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.

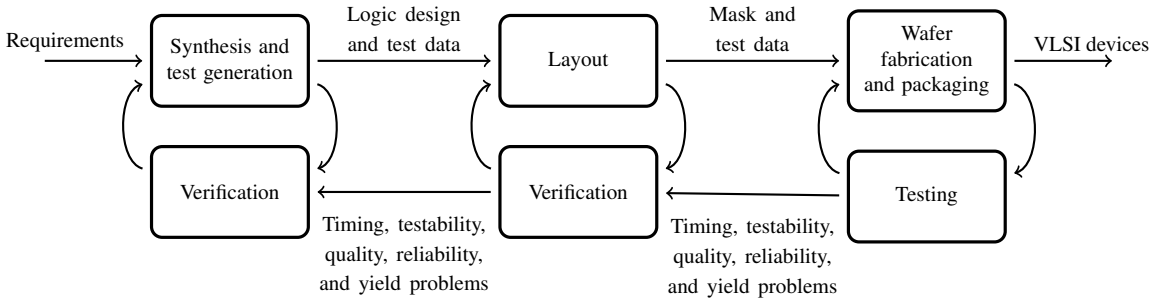

Figure 1. VLSI process workflow.

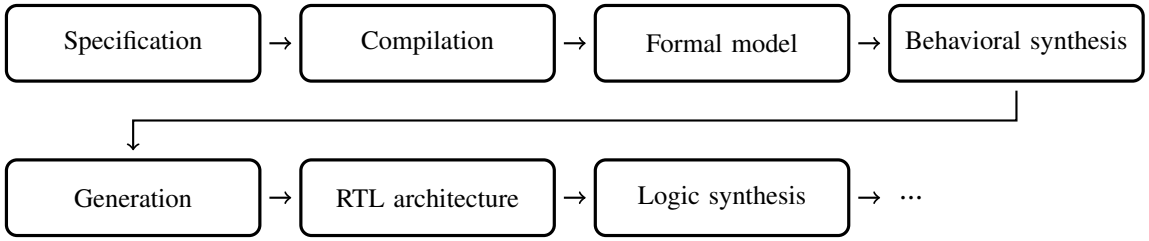

Figure 2. Reproduced workflow from Philippe et al. [2].

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

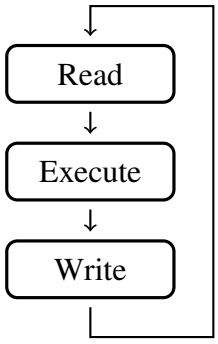

Figure 3. SME process flow for one clock cycle.

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.

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.

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

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

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.

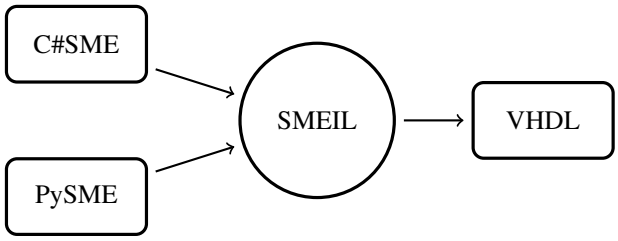

Figure 4. SMEIL transpiler structure.

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

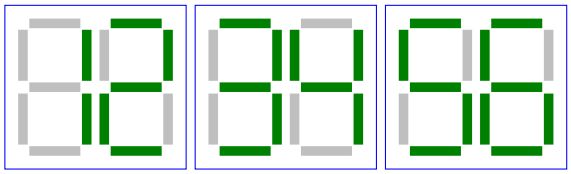

Figure 5. Digital clock with six seven segment displays, displaying 12:34:56.

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.

```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.

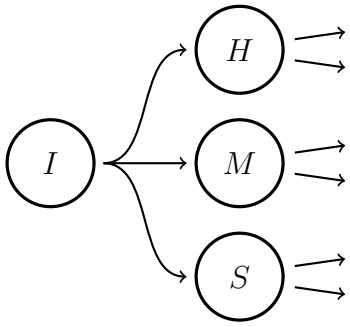

Figure 6. SMEIL network for a seven segment display clock. Each SMEIL process is represented by a cicle with a letter corresponding to the processes Input, Hours, Minutes and Seconds respectively.

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*), 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.

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

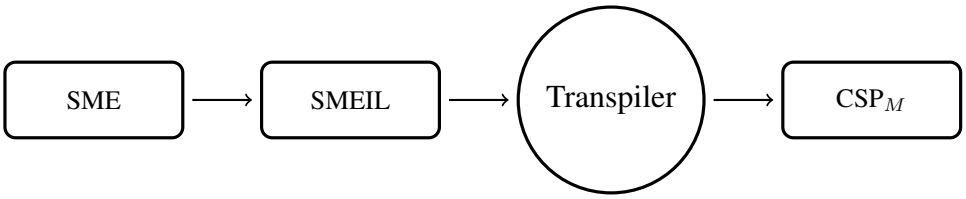

Figure 7. SME to CSP$_M$ transpiler.

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

```
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.

