ABSTRACT

In this paper the I/O models of the functional languages Haskell, Clean, Erlang and Amanda are researched. We are interested in how their different way of expressing I/O can be used in functional hardware description languages to describe the communication between separate components in hardware. To answer this question a case study has been done implementing a PS/2 interface. For each language it is evaluated whether it is able to express and simulate all the desired behaviour and how natural and concise the resulting source code is. Based on this we conclude that message passing, monads and uniqueness typing might be suitable. Further research is required to determine which approach is overall the best.

Keywords

Functional programming, hardware synthesis, I/O, Haskell, Amanda, Erlang, Clean

1. INTRODUCTION

A functional language used as a hardware description language has great advantages over traditional languages like VHDL. It allows for a much more natural and concise way of describing hardware and has been successfully used to build synchronous single actor systems.[3, 9] In real-life a piece of hardware is often part of a greater multi-actor system in which the other actors are external to the component being described. Communication between these components thus requires some sort of I/O. When describing a keyboard this I/O could be a key press from the user or sending the key codes to the connected device.

First we have to consider what this I/O exactly entails. In its most basic form I/O means reading or writing a high or low signal to a physical pin. For more advanced applications it is necessary however to be able to react to input without having to actively listen for it, for example when reacting to key presses sent by a keyboard. Moreover, different communication protocols, such as USB (and every other serial interface), require accurately timed sequential I/O operations. These aspects, and undoubtedly more, will need to be expressed in whatever language is used to describe interactive hardware.

If we want to use a functional language to do this we should have a look at the way I/O is normally described in it. Different models for expressing I/O exist, originating from different languages.

A construct called the I/O-Monad is being used in Haskell. Functions inside this I/O Monad are called actions. They are called impure or not referentially transparent because calls to these functions can not be safely substituted for their result. This is because the functions have side-effects or their return value is unpredictable. Actions can be bound together to form more complex I/O operations. The result of an action can never be used directly, but only from within the monad. This makes sure that the side-effects do not spread to the rest of the program ensuring a strict separation between the impure and pure parts of the program.[1, 5]

A different approach is taken by uniqueness typing. The functional language Clean uses it to make referentially transparent I/O possible. Every function that handles I/O is passed a unique parameter; a parameter that can never be referenced again. As a result of this a specific call to this function could safely be replaced by its result. In other words it’s referentially transparent. This specific call to the function will never generate a different result simply because this exact call can never be made again.[5]

Although technically not an I/O model, Erlang’s message passing mechanism deserves a look. In contrast to Clean and Haskell the functional parallel programming language Erlang is not (completely) referentially transparent. It also differs in the fact that function bodies exist out of multiple commands that are executed sequentially, much like procedures in imperative languages. In an Erlang program multiple parallel processes can communicate with each other by means of message passing. This happens when sender and receiver execute a send and receive operation respectively.[2]

Finally there is an event driven model. From a main loop the program constantly reads an input event, performs some actions that finally produce some output events and feeds these to a special output event handling function, which translates the events to the actual output. Then the next input event is read, starting the cycle anew. The functional language Amanda uses this model as its primary design pattern. Amanda makes no effort to isolate the impure input and output events from the rest of the program.[8]

Another interesting method of describing I/O events in
A case study is the chosen method of research. The chosen case is that of a PS/2 keyboard interface, which we will try to describe in the different functional languages. This interface was chosen because it incorporates many of the common aspects of I/O, such as exact timing requirements, synchronous sequential communication and event listening. Though it is an often reoccurring aspect, asynchronous communication has explicitly been left out of this list. It will be left for future work.

The description of the interfaces behaviour is given below. It is a simplified description based on PS/2 and might represent a dedicated chip on the keyboard.[4].

The keyboards PS/2 interface, which will from now on be referred to as simply ‘the keyboard’, should read data from the output buffer and send it to the host. The keyboard:

1. waits until the signals BUFFER NOT EMPTY from the keyboard and KBD DATA and KBD CLOCK from the host computer all become high (idle).
2. starts generating a clock signal on KBD CLOCK
3. reads and sends one bit from the output buffer at a time over KDB DATA. Every bit should be readable on the falling edge of the clock signal.
4. stops generating the clock signal.

