the SMEIL processes and creating the monitor processes, we need to

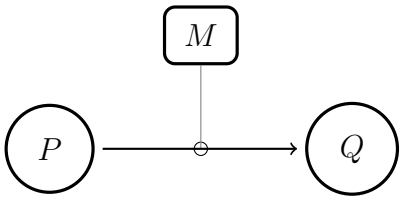

Figure 8. The monitor process M listens in on the communication between P and Q in order to assert the communicated values.

```
1 Seconds_out_first_digit_monitor(c) =
2     c ? x -> if 0 <= x and x <= 5 then SKIP else STOP
3 Seconds_out_second_digit_monitor(c) =
4     c ? x -> if 0 <= x and x <= 9 then SKIP else STOP
```

Listing 5. Example of the `Seconds` monitor processes from the generated CSP$_M$ code in the seven segment display example. See full example in Listing 8 in the appendix.

create the network described in the last part of the SMEIL program, see lines 53 to 59 in Listing 7 in the appendix. We wish only to assert the values the `time` processes are communicating to the monitor processes, and therefore we have to synchronize these processes into a single network in CSP$_M$. We create three network processes, one for each part of the network, and we create a nested synchronization, in order to have all monitor processes synchronized with the `time` process. An example of this network can be seen in on lines 61 to 66 in Listing 8 in the appendix. This network process is also the process that receives the input from the input channel. By not adding the receiving communication in the `time` processes, we avoid having to specify the name of the input channels before creating the network which simplifies the translation, as described in Section 3.2. In SMEIL, this information is part of the `network` section, and therefore it fits well within this part of the CSP$_M$ code.

After creating the network we add the actual assert function calls. For these kinds of assertions, where we want to check a range, the best solution is to assert that the network processes behave as the `SKIP` process. This is done by having the monitor process running the `SKIP` process if the value is within the range and the `STOP` process if not. Two examples can be seen in lines 2 and 4 in Listing 5. We assert this by using the FDR4 failures model on the the `SKIP` process along with hiding communication events, which can be seen in lines 68, 78 and 88 in Listing 8 in the appendix.

The different parts of transpiling the seven segment display example have been presented and in Figure 9 the corresponding network of the CSP$_M$ system is presented. The corresponding network in CSP$_M$ consists of 12 different processes, all created so that not only the network is simulated correctly, but also so the assertions we wish to make, are in place. The input is represented by a triangle, since it transpiles from an SME process to a CSP$_M$ channel. Each of the dotted squares represents the network of synchronizations for each `time` processes, which in itself is a process in CSP$_M$. For each network, we have the `time` processes and two monitor processes, for example, H, M_{H_1} and M_{H_2}.

In order to show that the verification is accurate, the example in Listing 6 contains an error that results in FDR4 failing the verification. In Listing 6 the example is only able to handle an input that is below 24 hours. This is because the calculation in the `Hours` process does not handle the wrap around at the 24$^{\text{th}}$ hour. This means that if the input represents more than 24 hours, the assertions will fail in FDR4 because one seven segment display suddenly has to display two digits instead of one. An example of such could be the input `131071`, which

