FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Truls Asheim — truls@asheim.dk

A Domain Specific Language for Synchronous Message Exchange Networks

Supervisors: Brian Vinter and Kenneth Skovhede

May 2018

Abstract

Synchronous Message Exchange (SME) is a Concurrent Sequential Processes (CSP)-derived model for hardware designs implementing globally synchronous message passing. SME implementations currently exist for several general-purpose languages, some of which, are translatable to VHDL for subsequent implementation on hardware. A common SME language could reduce the duplication and feature disparity present in these independent implementations. This thesis introduces a domain-specific language for implementing SME designs. It is usable both as a primary implementation language for SME models and as an intermediate target for general-purpose languages. We describe the language, its implementation and its features. Furthermore, we explain the specific requirements for a language within this domain. Finally, we evaluate the language through a number of simple, but realistic, hardware designs by showing how they may be implemented and tested.

Acknowledgments

First of all, I would like to thank my supervisors Brian Vinter and Kenneth Skovhede for their supervision and support, not only in context of this thesis, but also in the scope of previous projects that I have done under their guidance. I would also like to thank the eScience group at the Niels Bohr Institute for providing a welcoming environment. In particular by offering myself, and the other Master's Thesis students, office space and even inviting us along for their strategy retreat. It was highly appreciated! Last but not least, I would like to thank Ben Duy'Anh Gurskov Nielsen, Alberte Thegler, Henrik Grønholt Jensen and Mads Ohm Larsen for taking their time to read my thesis and providing insightful comments.

Contents

Abbrevations

ASIC Application Specific Integrated Circuit. 12

CPU Central Processing Unit. 1, 12

CSP Communicating Sequential Processes. 2, 6

CSV Comma-Separated Values. 9, 35

FPGA Field-Programmable Gate Array. 2, 12, 13

GPGPU General Purpose Graphics Processing Unit. 1, 2, 12

HDL Hardware Description Language. 1

HLS High-Level Synthesis. 1

IC Integrated Circuit. 12

IL Intermediate Language. 12, 35, 42

SLOC Source Lines Of Code. 10

SME Synchronous Message Exchange. 2, 14

SMEIL SME Implementation Language. 14

VHDL VHSIC Hardware Description Language. 1, 2, 10, 11, 13, 14, 42

1

Introduction

Special-purpose hardware has a wide range of different uses and can provide a significantly improved performance-to-watt ratio compared to General Purpose Graphics Processing Units (GPGPUs) and Central Processing Units (CPUs). However, the prevalence of such hardware is limited, in part, by poor design tools. Traditional hardware design workflows utilize Hardware Description Languages (HDLs) such as VHSIC Hardware Description Language (VHDL) or Verilog which require the programmer to specify the hardware design at a very low level. While this enables complete control over the resulting hardware, the productivity sacrifice is significant when compared to using general-purpose languages for writing software. Additionally, all aspects of a hardware design are often written in a HDL, including code for testing and verification. Traditional HDLs are fundamentally unfit for performing tasks commonly needed for simulating input for a design, such as reading and decoding an image file.

In the past few decades, there has been a significant interest in tools that improve the productivity of hardware design workflows. Vendors of reconfigurable hardware have focused primarily on High-Level Synthesis (HLS). These utilities transform algorithmic descriptions written in a general-purpose language to an HDL-description that can be implemented on hardware. The source languages for the most common HLS tools are C (e.g. Vivado HLS [43]) and OpenCL (e.g. Altera OpenCL [23]).

Hardware is inherently parallel, and utilizing this parallelism is imperative for achieving good performance. Therefore, efficiently transforming sequential C code to a hardware description requires inferring parallelism in a similar manner to, for example, OpenMP. To control the transformation, the programmer is required to add annotations to the C program. The quality and performance of the resulting hardware implementation depend greatly on the aptitude of the programmer to add these annotations correctly. This requires a deep understanding of the transformation process and the underlying architecture of the targeted hardware. The difficulties of creating auto-parallelizing compilers for impure general-purpose languages are well-known and arise in particular due to the challenges of resolving data dependencies [11, 14]. HLS utilities provide no revolutionary improvements in this regard and thus have a tendency to retain major sequential parts of the original program resulting in an inefficient hardware design.

Transforming OpenCL programs to hardware descriptions is a related scheme which is currently gaining popularity. This option seems more attractive as OpenCL is already an explicitly parallel language targeting heterogeneous computing platforms. However, OpenCL code needs to be tuned specifically to each target platform in order to achieve optimal performance [15]. Most existing OpenCL programs are written with GPGPUs in mind. Thus, these programs must be rewritten to perform optimally on Field-Programmable Gate Arrays (FPGAs), again requiring heavy use of annotations. This reduces the portability and productivity advantages of OpenCL. Furthermore, the OpenCL computing model requires the presence of a host device which makes it unsuitable for creating completely independent hardware components.

To approach this problem from a different angle, the Synchronous Message Exchange (SME) model[40, 41] has been introduced. SME is similar to Communicating Sequential Processes (CSP) [22], but replaces the rendezvous-style communication of CSP with globally synchronous message passing between processes driven by a hidden clock[1]. This allows the programmer to be explicit about concurrency, using a model which closely resembles signal propagation in hardware. Thus, SME simplifies performance reasoning compared to the HLS approaches described above which relies on inferred concurrency.

As the implementations of SME has advanced, SME has been utilized to create several successful hardware designs which have been implemented on FPGAs. For example, a MIPS processor implemented in SME was successfully synthesized and implemented on an FPGA [25]. These achievements have motivated and encouraged the continuing development of the model and related utilities, although we do not claim that it has reached the level of maturity of the HLS approaches previously mentioned.

SME by itself is just a model, which is not tied to a specific programming language or implementation. Currently, libraries for implementing SME models exist for the general-purpose languages C++ [3], C# [37] and Python [7]. The latter two have code-generation backends targeting VHDL. Maintaining feature-parity between these independent implementations proved infeasible due to the code-duplication involved. This created a demand to unify the common backend components of the divergent code bases of the two SME implementations. To achieve this, a common intermediate language for SME networks was needed. Additionally, combining SME networks written in different source languages was also a desired feature. While we could feasibly introduce an interface allowing this between Python and C#, the number of required interfaces increase exponentially for every language added. Having a common intermediate language would make this integration simple.

This thesis introduces the SME Implementation Language (SMEIL) and its accompanying implementation, LIBSME [5] (sometimes referred to as "the compiler"). SMEIL is a specialized language, featuring a familiar C-like syntax and structural constructs which are deeply rooted in the SME model. Furthermore, it provides a type-system which is tailored for hardware-specific subtleties that are difficult to express in general-purpose languages without deviating from established paradigms. An explicit design goal of SMEIL is to provide a simple and straightforward mapping of code structures commonly found in imperative general-purpose languages. For testing designs implemented in SMEIL, general-purpose languages are well suited since their full range of available libraries can be utilized. Therefore, LIBSME provides a simple and language-independent API allowing SME implementations written for general-purpose languages to communicate with SME networks written in SMEIL.

Although SMEIL was initially intended as an intermediate language target for existing SME implementations, the resulting language is also usable as an independent primary implementation language for SME models. The remainder of the thesis will

[1] An extended introduction to the SME model is given in Section 2.1

primarily describe the language from this perspective. To show its use as an intermediate language, we have adapted our previous implementation of a Python SME to VHDL compiler [2] to output SMEIL instead of VHDL directly. This is discussed in Section 6.1.

1.1 Motivations for a SME DSL

Initially, we considered just creating a common Abstract Syntax Tree (AST) representation. This approach would focus on generalizing the existing ASTs already used internally by the PySME and C# SME to VHDL transformers. An advantage of this strategy is that it carries a smaller implementational burden compared to creating a dedicated language. However, no simple and established frameworks exist for formally specifying an AST in a language-neutral way. A representation without a corresponding concrete syntax would also be difficult to understand and reason about, making it hard to verify the correctness of the generated intermediate code.

These observations meant that the resulting language got a concrete syntax and fulfilled the original design goal of providing a direct mapping of constructs from common general-purpose languages. The reason for this design goal was to ensure that adding new SME frontends would be as simple as possible, relieving them of performing sophisticated transformations. Thus, SMEIL inevitably became an independent DSL suitable as a primary implementation language for SME models. Exploring the concept of an independent SME DSL is interesting for a number of reasons. In particular, A DSL allows concise and elegant expression of concepts present in the target domain—the domain-specific needs of hardware are not considered in the design of general-purpose software languages.

C# SME. Translating SME models written in C# is comparatively straightforward since language properties can be statically specified and are enforced by the compiler. For example, if we declare a variable as being constant, we can be sure that its value will never change beyond its initial assignment and fixed-length arrays may be explicitly created as such. Likewise, when we declare a variable to be of a certain type, we can be sure that it will keep that type throughout the program. Being able to specify such restrictions is immensely useful as the transformation target (a hardware description), is also static. However, this also means that hardware-targeted SME models written in C# contains a lot of declarative "noise" required to confine the C# language to a feature set which is possible to implement on hardware. Since the C# syntax cannot be extended to natively declare SME elements, annotations are required to inform the SME runtime and translation system about how a C# object should be interpreted.

PySME. The situation is different for languages such as Python. The key selling point of Python is that it is a high-productivity language which is simple to use. It largely owes these attributes to the fact that it is a dynamic language. However, this makes it challenging to determine the static properties needed in a hardware description from a Python program without imposing a heavy annotational burden. Furthermore, building upon our C# example from before, ensuring that a Python variable retains its type throughout program execution require either sophisticated analyses or strong programmer discipline. Being able to provide such guarantees is a prerequisite for performing a semantically unchanged transformation. We base these assertions on our previous experience building a Python SME to VHDL compiler [7]. While we were able to transform SME networks written in Python to VHDL, the programmer could only use a narrow and strictly specified subset of Python in the hardware-targeted processes. The addition of unfamiliar features (annotations), and required re-learning of semantic assumptions, reduces the advantage of Python from the perspective of an experienced Python programmer. It is certainly possible to improve on our previous attempt at transforming Python. However, this is a significant effort which does not directly contribute to the capabilities of SME as a hardware design utility.

The key advantage of writing SME models in a general-purpose language is that test code can utilize the full range of libraries available for that language. It is crucial that this advantage is preserved for SME networks written in SMEIL. We explain how this is achieved in Chapter 4. A common objection against DSLs is the requirement of learning a new and unfamiliar language. However, the SME model itself needs to be learned in any case and the additional overhead SMEIL imposes, is minimal. From first-hand experience, a student of computer science familiar with CSP, but not SME, was able to start writing simple SMEIL programs after just a few hours of introduction.

Due to its origins as an intermediate language, its syntax is not the friendliest. The syntax attempted to strike a balance between being simple to parse while not being completely unreadable for humans.

1.2 Limitations

This thesis does not discuss the low-level details of hardware design beyond a brief introduction to hardware design workflows. As previously mentioned, the feasibility of SME as a hardware design tool has already been established through previous successful implementations. All of our references to hardware design are based on these previous experiences. The problems addressed in this thesis are purely related to the SME model and the results of our work does not alter the *fundamental* qualities of SME as a hardware design tool.

1.3 Contributions

We summarize the contributions of the thesis as follows:

- We present a new language for implementing SME networks. The language is suitable both as a primary implementation language for SME networks and as an intermediate language for other SME implementations.
- We provide a way to test models written in the language using a co-simulation approach.
- We provide a method for deriving the minimally required bit-widths of wires in the final design from the observed range of values assigned to them during simulation.
- We demonstrate an implementation of the above points in addition to VHDL code generation from designs written in the introduced language.

A paper based on this thesis has been submitted for publication as

> T. Asheim, "SMEIL: A Domain Specific Language for Synchronous Message Exchange Networks". In: *Proceedings of Communicating Process Architectures 2018* (2018)

1.4 Notation and Definitions

We frequently refer to hardware-design nomenclature: A *test bench* is a piece of software used for testing a hardware model by providing input data and verifying its output. *Synthesis* is the process of transforming a hardware-model written in a HDL to an actual description which can be implemented on hardware. We will occasionally refer to "assigning a value". The "value" here may, unless specified otherwise, be any assignable SMEIL construct (either a variable or a bus channel).

2

Background

In this chapter, we introduce the Synchronous Message Exchange (SME) model and briefly describe its origins, evolution, semantics, and implementations. The design of SMEIL draws from the lessons learned throughout the brief time period in which SME has existed. Here, we convey these insights to the reader. Additionally, we motivate the need for custom hardware.

2.1 Synchronous Message Exchange

The beginnings

The Synchronous Message Exchange model was conceived based on the experiences of a masters thesis project [35] which attempted to generate a hardware description from a model of a vector processor. The vector processor (described in [32]) was modeled with CSP using PyCSP [13], a CSP library for Python. The initial experiences using CSP for modeling the processor were promising. Especially the process abstraction of CSP proved to be well suited for representing the discrete components of a hardware design. Furthermore, the modularity originating from the *shared-nothing* property of CSP was advantageous as it allowed seamlessly interchanging fine- and coarse-grained implementations of the same discrete component.

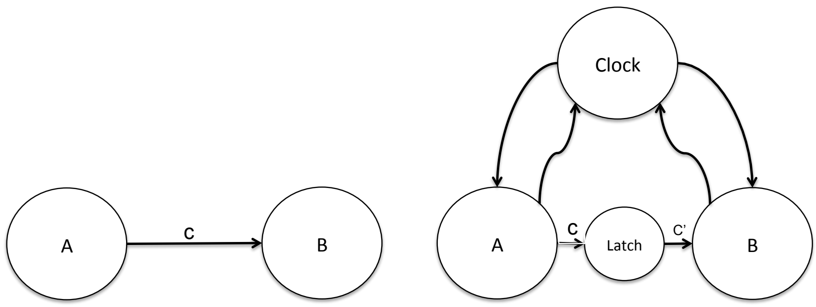

Figure 2.1: In order to enforce synchronous communication semantics on a simple CSP network, a large amount of additional complexity is needed. Figure from [41].

When the master thesis project, mentioned before, later attempted to convert the pure CSP model to a hardware description, they found the CSP approach less apt. Their experiences revealed a fundamental discrepancy between the data propagation models of hardware and of CSP. In CSP, a process is free to communicate at any time while in digital hardware, all communication is driven forward synchronously by a clock. Thus, to accurately model hardware using CSP, this clock had to be emulated by adding a single clock process with broadcasting channels to every process in the network. Back-channels also had to be inserted for notifying the clock process when a process had finished running. Furthermore, latch-processes had to be inserted into every channel going between processes, ensuring that values were not propagated in the middle of a clock cycle. The effect of adding these additional processes and channels is seen in Figure 2.1. Whenever the clock process emitted a signal, all processes in the network would run. When the processes completed their run, the latch processes ensured that values were propagated in the next cycle.

In the end, the thesis successfully managed to translate simple PyCSP networks to VivadoC, a language for HLS. Despite this, the overall conclusion was that, while CSP could be forced to adhere to globally synchronous semantics, the networks required to do so were prohibitively complex. Furthermore, only a small subset of the features in CSP was used to model the design. Particularly, a concept central to CSP, *external choice* which allows a process to determine if it should run based on whether it received a message, was not found to be applicable to hardware designs. However, not all was bad: As concluded by the original vector-processor design work, the shared-nothing property of CSP proved useful as the state of the network could only be altered by processes communicating. This made it easy to compose networks without worrying about inter-process dependencies.

Based on these experiences, the idea of using pure CSP as a hardware design tool was discarded. In its place, a derived model, called SME, was conceived which maintained the concepts of CSP that were found beneficial while adding a new, globally synchronous, communication model [41].

The model

The key concept of the SME model is the introduction of an implicit clock, eliminating the complexity induced by forcing CSP to adhere to globally synchronous message passing semantics.

Building on its CSP roots, the fundamental unit in an SME network is the *process*. Networks are built by connecting processes through buses. SME buses use the name “bus” instead of “channel”, to reinforce the hardware analogy and clarify its semantic equivalences with a physical signal bus found in hardware. Furthermore, where channels in traditional CSP only support a single value, a bus in SME is a bundle of individual channels connected to processes as an entity. This is also considered part of the hardware analogy since a single signal path in hardware often consists of several individual wires.

Execution flow

The SME concept of a “clock cycle” (Figure 2.2) goes through two distinct phases. During the *compute phase* all processes are run. During the *bus propagation* phase, values written to buses in the current cycle are positioned to be read by processes in the following cycle.

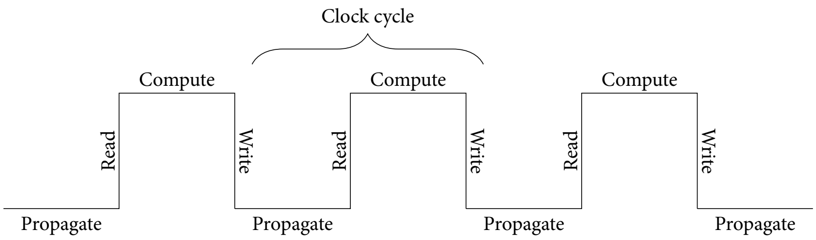

Figure 2.2: Illustration of the SME clock cycle concept.