It should be able to handle input from the host the keyboard is connected to. In order to receive a command from the connected host the keyboard:

1. waits until the signal KBD CLOCK becomes low for more than one clock cycle (to prevent the keyboard from sending data at the same time that the host attempts to send a command)
2. starts generating a clock signal on KBD CLOCK
3. reads one bit at a time from KDB DATA. Every bit should be read on the falling edge of the clock signal.
4. writes each bit to the input buffer
5. stops generating the clock signal

To make the written code and diagrams below a bit less verbose the following abbreviations will be used: $C = \text{KBD CLOCK}, D = \text{KBD DATA}, BNE = \text{BUFFER NOT EMPTY}, BO[i-1] = \text{the } i^{th} \text{ bit in the output buffer}, BI[i-1] = \text{the } i^{th} \text{ bit in the input buffer}$.

The case study will be executed by trying to create a functional description of the hardware’s behaviour that can be simulated. This will be done for each of the I/O models and their respective languages (Haskell, Clean, Amanda and Erlang). In doing so the existing semantics of these languages will be adhered to as closely as possible. These languages are, of course, not meant to describe hardware, so a few basic features will have to be added to close this gap. To make a fair comparison possible this extra functionality should be the same for each language.

Three extra features were found to be required. The hardware designer should be able to clearly specify:

1. when a described behaviour occurs. Since we have narrowed our scope to synchronous hardware, only flanks of the clock are viable options. The choice is thus limited to at which rising or falling edge the behaviour occurs.
2. which input is read. All communication on and between computer chips takes place in the form of small (electrical) currents via conducting tracks representing either a low or a high value. (The single bit of information they provide will in this paper be referred to as signal) So all that needs to be determined is which track is being read.

3. which output is written. Just like the input only signals will need to be written.

Together these features form a common interface (shared between the description in the different languages) to the hardware environment. Their exact syntax and semantics will however differ per language. These will thus be specified separately for each functional language. Their actual implementation is part of the not (yet) existing compiler that translates the behaviour to hardware and is thus outside the scope of this paper. However, for simulation purposes some sort of implementation will have to be provided in our source language that does not violate the specified semantics. It should allow the programmer to control the hardware’s environment during testing and notify him of what output generated and when. The extra code needed for simulation has been clearly separated from the behavioural description in the provided source codes for our case study.

After specifying the syntax and semantics of the common interface, writing simulation code for it and describing the PS/2 interface the results for each I/O model will be compared to determine which of the models is most suitable. This will be done by answering the main research question for each of them.

It is important to note that this answer, whilst attempting to be as objective as possible, will for a large part be empirical and qualitative due to the nature of the properties compared. Aspects that might be judged in a more quantitative way such as the performance of synthesizing or simulating the hardware (with a prototype compiler) are considered as future work.

Some of the discussed languages in this paper support alternative ways to perform I/O, deviating or even bypassing from their main I/O model that we are investigating. Since we are not interested in the actual languages but just the I/O models, these deviations will not be discussed.

3. THE I/O MONAD

3.1 An introduction

A monad is a mathematical construct that is used by Haskell to separate its I/O performing code (which is not referentially transparent) from the rest of the program. It also is used as a mechanism to chain functional operations referentially transparent) from the rest of the program. It works as follows. Functions inside a monad are marked by it as their return type is wrapped in it. This return type can not be accessed directly but needs to be unwrapped first. This can only be done by other functions inside the monad which again return a value wrapped in the monad. This ensures that the referentially non-transparent return values can never leak to pollute the pure part of the program.

The operation that unwraps values and feeds them into the next action exists within the monad and is called the bind operator. It is responsible for the sequential execution of the actions despite lazy evaluation. There is also a bind operator that binds two actions without passing on a return value for actions that do not return anything. Their type definitions are:

\[
\begin{align*}
(a \rightarrow m b) & \rightarrow m b \\
(a \rightarrow m b) & \rightarrow m b
\end{align*}
\]

\(m b\) denotes a value of type \(b\) wrapped in monad \(m\). The parentheses around the operators mean they are used infix, such that a sequence of actions can be bound together intuitively by \((\text{action1 >>= action2 >>= action3} \ldots)\) etcetera.

There is one more interesting function named \(\text{return}\) that simply wraps a value inside the monad:

\[
\text{return} :: a \rightarrow m a
\]

With this and the bind operators we have all the tools we need to successfully use the I/O model in Haskell. However, Haskell has a shortcut built in to hide the somewhat confusing bind operators. A bounded string of actions can be replaced by the \(do\) keyword followed by the sequence of actions written as if in an imperative language. The construct hides all the bind operators in between and enables the programmer to name unwrapped return values for use later in the sequence. This construct can be systematically transformed back into the basic monad syntax and can thus safely be used as part of the monadic I/O model [10].

