A Higher-Level Language for Hardware Synthesis

Richard Sharp and Alan Mycroft

Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK
{rws26,am}@cl.cam.ac.uk

Abstract. We describe SAFL+: a call-by-value, parallel language in the style of ML which combines imperative, concurrent and functional programming. Synchronous channels allow communication between parallel threads and π-calculus style channel passing is provided. SAFL+ is designed for hardware description and synthesis; a silicon compiler, translating SAFL+ into RTL-Verilog, has been implemented.
By parameterising functions over both data and channels the SAFL+ `fun` declaration becomes a powerful abstraction mechanism unifying a range of structuring techniques treated separately by existing HDLs.
We show how SAFL+ is implemented at the circuit level and define the language formally by means of an operational semantics.

1 Introduction

In 1975 a single Integrated Circuit contained several hundred transistors; by 1980 the number had increased to several thousand. Today, designs fabricated with state-of-the-art VLSI technology often contain several million transistors.

The exponential increase in circuit complexity has forced engineers to adopt higher-level tools. Whereas in the 1970s transistor and gate-level design was the norm, during the 1980s Register Transfer Level (RTL) Hardware Description Languages (HDLs) started to achieve wide-spread acceptance. Using such languages, designers were able to express circuits as hierarchies of components (such as registers and multiplexers) connected with wires and buses. The advent of this methodology led to a dramatic increase in productivity since, for some classes of design, time consuming place-and-route details could now be automated

More recently, *high-level synthesis* (sometimes referred to as *behavioural synthesis*) has started to have an impact on the hardware design industry. In the last few years commercial tools have appeared on the market enabling high-level, imperative languages (referred to as *behavioural languages* within the hardware community) to be compiled directly to hardware. Although these techniques undoubtedly offer increased levels of abstraction over RTL specification there is still room for even higher level HDLs, particularly when it comes to specifying interfaces between separate components (see Section 1.1). Since current trends predict that the exponential increase in transistor density will continue throughout the next decade, investigating higher-level tools for hardware description and synthesis will remain an important research area.

We present a language designed for hardware synthesis which, we argue, is higher level than existing behavioural synthesis packages. This paper builds on previous work in which we use SAFL [14] (Statically Allocated Functional Language) for circuit description. To reflect this we choose to call our new language SAFL+. Our optimising SAFL silicon compiler [16] has been extended to handle SAFL+ and the resulting system has been tested on a number of small designs. The contributions of this paper are:

– We extend SAFL with synchronous channels and assignment and argue that the resulting combination of functional, concurrent and imperative styles is a powerful framework in which to describe a wide range of hardware designs.
– Channel passing in the style of the π-calculus [12] is introduced. By parameterising functions over both data and channels the SAFL+ `fun` declaration becomes a powerful abstraction mechanism unifying a range of structuring techniques treated separately by existing HDLs (Section 2.3).
– We show how SAFL+ is implemented at the circuit-level (Section 3) and define the language formally by means of an operational semantics (Section 4).

1.1 The Motivation for Higher-Level HDLs

Register Transfer Level HDLs (e.g. RTL Verilog) describe hardware as a set of *blocks* parameterised over input and output ports. Once defined, the blocks can be instantiated (possibly multiple times) and explicitly connected with wires and buses. Although behavioural languages provide higher-level primitives for describing block internals, the block remains the primary abstraction mechanism used to structure large designs. For example, at the top level, a Behavioural Verilog program still consists of `module` declarations and instantiations albeit that the modules themselves contain higher-level constructs such as assignment, sequencing and while-loops.

Experience has shown that the notion of block is a useful syntactic abstraction, encouraging structure by supporting a "define-once, use-many" methodology. However, as a *semantic abstraction* it buys one very little; in particular: (i) any part of a block's internals can be exported to its external interface; and (ii) inter-block synchronisation mechanisms must be coded explicitly on an ad hoc basis.

Point (i) has the undesirable effect of making it difficult to reason about the global (inter-module) effects of local (intra-module) transformations. For example applying small changes to the local structure of a block (e.g. delaying a value's computation by one cycle) may have dramatic effects on the global behaviour of the program as a whole. We believe point (ii) to be particularly serious. Firstly, it leads to low-level implementation details scattered throughout a program—e.g. the definition of explicit control signals used to sequence operations in separate modules, or (arguably even worse) reliance on unwritten inter-module timing assumptions. Secondly, it inhibits compiler analysis: since inter-block synchronisation mechanisms are coded on an ad hoc basis it is very difficult for the compiler to infer a system-wide ordering on events. Based on these observations, we argue that structural blocks are not a high-level abstraction mechanism.

Through our previous work on SAFL we demonstrated that these problems can be alleviated by structuring code as a series of function definitions. The properties of functions make it easier to reason about the effects of local transformations. As a result we are able to make extensive use of source-to-source program transformation to assist with architectural exploration [15, 14]. The "invoke and wait for result" interface provided by functions removes the burden of explicitly specifying ad hoc inter-module synchronisation mechanisms. Furthermore our compiler is able to automatically infer a system-wide partial-ordering on events thus increasing the scope for global analysis and optimisation. For example consider the SAFL expression `f(g(3),h(5))`. From this (and the call-by-value property of SAFL) we can infer that `g` and `h` will be invoked in parallel, after which `f` will be invoked. Our SAFL compiler exploits a number of analyses and optimisations based on these event orderings (e.g. Soft Scheduling [17] and Data Validity Analysis [16]).

However, although SAFL is well-suited to describing certain types of hardware design, the facility for I/O is lacking. In addition we sometimes find the "call and wait for result" interface to be a little too restrictive. By extending SAFL with channel-communication, channel passing and assignment we intend SAFL+ to be a truly general purpose hardware description language. In Section 2.3 we show that SAFL+ supports a programming style which relaxes many of SAFL's restrictions without sacrificing analysibility.

