An Embedded Language Framework for Hardware Compilation

Koen Claessen[1] and Gordon Pace[2]

[1] Chalmers University, Gothenburg, Sweden
[2] Inria Rhône-Alpes, Grenoble, France

Abstract. Various languages have been proposed to describe synchronous hardware at an abstract, yet syntesisable level. We propose a uniform framework within which such languages can be developed, and combined together for simulation, synthesis, and verification. We do this by embedding the languages in Lava — a hardware description language (HDL), itself embedded in the functional programming language Haskell. The approach allows us to easily experiment with new formal languages and language features, and also provides easy access to formal verification tools aiding program verification.

1 Introduction

There are two essentially different ways of describing hardware. One way is *structural* description, where the designer indicates what components should be used and how they should be connected. Designing hardware at the structural level can be rather tedious and time consuming. Sometimes, one affords to exchange speed or size of a circuit for the ability to design a circuit by describing its behaviour at a higher level of abstraction which can then be automatically *compiled* down to structural hardware. This way of describing circuit is usually called a *synthesisable behavioural description*[1]. Behavioural descriptions are also often used to describe the specification of a circuit.

There exist a number of languages that one can use to structurally describe hardware. An example is the synchronous language Lustre [8, 9], which can be compiled into hardware structurally [21]. Languages that can be used for synthesisable behavioural description are for example Esterel [2] and Occam [17]. The popular industrial description languages VHDL and Verilog allow both kinds of descriptions.

In this paper, we will only deal with synchronous hardware, that is, all latches in a circuit listen to one omnipresent global clock. Moreover, at every clock cycle, if each input to a circuit is defined, each point in the circuit stabilises to exactly

[1] These are to be distinguished from *behavioural descriptions* (as used in industrial HDLs such as Verilog and VHDL) which are used to describe the functionality of a circuit, but are do not necessarily have a hardware counterpart.

one voltage, low or high. However, we do not require that every feedback loop in the circuit contains a latch.

There are two main classes of synthesisable languages: ones where the description determines the timing behaviour (cycle by cycle) of the resultant circuit, and ones with no explicit timing control, and where the compilation only guarantees that the output at the end of the algorithm (or at designated points in the algorithm) matches that of the circuit. Languages with strict timing are necessary to describe circuits such as protocol implementations, and reactive systems, where the circuit continuously runs, sampling inputs, and behaving accordingly. In practice, some compilation schemata fall somewhere in between these two classes. In particular, commercial synthesis tools for Verilog and VHDL usually provide the user with the option of choosing how strictly the timing behaviour specified is adhered to. In the rest of the paper, we will be talking *exclusively* of strict timing compilation, but the approach is equally applicable to languages with loose timing.

Embedded Description Languages
Using a technique from the programming language community, called *embedded languages* [11], we present a framework to merge structural and behavioural hardware descriptions. An embedded description language is realised by means of a library in an already existing programming language, called the *host language*. This library provides the syntax and semantics of the embedded language by exporting function names and implementations.

The basic embedded language we use is *Lava* [5]. Lava is a structural hardware description language embedded in the functional programming language Haskell [18]. From hardware descriptions in Lava, EDIF netlist descriptions can be automatically generated, for example to implement the described circuit on a Field Programmable Gate Array (FPGA). This has previously led to highly efficient implementations of complicated circuits [6, 25].

Embedding a language is a powerful concept because descriptions in the embedded language are first-class objects in the host language. In the case of Lava, this means that hardware descriptions can be generated, analysed and transformed using a full-blown programming language.

The idea is now to build a layer on top of Lava, which embeds a synthesisable behavioural description language. In order to do this, we have to specify the syntax of the behavioural language, and the way it is compiled into a structural hardware description. It is possible to describe all this in the Lava framework: the syntax is described as a Haskell datatype, and the compilation process described as a Lava circuit description.

But why stop there? It is possible to embed several different behavioural description languages, each with their own features, advantages and disadvantages. In this way, we can describe a hardware system, using different languages for different parts, all within a single framework.