3.2 Implementation

The common interface can now be implemented in Haskell. First we have defined signals and their values to be their own type as this will allow for much more readable code.

\[
\begin{align*}
data \text{ Signal} & = \text{KBD, CLOCK} \\
data \text{ BUFFER} & = \text{NOT, EMPTY} \text{ BUFFER in} \text{ BUFFER out} \\
data \text{ Int} \text{ Value} & = \text{High, Low}
\end{align*}
\]

To read and write signals two functions can be created. These will have to operate within the \(I\O\) monad as they produce side-effects. To simulate them a call to these functions generates a prompt where the user can input the value of a read signal. Their type definition is as follows:

\[
\begin{align*}
\text{test} :: \text{Signal} \rightarrow \text{IO Value} \\
\text{set} :: \text{Signal} \rightarrow \text{Value} \rightarrow \text{IO ()}
\end{align*}
\]

We still need to specify at which clock flank certain behaviour occurs. There are multiple ways to deal with time. During the execution of the protocol it is a very common pattern that some action(s) need to be performed sequentially, then the hardware needs to wait for a next flank and, after that, continue performing actions. In our keyboard example this happens during sending and receiving. It makes a lot of sense to model the waiting for a next flank also as an action such that these sequential operation do not need to be interrupted. Based on this thought these two waiting-functions were defined:

\[
\begin{align*}
\text{nextUp} :: \text{Int} \rightarrow \text{IO ()} \\
\text{nextDown} :: \text{Int} \rightarrow \text{IO ()}
\end{align*}
\]

wait for the \(n\)th next rising edge of the clock

wait for the \(n\)th next falling edge of the clock
The function calls can be interpreted by the programmer as a blocking wait statement delaying the actions following it. Alternatively, an action that needs to be delayed could be passed as an argument, which might be more intuitive for a single action. Multiple following actions could then be passed by combining them in a newly defined single action. One piece of sequentially executed code with intermediate delays would then be fragmented into multiple user defined actions. This is not desirable as it requires more code and is confusing.

With this interface at hand we can specify how our hardware design in Haskell will be structured. The following action diagram does this. Converting this diagram into code is pretty straightforward. Below are the start and send functions that implement their eponymous parts in the diagram. The receive function is implemented largely analogous to the send function.

```haskell
start :: IO ()
start = do
  c <- test KBD_CLOCK
  d <- test KBD_DATA
  bne <- test BUFFER_NOT_EMPTY
  if c==Low then receive 8 >> start
  else if c==High && d==High then
    send 8 >> start
  else nextDown >> start

send :: Int -> IO ()
send 0 = return ()
send nrOfBits = do
  set KBD_CLOCK Low
  nextUp
  set KBD_CLOCK High
  v <- test (BUFFERout nrOfBits)
  set KBD_DATA v
  nextDown
  send (nrOfBits - 1)
```

The event listening aspects, namely reacting to KBD_CLOCK becoming low or BUFFER_NOT_EMPTY, KBD_DATA and KBD_CLOCK becoming high, are simply implemented by, in a loop, continuously checking on these signals’ status.

### 3.3 Evaluation

The sequential notation of code in a monadic binding allows us to easily express the sequential aspect of the behaviour. It also nicely incorporates the temporal behaviour thanks to the waiting functions. Furthermore testing signals in a loop is a quite intuitive way of modelling event listening since it is the way it really works in hardware.

### 4. UNIQUENESS TYPING

#### 4.1 An introduction

Uniqueness typing is another approach to make I/O referentially transparent. Function parameters can be defined as being unique. A way to denote this in Clean is by prepending an asterisk to their type. This means that “it is guaranteed that at run-time the corresponding actual object is local, i.e. there are no other references to it.” [11] Trying to pass it to a second function for example will result in an error. As explained in the introduction a function with unique parameters must be referentially transparent, for the property that a specific call to it can safely be replaced by its result always holds.