Related Work Many parallels can be drawn between SAFL+ and the HardwareC [8] language since both provide synchronous channels and allow function definitions to be treated as shared resources. The major differences are: (*i*) whereas HardwareC offers purely imperative features, SAFL+ also supports a functional style (we find SAFL+'s `let`-construct for declaring immutable bindings to be particularly useful for describing data-dominated hardware); (*ii*) the expressivity of SAFL+ is greater due to our less restrictive scheduling policy [17]; and (*iii*) HardwareC provides a `block` primitive for structural-level declarations. In contrast SAFL+ only allows function declarations.

An interesting observation is that, as a direct result of SAFL+'s channel passing facility, all four of HardwareC's structuring primitives (`block`, `process`, `procedure` and `function`) can be seen as special cases of SAFL+ `fun` declarations (see Section 2.3). Since SAFL+ only requires a single structuring primitive it yields a simpler semantics.

Hoe and Arvind [5] describe TRAC: a hardware synthesis system which generates synchronous hardware from a high-level specification expressed as a term-rewriting system. Broadly speaking, terms correspond to states and rules correspond to combinatorial logic which calculates the next state of the system. Restrictions imposed on the structure of rewrite rules facilitate the static allocation of storage. This closely corresponds to the tail-recursion restriction imposed on SAFL+ programs to achieve static allocation [14].

Previous work on compiling declarative specifications to hardware has centred on how functional languages can be used as tools to aid the *structural* description of circuits—e.g. muFP [18], Lava [3] and the DDD algebra [7]. Functional programming techniques (such as higher order functions) are used to express concisely the regular, repetitive structures that often appear in hardware circuits. In this framework, different interpretations of primitive functions correspond to various operations (e.g. behavioural simulation and netlist generation). Our work differs from this in that we adopt a *behavioural* approach abstracting circuit-level details as far as possible. For example, when expressing a design in Lava, a programmer must explicitly ensure that design-level constraints (e.g. gate fan-out limits) are satisfied. In contrast we consider this to be a low-level detail: ensuring a circuit conforms to low-level design-rules is the job of our SAFL+ compiler. We take SAFL+ constructs (rather than gates) as primitive. Although this restricts the class of circuits we can describe to those which satisfy certain high-level properties, it increases the scope for high-level analysis and optimisation.

A number of languages have been developed which provide structural abstractions similar to those available in the Lava/muFP/DDD framework. For example HML [9] is one such language based on Standard ML [13]; Jazz [1] combines a polymorphic type-system with object oriented features; Hawk [11], like Lava, is embedded in Haskell, but focuses on simulation rather than synthesis.

2 SAFL+ Language Description

In this section we present the syntax of SAFL+ and informally describe its semantics. The language semantics are defined formally in Section 4.

SAFL+ is a concurrent, first-order, call-by-value language which, in the style of ML [13], supports a combination of functional and imperative programming. Function call arguments and `let`-definitions are evaluated in parallel; synchronous channels allow parallel threads to communicate with each other.

Function declarations take the form:

$$\texttt{fun}\ f(x_1, \ldots, x_k)\ [c_1, \ldots, c_j] = e \qquad \text{(where } k, j \geq 0\text{)}$$

We make a syntactic distinction between arguments used to pass data, $x_1, \ldots, x_k$, and arguments used to pass channels $c_1, \ldots, c_j$. Iteration is provided in the form of self-tail-recursive calls. As with SAFL, general recursion is forbidden to permit static allocation of storage [14]. Programs have a distinguished function, `main`, which represents an external world interface—at the hardware level it accepts values on an input port and may later produce a value on an output port. (In the current version of the language, `main` does not have channel parameters).

In the following a ranges over primitive functions (such as `+`, `*` etc.), f ranges over user-defined functions and l ranges over record field labels. We use r for array variables, c for channel variables, x for other variables, and i for integer constants. A vector of parameters, $x_1, \ldots, x_k$, is sometimes abbreviated to $\vec{x}$. The abstract syntax of SAFL+ programs, p, is presented in Fig. 1. The `static`

$$
\begin{array}{rcll}
e & \leftarrow & x \quad | \quad i \quad | \quad () & \text{(Variable, Integer constant, Unit constant)}\\
& | & \{l_1 = e_1, \ldots l_k = e_k\} \quad | \quad e.l & \text{(Record creation/selection)}\\
& | & r[e] \quad | \quad r[e] \ := \ e & \text{(Array read/write)}\\
& | & c? \quad | \quad c \,!\, e & \text{(Channel read/write)}\\
& | & a(e_1, \ldots, e_k) & \text{(Call to primitive function)}\\
& | & f(e_1, \ldots, e_k)[c_1, \ldots, c_j] & \text{(Call to user-defined function)}\\
& | & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \text{(Conditional)}\\
& | & \texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0 & \text{(Parallel let)}\\
& | & \texttt{static } p \texttt{ in } e & \text{(Local declarations)}\\
& | & e \parallel e \quad | \quad e;\, e & \text{(Parallel/sequential composition)}\\
\\
d & \leftarrow & \texttt{fun } f(x_1, \ldots, x_n)[c_1, \ldots, c_n] = e & \text{(Function declaration)}\\
& | & \texttt{channel } c & \text{(Channel declaration)}\\
& | & \texttt{channel external } c & \text{(I/O Channel declaration)}\\
& | & \texttt{array } [i]\ r & \text{(Array declaration)}\\
\\
p & \leftarrow & d \quad | \quad d\ p &
\end{array}
$$

Fig. 1. The abstract syntax of SAFL+ programs, p

construct, used to introduce local definitions, is provided purely for syntactic convenience. It is borrowed from the C language as a way of providing top-level definitions which are only accessible locally. It is not to be confused with the kind of *dynamic* channel-creation present in the π-calculus.