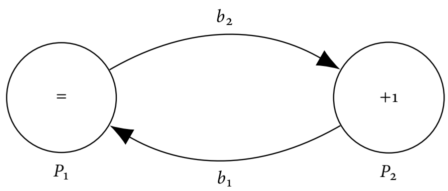

Figure 2.3: A simple SME network consisting of two processes. One simply forwards the received value while the other increments it by one.

op \ c	1	2	3	4	5	6	7	8	9
$P_1 \leftarrow b_1$	0	1	1	2	2	3	3	4	4
$P_1 \rightarrow b_2$	0	1	1	2	2	3	3	4	4
$P_2 \leftarrow b_2$	0	0	1	1	2	2	3	3	4
$P_2 \rightarrow b_1$	1	1	2	2	3	3	4	4	5

Figure 2.4: A table showing values read and written for every clock cycle of SME networks. Note that when we refer to a *trace* later, it is different from the table shown here. An SME trace file normally only contains the values of the reading ends of bus channels following every cycle.

Each channel in a bus has separate reading- and writing-ends. During the compute phase of a cycle, the reading-ends of channels are kept constant. The writing end of a channel has a single-element overwriting buffer. Therefore, when a process writes to a channel, the result is not immediately visible on the reading end. The bus propagation phase copies all values from the writing end to the reading end. Thus, values written in cycle c, will be read in cycle $c + 1$. Another way to look at this is that, from the perspective of processes, the values of all buses change simultaneously since value propagation happens when the processes are not running.

An example

Even though the concepts of the SME model are uncomplicated, gaining an intuition of value propagation governed by globally synchronous semantics is harder. In an attempt to convey this intuition, we show an example of a simple network, seen in Figure 2.3. We return to a slight variation of this example later, but for now, the network consists of two processes P_1 and P_2 and two buses connecting them, b_1 and b_2. In this network, a value is passed around in a circular fashion. The process P_1 simply forwards the value it receives while the P_2 process increments it by 1. In Figure 2.4 we see the actual values

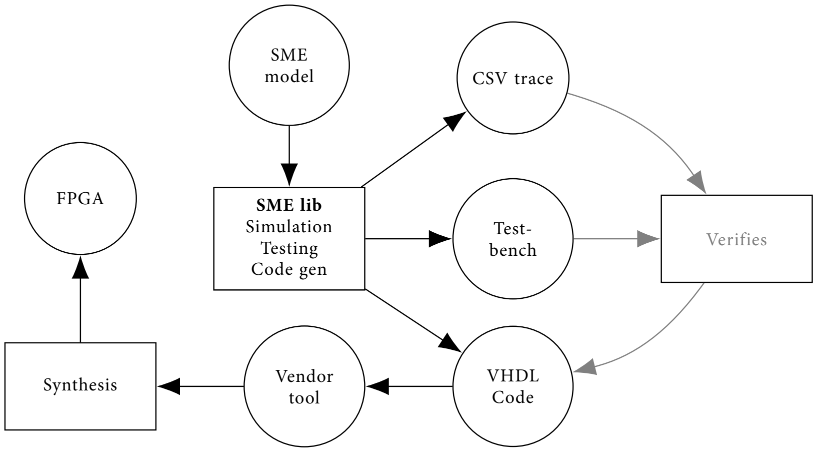

Figure 2.5: A simplified overview of the steps taken from SME model to hardware implementation.

read and written by every process for every cycle. Note that before every cycle shown in the table, an implicit bus propagation is run, driving forward the bus values. The arrows denote the operation performed. A process can either *write into* or *read from* a bus. So, the operation $P_1 \rightarrow b_1$ means that P_1 writes to b_1. The reading-ends of all buses initially start out as 0. Thus, in the first cycle, this value is read by both processes. In the second cycle, we see the effect of the delayed value propagation: P_2 reads 0 again, even though it wrote 1 in the previous cycle. Due to the single-cycle delay in value propagation through a bus, the 0 read now in cycle c was written by P_1 in cycle $c - 1$. This pattern continues and we show the first 9 cycles here. In cycle 9, the value written by P_2 is 5.

Using SME

The purpose of modeling a design in SME is to eventually convert it to an actual hardware description. Regardless of which SME implementation is used (including the one described in this thesis), the process goes through the same general steps shown in Figure 2.5. The first step is to write the SME model and related tests in a language with the required SME support. Then, this model is read by the SME library which simulates the design and runs the related tests in order to verify correctness. Three results are generated from the simulation: A rendering of the SME design in an HDL (only VHDL has been used so far), a test bench written in the HDL used for verifying the generated code and finally, a Comma-Separated Values (CSV)-file containing a value trace. The CSV-file is read by the test bench which uses it as a source of input values to provide to the hardware model and for checking that its output values are as expected. The generated HDL code is then passed to a vendor tool for synthesis and eventual implementation on hardware.

The present work only affects the stages up until passing the generated HDL to a vendor tool.

Implementations

A number of different library implementations of the SME model exist.

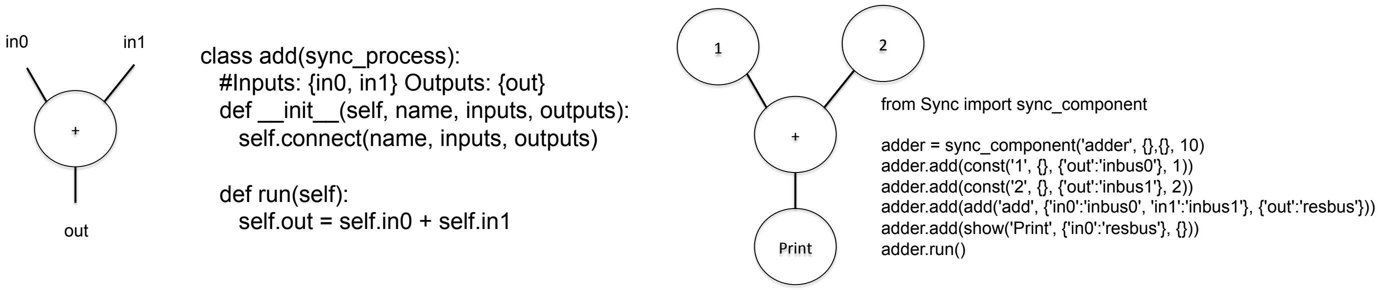

Figure 2.6: An implementation of an adder (left) and a network using it (right) written using the original SME framework. Figure from [41].

1st PySME

The initial implementation of SME was extremely simple: A mere 69 Source Lines Of Code (SLOC) of Python was all that was needed to create a library allowing Python programs to be written following the SME model. This implementation was, of course, quite rudimentary, however, it underlines a key advantage of the SME model. A person can both understand the model and write an implementation from scratch in less than a day.

An example of a network for adding together two values written using this SME implementation is seen in Figure 2.6. As can be seen, this initial SME version created connections between processes using *named channels*. So connecting a channel between two processes was done just by referencing the same name as an input port and output port in the instances of two different processes.

2nd PySME

After promising experiences with the first version of SME, a revision to the model and its implementation was published [40]. This version contained a number of changes, the most important being the abandonment of using the aforementioned named channels for connecting processes. Instead, buses were now considered first-class independent components of an SME network. Furthermore, a bus was extended from being just a single channel to a bundle of channels. Also included was a new top-level construct, named `Network` used purely for defining buses and their connections to processes. The adder shown in Figure 2.6 using this version of SME is seen in Figure 2.7.

Modeling buses as active components of the network opened up a number of possibilities, in particular generating the CSV trace-files mentioned above. A disadvantage of this approach was that defining the connections of a complex network quickly became unwieldy, as may already be visible from the very short example in Figure 2.7.

At this point, SME was only used for simulation and prototyping of hardware designs. The completed prototypes were then manually translated to VHDL and verified using the trace-file generated by simulating the original SME implementation. This was a tedious process, but it showed the viability of translating SME models to a hardware description.

```
from bSME import *

class Const(Function):
  # [..]

class Add(Function):
  def setup(self, args):
    self.cbus1, self.cbus2, self.out = args
    self.out["val"] = 0
  def run(self):
    self.out["val"] = self.cbus1["val"] +
                      self.cbus2["val"]

class Printer(Function):
  # [..]
```

```
class Adder(Network):
  def wire(self, args):
    self.constbus1 = Bus("Const1", "val")
    self.constbus2 = Bus("Const2", "val")
    self.resbus = Bus("Result", "val")
    self.c1 = Const("c1", [self.constbus1, 30])
    self.c2 = Const("c2", [self.constbus2, 5])
    self.add = Add("add", [self.constbus1,
                 self.constbus2,
                 self.resbus])
    self.printer = Printer("print",
                     [self.resbus])
Adder("Adder").clock(10)
```

Figure 2.7: The adder shown in Figure 2.6 implemented using the updated version of the SME framework.

C# SME

A C# implementation of the new version of SME was also created [37]. The primary change from the Python SME implementations was the omission of a `Network`-like construct. Instead, connections between a pair of processes were established if they referenced the same bus. So instead of process instances being parameterized with their connections, processes established connections themselves by referencing buses. This proved to be a more comprehensible way of building networks as the information about connections was spread out throughout the program instead of being confined to a single class. However, a shortcoming of this approach was the one-to-one correspondence between process and bus declarations and instances. This limitation was alleviated in later versions of C# SME library by the introduction of *scopes* which allowed several instances of the same process to exist as long as they were defined in different scopes.

This version of SME also facilitated automatic translation to VHDL.

3rd PySME

Based on the success of translating SME models written in C# to VHDL, a project was started to bring the same capability to the Python version of SME [7]. The challenges of deriving static code from a dynamic language were briefly mentioned in the introduction. Because of these challenges, the previous PySME implementation was altered to require the programmer to state her intentions more clearly. For example, in the 2nd PySME, declaring a bus or instantiating a process could be done simply by assigning a variable in the `wire` function of a network class. This made analyzing the code difficult since the programmer's intention was not clearly stated. Thus, an `add` method was added to the `Network` class. This is the version of PySME used in the thesis.

Conclusions

The design of SMEIL draws heavily from the lessons learned by the different approaches used by these SME implementations. First of all, we concluded that requiring all buses and connections to be declared in one place quickly become difficult to understand.

The other is, that it would be advantageous to associate process instances with bus instances to avoid the same pitfall as the original C# implementation. This is also helpful when using SMEIL as an Intermediate Language (IL) since buses and their connections can be translated straightforwardly regardless of the originating implementation.

2.2 Motivations for custom hardware

Digital circuits are fundamentally a collection of logic gates which are connected in a specific configuration to fulfill a particular purpose. In Integrated Circuits (ICs), this configuration is hard-wired – etched into silicon using a lithographic process. An example of a hard-wired IC is the common CPU. CPUs are highly versatile devices, capable of computing anything computable. However, in addition to their low price, this versatility is their main advantage and they excel at nothing else. Since the advent of the first microprocessor, almost 5 decades ago, this problem has been widely recognized. Therefore, a steadily increasing amount of special-purpose hardware is being added in order to relieve the main CPU pipeline of common and computationally intensive tasks. An early, and highly successful, example of this is the introduction of co-processors for performing floating-point calculations. As personal computers were increasingly being used for tasks relying heavily on floating-point arithmetic, emulating this in software increasingly became a limiting factor for performance. To solve this, specialized hardware units, known as Floating Point Units (FPU) were added to significantly speed up floating point calculations compared to what was possible using software emulation. Numerous similar examples exist, for example, the AES-NI instruction set built into recent CPUs, providing access to specialized circuitry for performing encryption and decryption using the ubiquitous Advanced Encryption Standard (AES) algorithm.

The current use of specialized hardware only scratches the surface of applications for which it would be beneficial. For example, by offering a significantly improved performance-per-watt ratio [17]. This is especially important in the light of the ever-increasing power consumption of data centers [9] which counter global efforts to decrease emissions. In an ideal world, everyone would have cheap and easy access to creating hardware specialized for their particular application. Unfortunately, such hardware, known as Application Specific Integrated Circuits (ASICs), is extremely expensive and time-consuming to design and put into production. Furthermore, a very large number of units has to be ordered before achieving a reasonable cost-per-unit. Finally, mistakes are expensive since once the hardware is made it can never be changed.

Field-Programmable Gate Arrays (FPGAs)

Reprogrammable computing, of which Field Programmable Gate Arrays (FPGAs) are the only prominent example, offers an attractive compromise between ASICs and more general devices such as CPUs or GPGPUs. While not nearly as good as ASICs [28], they can provide significant improvements in both performance and power consumption at a much lower cost.

FPGAs are ICs which allow their circuits to be changed after manufacture (hence, they are programmable in the field) and can, therefore, be reconfigured to fulfill any purpose. Of course, the circuitry on the chip cannot be physically changed, so FPGAs consists of an array of logic blocks which has configurable interconnects. The reprogramming of an FPGA happens by reconfiguring switches which determines the signal path. Since the individual components on an FPGA are fine-grained, they can be reconfigured for any computational purpose by rewiring signal paths.

The VHSIC Hardware Description Language (VHDL)

As briefly mentioned in the introduction, FPGAs are usually programmed using Hardware Description Languages (HDLs) such as VHDL or Verilog. We will focus on VHDL here, since that is the current target of SMEIL code generation.

VHDL was originally developed in the 1980's as a formal specification language for documenting hardware designs, with the first IEEE standard being released in 1987 [8]. Initially, it was purely a specification language, not intended to be run by computers. It was only later that simulators for VHDL were developed allowing designs written in the language to be automatically tested. Even later, tools were developed for performing *syntehsis* of VHDL which meant that a VHDL hardware description could be transformed to an actual circuit which could be implemented on hardware.

The standard of VHDL which is most widely supported by tools was released in 1993. This is despite the fact that an updated version of the standard was released in 2008. However, even the most recent iteration of the standard is fundamentally unchanged from the initial version. Therefore, VHDL has not benefited from decades of research in programming language productivity. As also written in the introduction, this becomes painfully obvious when VHDL is used for anything other than writing the actual hardware description. Most hardware design workflows today use traditional HDLs both for writing the actual hardware implementation and the corresponding testbench. Thus, the inadequacy of traditional HDLs for fulfilling the latter purpose creates both a productivity issue and a high barrier of entry for potential hardware designers.

Poor hardware design languages are not the only problem causing a high barrier of entry into the world of custom hardware. However, it is the problem that we, as software developers, are best equipped to help solving.

3

The SME Implementation Language

The language introduced in this thesis, the SME Implementation Language (SMEIL), is a small, strongly and statically typed, language with a C-like syntax featuring SME primitives as first-class constructs. In this section, we give an (informal) introduction to its syntax and semantics.

3.1 Guiding principles

As mentioned in the introduction, initial design decisions were primarily driven by the goal of making SMEIL usable as an intermediate language. In order to achieve this goal, it was important to provide a straightforward mapping of constructs found in languages such as Python and C#. These two languages, in particular, were the initial focus since they already had SME implementations with code generation backends for VHDL. Thus, the SME implementations were proven capable of more than just simulating simple SME networks. Furthermore, taking two imperative languages with different typing disciplines into consideration meant that SMEIL was less likely to adopt idiosyncrasies of either statically or dynamically typed languages. The body of SME code already existing for Python and C#, also meant that we could do more than just hypothesizing about the consequences of our SMEIL design choices. Furthermore, it allowed us to identify common use-patterns in order to determine the requirements of an SME language. The C-like syntax of SMEIL was chosen since it is simple to parse and contains no significant whitespace. Furthermore, curly-braces are used to clearly distinguish blocks and semicolons clearly mark the end of statements.

The guiding principles driving the initial design of the language were:

Language independence. Since SME networks can be written in several different languages, SMEIL should have no elements which are specific to a certain source language.

Structural richness. A goal of the SME model is that the generated code should be readable and have a relationship with the original source code. Therefore, SMEIL should have rich constructs for specifying the structure of SME networks.

Readability. Ensuring that the language has a readable and accessible representation aids debugging and makes it easier to understand. For this reason, SMEIL has a human-readable concrete syntax.

Composability. The language should provide unrestricted composability to ensure that networks can be subdivided for optimal flexibility.

Principle of least astonishment. As a continuation of the goal to ensure a straightforward mapping of constructs found in, for example, C#, the semantics of constructs not unique to SME are as expected. This principle applies everywhere in the language except for reading to and writing from, bus channels since these are features unique to SME.

3.2 Language reference

In this section, we describe the SMEIL language and its grammar from beginning to end. Towards the end of the chapter, we look a small example to see how everything comes together. The grammar of SMEIL (in BNF format) is introduced in fragments as we go along. Note that all the grammar fragments come together, so one fragment may refer to a production declared in another fragment.