This construction can be used to perform I/O operations. Each one is performed on a unique world (*World) and results in a different, again unique world. This constant passing of a *World parameter has the added advantage that functions can easily be executed sequentially; by using the fact that each functions input depends on its predecessors output which is thus executed before it. This is done as follows:

```haskell
function world =
  let
    world1 = performIO world
    world2 = performIO world1
    world3 = performIO world2
  in world3
```

This is very useful but the not that well readable. In Clean this problem was solved by the introduction the let-before-construct. It differs from a standard let-construct in that the declarations happen before the function body and that special scope rules apply. The variables on the left-hand side of the assignments are outside of the scope of the right-hand side of that definition, but they do appear in the scope of the other definitions that follow. This allows us to continually override the world variable, creating the
functions again; this is pretty straightforward. Here are the start and send functions used to clarify it. Converting this diagram into Clean code is required for the next operation. Also the types Signal and Value have again been defined. This results in the following type definitions:

\[
\text{test} :: \ast \text{File}, \text{Signal} \rightarrow (\ast \text{File}, \text{Value})
\]
\[
\text{set} :: \ast \text{File}, \text{Signal}, \text{Value} \rightarrow \ast \text{File}
\]
\[
\text{nextUp} :: \ast \text{File} \rightarrow \ast \text{File}
\]
\[
\text{nextDown} :: \ast \text{File} \rightarrow \ast \text{File}
\]

The functions do not interact directly with the unique *World, but with *File which is part of it. This is merely for simulation purposes. The functions implementations read and write to the console which is of type *File for simulation purposes. The functions implementations have again been defined. This results in the following type definitions:

\[
\text{start} :: \ast \text{File} \rightarrow \ast \text{File}
\]
\[
\text{send} \ i \ o \ 0 = \ i o
\]
\[
\text{send} \ i o \ \text{nrOfBits} = \ i o \ \text{set} \ i o \ \text{KBD}_{\text{CLK}} \ \text{Low}
\]

As a result of the similarities with the Haskell implementation, the same code structure applies to the Clean implementation. Therefore the same action diagram can be used to clarify it. Converting this diagram into Clean code is pretty straightforward. Here are the start and send functions again:

\[
\text{start} :: \ast \text{File} \rightarrow \ast \text{File}
\]
\[
\text{send} \ i o \ 0 = \ i o
\]
\[
\text{send} \ i o \ \text{nrOfBits} = \ i o \ \text{set} \ i o \ \text{KBD}_{\text{CLK}} \ \text{Low}
\]

4.2 Implementation
The implementation of the common interface and the keyboard using uniqueness typing is largely similar to our monadic implementation. The same I/O read, write and wait functions can be defined in Clean. The only difference is that instead of operating as an action bound by a monad the functions return a new unique world that is required for the next operation. Also the types Signal and Value have again been defined. This results in the following type definitions:

\[
\text{test} :: \ast \text{File}, \text{Signal} \rightarrow (\ast \text{File}, \text{Value})
\]
\[
\text{set} :: \ast \text{File}, \text{Signal}, \text{Value} \rightarrow \ast \text{File}
\]
\[
\text{nextUp} :: \ast \text{File} \rightarrow \ast \text{File}
\]
\[
\text{nextDown} :: \ast \text{File} \rightarrow \ast \text{File}
\]

The implementation of the common interface and the keyboard using uniqueness typing is largely similar to our monadic implementation. The same I/O read, write and wait functions can be defined in Clean. Therefore the same action diagram can be used to clarify it. Converting this diagram into Clean code is pretty straightforward. Here are the start and send functions again:

\[
\text{start} :: \ast \text{File} \rightarrow \ast \text{File}
\]
\[
\text{send} \ i o \ 0 = \ i o
\]
\[
\text{send} \ i o \ \text{nrOfBits} = \ i o \ \text{set} \ i o \ \text{KBD}_{\text{CLK}} \ \text{Low}
\]

As a result of the similarities with the Haskell implementation, the same code structure applies to the Clean implementation. Therefore the same action diagram can be used to clarify it. Converting this diagram into Clean code is pretty straightforward. Here are the start and send functions again:

\[
\text{start} :: \ast \text{File} \rightarrow \ast \text{File}
\]
\[
\text{send} \ i o \ 0 = \ i o
\]
\[
\text{send} \ i o \ \text{nrOfBits} = \ i o \ \text{set} \ i o \ \text{KBD}_{\text{CLK}} \ \text{Low}
\]

4.3 Evaluation
The differences with Haskell are not that many. Just as the do notation in Haskell, the let-before construct allows the programmer to easily and quite naturally specify sequential operations and their timing (thanks once again to the wait functions). It must however be noted that the let-before construct is more restrictive in its use. Within a let-before statement it is not possible to use if and case commands. Although guards and pattern matching provide the same expressive power, some flexibility for the programmer and readability is lost. We could of course choose to not use let-before, but its disadvantages still outweigh explicit uniqueness typing which makes sequential operations hard to read.