Examples of uses of embedding in this way are: behavioural in structural, where we use a behavioural language to describe some parts, and plug these parts together using a structural language; multiple behavioural in structural, the same, but having several different behavioural languages; structural in behavioural, so that we can describe a sub-procedure of the behavioural algorithm structurally; and even behavioural in behavioural, where we can describe sub-procedures for one behavioural language by using another behavioural language. All these examples are useful in describing circuits as well as their specifications.

Some of these examples are non-trivial to achieve, and we do not claim to have a generic solution to them. Our contribution proposes a common framework, in which one can quickly experiment with different approaches and new behavioural languages. The framework we propose, Lava, is powerful enough to use for describing new languages, giving semantics to them, implementing them, and combining them. In the context of developing behavioural description languages, it is very convenient to have circuit descriptions, analyses, transformations, and implementation and verification methods backed up by a full-blown programming language.

In section 2 we briefly introduce Lava and show how a simple high level language, that of regular expressions, can be embedded in Lava and how instances of this language can then be manipulated syntactically and compiled into circuits. In section 3 we illustrate how the embedded language approach extends easily to more complex languages by presenting a small, imperative style language, Flash. Section 4 then discusses more advanced issues: various ways of combining different high level languages, verification of compiled programs and exploring potentially dangerous combinational loops.

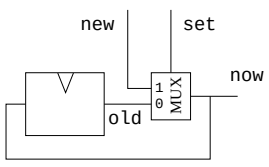
2 Embedding Hardware Description Languages

Circuit Descriptions in Lava

Circuit descriptions in Lava correspond to function definitions in Haskell. The Lava library provides primitive hardware components such as gates, multiplexers and delay components. We give a short introduction to Lava by example.

Here is an example of a description of a register. It contains a multiplexer, `mux`, and a delay component, `delay`. The delay component holds the state of the register and is initialised to `low`.

```
setRegister (set, new) = now
  where
    old = delay low now
    now = mux (set, (old, new))
```



Note that `setRegister` is declared as a circuit with two inputs and one output. Note also that definitions of outputs (`now`) and possible local wires (`old`) are given in the `where`-part of the declaration.

After we have made a circuit description, we can simulate the circuit in Lava as a normal Haskell function. We can also generate VHDL or EDIF describing the circuit. It is possible to apply circuit transformations such as retiming, and to perform circuit analyses such as performance and timing analysis. Lava is connected to a number of formal verification tools, so we can also automatically prove properties about the circuits.

Generic and Parametrized Circuit Definitions
We can use the one bit register to create an n-bit register array, by putting n registers together. In Lava, inputs which can be arbitrarily wide are represented by means of lists. A generic circuit, working for any number of inputs, can then be defined by recursion over the structure of this list.

```
setRegisterArray (set, [])       = []
setRegisterArray (set, new:news) = val:vals
  where
    val  = setRegister (set, new)
    vals = setRegisterArray (set, news)
```

Note how we use pattern matching to distinguish the cases when the list is empty (`[]`) and non-empty (`x:xs`, where `x` is the first element in the list, and `xs` the rest).

Circuit descriptions can also be parametrized. For example, to create a circuit with n delay components in series, we introduce n as a parameter to the description.

```
delayN 0 inp = inp
delayN n inp = out
  where
    inp' = delay low inp
    out  = delayN (n-1) inp'
```

Again, we use pattern matching and recursion to define the circuit. Note that the parameter `n` is *static*, meaning that it has to be known when we want to synthesise the circuit.

A parameter to a circuit does not have to be a number. For example, we can express circuit descriptions which take other circuits as parameters. We call these parametrized circuits *connection patterns*. Other examples of parameters include truth tables, decision trees and state machine descriptions. In this paper, we will talk about circuit descriptions which take behavioural hardware descriptions, or *programs*, as parameters.

Behavioural Descriptions as Objects
In order to parametrize the circuit definitions with behavioural descriptions, we have to embed a behavioural description language in Lava. We do this by declaring a Haskell datatype representing the syntax of the behavioural language. To illustrate the concepts with a small language, we will use regular expressions. The syntax of regular expressions is expressed as a Haskell datatype:

```
data RegExp = EmptyString
            | Input Sig
            | Star RegExp
            | RegExp :+: RegExp
            | RegExp :>: RegExp
```

The data objects belonging to this type are interpreted as regular expressions with, for example, $a(b+c)^*$ being expressed as:

```
Input a :>: Star (Input a :+: Input c)
```

Note that the variables `a`, `b` and `c` are of type `Sig` — they are *signals* provided by the programmer of the regular expression. They can either be outputs from another existing circuit, or be taken as extra parameters to the definition of a particular regular expression. We interpret the signal `a` being high as the character 'a' being present in the input.