Modules

⟨*module*⟩	::=	{ ⟨*import-stm*⟩ } ⟨*entity*⟩
		{ ⟨*entity*⟩ }
⟨*import-stm*⟩	::=	'`import`' ⟨*import-name*⟩ ⟨*qualified-specifier*⟩ '`;`'
	|	'`from`' ⟨*import-name*⟩
		'`import`' ⟨*ident*⟩ { '`,`' ⟨*ident*⟩ } ⟨*qualified-specifier*⟩ '`;`'
⟨*import-name*⟩	::=	⟨*ident*⟩ { '`.`' ⟨*ident*⟩ }
⟨*qualified-specifier*⟩	::=	{ '`as`' ⟨*ident*⟩ }
⟨*entity*⟩	::=	⟨*network*⟩
	|	⟨*process*⟩

The fundamental unit in a SMEIL program is a `module`. Similarly to, for eaxmple, Python, a module corresponds to a file. Only files can be modules since we, unlike Python, don't provide a way to make a directory act as a module[1]. Hierarchies of modules are built by importing one or more entities defined in a foreign module. Allowing SMEIL programs to be separated into several modules makes it simple to separate implementations in reusable components. A module contains import-statements and entities (described next). The syntax and semantics of import statements, are equivalent to those of Python and will be familiar to an experienced Python programmer. The handling of modules in SMEIL is described further in Section 5.2.

As an alternative to the current module system, we considered a model simply based on source includes. The implementation of such a system would be similar to that of the C pre-processor, possibly with implicit include guards preventing a single file from being imported more than once. The primary problem with this approach, despite being simpler to implement, is that include-based "module" systems feel archaic and require the names of all included entities to be unique. Languages relying on the

[1]In Python, this is done by creating a `__init__.py` file in a directory

C pre-processor get around this by, as a convention, prefixing all function names with the name of the library, however, this is restrictive in some cases.

The module system of SMEIL contributes towards the goal of creating a library of reusable components for SME.

Entities

```
⟨network⟩        ::= ‘network’ ⟨ident⟩ ‘(’ [ ⟨params⟩ ] ‘)’
                     ‘{’ ⟨network-decl⟩ ‘}’

⟨process⟩        ::= [ ‘sync’ | ‘async’ ] ‘proc’ ⟨ident⟩
                     ‘(’ [ ⟨params⟩ ] ‘)’ { ⟨declaration⟩ } ‘{’ { ⟨statement⟩ } ‘}’

⟨network-decl⟩   ::= ⟨instance⟩
                   | ⟨bus-decl⟩
                   | ⟨const-decl⟩
                   | ⟨gen-decl⟩

⟨declaration⟩    ::= ⟨var-decl⟩
                   | ⟨const-decl⟩
                   | ⟨bus-decl⟩
                   | ⟨enum⟩
                   | ⟨function⟩
                   | ⟨instance⟩
                   | ⟨generate⟩

⟨params⟩         ::= ⟨param⟩ { , ⟨param⟩ }

⟨param⟩          ::= { ‘[’ { ⟨expression⟩ } ‘]’ } ⟨direction⟩ ⟨ident⟩

⟨direction⟩      ::= ‘in’ (input signal)
                   | ‘out’ (output signal)
                   | ‘const’ (constant input value)
```

SMEIL programs are composed of two basic building blocks: `process` and `network`. Together, we refer to them as *entities*. Networks are used purely to define relations between entities. Therefore, they may only contain static declarations since the structure of an SME network must be static at compile time. This is a consequence of the static target of SMEIL which makes it impossible to, for example, instantiate a new process at runtime.

Processes consist of a declarative part and a statement part (the body). The declarative part defines all the variables and buses used in a process while the body is a collection of sequential statements which are evaluated once per clock cycle.

To simplify program analysis, it is not possible to declare new variables inside the body of a process. All variables used in the body must, therefore, be declared in the declarative part of the process ahead-of-use. For users of SMEIL as an intermediate language, the inconvenience is very slight since it is easy to gather all variables used in a code block and add the appropriate declarations.

Entities may take parameters which are set upon instantiation. Two types of parameters are supported: input- and output buses and constants. We discuss how to instantiate a process with a set of parameters later.

The purpose of the `sync` and `async` modifiers of processes, which are seen in the <*process*> production of the grammar, is to distinguish between processes which are run during every clock-cycle and processes which are only run when they receive a signal on their input buses. If neither modifier is specified, the process defaults to `sync`. `async` processes are not supported in the current implementation.

Declarations

This section introduces the declarations in SMEIL. All of the declarations described here may be used in the declarative part of a `process` while only some are allowed in `networks`. Note that an <*expression*> occurring in declarations must be compile-time static. Furthermore, due to limitations of the current compiler implementation, they are in some cases restricted to integers.

Variables and constants

⟨*var-decl*⟩	::=	'`var`' ⟨*ident*⟩ '`:`' ⟨*type-name*⟩ ['`=`' ⟨*expression*⟩] [⟨*range*⟩] '`;`'
⟨*range*⟩	::=	'`range`' ⟨*expression*⟩ '`to`' ⟨*expression*⟩
⟨*const-decl*⟩	::=	'`const`' ⟨*ident*⟩ '`:`' ⟨*type-name*⟩ ['`=`' ⟨*expression*⟩] '`;`'

Constants are used for declaring named constant values, for instance

```
const secs_per_hour: uint = 3600;
```

Constants should always be declared with an unconstrained type as it allows for more accurate type unification (see Section 3.3). Due to this, the compiler will emit a warning for constants declared with constrained types, such as `u8`.

Variable are used for holding process-local mutable values. A semantic variation compared to general-purpose languages is that the state of variables in SMEIL persists between process runs (clock cycles). In a way, they are similar to function-local `static` variables in C whose value also persists between function calls.

In addition to a type, variables may also take a specified `range` of values. For example, the following declarations

```
var seconds: uint range 0 to 59 = 0;
var seconds: u6 range 0 to 59 = 0;
var seconds: u6 = 0;
```

are all equivalent. The following declaration, on the other hand,

```
var seconds: u4 range 0 to 59 = 0;
```

is rejected by the type checker since representing the number 59 requires more than 4 bits.

The assignment (`= 0`) of all of these declarations sets the initial value of the variable.

The `range` option was added to provide an intuitive method for specifying the expected value range of a variable. However, a further use is described in Section 4.4. Currently, the given range is only used to calculate the number of bits required to hold the value. Only the range given by the bit-size is enforced during simulation. This is to more closely emulate the resulting hardware implementation.

Enumerations

⟨*enum*⟩ ::= '`enum`' ⟨*ident*⟩ '`{`' ⟨*enum-field*⟩ { '`,`' ⟨*enum-field*⟩ } '`}`' '`;`'

⟨*enum-field*⟩ ::= ⟨*ident*⟩ ['`=`' ⟨*integer*⟩]

Enumerations are a useful way of specifying closely associated named constants. They are used in a number of designs made with the C# SME library, for example, in the MIPS processor implementation [25] where the MIPS opcodes are defined in an `enum`. Referencing symbolic constants instead of numeric constants improve source code readability. Thus, to fulfill our goal of providing straightforward mappings from constructs commonly used in other SME implementations, enumerations were added to SMEIL.

Semantics are similar to other C-like languages. For example,

```
enum numbers {
  zero,
  three = 3,
  four,
  ten = 10
};
```

declares the enumeration `numbers` where the members are named in correspondence with their numeric values. When enumeration members are used in an expression they are typed according to the largest member. So all members of the enumeration above have the type `u4`.

Bus declarations

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

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

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

As previously mentioned, buses in SME are used for communication between processes. They provide a collection of one or more channels, of varying types, which are assigned to processes as a single unit (i.e., all channels of a bus are connected at the same time). The cardinality of buses is unrestricted, meaning that they may form many-to-many relationships between processes. Thus, SME buses mirror hardware buses, consisting of physical wires, where many components may be connected to the same wire. As a consequence of this, a bus may only have a single *driver* (an entity sending a signal on the bus) per clock cycle, since otherwise, the signal read from the bus is *unresolved* (i.e., its value is undecidable).

A bus in SMEIL is declared using a `bus` block. For example

```
exposed bus pixel {
  r: u8;
  g: u8;
  b: u8;
};
```

declares a bus named `pixel` used for transmitting the pixels of an image separated into their red, green and blue color channels. It contains three individual channels named `r`, `g` and `b` each of which is typed as 8-bit unsigned integers. The `exposed` modifier signifies that the bus is used for external interactions, either through co-simulation (Chapter 4) or through the top-level entity (see the next section) of the generated VHDL code (Section 5.2).

Exposed buses must be defined within, or in entities directly instantiated from, the top level entity. Otherwise, incorrect code will be generated. It is a future goal to check this statically such that appropriate error messages are emitted.

Similarly to variables, mentioned above, bus channels allow the specification of a `range`.

Entity instances

⟨*instance*⟩ ::= ‘`instance`’ ⟨*instance-name*⟩ ‘`of`’ ⟨*ident*⟩
‘`(`’ [⟨*param-map*⟩ ‘`,`’ ⟨*param-map*⟩]‘`)`’ ‘`;`’

⟨*instance-name*⟩ ::= ⟨*ident*⟩ ‘`[`’ ⟨*expression*⟩ ‘`]`’ (indexed instance)
| ⟨*ident*⟩ (named instance)
| ‘`_`’ (anonymous instance)

⟨*param-map*⟩ ::= [⟨*ident*⟩ ‘`:`’] ⟨*expression*⟩

A powerful feature of SME is its ability to define compositions of reusable networks. In SMEIL, networks are constructed by instantiating entities and connecting the instances using buses. Possible ways of composing a network are subject to only a few restrictions.

Furthermore, having several instances of the same process that are parameterized with different bus connections or different constant values is often convenient. Therefore, both processes and networks may have parameters which are set upon instantiation. The `instance` declaration is used to instantiate an entity with a specified set of parameters. An instance may optionally be given a name which can be used for referencing buses declared within the entity instance. Figure 3.1 shows three different ways that a network may be constructed through bus references. If an instance is unnamed (anonymous), connections between the instances can be made by referring to buses through their public names as seen in Figure 3.1a.

Several anonymous instances of an entity may exist within the same network. To avoid ambiguous networks, a scope may only contain a single anonymous instance of a particular entity. Figure 3.2 shows two networks utilizing anonymous entity instances. They both attempt to create the same number of instances of each process, however, one makes two anonymous instantiations of the same process and is therefore invalid.

Also note that <*instance-name*> allows the name of an instance to be followed by an optional array index (e.g., `instance a[i] of ..`), creating an *indexed instance*. This is intended to be used together with `generate`-declarations (described below) in order to create an array of instances. Like `generate`-declarations, this feature is not yet implemented.

When an entity is instantiated, a copy is created of all the resources declared within it. In particular, instantiating a process containing a bus will also create a new instance of that bus.

All well-formed SMEIL programs must contain a `network` declaration which is used as the top-level entity containing the exposed interfaces of the network. We chose not to add any explicit annotation for indicating the top-level entity since it can be unambiguously identified using the following method: A graph is generated from the

```
proc A ()
  bus b_a { chan: int; };
{
  B.b_b.chan = b_a.chan
}

proc B ()
  bus b_b { chan: int; };
{
  A.b_a.chan = b_b.chan;
}

network N ()
{ instance _ of A();
  instance _ of B();
}
```

(a) A network created by processes directly using buses through their hierarchical names.

```
proc A (in i)
  bus b_a { chan: int; };
{
  b_a.chan = i.chan;
}

proc B (in i)
  bus b_b { chan: int; }
{
  b_b.chan = i.chan;
}

network N () {
 instance a of
   A(b.b_a);
 instance b of
   B(a.b_b);
}
```

(b) A network created using processes taking their input bus as a parameter. The connection between the two processes is made by the network N.

```
proc A (in i, out o) {
  o.chan = i.chan;
}

network N () {
  bus b_a { chan: int; }
  bus b_b { chan: int; }

 instance a of
   A(b_a, b_b);
 instance b of
  A(b_b, b_a);
}
```

(c) This network only contains a single network taking both its input and output buses as parameters. The same network as in (a) and (b) is then built by passing bus references to two instances of `A`.

Figure 3.1: The three different networks shown here are equivalent and demonstrates different ways of connecting processes in SMEIL.

```
proc B () {}
proc C ()
  instance _ of B();
{}
network N {
  instance _ of B();
  instance _ of C();
}
```

(a) Since the two anonymous instances of process `B` are instantiated from different entities, this network is valid.

```
proc B () {}
proc C ()
{}
network N {
  instance _ of B();
  instance _ of B();
  instance _ of C();
}
```

(b) The duplicate anonymous instances of `B` in `N` makes this network invalid.

Figure 3.2: Two networks showing a valid and invalid use (respectively) of anonymous entity instances.

SMEIL program containing one node per entity and edges between entities that instantiate each other. The nodes of the graph are then topologically sorted and the head of the sorted list is the top-level entity of the network. We also ensure that the graph is acyclic as process instantiation cycles would expand indefinitely. Note that even though the connections of a network such as ADDONE (seen in Section 3.5) is cyclic, its instantiation graph remains acyclic since a single network instantiates the two processes.

Generators

⟨*gen-decl*⟩	::=	'generate' ⟨*ident*⟩ '=' ⟨*expression*⟩ 'to' ⟨*expression*⟩ '{' { ⟨*network-decl*⟩ } '}'

The `generate` declaration adds limited metaprogramming capabilities to SMEIL. This declaration is not supported by the current implementation since it was not directly needed by any of the examples that we implemented in SMEIL. However, it is often useful to create networks with a parameterized structure. For example, our MD5 example shown in Section 6.5 was derived from an implementation which use parameterized network instantiations to create networks containing a variable number of MD5 compute cores.

The semantics of `generate` declarations are simple:

```
generate i = 0 to 2 {
  instance a_inst[i] of a(val: i);
}
```

is equivalent to

```
instance a_inst_0 of a(val: 0);
instance a_inst_1 of a(val: 1);
instance a_inst_2 of a(val: 2);
```

except that in the latter example the instances are not indexed using array notation.

We could have used the more common name `for` instead of `generate`. However, the current name was chosen to emphasize that the declaration is not a loop executing statements for their side-effects but is rather a code-generation construct.

Statements

⟨*statement*⟩	::=	⟨*name*⟩ '=' ⟨*expression*⟩ ';' (assignment)
	|	'if' '(' ⟨*condition*⟩ ')' '{' { ⟨*statement*⟩ } '}' { ⟨*elif-block*⟩ } [⟨*else-block*⟩]
	|	'for' ⟨*ident*⟩ '=' ⟨*expression*⟩ 'to' ⟨*expression*⟩ '{' { ⟨*statement*⟩ } '}'
	|	'switch' ⟨*expression*⟩ '{' ⟨*switch-case*⟩ { ⟨*switch-case*⟩ } ['default' '{' ⟨*statement*⟩ { ⟨*statement*⟩ } '}'] '}'
	|	'trace' '(' ⟨*format-string*⟩ { ',' ⟨*expression*⟩ })';'
	|	'assert' '(' ⟨*condition*⟩ [',' ⟨*string*⟩])';'
	|	'break' ';'
⟨*switch-case*⟩	::=	'case' ⟨*expression*⟩ '{' { ⟨*statement*⟩ } '}'
⟨*elif-block*⟩	::=	'elif '(' ⟨*condition*⟩ ')' '{' { ⟨*statement*⟩ } '}'

⟨*else-block*⟩ ::= ‘`else`’ ‘`{`’ { ⟨*statement*⟩ } ‘`}`’

⟨*format-string*⟩ ::= ‘`"`’ ⟨*format-string-part*⟩ ‘`"`’ ‘`;`’

⟨*format-string-part*⟩ ::= ‘`{}`’ (placeholder string)
| ⟨*string-char*⟩ (Any printable character)

The semantics of statements in SMEIL corresponds to their counterparts in C-like languages. Thus, we will not devote a lot of attention to describing them here. A few things to note:

Assignments. In assignments, we make no distinction between what is being assigned. The same syntax is used both for assigning to bus channels and to variables. A common trait of HDLs is that they make a syntactic distinction between the two. VHDL, for example, uses `:=` and `<=` for variables and signals respectively. In the design of SMEIL, we concluded that there was no need to make this distinction: The compiler is always able to distinguish which kind of object is being assigned to, based on the type that it was declared as.

Loops. `for`-loops have, compared to C, a slightly more restricted syntax. For example, the following,

```
for i = 1 to 10 {
  trace("{}", i);
}
```

iterates through the range 1-10 (inclusive).

Tracing and asserting. The `trace` and `assert` statements of SMEIL reports on the state of the network and enforces runtime constraints, respectively. A `trace` statement takes a string optionally containing replacement “holes” (similar to `printf`) followed by a number of arguments matching the number of holes. For example,

```
foo = 1; bar = 2;
trace("foo {} bar {}", foo, bar);
```

will print

```
foo 1 bar 2
```

every time the process is executed.

Assertions are useful to specify invariants which must be maintained during program execution. In SMEIL, `assert` statements take a condition and an optional message. When an assert statement is evaluated, the condition is checked, and program execution is halted if a condition is violated. If present, the message is printed as part of the assertion error message. For example:

```
assert(i > 10, "i must always be greater than 10");
```

Switch statements. `switch` statements are similar to their C-counterpart except for the omission of implicit fallthrough. There are two reasons for this: The equivalent to a switch-statement in VHDL have no fallthrough-capabilities built in. Thus, supporting fallthrough in SMEIL would complicate translating SMEIL to VHDL. Furthermore, implicit fallthroughs are a misfeature of C as they often lead to incorrect code containing unintentional fallthroughs. There are plans to eventually add *explicit* fallthroughs.