Our existing compiler provides a number of simple syntactic sugarings: the declaration `array [1] r` can be written `reg r`—when accessing such arrays one writes `r` instead of `r[0]`; functions without channel parameters can omit their square brackets completely (both in definition and calls); and `case`-statements are translated into nested conditionals in the usual way.

2.1 Resource Awareness

Our approach is to model hardware as a fixed set of communicating and (possibly) shared resources. As can be seen from Fig. 1, a program consists of a series of resource declarations. There are three different types of resource, each of which addresses a key element of hardware design:

Resource type	Purpose	Hardware Representation
Function	Computation	General Purpose Logic
Channel	Communication	Buses, Wires and Control Logic
Array	Storage	Memories or Registers

We say that SAFL+ is *resource-aware* since each declaration, d, (be it a function, channel or array declaration) corresponds to a *single* hardware block, H_d. Multiple references to d at the source-level (e.g. multiple calls to a function or multiple assignments to an array) correspond to the sharing of H_d at the circuit-level.

A call, $f(\vec{x})[\vec{c}]$, corresponds to: (i) acquiring mutually exclusive access to resource, H_f; (ii) passing data $\vec{x}$ and channel-parameters $\vec{c}$ into H_f; (iii) waiting for H_f to terminate; and (iv) latching[1] the result from H_f's shared output.

For a concrete example of SAFL+, Fig. 3 describes a lock shared between functions `f1` and `f2`. Synthesising this example leads to three resources: $H_{\mathtt{f1}}$, $H_{\mathtt{f2}}$ and $H_{\mathtt{lock}}$ with $H_{\mathtt{lock}}$ shared between resources $H_{\mathtt{f1}}$ and $H_{\mathtt{f2}}$.

Sharing issues, such as ensuring mutually exclusive access to resources, are dealt with automatically by our compiler. In [15] we describe how arbiters are generated to protect shared resources from concurrent accesses. A global analysis is presented which allows redundant arbiters to be optimised away.

Resource-awareness means that, although a SAFL+ compiler is free to optimise the internals of `fun` definitions, it must respect the circuit structure specified by the programmer (i.e. one declaration = one hardware-level resource). We apply source-to-source program transformation as a pre-compilation phase to express resource sharing/duplication and other area-time tradeoffs [14]. Although such transformations are applied manually at the moment, tools to assist with the transformation process and automatically explore the design space are currently being developed.

2.2 Channels and Channel Passing

SAFL+ provides synchronous channels to allow parallel threads to synchronize and transfer information. Channels can be used to transfer data locally within a function, or globally, between concurrently executing functions.

Our channels generalise Occam [6] and Handel-C [4] channels in a number of ways: SAFL+ channels can have any number of readers and writers, are bidirectional and can connect any number of parallel processes. As in the π-calculus, if there are multiple readers and multiple writers all wanting to communicate on the same channel then a single reader and a single writer are chosen non-deterministically.

At the hardware level a channel is implemented as a many-to-many communications bus supporting the atomic transfer of single values between readers and writers (see Section 3). No language-support is provided for *bus-transactions* (e.g. lock the bus for 20 cycles and write the following sequence of data values onto it). In Section 2.3 a SAFL+ code fragment is presented which shows how such transactions can be implemented by using explicit locking.

Channels declared as `external` are used for I/O: writing to an external channel corresponds to an output action; reading an external channel corresponds to reading an input. There is no synchronisation on external channels although writes are guaranteed to occur under mutual exclusion. For example, for an external channel `c`, the only two possible output sequences occurring as a result of evaluating expression `(c!2 || c!3)` are $\langle 2, 3\rangle$ or $\langle 3, 2\rangle$. (See Section 4).

[1] A data-flow analysis is used to optimise these latches away under certain circumstances [16].

The code in Fig. 2 illustrates channel-passing in SAFL+ by defining two resources parameterised over channel parameters: `Accumulate` reads integers from a channel, returning their total when a 0 is read; and `GenNumbers` writes a decreasing stream of integers to a channel, terminating when 0 is reached. The function, `sum(x)` calculates the sum of the first `x` integers by composing the two resources in parallel and linking them with a common channel, `connect`. (Note that the parallel composition operator, `||`, waits for both its components to terminate before returning the value of the rightmost one.)

```
fun Accumulate(state) [c] =
  let val read_value = c?
   in if read_value=0 then state
                     else Accumulate(state+read_value)  end

fun GenNumbers(state) [c] =
  c!state; if c=0 then () else GenNumbers(state-1)

fun sum(x) =
  static channel connect
  in GenNumbers(x) [connect] || Accumulate(0)[connect]   end
```

Fig. 2. Example showing SAFL+ channel-passing

Channel parameters are not passed on recursive calls. Once a function resource, f, has been acquired by means of an *external* (i.e. non-recursive) call, $f(\vec{x})[\vec{c}]$, f's channel parameters remain bound to $\vec{c}$ until f terminates. See the operational semantics presented in Section 4 for a more precise description.

2.3 The Motivation for Channel Passing

By parameterising functions over both data and channel parameters, the SAFL+ `fun` definition becomes a powerful abstraction mechanism, encapsulating a wide range of structuring primitives treated separately in existing HDLs:

- Pure functions can be expressed by omitting channel parameters:
  `fun f(x,y) = ...`
- Structural-level blocks (*cf*. Verilog's `module` construct) can be expressed as non-terminating `fun` declarations parameterised over channels:
  `fun module() [in1,in2,out] = ...; module()`
- HardwareC `process` declarations can be expressed as non-terminating `fun` definitions (possibly without channel or data parameters):
  `fun process() = ...; process()`
- HardwareC `procedures` can be expressed as `fun` declarations that return a unit result:
  `fun procedure(x,y) = ...; ()`