Since regular expressions are now simply data objects, we can generate these expressions using Haskell programs. Thus, for example, we can define a `power` function for regular expressions:

```
power 0 e = EmptyString
power n e = e :>: power (n-1) e
```

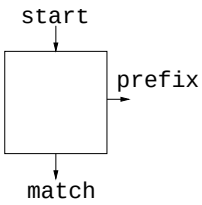
Similarly, regular expressions can be manipulated and modified. For example, a simple rewriting simplification can be defined as follows:

```
simplify (EmptyString :>: e) = simplify e
simplify (EmptyString :+: e)
  | containsEmpty e          = simplify e
  | otherwise                = EmptyString :+: simplify e
simplify (Star (Star e))     = simplify (Star e)
...
```

Another useful algorithm which can be expressed is the one presented in [20], which reduces (in linear time) a regular expression e to another one f such that the empty string does not occur in f and e^* is the same language as f^*. Thus, from now on, we assume that the body of a `Star` cannot produce the empty string.

Compiling Regular Expressions into Circuits

The circuits we generate for regular expressions have one input `start` and two outputs `match`, and `prefix`. When `start` is set to high, the circuit will start sampling the signals. The output `match` is then set to high when the resulting sequence of signals is included in the language represented by the expression. The output `prefix` corresponds to a wire which indicates whether the compiled circuit is still active, and the parsing of the regular expression has not yet failed with respect to the received inputs. Note that the circuit will get extra inputs, which correspond to the parsed symbols. They are part of the regular expression, by means of the `Input` construct.

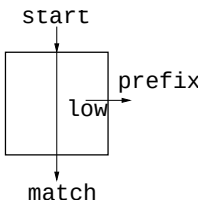


The type of the resulting circuit is thus:

```
type Circuit_RegExp = Sig -> (Sig, Sig)
```

since the resulting circuit has one input and two outputs. We express the compilation process as a circuit definition parametrized by a regular expression:

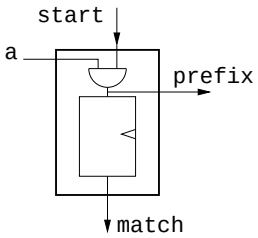
```
regexp :: RegExp -> Circuit_RegExp
```



The Empty String

The compilation of the empty string is straightforward, given the usage of the `prefix` and `match` wires:

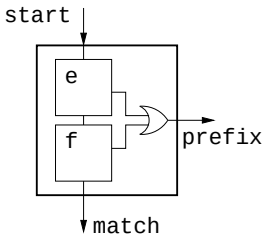
```
regexp EmptyString start = (prefix, match)
 where
    prefix = low
    match  = start
```



Signal input

The regular expression `Input a` is matched if, and only if the signal `a` is high when the circuit is started.

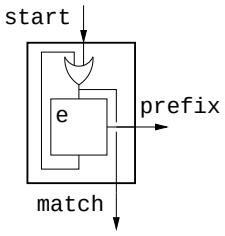
```
regexp (Input a) start = (prefix, match)
  where
    prefix = and2 (start, a)
    match  = delay low prefix
```



Sequential composition

The regular expression `e :>: f` must start accepting expression `e`, and upon matching it, start trying to match expression `f`.