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 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.

```
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.

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.

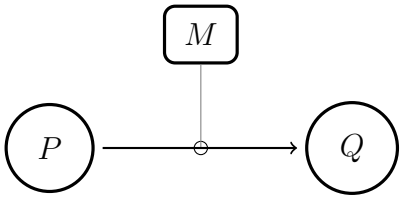

Figure 8. The monitor process M listens in on the communication between P and Q in order to assert the communicated values.

```
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.

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

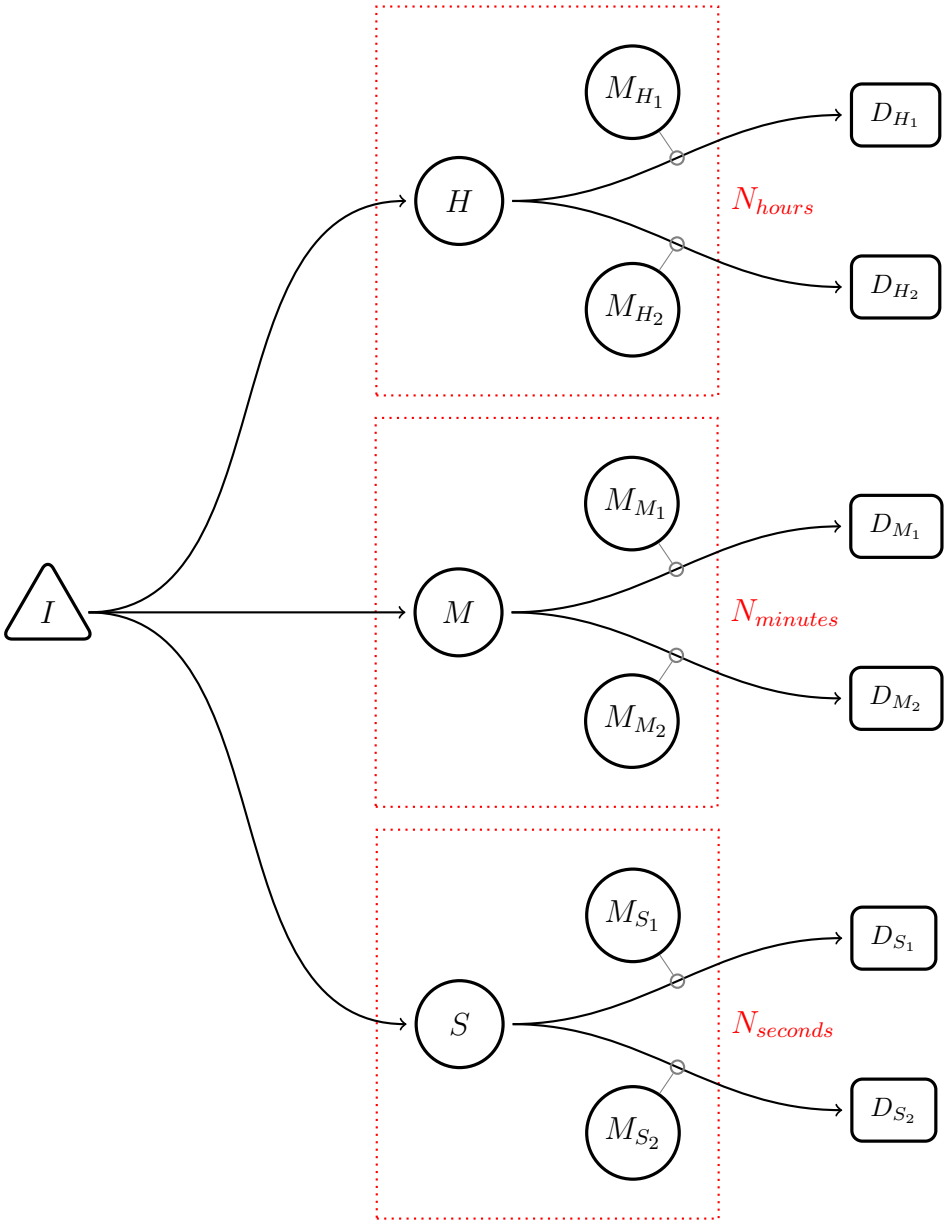

Figure 9. A seven segment display clock network in CSP$_M$. I represents the input channel. N_{hours}, $N_{minutes}$ and $N_{seconds}$ represent the network processes with H, M and S as the `time` processes. The results from the `time` processes are communicated to the displays. The displays are represented 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.

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 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

```
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
7  Hours(hours_in) =
8  let
9      hours = hours_in / 3600
10     ⋮
11
12 Hours_out_first_digit_monitor(c) =
13     c ? x -> if 0 <= x and x <= 2 then SKIP else STOP
14 Hours_out_second_digit_monitor(c) =
15     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.

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.

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

```
1  proc clock ()
2      bus clock_out {val: u17 range 1 to 86401;};
3      var i: u17 = 0 range 0 to 86401;
4  {
5      i = i + 1;
6      clock_out.val = i;
7  }
8
9  proc hours (in hours_in)
10     bus hours_out {first_digit: u2 range 0 to 2;
11                    second_digit: u4 range 0 to 9;};
12     var hours: u5 range 0 to 23;
13     var hours_first_temp: u2 range 0 to 2;
14     var hours_second_temp: u4 range 0 to 9;
15 {
16     hours = hours_in.val / 3600 % 24;
17     hours_first_temp = hours / 10;
18     hours_second_temp = hours % 10;
19     hours_out.first_digit = hours_first_temp;
20     hours_out.second_digit = hours_second_temp;
21 }
22
23 proc minutes (in minutes_in)
24     bus minutes_out {first_digit: u3 range 0 to 5;