As well as unifying a number of common abstraction primitives, SAFL+ also supports a style of programming not exploited by existing HDLs. Recall the definition of `Accumulate` in Section 2.2. The `Accumulate` function can be seen as a hybrid between a structural-level block (since it is parameterised over a port, `c`) and a function (since it terminates, returning a result). More generally, by passing in locally defined channels, a caller, f, is able to synchronise and communicate with its callee, g, during g's execution. For example, consider the SAFL+ code in Fig. 3 which declares a lock shared between functions `f1` and

```
fun lock()[acquired, release] = acquired!(); release?

fun f1() = static channel go   channel done
           in (lock()[go,done] ||
                (go?;   (* f1's critical region *) done!()) )   end

fun f2() = static channel go   channel done
           in (lock()[go,done] ||
                 (go?;  (* f2's critical region *) done!()) )   end
```

Fig. 3. Mutual exclusion implemented by `lock`

`f2` to implement mutual exclusion from the critical regions. The `lock` function is parameterised over two channels: `acquired` is signalled as soon as `lock` starts executing, indicating to the caller that the lock has been acquired; `release` is used by the caller to signal that it has finished with the lock (at which point `lock` terminates). Recall that resource-awareness means that `lock` represents a single resource shared by functions `f1` and `f2`: the compiler ensures that only one caller can acquire it at a time. By passing in locally defined channels, functions `f1` and `f2` are able to communicate with `lock` during its execution.

3 Translating SAFL+ to Hardware

In [16] we describe in detail how we translate the functional subset of SAFL+ into synchronous hardware. The basic principle involves translating each function definition into a single hardware block consisting of logic to serialise concurrent accesses and registers to latch arguments. Tail-recursive calls[2] are translated into feedback loops at the circuit level.

Here we extend this by showing how the non-functional features (i.e. channels and arrays) can be integrated into our existing framework. As in [16] we adopt the graphical convention that thick lines represent data-wires and thin lines represent control signals.

A channel is translated into a shared bus surrounded with the necessary control logic to arbitrate between waiting readers and writers. Fig. 4 shows

[2] The only form of recursion allowed is tail-recursion.

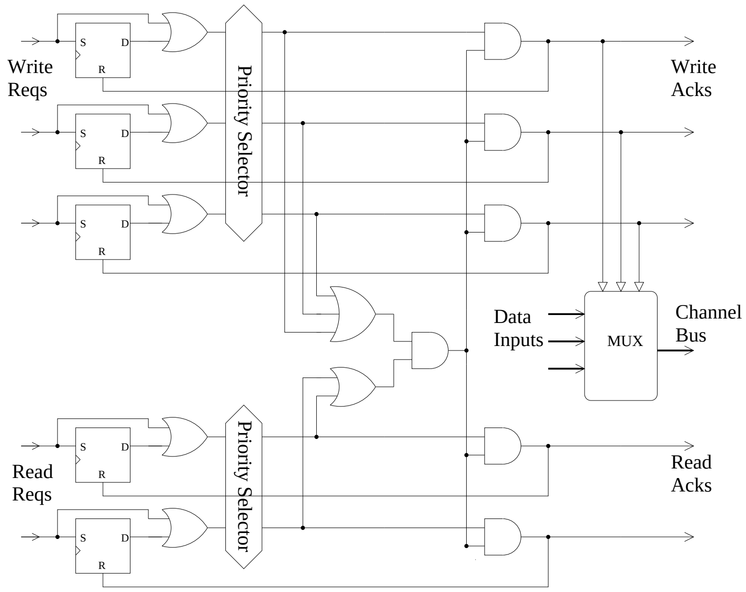

Fig. 4. A Channel Controller. The synchronous RS flip-flops (R-dominant) are used to latch pending requests (represented as 1-cycle pulses). Static fixed priority selectors are used to arbitrate between multiple requests. The 3 data-inputs are used by the three writers to put data onto the bus.

channel control circuitry in a case where there are two readers and three writers. Since we are primarily targeting FPGAs we choose to multiplex data onto the bus rather than using tri-state buffers. To perform a read operation the reader signals its read-request and blocks until the corresponding read-acknowledge is signalled. We extend the convention of our previous compiler [16] that the read-acknowledge line remains high for one cycle during which time the reader samples the data from the channel. To perform a write operation the writer places the data to be written onto a channel's data-input and signals the corresponding write-request line; the writer blocks until the corresponding write-acknowledge is signalled. Our current compiler synthesises static fixed-priority arbiters to resolve multiple simultaneous read requests or multiple simultaneous write requests. However, since the SAFL+ semantics do not specify an arbitration policy, future compilers are free to exploit other selection mechanisms.

Our SAFL+ compiler performs a static flow-analysis to determine which *actual channels* (those bound directly by the `channel` construct) a given formal-channel-parameter may range over. This information enables the compiler to statically connect each channel operation (read or write) to every possible actual channel that it may need to access dynamically. At the circuit level channel values are represented as small integers which are passed as additional parameters on a function call.

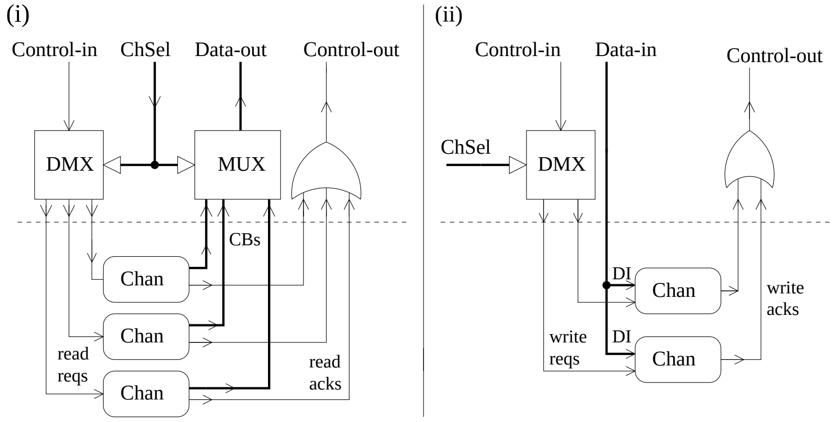

Fig. 5. (*i*) A READ node connected to 3 channels; (*ii*) A WRITE node connected to 2 channels. Each of the boxes labelled 'Chan' is a channel (as in Fig. 4). Although each such channel may well have other readers/writers these are not shown in the figure. The data-wires labelled 'CB' are the channel buses, those labelled 'DI' are channels' data-inputs (multiplexed onto the channel buses—see Fig. 4). 'ChSel' is the channel-select-input. Note that (although not shown in this figure) channel buses may be shared among many readers. The dotted line represents the boundary between the resource performing the channel operation and the channels themselves.