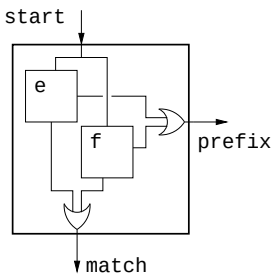
```
regexp (rexp1 :>: rexp2) start = (prefix, match)
  where
    (prefix1, match1) = regexp rexp1 start
    (prefix2, match)  = regexp rexp2 match1
    prefix = or2 (prefix1, prefix2)
```



Loops

The circuit accepting regular expression `Star e` is very similar to that accepting `e`, but it is restarted every time the inputs match `e`.

```
regexp (Star rexp) start = (prefix, match)
  where
    (prefix, match') = regexp rexp match
    match = or2 (start, match')
```



Non-deterministic choice
The inputs match regular expression `e :+: f` exactly when they match expression `e` or `f`.

```
regexp (rexp1 :+: rexp2) start = (prefix, match)
  where
    (prefix1, match1) = regexp rexp1 start
    (prefix2, match2) = regexp rexp2 start
    prefix = or2 (prefix1, prefix2)
    match  = or2 (match1, match2)
```

A circuit resulting from such a compilation scheme is not necessarily efficient enough. Often, there are optimisations we can make, such as constant folding (when the input to a gate is always low or always high), sharing introduction (when we have identical gates with identical inputs), tree introduction (changing a linear chain of associative gates into a balanced tree), and constant introduction (when a circuit point provably always has the same value). Sometimes, more rigorous optimisation methods are necessary; in this case we can use external circuit optimisation tools such as SIS [7].

3 Compiling Flash

In this section, we will show a slightly bigger example of a language, we will call *Flash*. It is quite a basic language, but it illustrates many of the issues one encounters when dealing with hardware compilation. As it is meant just an example, we deal quite informally with the semantics of Flash. More formal treatment of the semantics of similar languages can be found in [1, 17].

Flash Syntax
As before, we first declare a Haskell datatype that embeds the syntax of Flash.

```
data Flash = Skip
           | Delay
           | Shout
           | IfThenElse Sig (Flash, Flash)
           | While Sig Flash
           | Flash :>> Flash
           | Flash :|| Flash
```

Flash is a simple imperative programming language containing the usual statements like skip, sequential composition (`:>>`), if-then-else, and while. For simplicity, the language has no expressions. Instead, we can use Lava gates directly to create a signal representing the condition in both the if-then-else and the while loop.

To create some interesting output, we have added a `Shout` statement. This statement is in the spirit of the Esterel `emit` statement [2]. It makes a special output

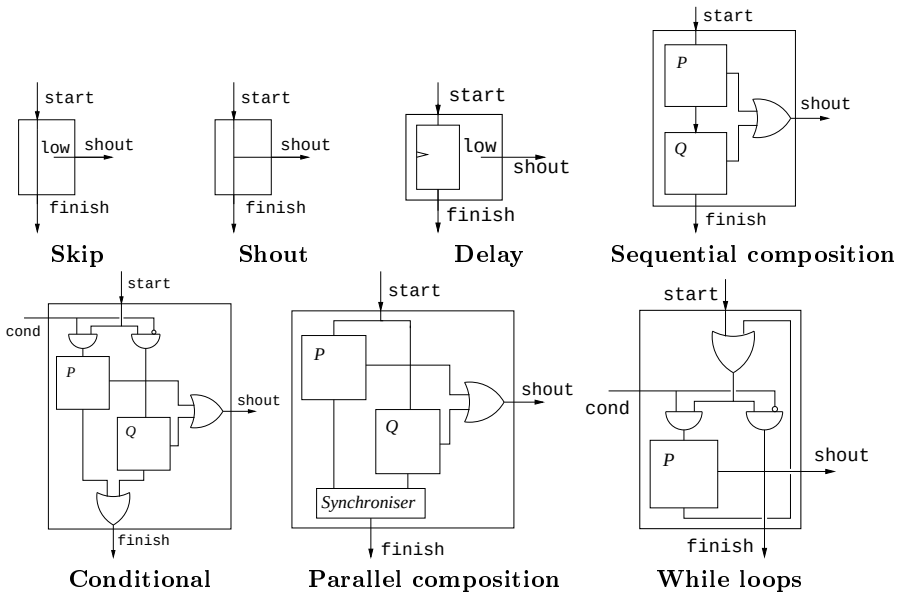


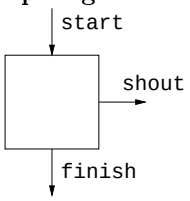
Fig. 1. Compiling Flash

of the circuit, called `shout`, high whenever `Shout` is executed. Further, we also have parallel composition (`:||`), which has a fork-join semantics. Lastly, the delay statement is the only statement that takes time. When executed, it blocks the process until the next clock cycle. Note that `Shout` takes no time to execute.