25                      second_digit: u4 range 0 to 9;};
26     var minutes: u6 range 0 to 59;
27     var minutes_first_temp: u3 range 0 to 5;
28     var minutes_second_temp: u4 range 0 to 9;
29
30 {
31     minutes = minutes_in.val / 60 % 60;
32     minutes_first_temp = minutes / 10;
33     minutes_second_temp = minutes % 10;
34     minutes_out.first_digit = minutes_first_temp;
35     minutes_out.second_digit = minutes_second_temp;
36 }
37
38
39 proc seconds (in seconds_in)
40     bus seconds_out {first_digit: u3 range 0 to 5;
41                      second_digit: u4 range 0 to 9;};
42     var seconds: u6 range 0 to 59;
43     var seconds_first_temp: u3 range 0 to 5;
44     var seconds_second_temp: u4 range 0 to 9;
45 {
46     seconds = seconds_in.val % 60;
47     seconds_first_temp = seconds / 10;
48     seconds_second_temp = seconds % 10;
49     seconds_out.first_digit = seconds_first_temp;
50     seconds_out.second_digit = seconds_second_temp;
51 }
52
53 network clock_network ()
54 {
55     instance g of clock();
56     instance h of hours(g.clock_out);
```

```
57      instance m of minutes(g.clock_out);
58      instance s of seconds(g.clock_out);
59  }
```

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

```
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 -> if 0 <= x and x <= 2 then SKIP else STOP
25  Hours_out_second_digit_monitor(c) =
26      c ? x -> if 0 <= x and x <= 9 then SKIP else STOP
27
28
29  Minutes(minutes_in) =
30  let
31      minutes = minutes_in / 60  % 60
32      minutes_first_temp = minutes / 10
33      minutes_second_temp = minutes % 10
34  within
35      minutes_out_first_digit ! minutes_first_temp ->
36      minutes_out_second_digit ! minutes_second_temp ->
37      SKIP
38
39  Minutes_out_first_digit_monitor(c) =
40      c ? x -> if 0 <= x and x <= 5 then SKIP else STOP
41  Minutes_out_second_digit_monitor(c) =
42      c ? x -> if 0 <= x and x <= 9 then SKIP else STOP
43
44
45  Seconds(seconds_in) =
46  let
47      seconds = seconds_in % 60
48      seconds_first_temp = seconds / 10
49      seconds_second_temp = seconds % 10
50  within
51      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.