Our intermediate code [16] is augmented with READ and WRITE nodes representing channel operations. In cases where our flow-analysis detects that a channel operation may refer to a number of possible actual channels, multiplexers and demultiplexers are used to dynamically route to the appropriate channel. READ nodes have a control-input (used to signal the start of the operation), a control-output (used to signal the completion of the operation), a channel-select-input (used to select which actual channel to read from) and a data-output (the result of the read operation). Similarly WRITE nodes have a control-input, a control-output, a channel-select-input and a data-output. Fig. 5 shows READ and WRITE nodes connected to multiple channels.

We extend the translation of `fun` declarations described in [16] to include extra registers to latch channel-parameters. At the circuit-level channel-parameters are fed into the select lines of the multiplexers and demultiplexers seen in Fig. 5. In this example 'ChSel' would be read directly from the registers storing the enclosing function's channel-parameters.

Arrays are represented as RAMs wrapped up in the necessary logic to arbitrate between multiple concurrent accesses. Our compiler translates array declarations, `array [i] r`, into SAFL+ function definitions with signature:

```
fun r (addr:int, data:int, wr_select:bit) : int
```

Calling `r` always returns the value stored at memory location `addr`. If `wr_select` is 1 then location `addr` is updated to contain `data`. Hence array assignments, `r[e1] := e2`, are translated into function calls of the form `r(e1,e2,1)` and array accesses, `r[e]`, are translated into calls of the form `r(e,0,0)`. Treating arrays as SAFL+ functions in this way allows us to use the compiler's existing machinery to synthesise the necessary logic to serialise concurrent accesses to the array and latch address lines. The compiler automatically generates the body of `r`, which consists solely of RAM.

4 Operational Semantics for SAFL+

In this section we define the meaning of the SAFL+ language formally through an operational semantics. Although, at first sight, the semantics may seem theoretical and far-removed from hardware-implementation we argue that this is not the case. It is worth pointing out that many of the symbols in Fig. 9 have a direct correspondence to circuit-level components. For example, *channel resources*, $\langle v \rangle_c$ (see below), represent channel controller circuits (as shown in Fig. 4) and the (*Call*) rule (see Fig. 9) corresponds directly to transferring data into the callee's argument registers (circuits corresponding to this are presented in [16]).

A SAFL+ program consists of a series of function definitions of the form:

$$\texttt{fun}\ f\ (x_1, \ldots, x_k)\ [c_1, \ldots, c_j] = b_f$$

We write b_f for the body of function, f, $x_1, \ldots, x_k$ for formal parameters and $c_1, \ldots, c_j$ for channel parameters. For the sake of brevity, we define $\vec{x}$ to mean $x_1, \ldots, x_k$ and, similarly, $\vec{c}$ to mean $c_1, \ldots, c_j$.

Due to the static nature of SAFL+, we can simplify matters by assuming that: (i) SAFL+ programs have been α-converted to make all variable names distinct; and (ii) scope-flattening has been performed, bringing local declarations to the top level and eliminating `static` statements. (Note that bringing a locally defined function to the top level may require extra arguments to be added to the function in order to pass in values for its free variables.)

We give the semantics by describing how one *program state*, P, evolves into another, say Q, by means of a *transition*: $P \xrightarrow{\alpha} Q$, where α represents an optional I/O action taking one of the following forms:

$\bar{\mathbf{c}}\langle v \rangle$	Output v on external channel $\mathbf{c}$
$\mathbf{c}(v)$	Read a value v from external channel $\mathbf{c}$
$go(\vec{v})$	Pass parameters $\vec{v}$ into the `main` function
$done(v)$	Read result v from the `main` function

Note that we use a bold-face $\mathbf{c}$ to range over external channels (in contrast to c, which ranges over non-external channels).

A program state consists of a parallel composition of *function resources*, *channel resources* and *array resources* (see Fig. 6). Our presentation borrows notation and ideas from Marlow *et al* [10].

Each non-external channel declaration, `channel` c, corresponds to a channel resource. When an empty channel resource (written $\langle\rangle_c$) reacts with a waiting writer a value, v, is transferred and c becomes full (written $\langle v \rangle_c$). On reacting with a waiting reader, the value is consumed and the c enters an acknowledge state (written $\langle \mathrm{Ack} \rangle_c$). The Ack interacts with the writer, notifying it that communication has taken place and returning c to the empty state, $\langle \rangle_c$. The explicit use of Ack models the synchronous nature of SAFL+ channels ensuring that a writer is blocked until its data has been consumed by a reader.

Array resources, $[\mathcal{S}_i]_r$, correspond to array declarations, `array [i]` r. The contents of the array, $\mathcal{S}_i$, is a function mapping indexes $0 \ldots (i-1)$ onto values. We write $\mathcal{S}_i\{j \mapsto v\}$ to denote the function which is as $\mathcal{S}_i$ but maps index j onto value v. Accessing elements outside the bounds of an array leads to undefined behaviour. To reflect this we define $\mathcal{S}_i(j)$ to be an undefined value if $j \geq i$. Furthermore if $j \geq i$ then $\mathcal{S}_i\{j \mapsto v\}$ represents an undefined state mapping indexes $0 \ldots (i-1)$ onto undefined values.