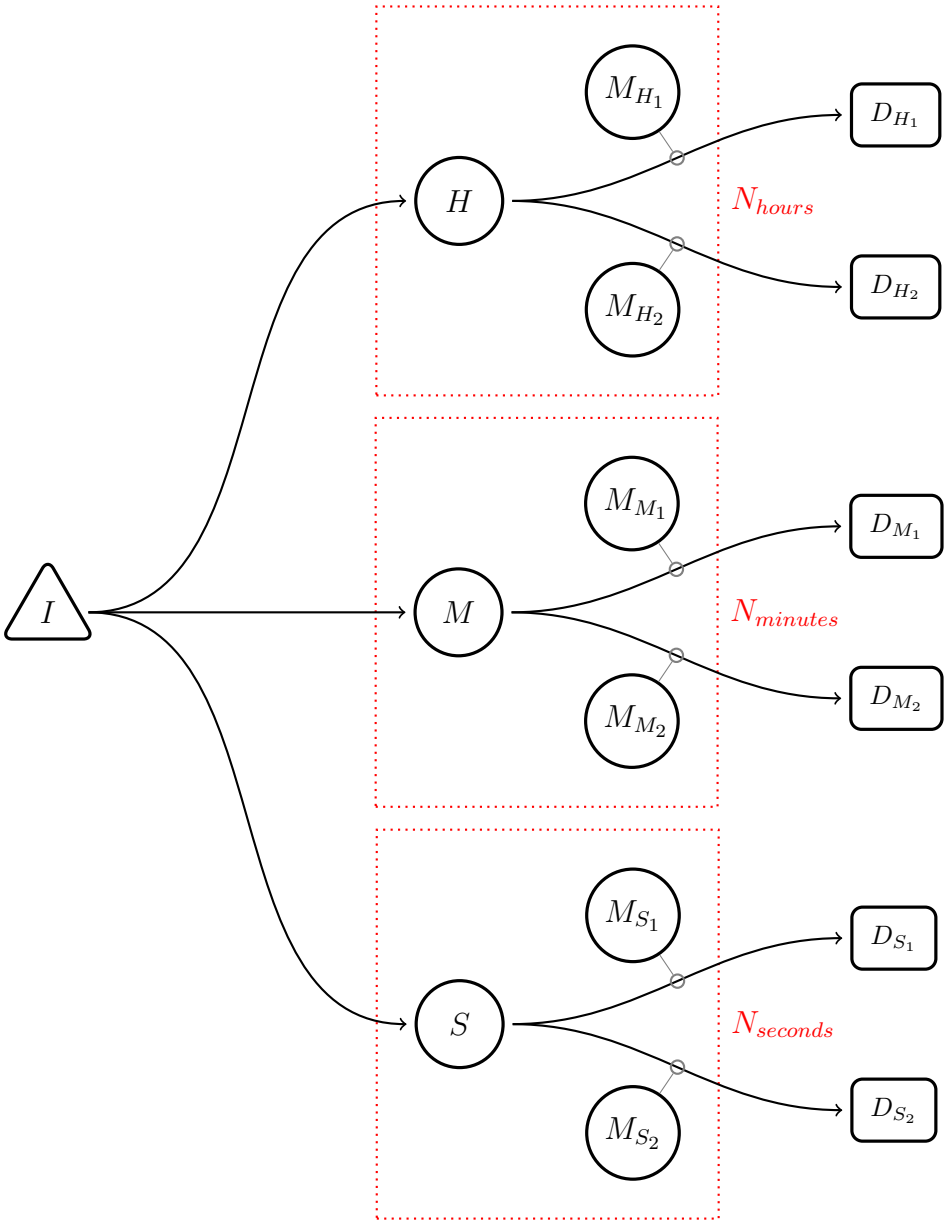

Figure 9. A seven segment display clock network in CSP$_M$. I represents the input channel. N_{hours}, $N_{minutes}$ and $N_{seconds}$ represent the network processes with H, M and S as the `time` processes. The results from the `time` processes are communicated to the displays. The displays are represented by a square since they are not actual CSP$_M$ processes. Each display communication also has a monitor process which assert the legal communication values.

represents 36 hours, 24 minutes and 31 seconds, or 1 day, 12 hours, 24 minutes and 31 seconds. When trying to assert the code from Listing 6 in FDR4, the assertion fails. The counterexample shows that the number 3 is communicated on `hours_out_first_digit`, which is not allowed according to the monitor process on lines 12 and 13 in Listing 6.

This example of failure shows how verifying the solution with a tool like FDR4 actually catches errors that the programmer might have overseen. In this case, the error is simply corrected by adding `% 24` on the end of line 9 in Listing 6 and can be seen corrected in Listing 8 in the appendix at line 15. Now when we try to assert the example in FDR4, it passes. By using modulo on the result, we ensure that we still get the accurate time of day, no matter how many full days the input represents.

The full SMEIL and CSP$_M$ code for the seven segment display example can be seen in Listing 7 and in Listing 8 in the appendix.

```
channel clock_out_val : {0..131071}

channel hours_out_first_digit : {0..3}
channel hours_out_second_digit : {0..15}
    ⋮

Hours(hours_in) =
let
    hours = hours_in / 3600
    ⋮

Hours_out_first_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 2 then SKIP else STOP
Hours_out_second_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 9 then SKIP else STOP
```

Listing 6. Example of an erroneous version of the `Hours` process from the CSP_M seven segment display example seen in Listing 7 and in Listing 8 in the appendix.