For example, a Flash program to output a clock-like output alternating between high and low could be written as:

```
alternate = While (high) (Shout :>> Delay :>> Delay)
```

Compiling Flash



The circuits that we compile Flash programs into have one input, `start`, which is set to high to start the program. They will have two outputs: `shout`, which becomes high when the program shouts, and `finish`, which becomes high when the program is done.

In figure 1, we see the compilation schemata for the various language constructs of Flash. We show the Lava code for some of the constructs.

The case for the while loop looks as follows:

```
flash (While cond prog) start = (shout, finish)
  where
    (shout, finish') = flash prog start'
```

```
    restart = or2 (start, finish')
    start'  = and2 (restart, cond)
    finish  = and2 (restart, inv cond)
```

We might (re)start the body of the while loop, if the whole loop is started or if the body has just finished. In that case, depending on the condition, we restart the body or we finish. Note that we have created a loop since `finish'` depends on `start'` depends on `restart` depends on `finish'`. In fact, this loop might be a combinational loop — we say more about this in section 4.

Here is how we translate parallel composition:

```
flash (prog1 :|| prog2) start = (shout, finish)
  where
    (shout1, finish1) = flash prog1 start
    (shout2, finish2) = flash prog2 start
    shout  = or2 (shout1, shout2)
    finish = synchroniser (finish1, finish2)
```

We start both processes as soon as the parallel composition is started. We shout when one of the processes shouts. But when do we finish? We use a little circuit, called *synchroniser*, which keeps track of both processes, and generates a high on the finish signal exactly when both processes have finished.

```
synchroniser (finish1, finish2) = finish
  where
    both   = and2 (finish1, finish2)
    one    = xor2 (finish1, finish2)
    wait   = delay low (xor2 (one, wait))
    finish = or2 (both, and2 (wait, one))
```

The wire `both` is high when both processes are finishing at the same time. The wire `one` is high when exactly one process is finishing. The wire `wait` is high when one process has finished but not the other.

4 Advantages of Embedding

In this section, we discuss some of the advantages of embedding behavioral languages in a general hardware description framework like Lava.

Combining Languages
The choice of the right language to solve a problem is crucial both to simplify the algorithm, and to generate more efficient circuits. For example, regular expressions can be very useful to generate circuits which validate their input, but, since they have no outputting mechanism, it becomes very difficult (or impossible) to perform calculations and output their results.

Consider the problem of designing a circuit that accepts input sequences that behave like a clock with half-period n. This circuit might be useful for monitoring real input, or when expressing properties for later formal verification. It is easy to write a generic regular expression with the specified behaviour:

```
acceptClock n c = Star (  power n (Input c)
                       :>: power n (Input (inv c))
                       )
```

Now consider using a regular expression to design a circuit that monitors two inputs, accepting them only if they behave like clocks with half-periods n and m. The size of the smallest regular expression capable of doing this has a size of the order of magnitude of the least common denominator of n and m, which is too big in practice.

There are two solutions. One is to design a new language, in which it is easy to describe circuits as the one mentioned above. In fact, it would suffice to add *conjunction* as a regular expression operator, which would require some extra compile-time effort. The other is to combine the solutions to the two subproblems (recognising each clock) at the structural level using Lava:

```
acceptTwoClocks n m (c1,c2) = ok
  where
    (ok1,_) = regexp (acceptClock n c1) start
    (ok2,_) = regexp (acceptClock m c2) start
    start   = delay high low
    ok      = and2 (ok1, ok2)
```

Obviously, the used subprograms need not be in the same language. For example, if we want to run a Flash program `prg` only to abort it as soon as the input does not match a regular expression `rexp`, we can use the following parameterised circuit:

```
abort rexp prg start = (shout', finish)
  where
    (shout, finish) = flash  prg  start
    (prefix, _)     = regexp rexp start
    shout'          = and2 (shout, prefix)
```

Nesting Languages

A problem with the approach mentioned above is that we deal with the input and output of the produced circuits at a rather low-level. This is quite error-prone, and it becomes difficult to change the shape of the produced circuits.