Each SAFL+ function declaration, `fun` $f\ (\vec{x}\,)\ [\vec{c}\,] = b_f$, is represented by a function resource. At any given time a function resource may be *busy* (performing a computation) or *available* (waiting to perform a computation). An available function resource, f, is written $\mathbb{O}_f$, signifying that f is not in use; a busy function resource takes the form $(\!|e|\!)_f$ signifying that f is currently in *evaluation state* e. The syntax of evaluation states (see Fig. 6) is essentially the same as the syntax of SAFL+ expressions augmented with the $\mathcal{W}_g$ construct which represents waiting for a result from function resource g. To save space, conditional expressions, `if` e_1 `then` e_2 `else` e_3, are shortened to $e_1 \triangleright e_2 : e_3$.

As with the Chemical Abstract Machine [2] program states can be viewed as a "solution" of reacting resources. We formalise this notion in the standard way by defining structural congruence, $\equiv$, to be the least congruence which satisfies the (*Comm*) and (*Assoc*) equations of Fig. 7. Rule (*Par*) allows transitions within parallel compositions and (*Equiv*) makes it possible to use the structural congruence relation to bring different parts of the program state together.

SAFL+ is an implicitly parallel language—an expression may contain a number of sub-expressions which can be evaluated concurrently. To formalise this notion we use a *context*, $\mathbb{E}$, to highlight the parts of an evaluation state which can be evaluated concurrently (see Fig. 8). Intuitively a context is an evaluation state $\mathbb{E}[\cdot]$ with a hole $[\cdot]$ into which we can insert an evaluation state, e, to derive a new evaluation state $\mathbb{E}[e]$.

A useful mental model is to consider a frontier of evaluation which is defined by $\mathbb{E}$ and advanced by applying the transition rules (see Section 4.1 and Fig. 9).

4.1 Transition Rules

For clarity, we present the transition rules for SAFL+ in two parts: Fig. 9(a) gives the rules for SAFL+ without channel passing. Section 4.2 explains how the rules are modified to handle channel passing in Fig. 9(b).

Substitution of values, $v_1 \ldots v_n$, for variables, $x_1 \ldots x_n$ in an evaluation state, e, is written, $\{v_1/x_1, \ldots, v_n/x_n\}e$, and for convenience abbreviated to $\{\vec{v}\,/\vec{x}\,\}e$.

$$
\begin{array}{lcll}
P & \leftarrow & (\!|e|\!)_f & \text{(busy function)} \\
 & | & \mathbb{O}_f & \text{(available function)} \\
 & | & \langle\rangle_c & \text{(empty channel)} \\
 & | & \langle v \rangle_c & \text{(full channel, holding value } v\text{)} \\
 & | & \langle \text{Ack} \rangle_c & \text{(channel in acknowledge state)} \\
 & | & P \mid P & \text{(parallel composition)} \\
 & & & \\
e & \leftarrow & v & \text{(value)} \\
 & | & \mathcal{W}_g & \text{(awaiting result from } g\text{)} \\
 & | & \{l_1 = e, \ldots, l_k = e\} \quad | \quad e.l & \text{(as in Fig. 1)} \\
 & | & x \quad | \quad f(e, \ldots, e)[c_1, \ldots, c_n] \quad | \quad a(e, \ldots, e) & \ldots \\
 & | & \mathtt{let}\ (x_1, \ldots, x_k) = (e, \ldots, e)\ \mathtt{in}\ e & \ldots \\
 & | & e \triangleright e : e \quad | \quad c\,!\,e \quad | \quad c? \quad | \quad e \parallel e & \ldots \\
 & | & e; e \quad | \quad x[e] := e \quad | \quad x[e] & \text{(as in Fig. 1)} \\
 & & & \\
v & \leftarrow & i & \text{(integer, } i \in \mathbb{N}\text{)} \\
 & | & () & \text{(unit)}
\end{array}
$$

Fig. 6. The Syntax of Program States, P, Evaluation States, e, and values, v

$$
\begin{array}{rcll}
P \mid Q & \equiv & Q \mid P & (\mathit{Comm}) \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & (\mathit{Assoc})
\end{array}
$$

$$
\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R}\ (\mathit{Par}) \qquad\qquad \frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q}\ (\mathit{Equiv})
$$

Fig. 7. Structural congruence and structural transitions

$$
\begin{array}{l}
\mathbb{E} \leftarrow [\cdot] \\
\quad | \ f(\mathbb{E}, e_2, \ldots, e_k) \quad | \quad \ldots \quad | \quad f(e_1, \ldots, e_{k-1}, \mathbb{E}) \\
\quad | \ a(\mathbb{E}, e_2, \ldots, e_k) \quad | \quad \ldots \quad | \quad a(e_1, \ldots, e_{k-1}, \mathbb{E}) \\
\quad | \ \{n_1 = \mathbb{E},\ n_2 = e_2,\ \ldots,\ n_k = e_k\} \\
\qquad\qquad\qquad\qquad \ldots \\
\quad | \ \{n_1 = e_1,\ \ldots,\ n_{k-1} = e_{k-1},\ n_k = \mathbb{E}\} \\
\quad | \ \mathtt{let}\ (x_1, \ldots, x_k) = (\mathbb{E}, e_2, \ldots, e_k)\ \mathtt{in}\ e \\
\qquad\qquad\qquad\qquad \ldots \\
\quad | \ \mathtt{let}\ (x_1, \ldots, x_k) = (e_1, \ldots, e_{k-1}, \mathbb{E})\ \mathtt{in}\ e \\
\quad | \ r[\,\mathbb{E}\,] \quad | \quad r[\,\mathbb{E}\,] := e \quad | \quad r[\,e\,] := \mathbb{E} \quad | \quad \mathbb{E}.l \\
\quad | \ \mathbb{E} \triangleright e_1 : e_2 \quad | \quad \mathbb{E}; e \quad | \quad \mathbb{E} \parallel e \quad | \quad e \parallel \mathbb{E} \quad | \quad c\,!\,\mathbb{E}
\end{array}
$$

Fig. 8. A context, $\mathbb{E}$, defining which sub-expressions may be evaluated in parallel

The rules in Fig. 9 are divided into six categories:

- (*Call*) and (*Return*) deal with interaction between functional resources.
- (*Ch-Write*), (*Ch-Read*) and (*Ch-Ack*) model communication over channels.
- (*Input*) and (*Output*) deal with I/O through external channels.
- (*Ar-Write*) and (*Ar-Read*) handle access to array resources.
- (*Start*) and (*End*) correspond to external call/return of `main()`.
- The remainder of the rules represent local computation within a function.