5. EVENTDRIVEN FP
5.1 An introduction
The event driven model we deal with here works following a strict structure. First there is an event generator; a function that spawns a new event each time it is called. This program constantly, in a loop, listens for events (i.e. it calls the generator) and calls an event handler function that decides what to do with them. As a result of the execution of an event handler function an output event is returned and as the main loop loops a stream of output events is generated by the program. In order to translate these in more meaningful output the main loop is wrapped in an output handler function. This function constantly evaluates new output from the main loop, causing a new input event to be read and handled, and works thus as the motor behind the lazily evaluated application.

Since after each event the flow of execution returns to the main loop no state is being preserved by it. The state of the program thus needs to be stored explicitly and continuously passed around. This state variable is being used by the event handler to decide what to do.[8]

5.2 Implementation
For our implementation in Amanda modelling events is a good starting point, since they are the most important aspects of the event driven model. The entire range of possible input and output events exists out of two events for each signal plus the clock, which is just a signal with a special meaning; one for it becoming high and one for it becoming low. (In particular the current flank of the clock is determined by the last event of the clock signal.) It thus makes sense to define an event as a tuple of a signal and a value. For example \((s,v)\) denotes the event that signal \(s\) just took on the value \(v\). (signal and value are also defined types) The event generator \(\text{hardwarein}()\) can now easily be defined as an infinite list of these tuples. For simulation each event is prompted to the user.

\[
\text{event} \ = \ (\text{signal}, \text{value})
\]
\[
\text{hardwarein} :: \left[\text{event}\right]
\]
\[
\text{hardwarein} = \text{promptEvent()} : \text{hardwarein}
\]

An input is 'read' when an event of that signal is generated. When in reaction to this another signal needs to be read the program needs to rely on its memory of that signals state. This requires that whenever an event of an input signal is received the program needs to update its memory to reflect that signals new state. The only common interface element that is still missing is a way to describe which output is written. Faithful to the event driven model this takes the form of a function that is being fed a list of events:
Imagine however the following scenario: Initially decisive for the external behaviour. can only be used if the order in which events occur is not occurred since the last flank. However, this method does not that occurs at the current flank of the clock when e occurred since the last flank. However, this method does not take into account the order at which events occurred. It can only be used if the order in which events occur is not decisive for the external behaviour.

Imagine however the following scenario: Initially BUFFER NOT EMPTY is high. KBD CLOCK is high and KBD DATA is low, thus the keyboard is waiting to send data. One clock cycle later KBD CLOCK has become low and KBD DATA has become high. If first the KBD DATA event is handled the keyboard will react to this by starting to send. Alternatively, if KBD CLOCK is handled first, the keyboard will start to receive. In other words, the keyboards behaviour is not independent of the order of events.

This can be solved by no longer basing behaviour on individual events, but on the whole of events that occurred. We could do this by handling one mega event, describing all input signal state changes. Alternatively, the behaviour can be based on the state resulting from each previous event. At each flank of the clock first all the input events are processed (in arbitrary order) yielding a new state. After this the behaviour at this flank is determined based on only this new state. Neither of these methods appears to have a significant advantage over the other. This subject will be revisited to conclude that this is indeed the case. This solution chosen to implement is the later one, because it was easier to write simulation code for this approach. It has been implemented by a state updating handler function for each input signal and an event generator that, at the flank of the clock, will first generate events for all input signal state changes and than one for the clock that triggers the behavioural description.

A state diagram of this design is provided below. Signal values have been abbreviated to H and L. The blocks with rounded sides represent states. Their labels only contain the relevant parts of the state. State changes due to input events are denoted by semi-dotted lines. State changes that are a result of the the keyboards behaviour are denoted by normal lines annotated by the clock flank at which they occur. The cloud shaped node is just a collection of additional irrelevant states.

The design can be translated in Amanda code as follows. First the state is defined as a record. Besides the input signal states it contains a variable for the mode and the iteration. The mode determines which part of the behaviour is currently executed. It is roughly analogue to the current function that is being executed in Haskell or Clean. The iteration is a number that is used to iterate through the bits of the output and input buffer.

```
execEvent :: [event] -> [char]
```

The return type of this function is irrelevant since the list of events it must consume before returning is infinite. However, for the purpose of simulating the behaviour it produces a string.

When describing the desired behaviour with this event driven model we encounter a problem. An event driven program normally handles events when they occur. In synchronous hardware however events can only be handled at up or down going flanks of a clock. We thus define an event handler for an event e to describe the behaviour that occurs at the current flank of the clock when e occurred since the last flank. However, this method does not take into account the order at which events occurred. It can only be used if the order in which events occur is not decisive for the external behaviour.