Expressions

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

⟨*bin-op*⟩ ::= ‘`+`’ (addition)
| ‘`-`’ (subtraction)
| ‘`*`’ (multiplication)
| ‘`/`’ (division)
| ‘`%`’ (modulo)
| ‘`==`’ (equal)
| ‘`!=`’ (not equal)
| ‘`<<`’ (shift left)
| ‘`>>`’ (shift right)
| ‘`<`’ (less than)
| ‘`>`’ (greater than)
| ‘`>=`’ (greater than or equal)
| ‘`<=`’ (less than or equal)
| ‘`&`’ (bitwise-and)
| ‘`|`’ (bitwise-or)
| ‘`^`’ (bitwise-xor)
| ‘`&&`’ (logical conjunction)
| ‘`||`’ (logical disjunction)

⟨*un-op*⟩ ::= ‘`-`’ (negation)
| ‘`+`’ (identity)
| ‘`!`’ (logical negation)
| ‘`~`’ (bitwise-not)

The syntax of expressions the syntax and precedence rules (Table 3.1) are similar to those of C-like languages. Note that there is no implicit truthness of non-boolean operators (see Section 3.3). Thus, logical operators (e.g. `&&`) only accept boolean values making an expression such as `4 && 3` invalid. Relational operators (e.g. `<=`) returns a boolean value as their result.

Lexical elements

⟨*literal*⟩ ::= ⟨*integer*⟩
| ⟨*floating*⟩
| ‘`"`’{ ⟨*char*⟩ }‘`"`’ (string literal containing printable chars)
| ‘`[`’ ⟨*integer*⟩ { ‘`,`’ ⟨*integer*⟩ } ‘`]`’ (array literal)
| ‘`true`’
| ‘`false`’

⟨*ident*⟩ ::= ⟨*letter*⟩ { (⟨*letter*⟩ | ⟨*number*⟩ | ‘`_`’ | ‘`-`’) } (identifier)

Precedence	Operators
0	`+ - ! ~` (unary)
1	`* / %`
2	`+ -`
3	`<< >>`
4	`< > <= >=`
5	`== !=`
6	`& ^ |`
7	`&&`
8	`||`

Table 3.1: Operator precedence of SMEIL

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

⟨*array-index*⟩ :: '*' (wildcard)
| ⟨*expression*⟩ (element index)

⟨*integer*⟩ ::= ⟨*number*⟩ { ⟨*number*⟩ } (decimal number)
| '`0x`' ⟨*hex-digit*⟩ { ⟨*hex-digit*⟩ } (hexadecimal number)
| '`0o`' ⟨*octal-digit*⟩ { ⟨*hex-digit*⟩ } (octal number)

⟨*number*⟩ ::= '`0`' - '`9`'

⟨*letter*⟩ ::= '`a`' - '`z`' '`A`' - '`Z`'

The last part of the grammar left to cover is the lexical elements of SMEIL. Identifiers are any sequence of letters, numbers, underscores, and dashes which start with a letter. Names is a sequence of identifiers separated by dots. Array element accesses (e.g., `foo.bar[i]`) may also occur as part of a name.

3.3 Type system

SMEIL is a strongly, statically typed language with a simple type system that is checked at compile-time. Since hardware is static and SMEIL is targeted towards creating hardware descriptions, we want a type system which is capable of enforcing as many static invariants as possible. This means that no implicit type coercion is performed except between signed and unsigned integers. Consequently, there is no implicit notion of truthness (i.e., `if (1) {` results in a type error) and only expressions of boolean type can be used in conditionals. These restrictions ensure that SMEIL can be straightforwardly transformed to a wide range of target languages; it is easy to transform a statically typed language to one which is dynamically typed, but not the other way around.

The primary feature which distinguishes the type systems of SMEIL and general-purpose languages is the support for bit-precise types. General-purpose languages target CPUs which has fixed-width registers and are typically unable to work with units of

⟨*type*⟩	::=	‘`i`’ ⟨*integer*⟩ (signed integer)
	|	‘`int`’ (arbitrary-width signed integer)
	|	‘`u`’ ⟨*integer*⟩ (unsigned integer)
	|	‘`uint`’ (arbitrary-width unsigned integer)
	|	‘`f32`’ (single-precision floating point)
	|	‘`f64`’ (double-precision floating point)
	|	‘`bool`’ (boolean value)
	|	‘`[`’ [⟨*expression*⟩] ‘`]`’ ⟨*type*⟩ (array of type)

Figure 3.3: Grammar for type declarations in SMEIL.

Type 1	Type 2	Unifies to
ia	ib	i $\max\{a, b\}$
ua	ub	u $\max\{a, b\}$
ia	ub	i $\max\{a, b\} + [a \leq b]$
ua	ib	i $\max\{a, b\} + [a \geq b]$
uint	ia	i $(a + 1)$
uint	ua	u a
int	ia	i a
$[n]t_1$	$[m]t_2$	t_1 unified with t_2
t	t	t
otherwise		error

Table 3.2: SMEIL type unification rules. $[P]$ are Iverson brackets: $[P]$ = 1 if P is true

data smaller than a byte. When targeting custom hardware, we are free to define wires of exactly the width we need. Determining the minimum width of a wire is a prerequisite for avoiding wasted space leading to a less efficient hardware implementation.

SMEIL supports integers constrained to a specific bit-length, unlimited-size integers, booleans, double and single precision floating point and string. Fixed-length arrays of these primitive types may also be created. Floating-point numbers are only there for completeness but are currently not supported in hardware-translations due to the spotty floating-point support in FPGAs (although this situation is improving). The naming scheme for types is simple and follows a predictable pattern (the full grammar is shown in Figure 3.3). For integer types, the prefixes `i` and `u` refers to signed and unsigned integers respectively. A prefix is followed by a number determining the bit-length of the type. For example, `i13` is a 13-bit signed integer. Unlimited-size integers are also supported (more on those in Section 4.4) and are denoted simply as `int` and `uint`. Finally, `bool`, `f32` and `f64` denotes booleans and single- and double-precision integers respectively. Array types are created by prefixing a type with a number of elements. For example, `[10]i4` denotes an array of 10 4-bit signed integers.

The type checker of LIBSME determines the validity of types in a SMEIL program through a number of simple type unification rules (Table 3.2). For all non-integer types, the rules are simple: only truly identical types unify. For integer types with a constrained bit-length, the following rules apply: Two integer types with different bit-lengths are unified to the largest. Two types of different signedness are unified to a signed integer with a size taking the sign-bit into account. The reasoning behind this is the following: when unifying a signed and an unsigned number, the resulting type should be able to hold the largest number representable by either of the two types. For example, if we unify the types `u8` and `i8`, the result is `i9` instead of `i8`. Otherwise, if the unsigned number was larger than 2^7 the type conversion itself would cause an overflow. Also, as seen from the table, the lengths of arrays are not taken into account as this would be overly restrictive. For instance, the expression `a[3] + b[2]` would be invalid if `a` and `b` were arrays of different lengths.

Types are enforced on assignment meaning that the following declarations are invalid:

```
const foo: i32 = 3;
var bar: u16 = foo;
```

since they assign `foo`, a 32-bit signed integer constant, to `bar` a 16-bit signed integer variable. However, the following declarations

```
const foo: i32 = 3;
var bar: uint = foo;
```

are valid since `i32` and `uint` unifies to `i32`.

We realize that this model has several limitations. In particular, unifying two differently sized types to the largest does not ensure that the result of a binary operation on two values will not overflow the destination type. On the other hand, unifying types to sizes large enough that the result can not overflow often leads to a significant overestimation of required bit-widths. Likewise, binary operators which may produce a result smaller than either of its operands are not taken into account. The currently implemented type checker also does not consider the constness of values, which cause it to make wrong assumptions in some circumstances. However, if used correctly, our model for observationally derived types (see Section 4.4) will provide some assurance that an overflow will not happen. The justification of the type system its current form is that it is an improvement compared to the languages that SMEIL replace, such as VHDL and in use, it has detected bugs.

Enforcing bus shapes

The type checker reduces buses to a representation consisting of channel names and their types. We refer to this as the bus *shape*. Two shapes unify if they have identical channel names and types. Since entities accept buses passed as parameters, we must ensure that no entity is instantiated with a bus that does not contain the expected channels. Figure 3.4 shows two processes that are both rejected by the bus shape unifier. In the program shown in Figure 3.4a a bus, `coords`, containing the channels `x` and `y` is assigned to two processes `A` and `B`. This is fine for process `A` since it expects a bus with those two channels. However, its assignment to process `B` results in a failure since it expects a bus with channels `x` and `z`. In the other example, shown in fig. 3.4b, a process `A` is instantiated twice with two buses `coordsA` and `coordsB` containing differently named channels. This fails because the shapes of the buses cannot be unified.

Enforcing bus directionality

We also make sure that bus directionality is enforced. Buses passed as parameters are explicitly declared as being used for either input or output. It is not possible to explicitly

```
proc A (in b) {
  trace("Coordinates: {}x{}",
        b.x, b.y);
}

proc B (in b) {
  trace("Coordinates: {}x{}",
        b.x, b.z);
}

network N () {
  bus coords {
    x: int;
    y: int;
  };
  instance _ of A(coords);
  instance _ of B(coords);
}
```

(a) Process instantiated with an incompatible bus.

```
proc A (in b) {
  trace("Coordinates: {}x{}",
        b.x, b.y);
}

network N () {
  bus coordsA {
    x: int;
    y: int;
  };
  bus coordsB {
    x: int;
    z: int;
  };

  instance a1 of A(coordsA);
  instance a2 of A(coordsB);
}
```

(b) One process instantiated with two incompatible buses.

Figure 3.4: Two networks which are rejected by the bus shape unifier.

specify the directionality of a bus declared within a process. Such buses are designated as either input or output based on their first use. If contradicting bus uses are encountered (e.g. reading from an output bus), a compile-time error is raised.

The same mechanism which enforces directionality also checks if variables are used. Warnings are emitted for unused variables as these may be an indication of subtle bugs in the program.

3.4 Scoping rules

All declarations are private and may only be used within the entity where they are declared. The exception to this is buses which, as described previously, constitutes the public interface of an entity, used for establishing communication between two entities. The detailed scoping rules for SMEIL are as follows:

Modules. All top-level declarations (processes and networks) are public and may be imported by other modules.

Networks and processes. Most declarations (variables, instances, etc.) are private to the entity they are declared within. Buses declared within an entity can be accessed either directly through the declared name of the entity or through the name of an instance of the entity.

`for`-loops. The counter variable of `for`-loop may not be declared prior loop entry and leaves the scope when the loop exits.

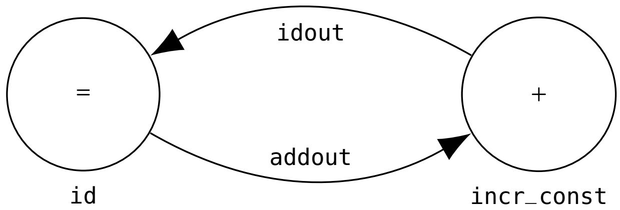

Figure 3.5: A simple SME network consisting of two processes. One simply forwards the received value while the other increments it by a constant. The names of the processes and buses corresponds with the names in Figure 3.6.

```
proc id(in inbus)
  bus idout {
    val: int;
  };
  var it: uint = 0;
{
  idout.val = inbus.val;
  trace("Iteration: {} Value: {}",
    it, inbus.val);
  it = it + 1;

}
```

```
proc incr_const(in inbus, const val)
  bus addout {
     val: int;
  };
{
  addout.val = inbus.val + val;
}

network incr() {
  instance addone_inst of
    incr_const(id_inst.idout, val: 10);
  instance id_inst of
    id(addone_inst.addout);
}
```

Figure 3.6: An example program written in SMEIL.

3.5 A small example

Before closing this chapter, we show an example of a small, but complete SME network, implemented in SMEIL, to give a feel of the language and show how everything fit together. The ADDONE network, illustrated in Figure 3.5, consists of two processes and two buses. One process, named `id` simply passes along the value received while the other process, named `incr_const`, increments it by a value passed as a constant parameter. The source code for the example is shown in Figure 3.6.

Each of the two processes is declared using a `proc` block. Immediately following, are the declarations belonging to the processes. In this case, both of the processes declares the bus used for sending their output values. Both processes are also parameterized with a bus on which they receive their input values. The `incr_const` process takes an additional parameter which is a constant value added to the input value. The curly-braces in a process contain the statements constituting its body, that is, the actions that are performed when the process is executed during a clock cycle. Both of the buses `addout` and `idout` are used within the process where they are declared. Therefore they are not explicitly annotated as being used for either input or output. However, since the buses are assigned in the body of the processes they are designated as output buses. The network `incr` instantiates the two processes and connects the output one to the input of the other using instance parameters. Furthermore, for the instance `addone_inst` the constant parameter `val` is also set, making the `incr_const` process add 10 to the value it receives every time its run. Declarations may be given in any order, allowing the mutually dependent process instantiations in the `incr` network.

4

Co-simulation

The usefulness of SMEIL would be limited without a way to provide external interactions with SME networks written in SMEIL. The simplicity of SMEIL can be attributed to its narrow scope: it is only intended for writing hardware models and not for writing test benches. For this, the full power of a general-purpose language is needed as the test-code can be written without hardware-related considerations and using all available libraries. For example, a test bench may read an image from disk or visualize the results of a simulation. Extending SMEIL to be able to perform such tasks is a substantial undertaking that does not further its primary purpose as a hardware-modeling language. Co-simulation [33] is the process of two separate entities (in this case two SME networks) which communicates through channels transparently established by the SME libraries.

4.1 Co-simulation with SMEIL

For performing co-simulation with SMEIL, we expose a C API from LIBSME. The API is intended to be implemented by SME libraries for general-purpose languages.

There are three aspects to the API: Firstly it provides a way to enumerate the buses exposed from a SMEIL model. Secondly, it provides calls for reading to- and writing from bus channels and driving forward the simulation. Finally, it offers calls for ordering the production of output, such as VHDL code generation.

A noticeable feature of the API is its support for arbitrary-length integers. This means that the bit-width of buses used by simulated models are not restricted by the register-width of the CPU of the underlying host. A hardware model may use values of any bit-width and the model should, therefore, not inherit the limitations of the platform that it is simulated on.

When the co-simulation API is used LIBSME operates as a "puppet", being controlled by the calling program (the "puppeteer"). Only buses declared with the `exposed` (Section 3.2) modifier in SMEIL are accessible through the LIBSME API. The calling library drives forward the simulation by calling a function for ticking the clock and for reading and writing from/to the exposed buses of a SMEIL program. In the following section, we give a detailed introduction to the API. For brevity, we refer to an SME library implementing the API as the "client" in the following.

The approach presented here is conceptually similar to the Verilog Procedural Interface (VPI) [16] which is used for interfacing with Verilog and VHDL simulators.

However, a big advantage of the SMEIL approach is that SME is used on both sides of the co-simulation. Hence, both the functional and verification parts of the network act as a single unified entity. Thus, the programmer does not need to consider integrating different abstract interfaces.

4.2 API reference

This section documents the public API of the co-simulation interface of LIBSME. Towards the end of the section, we give a short overview of how the API is used.

Exported data structures

```
typedef enum Type {
  SME_INT,
  SME_UINT,
  SME_FLOAT,
  SME_DOUBLE,
  SME_BOOL
} Type;

typedef struct SMEInt {
  int len;
  int alloc_size;
  int negative;
  char* num;
} SMEInt;

typedef struct Value {
  Type type;
  union {
    bool boolean;
    SMEInt* integer;
    double f64;
    float f32;
  } value;
} Value;

typedef struct ChannelRef {
  char* bus_name;
  char* chan_name;
  Type type;
  Value* read_ptr;
  Value* write_ptr;
} ChannelRef;

typedef struct BusMap {
  int len;
  ChannelRef** chans;
} BusMap;
```

The data structures listed above constitute the primary interface for exchanging data with LIBSME.

Public API

`SmeCtx* sme_init()`
: Initializes and returns the SME library context.

`bool sme_open_file(SmeCtx* ctx, const char* file, int argv, char** argc)`
: Loads an SMEIL file, while applying the supplied arguments to LIBSME.

`bool sme_has_failed(SmeCtx* ctx)`
: Returns true if an operation within the LIBSME library failed.

`char* sme_get_error_buffer(SmeCtx* ctx)`
: Returns a string containing the error message emitted by LIBSME. The memory pointed to may not be freed except by calling the `sme_free` function.

`void sme_free(SmeCtx* ctx)`
: Frees the SME library context and related resources.

`bool sme_tick(SmeCtx* ctx)`
Ticks the clock of an SME simulation synchronously. When this function returns, all processes defined within LIBSME will have run and written to their buses.

`bool sme_finalize(SmeCtx* ctx)`
Finalizes a simulation and dumps the recorded trace file (if any) to the file system. This function should always be called following the final call to `sme_tick`.