A cleaner approach is not to express the combination of programs at the structural level, but at the behavioural one. Thus, for example, one could allow adding Flash subprograms to regular expressions by augmenting the syntax of regular expressions by:

```
data RegExp = ...   | ImportFlash Flash
```

Consider the problem of generating a circuit which recognises the input of `a`, `b` and `c` in any order. If this is required in a sub-expression of a regular expression, the result of expanding the expression can lead to a blow up in circuit size. However, a Flash program for this is rather simple to write:

```
wait s        = While (inv s) Delay
perm3 (a,b,c) = (wait a :|| wait b :|| wait c) :>> Delay
```

If this is required within a regular expression, one can easily use it as for example:

```
Star (ImportFlash (perm3 (a,b,c)) :+: ImportFlash (perm3 (d,e,f)))
```

Fiddling with the interfaces to make them match is thus done only once when the compilation of a regular expression of the form `ImportFlash p` is defined. However, this approach still has the undesirable effect that for every new language one uses, the compilers for all other languages need to be modified to be able to import programs from the new languages into the old ones.

A more extendable approach would be to add one `Import` construct for each language:

```
data RegExp = ...   | ImportRegExp Circuit_RegExp
data Flash  = ...   | ImportFlash  Circuit_Flash
```

Now, in order to import Flash programs in regular expressions, all we have to provide is a parameterised circuit `flash_regexp`, which converts from one format to the other.

```
flash_regexp flashc start = (prefix, match)
  where
    (shout, finish) = flashc start
    prefix = shout
    match  = finish
```

Needless to say, there are other ways in which a Flash circuit can be transformed into one which can be used by regular expressions. For example, one can generate (or calculate) an `active` wire from Flash circuits which corresponds to the regular expression `prefix` wire. In defining these 'conversion' circuits, we have to be careful here not to invalidate the invariants that the languages involved assume and obey. The technique mentioned in the next section can be used to help with this. 'Calling' another language now simply becomes a matter of using the `Import` construct and the right conversion circuits.

Error Wires

Often, something can go wrong during the execution of a program. What exactly can go wrong depends on the semantics of the language. A standard example in a language with arithmetic expressions is division by zero. It is not clear what

the corresponding compiled circuit would do in that case, since we do not want the circuit simply to 'abort'.

In a language with parallel composition, things can go wrong due to parts of the circuit requiring single access: two processes trying to send a message on the same channel at the same time, two processes updating a shared variable at the same time, etc. If the semantics of the language disallows these situations, then we should make sure that the programs we compile to hardware are well-behaved.

The solution we propose is to have an extra output to the circuit which goes high as soon as something goes wrong with the program execution — an *error* wire. This wire and the logic generating it will not appear in the final implementation of the circuit, but will be used to *verify* (by means of model checking methods) that the program in question is error-free.

Consider a change to the semantics of Flash, requiring that only one process can shout at the same time. We would like to be warned at compile time if a program violates that property. Thus, we add an error wire to the output of Flash circuits, and adapt the compilation scheme accordingly. Here is the interesting case, parallel composition:

```
flash (prog1 :|| prog2) start = (shout, error, finish)
  where
    (shout1, error1, finish1) = flash prog1 start
    (shout2, error2, finish2) = flash prog2 start
    shout  = or2 (shout1, shout2)
    both   = and2 (shout1, shout2)
    error  = orl [error1, both, error2]
    finish = synchroniser (finish1, finish2)
```

There is an error in parallel composition of two programs if there was an error in (at least) one of the processes, or if both processes shout at the same time. We can now declare a *property*, a circuit which outputs are always high if and only if a certain property holds.

```
prop_FlashProgramOk prog start = inv error
  where
    (_, error, _) = flash prog start
```

The output `ok` is high if and only if there is no error in the program `prog`. We can check this property using the Lava command `verify`.

```
Lava> verify (prop_FlashProgramOk (alternate :|| (Delay :>> alternate))
Verify: ... Valid.

Lava> verify (prop_FlashProgramOk (Shout :>> Delay :|| Shout))
Verify: ... Falsifiable.
<high>
```

The error wire technique can also be used to find bugs in the compilation scheme itself. Many languages have certain invariants that hold for every program. By raising the signal on the error wire when the invariant is violated, and verifying the absence of this error for random programs (by using a technique similar to the one developed in [4]), we can find bugs or increase our confidence in the compilation scheme.