5. Future Work

With this work, we have taken a small step towards creating a simpler method for software developers to model hardware as well as verify properties within this model. In future work, we would like to extend this to software-hardware co-design, with which we would be able to assert deadlocks.

It would be desirable to be able to automatically create a human-readable report on the ranges and communications that are used within the system. This could become a standard addition to the documentation of the system, which would give a programmer an easy overview of a complicated system and would also allow for easier contemplation over the system.

Another, more complex idea for future work, is to implement support for multi-channel invariants. This is not something that can easily be simulated and therefore it would require some work, but it would provide the ability to express more complex assertions.

6. Conclusions

We have presented a transpiler that transpiles SME intermediate language (SMEIL) into CSP_M for then to use the Failure-Divergences Refinement tool (FDR4) to assert properties in a CSP_M network. We provide a simple approach that makes it more accessible for software programmers to program hardware and thereby bridging a gap between software programmers and the needs of the industry. Instead of having to create advanced test-benches, our tool provides a simple way to verify the hardware model via FDR4s assertion functionalities. We can assert that the observed values of a channel, in a simulated SMEIL program, are in fact the only possible values communicated on that specific channel. We have also shown this to work in an example case of a seven segment display.

Acknowledgements

Thanks to Uwe Zimmermann who made the seven segment example in TikZ on `http://www.texample.net/tikz/examples/segment-display/`.

References

[1] Vishwani Agrawal and Samuel H. C. Poon. VLSI Design Process. In *Proceedings of the 1985 ACM Thirteenth Annual Conference on Computer Science*, CSC '85, pages 74–78, New York, NY, USA, 1985. ACM.

[2] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[3] Jeremy Kepner. Hpc productivity: An overarching view. *The International Journal of High Performance Computing Applications*, 18(4):393–397, 2004.

[4] Brian Vinter and Kenneth Skovhede. Synchronous message exchange for hardware designs. *Communicating Process Architectures*, pages 201–212, 2014.

[5] Brian Vinter and Kenneth Skovhede. Bus centric synchronous message exchange for hardware designs. *Communicating Process Architectures*, pages 245–257, 2015.

[6] Truls Asheim. SMEIL: A Domain-Specific Language for Synchronous Message Exchange Networks. To be published, 2018.

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

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

[9] Kenneth Skovhede and Brian Vinter. Building hardware from C# models. In *FSP 2016; Third International Workshop on FPGAs for Software Programmers; Proceedings of*, pages 1–9. VDE, 2016.

[10] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-1987*, 1998.

[11] John Markus Bjørndalen, Brian Vinter, and Otto J Anshus. PyCSP-Communicating Sequential Processes for Python. In *Cpa*, pages 229–248, 2007.

[12] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[13] Truls Asheim. Implementing high performance synchronous message exchange, 2015. Bachelor's Thesis.

[14] Truls Asheim, Kenneth Skovhede, and Brian Vinter. VHDL Generation From Python Synchronous Message Exchange Networks. *Proceedings of Communicating Process Architectures 2016*, 2016.