`bool sme_propagate(SmeCtx* ctx)`
Propagates the values of both internal and external facing buses defined in LIBSME. Run this function before the clock is advanced (by calling `sme_tick`) in the simulation loop and it should be run together with any bus propagation that needs to be performed by the calling code. When this function returns, the values of all buses defined within LIBSME have been propagated.

`void sme_integer_resize(SMEInt* num, int len)`
When manipulating values of type `SMEInt` (arbitrary-size integers) the `sme_resize_integer` function will make sure that the memory pointed to by `Value.value` is large enough to hold the number that you intend to store. The function takes a pointer to the `SMEInt` structure and a parameter len which is the number of bytes needed to store the number as base-256. This function must be called before every direct manipulation of `SMEInt.num`. For a safer interface, see `sme_integer_store`.

`void sme_integer_store(SMEInt* num, int len, const char val[])`
Stores the base-256 representation of an integer in an `SMEInt`.

`void sme_set_sign(SMEInt* num, int sign)`
Sets the sign of an `SMEInt`. Possible values for `sign` are 0 meaning the number is positive and 1 for a negative value.

`BusMap* sme_get_busmap(SmeCtx* ctx)`
Returns a pointer to a `BusMap` structure containing the exposed buses of the SME network. This function is intended to be used by implementers of LIBSME to generate internal representations of their SME buses. It is the callers responsibility to free the memory returned by the function by calling `sme_free_busmap`.

`void sme_free_busmap(BusMap* bm)`
Frees a BusMap structure allocated by `sme_get_busmap`.

Using the API

The initial interaction with the API happens through a call to the `sme_init` function. The `SmeCtx` pointer returned is an opaque reference to the SME library context which must be included in every future interaction with the library. The next step is to load a SMEIL program by calling `sme_open_file`. Then, the implementing client must retrieve a *busmap* from LIBSME in order to learn how to access the `exposed` buses of the SMEIL program. This is done by calling the `sme_get_busmap` function. As seen, it returns a pointer to a `BusMap` struct which then again points to an array of `ChannelRef` structs. Each of these contains an individual bus channel. In addition to the type of the channel, a `ChannelRef` also contains two pointers to `Value` structs.

One for the reading- and writing-end of the channel respectively. Reading to- and writing from the bus channel is done by directly accessing these `Value`s. A `Value` consists of the actual value plus its type. The only value which is not simply represented using a native C type is `SMEInt` which, as seen, contains a `char*`. This is a pointer to the memory used for storing the individual bytes of a base-256 integer. Before this memory is written, the `sme_integer_resize` function must be called to ensure that the memory is large enough to contain the number that the client intends to store.

After each of these calls, the `sme_has_failed` must be called in order to check if the previous operation resulted in an error. If the function returns `true`, the `sme_get_error_buffer` function returns a pointer to a buffer containing a human-readable description of the error which occurred.

In the main simulation loop of the implementing client library the following calls to the API are usually made: In the bus propagation phase, the client must propagate its own buses and then call the `sme_propagate` function to propagate the buses defined the SMEIL program. Following bus propagation, the client library must run its own processes and the processes defined in the SMEIL program. The processes defined in the SMEIL program are run by calling the `sme_tick` function.

When the simulation is complete and the desired number of cycles has been performed, before freeing memory, a final call to `sme_finalize` must be issued in order to dump the recorded trace file to disk.

The intention of this API is that it should be as general as possible such that it can be implemented by any language providing an adequate Foreign Function Interface (FFI) for C APIs.

4.3 Co-simulation using PySME

In order to show a client implementation of the API described above, we have extended the PySME library [6] with support for co-simulation enabling seamless interaction between SME networks written in Python and SMEIL. In practice, extending the PySME library was straightforward and required less than a day of implementation work by a person with expert knowledge of both code-bases. We expect that a similar effort is required to extend other SME implementations (such as C# SME and C++ SME).

As an example, we revisit the ADDONE example introduced in Section 3.5, this time implementing one half of it in Python as seen in Figure 4.1. The `@extends` decorator is all that is needed to make the buses exposed from the SMEIL network available for the Python program. Behind the scenes, LIBSME is loaded and the `addone.sme` file is parsed, typechecked and the LIBSME SMEIL simulator is initialized. We gave a detailed description of this interaction in the previous section. A SMEIL-defined bus is referenced from PySME by instantiating an `ExternalBus`, providing the name of a bus defined in the SMEIL program as its parameter. The semantics of an `ExternalBus` is identical to those of a bus defined within Python. Any SMEIL type except strings and arrays may be passed along a bus. Strings play a very limited role in SMEIL and are therefore unlikely to be supported. Arrays, on the other hand, are desirable to include in future extensions. Integers are encoded in base-256 as a sequence of bytes, allowing arbitrarily-sized integers to be used between co-simulated entities.

When this program is run, the PySME library calls the LIBSME library for every cycle to stepwise progress the simulation. During the simulation LIBSME may, if asked to do so, record a trace of the communication taking place over the buses to a file. This

```
proc addone(in inbus, const val)
    exposed bus addout {
       val: i32;
    };
{
    addout.val = inbus.val + val;
}

network addone_net() {
    exposed bus idout {
        valid: bool;
        val: i32;
    };

    instance addone_inst of
        addone(idout, 1);
}
```

(a) The SMEIL code in `addone.sme`.

```
from sme import *

class Id(SimulationProcess):
  def setup(self, ins, outs, result):
    self.map_outs(outs, "out")
    self.map_ins(ins, "inp")

  def run(self):
    result[0] = self.out["val"]
    self.out["val"] = self.inp["val"]

@extends("addone.sme", ["-t", "trace.csv"])
class AddOne(Network):
  def wire(self, result):
    plus_out = ExternalBus("addone_inst.addout")
    id_out = ExternalBus("idout")
    p = Id("Id", [plus_out],
      [id_out], result)
    self.add(plus_out)
    self.add(id_out)
    self.add(p)

if __name__ == "__main__":
  sme = SME()
  result = [0]
  sme.network = AddOne("", "AddOne",
                       result)
  sme.network.clock(100)
  print("Final result was ", result[0])
```

(b) The corresponding Python code.

Figure 4.1: Example code showing the an interaction between SMEIL (left) and PySME (right).

trace file is then later used as the data source for the VHDL test bench which is used to verify the generated VHDL code.

This is a highly flexible model as co-simulation is enabled with minimal intrusion on existing PySME code. For example, should LIBSME be extended with a high-performance simulation backend for SMEIL, existing programs can take advantage of this without modifications. The implementation of LIBSME may even be replaced entirely, as long as the current API is maintained. Furthermore, it also facilitates an incremental design strategy, where a Python prototype can gradually be rewritten in SMEIL.

Notice in the PySME code of Figure 4.1 that the `addout` bus is referenced through its instance name `addone_inst.addout` and the `idout` bus is referenced directly by its name. This is because the latter bus is declared directly inside the top-level entity while the former is declared within the `addone` process which is instantiated as `addone_inst`. This naming scheme is used to ensure that exposed buses have unique names.

We show more examples of using co-simulation for testing SMEIL networks in Chapter 6.

```
proc A ()
 bus b {
  chan: int;
 };
 var c: i10;
{
 c = b.chan;
}
```

(a) Unconstrained types.

```
proc A ()
 bus b {
  chan: i6 range 0 to 29;
 };
 var c: i10;
{
 c = b.chan;
}
```

(b) Valid.

```
proc A ()
 bus b {
  chan: i16 range 0 to 30717;
 };
 var c: i10;
{
 c = b.chan;
}
```

(c) Invalid.

Figure 4.2: Shows a process entering the simulator with an unconstrained type (a) and examples of two possible resulting programs (b, c). The type changing between the examples is underlined.

4.4 Typing networks through simulation

As described in Section 3.3 SMEIL supports integers of both constrained and unconstrained bit-widths. In order to translate SMEIL to a hardware description, we require that all types in the program are constrained to a specific bit-width. In a hardware description, we need to statically specify the number of bits required by each value and therefore, arbitrary-width integers are not representable. However, it is often difficult to decide the optimal bit-width of a value in advance. In particular, this applies to internal variables whose values are derived from external inputs. To address this, LIBSME provides a method for re-typing a SMEIL program based on values observed during simulation.

When this feature is enabled, the maximum absolute value assigned to a variable is stored alongside its current value. Whenever the variable is assigned, the new value is compared to the current maximum which is then updated as needed.

When the simulation is concluded, the observed value ranges are converted to SMEIL types large enough to hold the range. For example, in the program shown in Figure 4.2b, we observed that the bus channel `b.chan` was assigned values between 0 and 29 during simulation. Therefore, the bus channel is assigned the type `i6` as we need 6-bits to hold the value 29 in a signed integer.

The types and observed ranges are spliced into the SMEIL AST and the re-typed program is then passed through the type checker. This ensures that constraints originating from fixed-size types in the original program are not violated. This process is illustrated in Figure 4.2 which shows how observationally derived types are spliced into an existing program. Figure 4.2c shows the violation of an existing constraint in the program. Since the value `c` has the fixed-sized type `i10`, the program will no longer be valid if `b.chan` observes values that are 16-bit long. A configuration flag `--no-strict-type-bounds` overrides this behavior by considering all types as unconstrained (i.e., `i10` is considered identical to `int`).

As seen, it is possible to mix types with constrained and unconstrained bit-widths, This is useful as we often know the possible range of external buses. Determining the ranges of values that derive from those buses are not always as easy. Therefore, we can let all internal buses and variables of a program be typed based on observed values while fixing external buses to a specific size. The type system of SMEIL will then enforce that we do not assign a larger dynamically determined value to a smaller fixed external value.

This feature can only be used safely if the following conditions apply: 1) All values deriving from input stimuli must increase monotonically in relation to the value of the input and 2) the testing code must ensure that the whole possible range of input stimuli is exhausted by test benches.

To allow easy examination of the types derived from value observations without having to examine the generated source code, LIBSME is able to display the SMEIL program with the modified types in place. That is if the program shown in Figure 4.2a was simulated, the resulting code shown in Figure 4.2b or Figure 4.2c would be what is actually shown to the user.

4.5 Alternative approaches

We considered a couple of alternative approaches before settling on this final design. As written in the introduction to this chapter, co-simulation was introduced as a method for providing external test inputs to SMEIL programs. Instead of adding the API for performing co-simulation, we could simply require that all SMEIL programs expecting external inputs would take these inputs from a CSV trace-file generated by simulating an equivalent SME network. In this scenario, the starting point would be a complete SME network implemented in, for example, PySME. Simulating this network would generate a trace-file containing a recording of values sent over buses. Then, the hardware-targeted processes of the PySME model would be translated to SMEIL and the recorded trace-file would be used for providing input to the SMIEL model.

Implementing this model would require less work, but it would only support using SMEIL as an intermediate language. The reason for this is the following: in order to generate the required trace file, a complete implementation of the SME network in a single language is required, thus, it would not be possible to implement a part of the network separately in SMEIL. Furthermore. the co-simulation model delegates the responsibility of generating the trace-file to LIBSME. Since LIBSME is also responsible for generating the final VHDL code, this makes it simpler to ensure that names and order of fields in the CSV file match those expected by the generated VHDL test bench. If the trace file was generated externally, it would also prevent changing the LIBSME implementation in a way which altered the naming and ordering of fields in the CSV files. Thus, the co-simulation model is advantageous even when SMEIL is used as an IL, and without it, SMEIL would not be usable as a primary implementation language at all.

External library generation. Insetad of implementing an interface to the simulator within LIBSME, we considered the alternative approach of generating an external library (for example in C++) and expose an API similar to the current. However, instead of loading the LIBSME library and asking it to load an SMEIL program, we would load the generated library directly. The advantages of this would be:

1. We would get two birds with one stone by both providing a method for co-simulation and a C++ code generator.

2. The SMEIL code would be compiled to native code and therefore execute significantly faster.

3. A library generated from a SMEIL program would be significantly smaller than all of LIBSME and could be distributed independently.

However, the disadvantages would be:

1. More complex client library integration: A library performing co-simulation with SMEIL would first need to compile the SMEIL code to a library, then load the library. We could handle this from LIBSME, but that would diminish the third advantage.

2. The implementation of observationally-derived typing (Section 4.4) would become more complicated as the observed types would need to be communicated back to LIBSME in order for them to be used in the code generated from the SMEIL program.

API considerations. The second consideration made was how to actually expose the API. As an alternative to the current C API, a web-based REST-style API was also considered. To implement this, LIBSME would contain a web-server which would listen to requests from a client library. The advantage of this approach would be that web-APIs are more ubiquitous that C-style APIs, and thus, they may be able to support a wider range of clients. On the other hand, we were concerned with the performance of such an approach as issuing an HTTP request carries a significantly higher overhead than performing a platform level C-call. Another concern with the currently chosen approach was whether it was sufficiently platform-neutral. However, we feel reasonably confident that the current approach is supported on all major platforms, although only Linux has been tested.

5

libsme design and implementation

In this section, we present the combined implementation of LIBSME and elaborate on implementation details.

5.1 Methods of interaction

SMEIL programs are run using the `libsme` library either through interaction with the C API of the library or by using the provided command line utility.

Direct code generation. A SMEIL program that contains only size-constrained types provide all the information that is needed to generate a hardware description. Therefore, VHDL code can be generated directly from a SMEIL program without the intermediate simulation step. Some advantages are lost when using this mode, as no test bench is created and the generated VHDL code must be manually modified and connected to a clock source for driving the simulation before it can be tested using a VHDL simulator.

Pure SMEIL simulation. This mode only applies to SMEIL networks which contain their own data generation process (see Section 6.2 and Section 6.5 for examples of such networks). SMEIL used like this is not very useful as it can only produce an output through `trace` statements (Section 3.2).

Co-simulation of SMEIL. The most common intended usage scenario for SMEIL is to use it together with an SME library for a general purpose language. This allows the generation of VHDL code, an associated test bench, and a trace file. The full details of the co-simulation interface of SMEIL were previously given in Chapter 4.

5.2 An overview

In the previous sections, we have described the individual parts of LIBSME without describing the integration of its components. Hence, we devote a section for that purpose here. An overview of the LIBSME library and its interactions is shown in Figure 5.1 and the individual stages are described below.

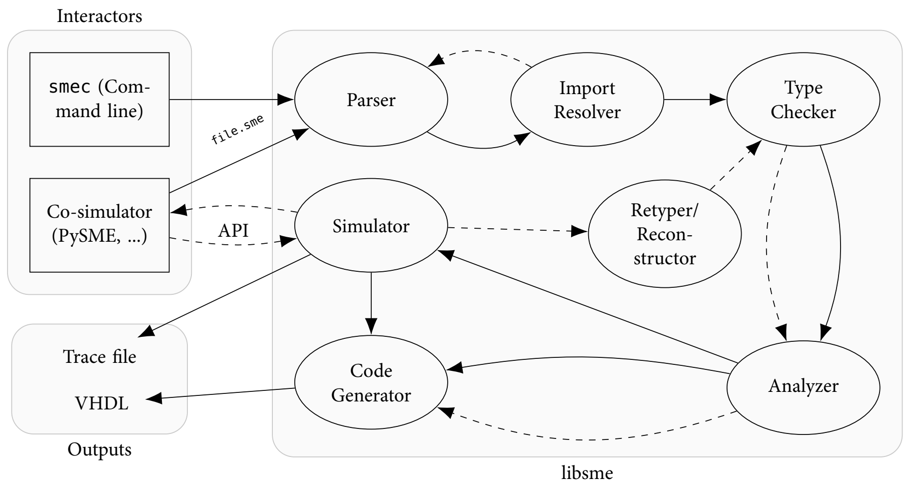

Figure 5.1: Overviews of interactions with and data flow within LIBSME. The dashed lines denotes paths which are followed conditionally depending on which mode LIBSME is executed in.

Parsing and Import Resolution

Regardless of how LIBSME is invoked (as described in the previous section) the SMEIL source is parsed and the resulting AST is passed through the import resolver. Here, the code is scanned for the presence of `import` statements. If any are found, the source files containing the imported modules are parsed in a recursive manner. Whenever we recursively import a module, we pass along the list of imported entities such that only the requested entities are imported. The tree of imported modules is then flattened by renaming hierarchical references. This process seeks to simplify the subsequent phases of the compilation process as module hierarchies do not have to be considered. The renamings are tracked and passed on to the following stages so that a reverse mapping may be performed later, for example for error messages.

Type Checking

The code is then passed through the type checker which enforce the typing rules described in Section 3.3. The type checker makes two passes through the code:

- The first pass locates all entity definitions (processes and networks) and adds them to the top-level symbol table. For every entity found, the declarations in that entity are added to a local symbol table which is associated with the entity.
- The second pass performs type checking on all declarations and statements in the previously discovered entities. During this process, the individual AST-nodes are annotated with their types. Having such type information available throughout the AST is of significant value for subsequent passes, such as code generation and simulation, since they are able to determine the type of an expression at any time by looking at its AST node.

The two-pass approach ensures that declarations can be given in any order. Requiring declarations to be made ahead-of-use would make the code shown in many of the examples shown throughout this thesis become significantly more convoluted.