Note that the left hand side of the (*Tail-Rec*) rule is not enclosed in a context. This reflects the fact that tail recursive calls cannot occur in parallel with any other expressions; hence a context is unnecessary.

4.2 Semantics for Channel Passing

To deal with channel passing, function resources need to store the channel parameters passed from an external call. We use the notation $(\!| \cdot |\!)_f^{\vec{c}}$ to represent a function resource which has been called with actual channel parameters, $\vec{c}$. For convenience we sometimes omit the channel parameters from a rule, defining $(\!| e_1 |\!)_f \rightarrow (\!| e_2 |\!)_g$ to mean $(\!| e_1 |\!)_f^{\vec{c}} \rightarrow (\!| e_2 |\!)_g^{\vec{c}}$.

The (*Call*), (*Ret*) and (*Tail-Rec*) rules are modified for channel passing as shown in Fig. 9(b).

5 Conclusions and Further Work

This paper has introduced and formally defined the SAFL+ language, motivating its use for hardware description and synthesis. We argue that the major advantages of SAFL+ over most existing high-level synthesis languages are:

- The combination of resource-awareness and channel-passing makes SAFL+ `fun` declarations a very powerful abstraction mechanism. Both structural blocks and functions can be seen as special cases of `fun` declarations.
- By structuring programs as a series of function definitions (as opposed to a collection of structural blocks), SAFL+ supports a wide range of analyses and transformations which are not applicable to conventional HDLs.
- SAFL+ has a formally defined semantics.

The project is in its early stages. Although we have implemented a silicon compiler for SAFL+ and tested it on small examples, we have yet to use the system to build a large system-on-a-chip design. This is very high-priority for our future work. Using SAFL+ to construct a large hardware design will test both the expressivity of the language and the efficiency of our compiler.

The translation of SAFL+ to hardware given in this paper (Section 3) outlines one of many possible implementation techniques. In future we plan to investigate the translation of SAFL+ to globally-asynchronous-locally-synchronous (GALS) hardware. This involves mapping function-resources into separate clock domains and extending our compiler to automatically instantiate the necessary inter-clock-domain interfaces.

(a) Rules for SAFL+ without channel passing:

$$
\begin{array}{rcll}
(\!|\mathbb{E}[g(v_1,\ldots,v_k)]|\!)_f \mid \mathbb{O}_g & \longrightarrow & (\!|\mathbb{E}[\mathcal{W}_g]|\!)_f \mid (\!|\{\vec{v}/\vec{x}\}b_g|\!)_g \qquad f \neq g & (\mathit{Call}) \\
(\!|v|\!)_f \mid (\!|\mathbb{E}[\mathcal{W}_f]|\!)_g & \longrightarrow & \mathbb{O}_f \mid (\!|\mathbb{E}[v]|\!)_g & (\mathit{Return}) \\
\\
(\!|\mathbb{E}[c\ !\ v]|\!)_f \mid \langle\rangle_c & \longrightarrow & (\!|\mathbb{E}[\mathcal{W}_c]|\!)_f \mid \langle v\rangle_c & (\mathit{Ch\text{-}Write}) \\
(\!|\mathbb{E}[c?]|\!)_f \mid \langle v\rangle_c & \longrightarrow & (\!|\mathbb{E}[v]|\!)_f \mid \langle \mathrm{Ack}\rangle_c & (\mathit{Ch\text{-}Read}) \\
(\!|\mathbb{E}[\mathcal{W}_c]|\!)_f \mid \langle \mathrm{Ack}\rangle_c & \longrightarrow & (\!|\mathbb{E}[()]|\!)_f \mid \langle\rangle_c & (\mathit{Ch\text{-}Ack}) \\
\\
(\!|\mathbb{E}[\mathbf{c}\ !\ v]|\!)_f & \xrightarrow{\bar{\mathbf{c}}\langle v\rangle} & (\!|\mathbb{E}[()]|\!)_f & (\mathit{Output}) \\
(\!|\mathbb{E}[\mathbf{c}?]|\!)_f & \xrightarrow{\mathbf{c}(v)} & (\!|\mathbb{E}[v]|\!)_f & (\mathit{Input}) \\
\\
(\!|\mathbb{E}[r[v_1]\ :=\ v_2]|\!)_f \mid [\mathcal{S}_i]_r & \longrightarrow & (\!|\mathbb{E}[()]|\!)_f \mid [\mathcal{S}_i\{v_1 \mapsto v_2\}]_r & (\mathit{Ar\text{-}Write}) \\
(\!|\mathbb{E}[r[v]]|\!)_f \mid [\mathcal{S}_i]_r & \longrightarrow & (\!|\mathbb{E}[\mathcal{S}_i(v)]|\!)_f \mid [\mathcal{S}_i]_r & (\mathit{Ar\text{-}Read}) \\
\\
\mathbb{O}_{\mathrm{main}} & \xrightarrow{go(\vec{v})} & (\!|\{\vec{v}/\vec{x}\}b_{\mathrm{main}}|\!)_{\mathrm{main}} & (\mathit{Start}) \\
(\!|v|\!)_{\mathrm{main}} & \xrightarrow{done(v)} & \mathbb{O}_{\mathrm{main}} & (\mathit{End}) \\
\\
(\!|\mathbb{E}[a(v_1,\ldots,v_k)]|\!)_f & \longrightarrow & (\!|\mathbb{E}[v]|\!)_f \quad \text{where } v = a(v_1,\ldots,v_k) & (\mathit{PrimOp}) \\
(\!|\mathbb{E}[\{\ldots,l=v,\ldots\}.l]|\!)_f & \longrightarrow & (\!|\mathbb{E}[v]|\!)_f & (\mathit{RecSelect}) \\
(\!|\mathbb{E}[0 \triangleright e_1 : e_2]|\!)_f & \longrightarrow & (\!|\mathbb{E}[e_2]|\!)_f & (\mathit{CFalse}) \\
(\!|\mathbb{E}[n \triangleright e_1 : e_2]|\!)_f & \longrightarrow & (\!|\mathbb{E}[e_1]|\!)_f \qquad n \neq 0 & (\mathit{CTrue}) \\
(\!|\mathbb{E}[\mathtt{let}\ \vec{x} = \vec{v}\ \mathtt{in}\ e]|\!)_f & \longrightarrow & (\!|\mathbb{E}[\{\vec{v}/\vec{x}\}e]|\!)_f & (\mathit{Let}) \\
(\!|\mathbb{E}[v;e]|\!)_f & \longrightarrow & (\!|\mathbb{E}[e]|\!)_f & (\mathit{Seq}) \\
(\!|\mathbb{E}[v_1 \parallel v_2]|\!)_f & \longrightarrow & (\!|\mathbb{E}[v_2]|\!)_f & (\mathit{Par}) \\
(\!|f(v_1,\ldots,v_k)|\!)_f & \longrightarrow & (\!|\{\vec{v}/\vec{x}\}b_f|\!)_f & (\mathit{Tail\text{-}Rec})
\end{array}
$$