Combinational Loops

Looking at the compilation scheme for the `while` construct, we can see that it is possible to introduce combinational loops: cycles in the circuit without a delay component.

The usual solution in this case is to require that body of the `while` loop takes time — the execution path goes through at least one `Delay` statement. But even with this restriction, the resulting circuit might still contain combinational loops. However, these combinational loops are not *bad*, in the sense that the actual circuit never produces undefined outputs. In this case, the combinational loops are called *constructive* [24].

Even when all combinational loops in a given circuit are constructive, most of the external formal verification tools that Lava is connected to, are not able to deal with these loops. Fortunately, the method of *temporal induction* [23] can naturally verify properties of cyclic circuit definitions. However, the method is only sound if all loops in the circuit are constructive loops.

Thus, before we implement or formally verify actual circuits containing possible bad loops, we have to prove that all loops all constructive. Lava provides a circuit analysis, called `constructive`, which does exactly that [3]. Here is how we can use it:

```
Lava> verify (constructive (flash (While high Delay)))
Verify: ... Valid.

Lava> verify (constructive (flash (While high Skip)))
Verify: ... Falsifiable.
<high>
```

What about parallel composition? When is it acceptable for a body of a while loop to contain a parallel composition? Take, for example, the following Flash program:

```
possibleProblem inp = While high
                        ( IfThenElse inp (Skip , Delay)
                       :|| Delay
                        )
```

In principle, we should be able to execute this, since for all programs `p`, the program `p :|| Delay` takes time to execute. Let us analyse the resulting circuit:

```
possibleProblemCirc inp = flash (possibleProblem inp)

Lava>  verify (constructive possibleProblemCirc)
Verify: ... Falsifiable.
<low, high>
<high, low>
```

This shows that the simple compilation scheme we have used to illustrate our examples is not sufficiently robust to handle this example. Obviously, one can require that both sides of a parallel composition should take time (when appearing immediately inside the body of a while loop). However, this is a stringent and rather unsatisfactory restriction. A better solution would be to use a more complex compilation of loops, as used, for example, in [1].

5 Conclusions

Related Work
Hardware compilation of high-level languages has been around for quite a while. The approach has been considered potentially practical mainly since the introduction of programmable circuits. The compilation for various languages have since appeared in the literature, see e.g. [15, 17, 16, 1]. An introductory overview of the methodology appears in [26].

It is widely recognised that different styles of synchronous languages lend themselves more easily to different applications. In [13, 14], Maraninchi and Rémond present Mode-Automata — a combination of state diagram based descriptions (based on Argos [12]) with the dataflow language Lustre [8]. The semantics of the resulting language are defined by a translation into plain Lustre. The approach is thus very similar to the one we use, except that they use external programs to read mode-automata and translate them into Lustre. The embedded language approach we use, allows us to translate and reason about the new language at the same level as our base HDL Lava. This allows a much more versatile approach to language combination.

Poigné and Holenderski [19] present a theoretical framework for combining synchronous languages by using synchronous automata as the common semantic level. These ideas have been implemented in the SYNCHRONY WORKBENCH where programs written in one of a number of languages (Esterel, Lustre, Argos, and Synchronous Eifel) can be combined together. The main difference between their work and that presented in this paper, is that we embed the languages we use, and our intermediate language, Lava, is itself an embedded language. This gives us certain advantages: it is easier to add new languages to the framework, and language combination can be easily adapted depending on the requirements.

Discussion
We have presented a uniform framework in which it is easy to implement and

hence experiment with synthesisable behavioural languages. By embedding these languages in Lava, we are able to define their compilation in a natural and easy way and, at the same time, benefit from the verification tools connected to Lava to improve compilation and verify programs. We also benefit from, the fact that we can directly generate EDIF netlists or VHDL from the circuits generated by our embedded compilers.

Using this approach, we have shown that we can formally reason about programs at a number of levels. First, taking advantage of the fact that our programs are just data objects in Haskell, we can apply syntactic reasoning by defining functions which modify the program. Second, using the verification tools linked to Lava, we can define observers (in one of the languages) to verify properties of hardware described using either structural Lava, or some other language. Third, the compilation process itself can make use of the verification tools to check dynamic properties which may be needed to guarantee correct compilation.