A single abstract representation of SMEIL is used throughout the compiler. Code simplifications could be made if an intermediate representation of SMEIL was used by the internal stages of the compiler. However, introducing such an intermediate representation would limit our ability to reconstruct the original SMEIL source code following re-typing (Section 4.4). Furthermore, maintaining an unchanged representation of the original source code means that the generated code more closely corresponds to the source code.

Analysis

The analysis phase examines the structure of a network. This is used for determining the top-level entity of the network which is needed both for deriving a runtime representation of the network and for subsequent code generation. From here, the AST may take two paths depending on the mode of invocation requested by the user. It is either passed on directly to the code generator, or simulated. If the AST was already retyped by the simulator, it is passed directly to the code generator.

Simulation

Simulation is performed to test a design. During the simulation, the value ranges assigned to every variable and bus channel are tracked such that we can use them for constraining integer types. Furthermore, the values of external-facing buses are logged and used to construct the CSV trace file used by the generated VHDL test bench. The simulator also performs accurate emulation of integer overflows. During simulation, if LIBSME is used for co-simulation with another SME network, it will exchange the values of external-facing buses with another SME network. After simulation, the AST may either be passed directly to the code generator or, if new types were assigned, returned to the type checker.

In very early phases of this project, we considered if implementing a simulator for SMEIL was even needed. After all, if SMEIL is used as an intermediate language, the source SME network could be simulated directly leaving SMEIL to be used purely for code generation. In this scenario, the trace file used for the test bench would simply be passed along with the SMEIL intermediate code and used for providing input to the generated VHDL test bench. However, we determined that without a simulator, SMEIL would be restricted to this particular use case only.

Code generation

The final stage, yielding the desired output, is the code generation phase which, as its name suggests, turns the typed and possibly simulated SMEIL AST into VHDL code.

SMEIL compiles to clean and readable VHDL code which is amenable to manual modifications. The code may be executed using a VHDL simulator or passed to FPGA vendor tools for synthesis and subsequent hardware-implementation (as described in Section 2.1). The generated code is a cycle-accurate representation of the original SMEIL network.

The fundamental structure of the SMEIL code is preserved in the generated VHDL code. One VHDL entity is generated per SMEIL entity and the body of an SME process is transformed into a VHDL `architecture` containing a single sequential process. For each of these processes, we also generate code for performing an asynchronous reset of all variables and outgoing signals. The naming hierarchy of the original SMEIL is preserved, making it easy to identify from where a particular section of the VHDL code was generated. In Section 6.3 we show an example of how a SMEIL process is transformed into an FPGA entity.

For verifying the generated VHDL code, a test bench is also generated. The test bench is a VHDL program which connects to the `exposed` buses of the SMEIL program. The CSV-trace file, containing the values recorded during simulation, is used by the VHDL test bench to drive inputs and verify outputs.

Alongside the generated code, a `Makefile` is generated for building and testing the VHDL code using the GHDL [19] simulator.

Integer types of SMEIL are represented in VHDL using the types provided by the standard `ieee.numeric_std` package. This package provides functions for performing signed and unsigned integer arithmetic with logic-vectors. For example, the types `i4` and `u12` are represented as `signed (3 downto 0)` and `unsigned (11 downto 0)` respectively.

Arrays require the creation of a new `type` in VHDL. A `type` declaring a 10-element array of 5-bit signed integers (`[10]i5` in SMEIL) is represented in VHDL as

```
type \[10]i5\ is array (0 to 9) of unsigned (4 downto 0);
```

These type declarations are stored in a separate package, `sme_types.vhdl`, which is shared between all entities of the design to avoid cluttering the generated code with duplicated declarations. SMEIL booleans are represented using the VHDL type `boolean`. As an alternative to this, a single `std_logic` type is commonly used. This type represents a wire in the hardware and is, therefore, able to have other states than just true or false. This may be useful in some circumstances.

The actual code is generated using the `language-vhdl-quote` library [4] which provides quasiquoters [29] for building VHDL ASTs using the concrete VHDL syntax. The major advantage of this approach is that it minimizes the chance of generating syntactically invalid VHDL since syntax errors are caught during compilation of LIBSME. Furthermore, a complete VHDL AST is constructed containing the contents of each generated VHDL file. This AST is then pretty-printed, yielding consistently formatted code which is difficult to achieve using more common techniques based on string templates.

Reconstruction

If observation based typing was enabled, the simulator will have annotated the SMEIL AST with types based on the observed values. By reconstructing a structure resembling the original AST, reusing the stages of the compiler is simplified. Furthermore, the results of the retyping are shown to the user using nicely formatted concrete SMEIL syntax.

5.3 Software-engineering considerations

The language chosen for implementation of LIBSME is Haskell. It would have been possible to carry out the implementation in any general-purpose language, but Haskell was chosen in particular because:

- Functional programming languages are well suited for writing compiler-related software, due to their support for Algebraic Data Types (ADTs) and pattern-matching. Also, a wide range of libraries exists for supporting the implementation of for example parsers and pretty-printers.

- The type-safety of Haskell trades a slightly slower development pace for a significant reduction of time spent debugging.

- The type system also significantly aids refactoring, something which proved useful several times while developing this project. When a data structure is changed in a Haskell program, the type system ensures that compile-time errors are raised for code affected by the change.

LIBSME comprises just short of 6000 SLOC of Haskell. Additionally, the wrapper module for holding the co-simulation state and neatly exposing the functions of the C-API is implemented in a module is approximately 500 SLOC of C. The VHDL parsing and quasiquotation library developed for use with this project consists of approximately 5500 SLOC of Haskell.

The implementation currently has several rough edges, but its fundamental structure is sane and it has been written with future extensions in mind. It also pays particular attention to usability-related features such as providing understandable error messages.

6

Evaluation

In this chapter, we first present an example of SMEIL used as an IL followed by four small examples implemented in SMEIL: A model clock using a 7-segment display, the core of a trading chip, a process for binning colors based on intensity and finally an MD5 hash bruteforcer.

Information on how to reproduce the runs shown below is given in Appendix A.

6.1 SMEIL as an intermediate language

We have made repeated references to the origins of SMEIL as a pure IL and described how and why the scope of the language was expanded to also include the use as an independent implementation language. Despite this, SMEIL is still very much intended to be also usable as an IL. As it may be obvious at this point, the design, implementation and testing have mostly focused on its use as a primary implementation language, with the IL angle remaining in the background. Ideally, we would have developed a code generation backend for C# at this point, targeting SMEIL, since C# SME is the most complete SME implementation. However, this was not possible within the available time-frame and in any case, the implementation would need to be carried out by a third party. Thus, we considered it to be outside the scope of the thesis.

To show that using SMEIL as an IL *can* be done, we have adapted our previously implemented Python to VHDL compiler to generate SMEIL instead of generating VHDL directly. This translation is not yet fully automated, but we explain how that could be done feasibly.

As an example, we will show an SME network, called SomeOps, is translated from PySME to SMEIL. This implementation was first introduced in [7] and is presented here with very minor modifications. The network, shown in Figure 6.1, is configured as follows: A generator process continuously emits two numbers onto a shared bus. Two processes are connected to the reading end of the bus which will add and multiply the numbers, respectively. The results of the calculation is then printed by the Printer process. The PySME code of the network is shown in Figure 6.2a. The code uses a slightly older version of the PySME library than used elsewhere in this thesis which has some minor differences, for example, the `self.tell` function which was renamed to `self.add`. SME processes are declared using a class extending one of the classes `External` or `Function`. The two classes are semantically identical, but are used to indicate if a process is intended purely for simulation or for hardware synthesis,

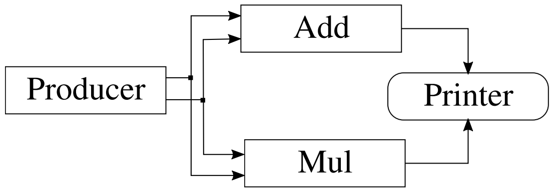

Figure 6.1: Schematics of the SomeOps network. Figure from [7].

respectively. Thus, `External` processes are expected to use features not representable on hardware. The translator handled these constructs by simply generating empty entity declarations in their place.

The generated SMEIL code is shown in Figure 6.2b. A `trace` statement was manually added to the `Printer` process for printing the calculation result. We can observe that it is possible to transform the PySME program to SMEIL in a straightforward manner. Different from most of the other SMEIL examples shown elsewhere, this code use a different method for creating connections between processes where both input and output buses are explicitly passed as parameters to the processes. This is the only possible way of connecting processes in PySME (as described in Section 2.1). Thus, our idea of providing a high-degree of flexibility in how network connections are created (discussed in Section 3.2) allows this PySME example to be translated without any structural transformations.

For this process to be automated all that is needed is

- Instead of converting `External` processes to empty processes in SMEIL, they should be omitted from the generated code.
- The PySME library should be able to replace buses going between `External` and `Function` processes in the Python program with connections to the `exposed` buses of the generated SMEIL code.

With these two changes in place, PySME programs will be able to use SMEIL as an intermediate target for generating VHDL and to create test-files through the co-simulation interface.

6.2 7-Segment Display

The example implements a model of an old-fashioned digital clock displaying the current time using 6 7-segment digits. The layout is depicted in Figure 6.3. The timer process continuously increments a numeric value, representing the number of seconds passed since the beginning of the day, which is stored in a process-local variable. For every cycle, the current number of seconds is emitted. This number is then broadcasted through a shared bus to a number of calculating processes which uses simple integer arithmetic to calculate the number of hours, minutes and seconds respectively. To better reflect an actual hardware implementation, `encoder` and `decoder` processes are inserted on the wire leading to the digit. They, respectively, encodes to and decodes from the bit-pattern (represented as hexadecimal numbers in the `encode` and `decode` processes) used to light up parts of the 7-segment display.

For representing the current time during simulation, the decoder processes are connected to a process which prints the current time in a readable format. The code for the process, with elisions, is shown in Figure 6.4.

```python
from sme import *
t = Types()

class Producer(Function):
  def setup(self, ins, outs):
    self.map_outs(outs, "outp")
    self.v1 = 0  # type: t.u7
    self.v2 = 0  # type: t.u7

  def run(self):
    self.outp["val1"] = self.v1
    self.outp["val2"] = self.v2
    self.v1 += 1
    self.v2 += 1
    if self.v1 > 100:
      self.v1 = 0
      self.v2 = 0

class Add(Function):
  def setup(self, ins, outs):
    self.map_ins(ins, "valbus")
    self.map_outs(outs, "addbus")

  def run(self):
    self.addbus["res"] = self.valbus["val1"] +
             self.valbus["val2"]

class Mul(Function): # Snipped (similar to Add)

class Printer(External):
  def setup(self, ins, outs):
    self.map_ins(ins, "addbus", "mulbus")

  def run(self):
    print(self.addbus["res"],
          self.mulbus["res"])

class SomeOps(Network):
  def wire(self):
    valbus = Bus("ValueBus", [t.u7("val1"),
                              t.u7("val2")])
    valbus["val1"] = 0
    valbus["val2"] = 0
    addbus = Bus("AddBus", [t.u8("res")])
    addbus["res"] = 0
    mulbus = Bus("MulBus", [t.u14("res")])
    mulbus["res"] = 0
    prod = Producer("Producer", [], [valbus])
    add = Add("Add", [valbus], [addbus])
    mul = Mul("Mul", [valbus], [mulbus])
    printer = Printer("Printer",
                      [addbus, mulbus], [])
    self.tell(printer) # 6x self.tell snipped
# Main function snipped
```

(a) Original Python SME code.

```
sync proc Add (in valbus, out addbus)
{
    addbus.res = valbus.val1 + valbus.val2;
}

sync proc Mul (in valbus, out mulbus)
{
    mulbus.res = valbus.val1 * valbus.val2;
}

sync proc Printer (in addbus, in mulbus)
{
    // Manually added
    trace("Add result: {} Mul result: {}",
        addbus.res, mulbus.res);
}

sync proc Producer (out outp)
    var v2: u7 = 0;
    var v1: u7 = 0;
{
    outp.val1 = v1;
    outp.val2 = v2;
    v1 = v1 + 1;
    v2 = v2 + 1;
    if (v1 > 100) {
        v1 = 0;
        v2 = 0;
    }
}

network SomeOps ()
{
    exposed bus AddBus {res: u8;};
    exposed bus MulBus {res: u14;};
    exposed bus ValueBus {val1: u7;
                          val2: u7;};
    instance Add of Add(ValueBus, AddBus);
    instance Mul of Mul(ValueBus, MulBus);
    instance Printer of Printer(AddBus, MulBus);
    instance Producer of Producer(ValueBus);
}
```

(b) Generated SMEIL code.

Figure 6.2: Example of PySME code automatically translated to SMEIL.

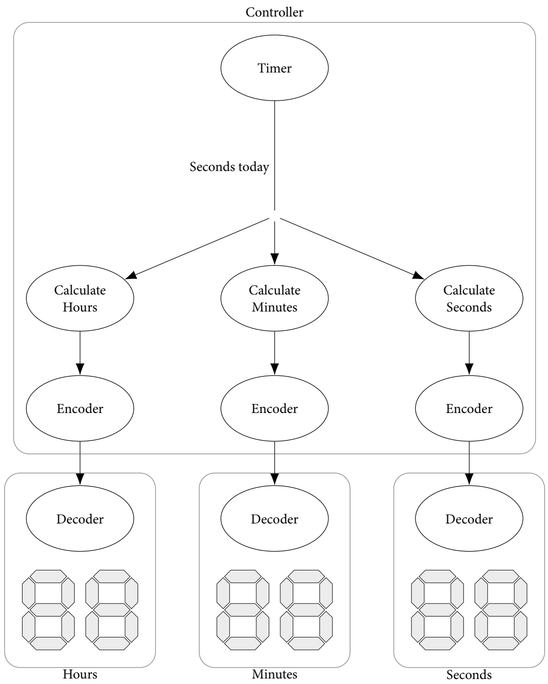

Figure 6.3: Model digital clock using a 7-segment display. A timer keeps track of the number of seconds elapsed since midnight and several processes calculates and lights. [a]

[a] 7-segment digit rendering based on "7 segment display labeled" (`https://commons.wikimedia.org/wiki/File:7_segment_display_labeled.svg`) by h2g2bob. CC BY-SA-3.0.

A real-world implementation of this design is simple to imagine: The timer process is replaced by an actual time-keeping device and the output of the encoder processes is connected directly to the 7-segment digits they drive. This network is implemented purely in SMEIL without depending on external processes for stimuli.

Since this example contains its own data generation source it is run using the command-line interface of LIBSME. The result of executing this network is shown below.

```
$ smec -i 7seg.sme -s 50
00:00:00
00:00:00
00:00:00
00:00:00
00:00:01
00:00:02
00:00:03
[..]
```

Note that the depth of the network is visible in that it takes a few cycles before the first time has propagated to the printer.

```
proc timer ()
  bus elapsed {
    secs: uint;
  };
  const secs_per_day: uint = 86400;
  var cur: u17;
{
  cur = (cur + 1) % secs_per_day;
  elapsed.secs = cur;
}

proc hrs (in time)
  bus vals {
    d1: uint;
    d2: uint;
  };
  const secs_per_hr: uint = 3600;
  var cur: uint;
{
  cur = time.secs/secs_per_hr;
  vals.d1 = cur/10;
  vals.d2 = cur%10;
}

// [..]

proc encode (in inval)
  bus vals {
    d1: uint;
    d2: uint;
  };
  const digits: [10]uint =
    [0x7E, 0x30, 0x6D,
     0x79, 0x33, 0x5B,
     0x5F, 0x70, 0x7F,
     0x7B];
{
  vals.d1 = digits[inval.d1];
  vals.d2 = digits[inval.d2];
}

proc decode (in inval)
  bus vals {
    d1: uint;
    d2: uint;
  };
{
  switch inval.d1 {
    case 0 {vals.d1 = 0; }
    case 0x7E { vals.d1 = 0;}
    // [..]
    case 0x7B { vals.d1 = 9;}
    default { assert(false); }
  }
  //[..]
}

proc disp (in val1, in val2, in val3) {
  trace("{}{}:{}{}:{}{}",
    val1.d1, val1.d2,
    val2.d1, val2.d2,
    val3.d1, val3.d2);
}

network clock() {
  instance t  of timer();
  instance h  of hrs(t.elapsed);
  instance ench of encode(h.vals);
  instance dech of decode(ench.vals);
  // [..]
  instance _ of disp(dech.vals,
                     decm.vals,
                     decs.vals);
}
```

Figure 6.4: Code of the 7-segment digital clock network.

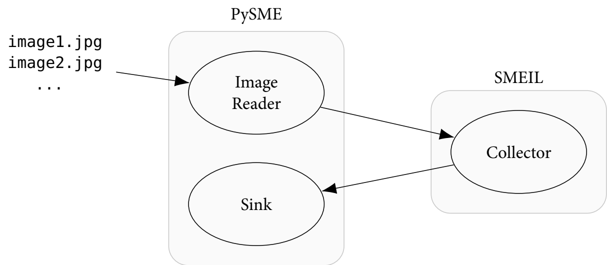

Figure 6.5: The process network of ColorBin.