(b) Modifications for Channel Passing:

$$
\begin{array}{rcll}
(\!|\mathbb{E}[g(v_1,\ldots,v_k)[d_1,\ldots,d_j]]|\!)_f^{\vec{c'}} \mid \mathbb{O}_g & \longrightarrow & (\!|\mathbb{E}[\mathcal{W}_g]|\!)_f^{\vec{c'}} \mid (\!|\{\vec{d}/\vec{c},\vec{v}/\vec{x}\}b_g|\!)_g^{\vec{d}} & (\mathit{Call}) \\
(\!|v|\!)_f^{\vec{c}} \mid (\!|\mathbb{E}[\mathcal{W}_f]|\!)_g^{\vec{d}} & \longrightarrow & \mathbb{O}_f \mid (\!|\mathbb{E}[v]|\!)_g^{\vec{d}} & (\mathit{Ret}) \\
(\!|f(v_1,\ldots,v_k)|\!)_f^{\vec{c'}} & \longrightarrow & (\!|\{\vec{c'}/\vec{c},\ \vec{v}/\vec{x}\}b_f|\!)_f^{\vec{c'}} & (\mathit{Tail\text{-}Rec})
\end{array}
$$

Fig. 9. Transition Rules for SAFL+

Acknowledgement

This work was supported by (UK) EPSRC grant GR/N64256 "A Resource-Aware Functional Language for Hardware Synthesis"; the first author was also sponsored by AT&T Research Laboratories Cambridge.

References

1. The Jazz Synthesis System. See: `http://www.exentis.com/jazz`.
2. Berry, G., and Boudol, G. The chemical abstract machine. *Theoretical Computer Science 96* (1992), 217–248.
3. Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. Lava: Hardware description in Haskell. In *Proceedings of the 3rd International Conference on Functional Programming* (1998), SIGPLAN, ACM.
4. Celoxica (Ltd.). Handel-C language datasheet. Available from Celoxica: `http://www.celoxica.com`.
5. Hoe, J., and Arvind. Hardware synthesis from term rewriting systems. In *Proceedings of X IFIP International Conference on VLSI* (1999).
6. Inmos (Ltd.). *Occam 2 Reference Manual.* Prentice Hall, 1998.
7. Johnson, S., and Bose, B. DDD: A system for mechanized digital design derivation. Tech. Rep. 323, Indiana University, 1990.
8. Ku, D., and De Micheli, G. HardwareC—a language for hardware design (version 2.0). Tech. Rep. CSL-TR-90-419, Stanford University, 1990.
9. Li, Y., and Leeser, M. HML, a novel hardware description language and its translation to VHDL. *Transactions on VLSI Systems*, 1 (February 2000).
10. Marlow, S., Peyton Jones, S., Moran, A., and Reppy, J. Asynchronous exceptions in Haskell. To appear. *Proceedings of ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)* 2001.
11. Matthews, J., Cook, B., and Launchbury, J. Microprocessor specification in Hawk. In *Proceedings of the IEEE International Conference on Computer Languages* (1998).
12. Milner, R. The polyadic π-calculus: A tutorial. Tech. Rep. ECS-LFCS-91-180, University of Edinburgh, October 1991.
13. Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised).* MIT Press, 1997.
14. Mycroft, A., and Sharp, R. A statically allocated parallel functional language. In *Proceedings of the International Conference on Automata, Languages and Programming* (2000), vol. 1853 of *LNCS*, Springer-Verlag.
15. Mycroft, A., and Sharp, R. Hardware/software co-design using functional languages. In *Proceedings of TACAS* (2001), vol. 2031 of *LNCS*, Springer-Verlag.
16. Sharp, R., and Mycroft, A. The FLaSH compiler: Efficient circuits from functional specifications. Tech. Rep. tr.2000.3, AT&T Laboratories Cambridge, 2000.
17. Sharp, R., and Mycroft, A. Soft scheduling for hardware, 2001. To appear. *Proceedings of the 8th International Static Analysis Symposium.*
18. Sheeran, M. muFP, a language for VLSI design. In *Proceedings of the ACM Symposium on LISP and Functional Programming* (1984).