Within our framework, we have implemented various languages or subsets of them, such as Esterel, Handel and Occam, fragments of process calculi such as CSP and CBS, and some restricted temporal logics. We have also embedded state machine descriptions and specification languages in the same framework. This included different control and data features including updatable variables, buffered and unbuffered channels, exceptions and broadcast communication. Describing the compilation of a language is rather straightforward, and in fact, we have successfully used this framework in the teaching of a graduate course on hardware description languages. The definition of the compilation function for a language is usually not much different from a denotational semantics of the language in terms of a dataflow network.

One of the important issues that we have not discussed in this paper is the question of the correctness of the compilation procedure. A number of approaches have been proposed [10, 22, 1] which are applicable to our compilation scheme. We are currently exploring how such proofs can also be presented uniformly within our framework. Preliminary work is encouraging and it is not difficult to prove that, for instance, the compilation of regular expressions presented in this paper satisfies regular expression equational axioms.

References

1. Gérard Berry. The constructive semantics of Pure Esterel. Unfinished draft, available from `http://www.esterel.org`, 1999.
2. Gérard Berry. The Esterel primer. Available from `http://www.esterel.org`, 2000.
3. K. Claessen. Safety property verification of cyclic circuits. In preparation, 2002.
4. K. Claessen and J. Hughes. QuickCheck: A light-weight tool for random testing of Haskell programs. In *Internatioanl Conference on Functional Programming*, 2000.
5. K. Claessen and M. Sheeran. A tutorial on Lava: A hardware description and verification system. Available from `http://www.cs.chalmers.se/~koen/Lava`, 2000.
6. Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *CHARME*. Springer, 2001.

7. E. M. Sentovich and K. J. Singh et al. SIS: A system for sequential circuit synthesis. Technical Report Berkeley, UCB/ERL M92/41, 1992.
8. N. Halbwachs. A tutorial of Lustre. Available from `http://www-verimag.imag.fr/SYNCHRONE`, 1993.
9. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
10. Jifeng He, Ian Page, and Jonathan Bowen. Towards a provably correct hardware implementation of Occam. In G.J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods*, number 683 in LNCS. Springer, 1993.
11. Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, 1996.
12. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*, number 630 in LNCS. Springer, 1992.
13. F. Maraninchi and Y. Rémond. Compositionality criteria for defining mixed-styles synchronous languages. In *International Symposium: Compositionality – The Significant Difference*, number 1536 in LNCS. Springer, 1997.
14. Florence Maraninchi and Yann Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*. Springer, 1998.
15. David May. Compiling Occam into silicon. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, University of Texas at Austin Year of Programming Series, chapter 3, pages 87–106. Addison-Wesley, 1990.
16. Ian Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.
17. Ian Page and Wayne Luk. In Wayne Luk and Will Moore, editors, *FPGAs*.
18. Simon Peyton Jones and John Hughes et al. Report on the programming language Haskell 98, a non-strict, purely functional language. Available from `http://haskell.org`, 1999.
19. A. Poigné and L. Holenderski. On the combination of synchronous languages. In *International Symposium: Compositionality – The Significant Difference*, number 1536 in LNCS, pages 490–514. Springer, 1997.
20. Pascal Raymond. Recognizing regular expressions by means of dataflow networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)*, number 1099 in LNCS. Springer, 1996.
21. F. Rocheteau and N. Halbwachs. Pollux, a Lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, 1991.
22. M. Schenke and M. Dossis. Provably correct hardware compilation using timing diagrams. Available from `http://semantik.Informatik.Uni-Oldenburg.DE/persons/michael.schenke/`, 1997.
23. Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, LNCS 1954. Springer, 2000.
24. T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference*, 1996.
25. Satnam Singh and Phil James-Roxby. Lava and JBits: From HDL to bitstream in seconds. In K.L. Pocek and J.M. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE Computer Society Press, 2001.
26. Niklaus Wirth. Hardware compilation: Translating programs into circuits. *Computer*, 31(6):25–31, 1998.