6.3 ColorBin

This network, named ColorBin (ported from the C# version in [38]), serially process the pixels in one or more images and categorize their intensity as low (closer to black), medium and high (closer to white). The generator reads images from the disk and separates each of their pixels into RGB components. The input bus also contains a boolean signal which is true along with the last pixel of each image, acting as a token to signify the end of an image. That way, the collector process (described next) can tell the images apart and reset its counters when a new image begins. The collector process examines each pixel, incrementing one of three intensity counter variables as appropriate. When it receives the last-pixel token, it sends the stored values of the intensity counters to the sink process which then collects the pixel intensity counts for each image.

The SMEIL source code for the collector process is shown in Figure 6.6. The VHDL code generated for the process is shown in Figure 6.7. The mapping of names and structure from SMEIL to VHDL is clearly seen, as is the immense verbosity of VHDL. When we generate expressions, we set parentheses in a very pessimistic manner. To ensure that the precedence of operators in SMEIL is preserved in VHDL, we set parentheses around every binary operation. Unfortunately, this adds some clutter to the generated code in the form of unnecessary parentheses. This matter is subject to future improvements, for example, by implementing the "unparsing" method described in [31] which, based on knowledge about operator precedence in the target language, reverse-transforms an AST using as few parentheses as possible.

This SMEIL network gets its data through co-simulation with a PySME network. Therefore, it is executed by running the Python-end of the network. As input to this network we provide three separate images. As output, it simply prints the pixel intensity statistics. In Section 6.6, we provide a benchmark comparing the time required for running this network on the three images both by simulating the original SMEIL code and the generated VHDL code.

6.4 High-frequency trading chip

We revisit an example from [7]. In high-frequency trading, a split-second decision needs to be made whether to buy, or sell, a stock. Reducing latency is paramount as you need to make transactions as fast as possible. This problem is, therefore, an interesting target for custom hardware as the intractable latencies induced by general-purpose hardware and software-implemented decision-making logic are avoided. The real-time value of a stock is passed through two calculator processes. Both calculate the

```
proc collector (in image_input)
  exposed bus bin_count_out {
    valid: bool;
    low: u32;
    med: u32;
    high: u32;
  };
// [..]
{
  if (image_input.valid) {
    color = ((image_input.R * 299) +
      (image_input.G * 587) +
      (image_input.B * 114)) / 1000;

    if (color > thresh_high) {
      counthigh = counthigh + 1;
    } elif (color > thresh_med) {
      countmed = countmed + 1;
    } else {
      countlow = countlow + 1;
    }
  }

  bin_count_out.low = countlow;
  bin_count_out.med = countmed;
  bin_count_out.high = counthigh;
  bin_count_out.valid =
    image_input.valid &&
    image_input.last_pixel;
}
```

Figure 6.6: SMEIL source code for the collector process of the ColorBin network.

exponential moving average of a stock, one using long decay and the other using short decay. The trading decision is based on detecting when the two averages cross. [27]

The network is shown in Figure 6.8 and the SMEIL source in Figure 6.9. The results of the two calculator processes described above is passed through the merge process which combines the long and short averages into a single bus. The core of the trader is written in SMEIL, while the processes providing input stimuli and data collections is written in Python as a PySME model. The input data is generated using a Brownian bridge [20] which is a stochastic process commonly used as a model for simulating realistic stock price developments. The results are collected and visualized in a graph for easy verification.

In an actual trading chip, the data generator is replaced with actual stock prices arriving through a network interface and the plot is replaced by market transactions. Both the testing and verification processes leverage existing Python libraries. The Brownian bridge generator is implemented using NumPy while the plot is made using `matplotlib`. Implementing these test processes in VHDL would be a massive undertaking, with Python, it is quite simple.

We execute this example by calling the Python script containing the PySME part of the implementation which will then execute together with the part of the network written in SMEIL via the co-simulation interface. The Python part of the network shows a graph visualizing the moving averages.

The test bench generated along with the VHDL code runs without errors and prints the message

```
ewma_tb.vhdl:166:13:@1275001275ns:(report note): completed
  successfully after 255 clockcycles
```

on completion.

6.5 MD5 bruteforcer

This example is a simplification of a bruteforcer of MD5-hashes developed to showcase the performance of FPGAs in comparison with CPUs and GPGPUs for a trivially parallelizable problem: Bruteforcing an MD5 hash [26]. The layout of the network is shown in Figure 6.10. The generator iteratively emits all combinations of 8 ASCII printable characters as a string. This string is then passed to the hasher, calculating the MD5 sum of the string. In the verifier, the calculated hash is compared to a pre-calculated hash of the input string that we wish to find. The predictiveness of the input generator means that we can ensure that the search terminates quickly by choosing a target string close to the starting string. Hence, short runs can be chosen for testing and long runs for benchmarking.

A complete implementation of this example exists for several targets: CPUs parallelized with OpenMP, OpenCL for GPGPUs, Xilinx HLS and finally C# SME. Both

```
entity collector is
  port (
    signal bin_count_out_valid: out boolean := false;
      -- [..]
  signal bin_count_out_high: out unsigned (31 downto 0) := to_unsigned(0, 32);
  signal image_input_valid: in boolean;
  -- [..]
  signal image_input_B: in unsigned (7 downto 0);
  signal clk: in std_logic;
  signal rst: in std_logic);
end entity collector;

architecture rtl of collector is
begin
  process (clk, rst) is
    constant thresh_high: integer := 200;
    -- [..]
    variable countlow: unsigned (31 downto 0) := to_unsigned(0, 32);
  begin
    if rst = '1' then
      bin_count_out_valid <= false;
      bin_count_out_low <= to_unsigned(0, 32);
      -- [..]
      countlow := to_unsigned(0, 32);
    elsif rising_edge(clk) then
      if image_input_valid then
        color := resize((((((image_input_R *
                             to_unsigned(299, 10))) +
                           ((image_input_G * to_unsigned(587, 10)))) +
                          ((image_input_B * to_unsigned(114, 10))))) /
                        to_unsigned(1000, 10)),  color'length);
        if (color > thresh_high) then
          counthigh := resize((counthigh + to_unsigned(1, 32)),
                             counthigh'length);
        -- [...]
        end if;
      end if;
      bin_count_out_low <= resize(countlow, bin_count_out_low'length);
      -- [..]
```

Figure 6.7: The VHDL code generated from the SMEIL code shown in Figure 6.6.

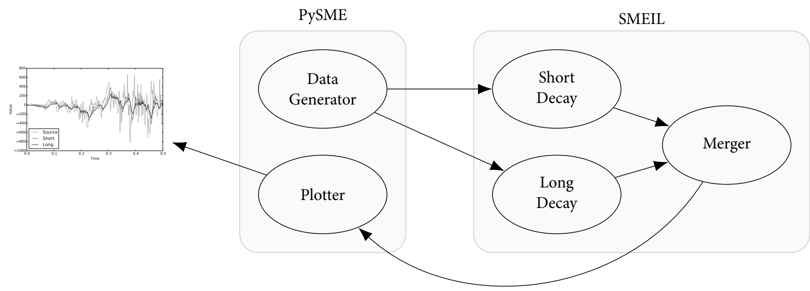

Figure 6.8: The network of simpletrader.

```
sync proc calc (in data, const decay)
  bus result {
    val: int;
    valid: bool;
  };
  var prev: int;

{
  if (data.valid) {
    result.valid = true;
    prev = (data.val >> decay) +
      (prev >> decay) *
      ((1 << decay) - 1);
    result.val = prev;
  } elif (!data.valid) {
    result.val = prev;
  } else {
    result.valid = false;
  }
}

sync proc merge (in long,
                 in short, out res) {
  if (long.valid && short.valid) {
    res.valid = true;
    res.long = long.val;
    res.short = short.val;
  } else {
    res.valid = false;
  }
}

network ewma () {
  const decay1: int = 2;
  const decay2: int = 3;

  exposed bus stream {
    val: int;
    valid: bool;
  };

  exposed bus output {
    short: int;
    long: int;
    valid: bool;
  };

  instance short of calc
      (data: stream, decay: decays1);
  instance long of calc
      (data: stream, decay: decays2);
  instance _ of merge
      (long: long.result,
       short: short.result,
       res: output);
}
```

Figure 6.9: SMEIL source code for the trader core.

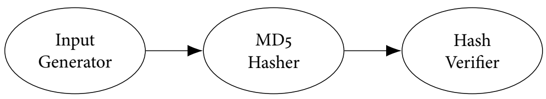

Figure 6.10: Structure of the MD5 bruteforcer network.

the two latter implementations synthesizes and runs on FPGAs. A comparison of these implementations showed that while GPUs were superior in raw performance, the performance-per-watt ratio favored FPGAs by more than an order of magnitude. Furthermore, the SME version is significantly more efficient than the Vivado HLS implementation, which relies on the concurrency inference discussed in the introduction.

This example mainly serves to show an SMEIL implementation of an SME model which has been synthesized to an FPGA. It also showcases an implementation of a non-trivial algorithm (MD5) in SMEIL. In particular, the MD5 algorithm relies heavily on bit-shifting of 32-bit unsigned integers. Therefore, it depends on the correctness of the integer overflow emulation of the LIBSME simulator in order to produce the expected result. The shortened source code of the SMEIL process for calculating an MD5 hash is shown in Figure 6.11. The process receives the string that should be hashed through the bus passed as its `input` parameter. The calculated hash is then sent on the `hashes` bus which is read by the verification process.

Since this network contain its own data generation source this network is run using the command line interface of LIBSME. A session is shown below

```
$ smec -i md5-simple.sme --no-warnings -s 50
[..]
verifying hash 3141438837 2285911344 1677794538 2336503479
verifying hash 1449451250 1644770476 3741827633 1220464617
1449451250 1644770476 3741827633 1220464617 found
```

where the bruteforcer network is configured to run for 50 cycles. The hash to look for has been pre-programmed in the verifier. Every tested hash is printed until the correct one is found. We have verified this implementation by testing it against the C-implementaiton of the same algorithm, ensuring that they produce identical results. On a side note, support for printing hexadecimal numbers should defenitely be added in future versions of SMEIL.

6.6 Performance

The simulation performance of a hardware design tool is not essential as it does not have an impact on the resulting implementation. Nevertheless, a slow simulator can waste valuable developer time by inducing a long develop-compile-test cycle.

The current implementation of SMEIL is not written with performance in mind and leaves a lot of performance-related low-hanging fruits unpicked. Indeed, its naïve interpreter makes repeated traversals of the SMEIL AST and the interface between PySME and LIBSME relies on the very general and inefficient `libffi` library. Lastly, Python itself is not cherished for its performance. In spite of this, LIBSME still exceeds the performance of the VHDL simulator, GHDL, which generates native code before simulating. The VHDL simulator of Xilinx Vivado, fails to complete the simulation due to memory exhaustion.

Simulating 352,686 cycles of the ColorBin (Section 6.3) network on an Intel Core i7 6700HQ CPU at 2.60GHz, requires 47 seconds using GHDL but only 30 seconds using LIBSME. Based on the benchmarks shown in [38], LIBSME achieves only slightly worse performance than C# SME in this particular test.

```
proc md5(in input)
  bus hashes {
    h0: u32;
    h1: u32;
    h2: u32;
    h3: u32;

    w0: u32;
    w1: u32;
  };

  const r: [64]uint = [
    [...]
    6, 10, 15, 21, 6, 10]

  const kk: [64]uint = [
    0xd76aa478, 0xe8c7b756
    [..]
    0xf7537e82, 0xbd3af235
    ];

// Variable declarations omitted

{
  h0 = 0x67452301;
  // [..]

  w[0] = input.w0;
  w[1] = input.w1;
  w[2] = 128;
  w[14] = 64;

  a = h0;
  b = h1;
  c = h2;
  d = h3;

for i = 0 to 63 {
  if (i < 16) {
    f = (b & c) | ((~b) & d);
    g = i;
   // [..]
  } else {
    f = c ^ (b | (~d));
    g = (7 * i) % 16;
  }

  tmp = d;
  d = c;
  c = b;
  x = a + f + kk[i] + w[g];
  c2 = r[i];
  b = b + (((x) << (c2)) |
      ((x) >> (32 - (c2))));
  a = tmp;
}

h0 = h0 + a;
// [..]
h3 = h3 + d;

hashes.h0 = h0;
// [..]
hashes.h3 = h3;

hashes.w0 = w[0];
hashes.w1 = w[1];
}
```

Figure 6.11: SMEIL source code for the MD5 hashing process.

7

Discussion

We started working on SMEIL with the intention of creating an intermediate representation for use with existing SME implementations. The resulting language covers a wider scope and has a larger number of use cases than was originally intended. In this chapter, we discuss uses for SMEIL and how it relates to current SME implementations.

7.1 Usability of SMEIL

As a language for representing SME networks SMEIL is complete in the sense that it implements the elements and semantics required by the SME model. We have also shown a number of actual implementations of SME networks illustrating the practical use of SMEIL. Thus, SMEIL is capable of representing the SME models which are currently implemented in other languages.

We have previously argued for the potential benefits of using SMEIL in favor of general-purpose languages for writing hardware-targeted SME models. However, there are also several disadvantages related to introducing a new and distinct language. One particular disadvantage is the abandonment of development environments, debuggers and other toolings that comes with an established general-purpose language. For SMEIL to offset these disadvantages, an additional development effort is required to replace this tooling. On the other hand, debuggers for conventional programming languages are not well suited for debugging concurrent models, such as SME. Therefore, introducing a new language may create an avenue for developing new toolings which are better suited for their purpose.

Using SMEIL as an IL

SMEIL is usable as an intermediate language for SME models written in a wide range of different general-purpose languages. The static structural definitions and the static type system of SMEIL forces a normalized representation of SME networks regardless of which source language the SMEIL code was generated from. Thus, when generating SMEIL from a dynamically typed source, it is the responsibility of the translator to ensure that the appropriate types are assigned. In PySME, for example, this problem was solved by allowing type-annotations to be written in comments [7]. The result of this is that a SMEIL program should not inherit any particular traits of the code that it was generated from. Therefore, SMEIL will allow combining SME networks that were originally written in different source languages.

We have already shown an example of how SMEIL can function as an intermediate language for the PySME SME implementation. However, the PySME library is restricted in terms of functionality compared to the C# SME library which has been used as the base for all of the previously referenced SME models implemented on hardware. Compared to C# SME, SMEIL and LIBSME still lacks a number of essential features. Achieving feature-parity with "state-of-the-art" SME libraries were considered outside the scope of this thesis due to the more mature state of these libraries. However, it means that more work is required in order to make SMEIL usable as an IL for C# SME.

7.2 Co-simulation and observation based typing

One of the strongest features of SMEIL is the co-simulation interface which allows it to seamlessly interact with SME networks written in other languages. It is the cornerstone of enabling SMEIL to be used both as an intermediate language and as a primary implementation language for SME models in a capable manner. For the intermediate-language use case, it helps with simplifying the implementation of SME libraries using it. For example by offloading the generation of trace-files. For use as a primary implementation language, it allows direct testing of designs without having to execute generated code. Furthermore, the co-simulation interface is essential for enabling a tightly integrated implementation of the system for observation-based typing.

Co-simulation is a frequently used technique for verifying hardware designs (see e.g. CoCoTB referenced in Section 8.1), however, the presented SME-based solution is unique in that the same model is used on both sides of the simulation. As one particular advantage, the compositionality of SME means that a process may be moved seamlessly between both sides of the co-simulation. We have only shown an implementation of the API for Python here but we believe that extending other SME implementations with support for the LIBSME co-simulation API will be possible. This is the case since the API is kept "as close to C" as possible. Hence, any language featuring a capable C interface should be able to use it.

Our approach for determining types based on observed input is, to the best of our knowledge, also not implemented by any existing HDLs. We are aware that there may be good reasons for this, due to the, as previously mentioned, inadequate safety provided by this approach. However, based on our limited testing, the productivity advantage of not having to place precise size bounds on all internal variables does seem to be real. Furthermore, in some cases, such as quickly prototyping a design on an FPGA, safety may not be the primary desired feature. Thus, if combined with static analysis to prove that the observationally derived size bounds really are safe, it may turn out to be a powerful approach.

7.3 Target language support

We have currently only implemented a code generation backend for VHDL. We focused on supporting VHDL since the primary intended target for SME models is hardware designs. For this thesis, we chose not to implement support for additional code generation backends and, instead, focus on making the core infrastructure related to SMEIL as complete as possible. However, having support for generating additional languages is desirable in several circumstances. Having, for example, C++ support will, as shown in [38], allow for faster simulation of SME networks and allow SME networks to be used as independent libraries with other languages.

Supporting other HDLs, such as Verilog, is also desirable since it would allow SME models to interface with hardware designs implemented in HDLs other than VHDL. Adding such support to SMEIL is quite feasible since most other "new" HDLs (we do a small survey in Section 8.1) support code generation for at least VHDL and Verilog. This shows that hardware designs which are implemented in VHDL can also be represented using Verilog.