Full SMEIL and CSP$_M$ code

```
proc clock ()
    bus clock_out {val: u17 range 1 to 86401;};
    var i: u17 = 0 range 0 to 86401;
{
    i = i + 1;
    clock_out.val = i;
}

proc hours (in hours_in)
    bus hours_out {first_digit: u2 range 0 to 2;
                   second_digit: u4 range 0 to 9;};
    var hours: u5 range 0 to 23;
    var hours_first_temp: u2 range 0 to 2;
    var hours_second_temp: u4 range 0 to 9;
{
    hours = hours_in.val / 3600 % 24;
    hours_first_temp = hours / 10;
    hours_second_temp = hours % 10;
    hours_out.first_digit = hours_first_temp;
    hours_out.second_digit = hours_second_temp;
}

proc minutes (in minutes_in)
    bus minutes_out {first_digit: u3 range 0 to 5;
                     second_digit: u4 range 0 to 9;};
    var minutes: u6 range 0 to 59;
    var minutes_first_temp: u3 range 0 to 5;
    var minutes_second_temp: u4 range 0 to 9;

{
    minutes = minutes_in.val / 60 % 60;
    minutes_first_temp = minutes / 10;
    minutes_second_temp = minutes % 10;
    minutes_out.first_digit = minutes_first_temp;
    minutes_out.second_digit = minutes_second_temp;
}


proc seconds (in seconds_in)
    bus seconds_out {first_digit: u3 range 0 to 5;
                     second_digit: u4 range 0 to 9;};
    var seconds: u6 range 0 to 59;
    var seconds_first_temp: u3 range 0 to 5;
    var seconds_second_temp: u4 range 0 to 9;
{
    seconds = seconds_in.val % 60;
    seconds_first_temp = seconds / 10;
    seconds_second_temp = seconds % 10;
    seconds_out.first_digit = seconds_first_temp;
    seconds_out.second_digit = seconds_second_temp;
}

network clock_network ()
{
    instance g of clock();
    instance h of hours(g.clock_out);
```

```
    instance m of minutes(g.clock_out);
    instance s of seconds(g.clock_out);
}
```

Listing 7. The full SMEIL code used for transpiling in the seven segment display example.

```
channel clock_out_val : {0..131071}

channel hours_out_first_digit : {0..3}
channel hours_out_second_digit : {0..15}

channel minutes_out_first_digit : {0..7}
channel minutes_out_second_digit : {0..15}

channel seconds_out_first_digit : {0..7}
channel seconds_out_second_digit : {0..15}


Hours(hours_in) =
let
    hours = hours_in / 3600  % 24
    hours_first_temp = hours / 10
    hours_second_temp = hours % 10
within
    hours_out_first_digit ! hours_first_temp ->
    hours_out_second_digit ! hours_second_temp ->
    SKIP

Hours_out_first_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 2 then SKIP else STOP
Hours_out_second_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 9 then SKIP else STOP


Minutes(minutes_in) =
let
    minutes = minutes_in / 60  % 60
    minutes_first_temp = minutes / 10
    minutes_second_temp = minutes % 10
within
    minutes_out_first_digit ! minutes_first_temp ->
    minutes_out_second_digit ! minutes_second_temp ->
    SKIP

Minutes_out_first_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 5 then SKIP else STOP
Minutes_out_second_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 9 then SKIP else STOP


Seconds(seconds_in) =
let
    seconds = seconds_in % 60
    seconds_first_temp = seconds / 10
    seconds_second_temp = seconds % 10
within
    seconds_out_first_digit ! seconds_first_temp ->
```

```
52      seconds_out_second_digit ! seconds_second_temp ->
53      SKIP
54
55  Seconds_out_first_digit_monitor(c) =
56      c ? x -> if 0 <= x and x <= 5 then SKIP else STOP
57  Seconds_out_second_digit_monitor(c) =
58      c ? x -> if 0 <= x and x <= 9 then SKIP else STOP
59
60
61  N_hours = clock_out_val ? variable ->
62            (Hours(variable)
63            [| {| hours_out_first_digit|} |]
64            Hours_out_first_digit_monitor(hours_out_first_digit))
65            [| {| hours_out_second_digit|} |]
66            Hours_out_second_digit_monitor(hours_out_second_digit)
67
68  assert SKIP [F= N_hours \ Events
69
70
71  N_minutes = clock_out_val ? variable ->
72              (Minutes(variable)
73              [| {| minutes_out_first_digit|} |]
74              Minutes_out_first_digit_monitor(minutes_out_first_digit))
75              [| {| minutes_out_second_digit|} |]
76              Minutes_out_second_digit_monitor(minutes_out_second_digit)
77
78  assert SKIP [F= N_minutes \ Events
79
80
81  N_seconds = clock_out_val ? variable ->
82              (Seconds(variable)
83              [| {| seconds_out_first_digit|} |]
84              Seconds_out_first_digit_monitor(seconds_out_first_digit))
85              [| {| seconds_out_second_digit|} |]
86              Seconds_out_second_digit_monitor(seconds_out_second_digit)
87
88  assert SKIP [F= N_seconds \ Events
```

Listing 8. The full CSP$_M$ code after transpiling the seven segment display example, as seen in Listing 7 in the appendix.