The current LIBSME implementation is written with support for multiple target languages in mind. This is done, by confining all VHDL-related details to the VHDL code generation backend. All other phases of the compiler are kept as general as possible. Thus, we do not foresee any problems related to adding additional code-generation backends for SMEIL.

We have not assessed the practicality of transforming SMEIL to languages following other paradigms, such as functional languages or different high-level hardware description languages. So far, however, there has not been a need for SME models to be able to target such languages.

8

Conclusions

8.1 Related Work

In addition to the HLS approaches mentioned in the introduction, several alternative hardware design modeling tools have been proposed both in the industry and in academia. Furthermore, a number of approaches to replace test benches written in traditional HDLs has been proposed.

MyHDL [24] is a Python-based HDL, essentially a DSL embedded in Python. It is intentionally implemented as a high-level version of traditional HDLs while enabling Python to be used for test benches. Since it inherits its worldview from traditional HDLs, it has a different goal than SME which provides an abstraction through the SME model.

Cx [39] is a dedicated DSL for writing hardware designs. The Cx language has several similarities with SMEIL, for example, the type system. Like SME, it allows the programmer to explicitly control concurrency by building networks of processes. However, despite claims on its website, Cx is a proprietary language requiring a license for long-term use, giving it a high barrier-of-entry.

CλaSH [42] and Lava[12] are two Haskell based approaches with different philosophies: Lava is a Haskell design pattern (with several implementations e.g., [18]) for specifying composable circuits at the gate-level. The extremely low-level approach means that it is targeted towards replacing and formalizing certain low-level uses of HDLs rather than as a general high-level hardware modeling tool. CλaSH, on the other hand, transforms a subset of high-level Haskell code to HDLs. This requires concurrency inference, but this is simpler to do for a purely functional language, such as Haskell, compared to an impure imperative language, such as C.

A more recent approach [1] also uses Haskell, but only as a host for an Embedded DSL. This EDSL translates to both VHDL and C, enabling the programmer to trivially change which parts of her program that runs on the CPU or the FPGA. The library automates setting up AXI interconnects between the CPU-part and the FPGA-part of the code. The advantage of this approach is that it enables simple hardware-software co-design. The primary disadvantage of this approach is that the CPU code must also be written in the DSL. For many applications, this can be overly restrictive since reuse of existing code and common libraries is not possible.

CAPF [34], Pyrope [21] and Chisel [10] are HDLs which provide data-flow based design models. CAPF and Pyrope are independent languages while Chisel is an EDSL in Scala. The data layouts that are good fits for these languages are also expressible using SME, albeit less elegantly. However, problems which are best represented as a sequential algorithm can be a poor fit for the data-flow paradigm.

The Coroutine Co-Simulation Test Bench (CoCoTB) [30] also implements a notion co-simulation between Hardware Descriptions and Python (A General-Purpose language). Using the Verilog Procedural Interface (VPI) which (despite the name) is implemented by both VHDL and Verilog simulators. This library presents a significant advantage over writing test benches exclusively in HDLs, however, the relative complexity of the VPI interface leaks into the CoCoTB interface, requiring a non-trivial amount of boilerplate code. Furthermore, it does not directly address the productivity issues associated with traditional HDLs and does not offer the unified simulation model used in SME.

8.2 Future Work

Other SME implementations, C# SME in particular (see e.g. [36]) have evolved in parallel with the development of SMEIL. Therefore, these are more comprehensive and support a wider range of features. Since SMEIL is, as previously mentioned, intended to serve as the only target language for SME, SMEIL should be brought on-par with other existing SME implementations. In the present work, a substantial amount of compiler-infrastructure groundwork has been laid, making these improvements a natural continuation of future SMEIL developments.

As the primary target for SMEIL is hardware, VHDL is the only code generation backend currently implemented. However, code-generation backends for additional languages should be added. For example, generating C++ code can make it possible to use SME programs with other software as a library and provide significantly faster simulation than the current interpretation-based approaches are able to offer [38].

All SME implementations currently target a single clock domain. Future efforts should be made towards supporting multiple clocks, running at different speeds.

In some cases, SMEIL may offer an insufficient amount of control over the generated VHDL code or the generated code may simply be inefficient compared to hand-optimized VHDL code. For this, we should allow inlining VHDL inside SMEIL by adding language constructs to specify how SME buses should be connected to VHDL signals. The simulation of such mixed code can be performed by running VHDL parts in a VHDL simulator.

Hardware-software co-design is an area that is actively researched. The idea is that specialized hardware is designed in parallel with the corresponding software such that each part of the design can be implemented on either hardware or software depending on what is best suited. Such heterogeneous designs require code for setting up the communication between the hardware and software parts. We should therefore extend our co-simulation approach to also allow SME networks to be distributed across several devices.

The presented approach, for automatically typing SMEIL networks based on observed input, makes the assumption that the complete possible space of input values is explored by the testing stimuli. The downside of this approach is that this assumption may be hard to fulfill. To address this, the current approach could be augmented by integer range analysis for proving the observed ranges.

To improve the user-friendliness and capabilities of SMEIL, there is a wide range of language features that we would like to add. A non-exhaustive list follows:

- In practice, not being able to add declarations, such as constants, enumerations. and functions, at the top-level of a SMEIL program proved too restrictive. This should be added.
- A syntax for direct bit manipulations. Currently, bit manipulation can only be performed through bitwise operators in a similar manner to C. It would be convenient to have an array-like syntax for achieving the same thing. The syntax could, for example, be similar to Python's array slicing feature.

8.3 Conclusion

We have presented SMEIL, a DSL for implementing SME networks and demonstrated its practical use through several examples. Although we have focused on using it as a primary implementation language for SME networks, it is also usable as an intermediate language for other SME implementations. We have shown this by providing a SMEIL code-generation backend for a Python to VHDL compiler.

SMEIL is based on the structural components of the SME model and provides a high-level C-like syntax with constructs commonly found in general-purpose imperative languages. This is needed in order to ensure that SME networks implemented in general-purpose languages can be translated without requiring sophisticated transformations.

The type system presented supports bit precise types which is an important feature for a hardware-targeted language. However, the requirement to specify a fixed bit-width for all types is sometimes impractical. Instead, arbitrary-length types may be specified which are then constrained based on values observed during simulation.

Simulation of SMEIL is performed in a manner which provides a cycle-accurate representation of the resulting hardware. During the simulation, a trace of channel communications is recorded. SMEIL compiles to readable VHDL code which can be used for a subsequent hardware implementation. Additionally, a test bench is generated which can be used to verify the correctness of the generated code. The test bench uses the trace recorded during simulation to allow continuing verification of the generated code even following manual refinement.

For testing SMEIL networks directly, an interface is provided for performing co-simulation with SME networks written in general-purpose languages. This approach proved highly successful in practice.

The presented language and its implementation do not yet provide the full feature-set of other, more mature, SME implementations. In spite of this, we are optimistic about its future prospects, both as an intermediate language and as an independent DSL for writing SME networks.

Bibliography

[1] M. Aronsson and M. Sheeran. "Hardware software co-design in Haskell". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. ACM. 2017, pp. 162–173 (cit. on p. 56).

[2] T. Asheim. *Almique source code*. `https://github.com/truls/almique`. [Online; accessed April 2018] (cit. on p. 3).

[3] T. Asheim. *Implementing High Performance Synchronous Message Exchange*. Bachelor's Thesis. 2015 (cit. on p. 2).

[4] T. Asheim. `language-vhdl-quote` *source code*. `https://github.com/truls/almique`. [Online; accessed April 2018] (cit. on p. 40).

[5] T. Asheim. *LibSME source code*. `https://github.com/truls/libsme`. [Online; accessed April 2018] (cit. on p. 2).

[6] T. Asheim. *PySME source code*. `https://github.com/truls/pysme`. [Online; accessed April 2018] (cit. on p. 32).

[7] T. Asheim, K. Skovhede, and B. Vinter. "VHDL Generation From Python Synchronous Message Exchange Networks". In: *Proceedings of Communicating Process Architectures 2016* (2016) (cit. on pp. 2, 3, 11, 42, 43, 47, 53).

[8] P. J. Ashenden. "VHDL standards". In: *IEEE Design & Test of Computers* 18.5 (2001), pp. 122–123 (cit. on p. 13).

[9] M. Avgerinou, P. Bertoldi, and L. Castellazzi. "Trends in Data Centre Energy Consumption under the European Code of Conduct for Data Centre Energy Efficiency". In: *Energies* 10.10 (2017), p. 1470 (cit. on p. 12).

[10] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. "Chisel: constructing hardware in a scala embedded language". In: *Proceedings of the 49th Annual Design Automation Conference*. ACM. 2012, pp. 1216–1225 (cit. on p. 56).

[11] U. Banerjee. *Data dependence in ordinary programs*. Department of Computer Science, University of Illinois at Urbana-Champaign, 1976 (cit. on p. 1).

[12] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. "Lava: hardware design in Haskell". In: *ACM SIGPLAN Notices*. ACM. 1998, pp. 174–184 (cit. on p. 56).

[13] J. M. Bjørndalen, B. Vinter, and O. J. Anshus. "PyCSP-Communicating Sequential Processes for Python." In: *Cpa*. 2007, pp. 229–248 (cit. on p. 6).

[14] R. Bodik, R. Gupta, and V. Sarkar. "ABCD: eliminating array bounds checks on demand". In: *ACM SIGPLAN Notices*. Vol. 35. ACM. 2000, pp. 321–333 (cit. on p. 1).

[15] D. Chen and D. Singh. "Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering". In: *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE. 2012, pp. 5–12 (cit. on p. 2).

[16] C. Dawson, S. K. Pattanam, and D. Roberts. "The verilog procedural interface for the verilog hardware description language". In: *Verilog HDL Conference, 1996. Proceedings., 1996 IEEE International*. IEEE. 1996, pp. 17–23 (cit. on p. 29).

[17] J. Fowers, G. Brown, P. Cooke, and G. Stitt. "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications". In: *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM. 2012, pp. 47–56 (cit. on p. 12).

[18] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling. "Introducing kansas lava". In: *International Symposium on Implementation and Application of Functional Languages*. Springer. 2009, pp. 18–35 (cit. on p. 56).

[19] T. Gingold. *GHDL*. `https://github.com/ghdl/ghdl`. [Online; accessed May 2018] (cit. on p. 40).

[20] P. Glasserman. *Monte Carlo methods in financial engineering*. Vol. 53. Springer Science & Business Media, 2003, pp. 79–99 (cit. on p. 48).

[21] Haven Skinner, Akash Sridhar, Rafael Trapani Possignolo and Jose Renau. *Pyrope*. `https://masc.soe.ucsc.edu/pyrope.html`. [Online; Accessed May 2018] (cit. on p. 56).

[22] C. Hoare. *Communicating Sequential Processes*. ISBN: 0-131-53271-5. London: Prentice-Hall, 1985 (cit. on p. 2).

[23] Intel Inc. *Altera OpenCL*. `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html`. [Online; Accessed May 2018] (cit. on p. 1).

[24] Jan Decaluwe. *MyHDL*. `http://myhdl.org`. Online; Accessed May 2018 (cit. on p. 56).

[25] C.-J. Johnsen. "Implementing a MIPS processor using SME". MA thesis. University of Copenhagen, 2017 (cit. on pp. 2, 18).

[26] C.-J. Johnsen, K. Skovhede, and B. Vinter. "Exploring multiple parallel hardware designs on an FPGA using Synchronous Message Exchange". Manuscript submitted for publication. 2018 (cit. on p. 48).

[27] A. Kablan and J. Falzon. "The Use of Dynamically Optimised High Frequency Moving Average Strategies for Intraday Trading". In: *Word Academy of Science, Engineering and Technology* (2012) (cit. on p. 48).

[28] I. Kuon and J. Rose. "Measuring the gap between FPGAs and ASICs". In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 26.2 (2007), pp. 203–215 (cit. on p. 12).

[29] G. Mainland. "Why it's nice to be quoted: quasiquoting for haskell". In: *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. ACM. 2007, pp. 73–82 (cit. on p. 40).

[30] potentialventures. *Coroutine Co-simulation Test Bench*. `https://github.com/potentialventures/cocotb`. [Online; accessed May 2018] (cit. on p. 57).

[31] N. Ramsey. "Unparsing expressions with prefix and postfix operators". In: *Softw., Pract. Exper.* 28.12 (1998), pp. 1327–1356 (cit. on p. 47).

[32] M. Rehr, K. Skovhede, and B. Vinter. "BPU Simulator". In: *Communicating Process Architectures 2013*. Ed. by P. H. Welch, F. R. M. Barnes, J. F. Broenink, K. Chalmers, J. B. Pedersen, and A. T. Sampson. Nov. 2013, pp. 233–248. ISBN: 978-0-9565409-7-3 (cit. on p. 6).

[33] F. Schloegl, S. Rohjans, S. Lehnhoff, J. Velasquez, C. Steinbrink, and P. Palensky. "Towards a classification scheme for co-simulation approaches in energy systems". In: *Smart Electric Distribution Systems and Technologies (EDST), 2015 International Symposium on*. IEEE. 2015, pp. 516–521 (cit. on p. 29).

[34] J. Serot, F. Berry, and S. Ahmed. "Implementing stream-processing applications on FPGAs: A DSL-based approach". In: *Field Programmable Logic and Applications (FPL), 2011 International Conference on*. IEEE. 2011, pp. 130–137 (cit. on p. 56).

[35] E. Skaarup and A. Frisch. "Generation of FPGA Hardware Specifications from PyCSP Networks". MA thesis. University of Copenhagen, Niels Bohr Institute, 2014 (cit. on p. 6).

[36] K. Skovhede and C.-J. Johnsen. "Building FPGA state machines from sequential code". Manuscript submitted for publication. 2018 (cit. on p. 57).

[37] K. Skovhede and B. Vinter. "Building hardware from C# models". In: *FSP 2016; Third International Workshop on FPGAs for Software Programmers; Proceedings of*. VDE. 2016, pp. 1–9 (cit. on pp. 2, 11).

[38] K. Skovhede and B. Vinter. "C++ support for better hardware/software co-design in C# with SME". In: *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers; Proceedings of*. VDE. 2017, pp. 1–8 (cit. on pp. 47, 51, 55, 57).

[39] Synflow. *The Cx Programming Language*. `http://cx-lang.org`. Online; Accessed May 2018 (cit. on p. 56).

[40] B. Vinter and K. Skovhede. "Bus Centric Synchronous Message Exchange for Hardware Designs". In: *Proceedings of Communicating Process Architectures 2015* (2015) (cit. on pp. 2, 10).

[41] B. Vinter and K. Skovhede. "Synchronous Message Exchange for Hardware Designs". In: *Proceedings of Communicating Process Architectures 2014* (2014) (cit. on pp. 2, 6, 7, 10).

[42] R. Wester. "A transformation-based approach to hardware design using higher-order functions". PhD thesis. University of Twente, 2015 (cit. on p. 56).

[43] Xilinx Inc. *Vivado HLS*. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`. [Online; Accessed May 2018] (cit. on p. 1).

A

Installation Instructions

This appendix contains information on how to install and run the SMEIL system in order to reproduce the runs shown throughout the thesis. These instructions has only been tested on a Fedora 27 machine and uses software versions (except stack) from its standard repositories. We expect them to work on any recent Linux distribution although we make no guarantees. These instructions are superseded by any instructions which may appear online.

A.1 Dependencies

The required dependencies for building and running are listed below

- `stack` — `https://haskell-lang.org/get-started`
- `gcc`
- `python3.6`
- `perl`
- `pip3`
- `git`
- The Python scripts used in the co-simulated examples have dependencies not listed here.

The individual software components referenced throughout this thesis:

- libsme — `https://github.com/truls/libsme`
- pysme — `https://github.com/truls/pysme`
- almique — `https://github.com/truls/almique`

A.2 Installation

PySME

1. Go to PySME directory
2. Run `pip3 install --user .`

libsme

- Go to the `libsme` directory
- Run `make` (this will take a while)
- When `make` completes it has created the `tools/runsme` script. Copy this script somewhere in your `$PATH`.
- Run `stack install` to install the `smec` executable to your `$PATH`.

almique

- Go to the `almique` directory.
- Run `stack build`

A.3 Running the examples

SMEIL as IL

Go to the `almique/examples` directory and run `stack exec almique someops.py`. You should now find a file named `SomeOps.sme` containing the generated SMEIL code.

7-segment

Go to the `libsme/examples/pure` directory. Run `smec -i 7seg.sme -s 50`

ColorBin

Go to the `libsme/examples/python/colorbin` directory. Run `runsme python3 colorbin.py`. The generated VHDL code can be found in the `output` directory. The VHDL code can be tested by running `make` in the `output` directory and executing the generated `coll_net_tb` file.

High-frequency trading chip

Go to the `libsme/examples/python/ewma` directory. Run `runsme python3 ewma-int.py`. The generated VHDL code can be found in the `output` directory. The VHDL code can be tested by running `make` in the `output` directory and executing the generated `ewma_tb` file.

MD5 bruteforcer

Go to the `libsme/examples/pure` directory. Run `smec -i md5-simple.sme --no-warnings -s 50`.