The main loop is a function that recursively calls itself with the new program state after the current event is handled by do. It also passes the next input events (ies).

```
main :: state -> [event] -> [event]
main state (ie:ies) = oes ++ (main newState ies)
  where (newState, oes) = do state ie
```

In order to simulate the behaviour main is called with an initial state and the event generator, passing its output to execEvent.

```
main = execEvent (run initState hardwarein)
```

The root event handler contains the actual behavioural description. It also contains a part that is responsible for the signal state updates.

```
do :: state -> event -> (state,[event])
```

### state updating handler functions

```
do state (KbdClock,v) = (state&{c=v}..[])
do state (KbdData,v) = (state&{d=v}..[])
do state (BufferNotEmpty,v)= (state&{bne=v}..[])
do state (BufferOut n,v) = (state&{bo=nbo}..[])
```

where

```
{bo=bo}=state
nbo = take n bo ++ [v] ++ drop (n+1) bo
```

### behaviour at flanks

```
do (state=(m=Start,c=e,d=d,bne=bne)) (Clock,L) = start (c,d,bne)
```

```
start state c d bne
  = (state&{m=Sending,i=0}..[(KbdClock,L)])
    .if c=H /
    d=H /
    c=H
  = (state&{m=Receiving,i=0}..[])
    .if c=L
  = (state..[])
  .otherwise
```

---

![Figure 2. State diagram of event driven keyboard implementation](image-url)
do \((\text{state}=(\text{m}=\text{Sending},i=1,bo=\text{bo}))\) \((\text{Clock},\text{H})\)  
\[= (\text{state} , [(\text{KbdClock},\text{H}),(\text{KbdData},(\text{bo}!\text{i}))])\]

do \((\text{state}=(\text{m}=\text{Sending},i=7))\) \((\text{Clock},\text{L})\)  
\[= (\text{state}\&\text{m}=\text{Start},\text{]}\) \]

do \((\text{state}=(\text{m}=\text{Sending},i=1))\) \((\text{Clock},\text{L})\)  
\[= (\text{state}\&\text{i}=\text{i}+1;[(\text{KbdClock},\text{L})])\]

do \((\text{state}=\text{Receiving}))\) \((\text{Clock},\text{H})\)  
\[= (\text{state} , [(\text{KbdClock},\text{H}))])\]

do \((\text{state}=\text{Receiving},i=7,d=d))\) \((\text{Clock},\text{L})\)  
\[= (\text{state}\&\text{m}=\text{Start},[(\text{KbdClock},\text{L}),
(\text{BufferIn } i,d)])\]

do \((\text{state}=\text{Receiving},i=1,d=d))\) \((\text{Clock},\text{L})\)  
\[= (\text{state}\&\text{i}=\text{i}+1;[(\text{KbdClock},\text{L}),
(\text{BufferIn } i,d)])\]

When we inspect this code a considerable amount of it turns out to be boilerplate code, namely the root handler function, the state updating handler functions and the state variables themselves for each defined input signal. These 'pollute' the state and root handler function as they are not relevant for the behaviour and draw some of the readers' attention from the parts that are relevant. However, since it is boilerplate code, it would be very easy to construct a compiler that generates it for us internally. Scratching the unnecessary lines from the source code allows us to reduce the total number of lines from 25 to 14.

Returning to the alternative approach that uses a mega event we can see why this approach would not significantly differ. Having event handlers match on both a signals state from the programs memory and the possible state change in from a mega event would be redundant and complex. Since a mega event might as well represent the state of all signals, not only those that changed, this can be fixed by letting it replace the programs state. The only difference that then remains is that the part of the event handler that describes the behaviour at the clock flanks pattern matches the mega event instead of the program state. The structure of the program (minus boilerplate code) remains the same.

5.3 Evaluation
Amanda's behaviour description is clearly less structured then the previous implementations. It comprises entirely out of one function; the root function handler. We are forced to write our code this way. Each meaningful state has to be handled separately in an new function body, dividing our hardware's behaviour in very small free standing pieces. At the most, to make this code more readable we can group related handlers together, such as has been done in our source for the handlers that deal with sending and with receiving. However, it still lacks the clear sequential execution flow of the much more naturally composed send and receive functions in Haskell and Clean.

Excessive use is made of state variables. Other than the signal states, there are only two hardware state variables, however quite some consideration was required during design to choose these such that a minimal amount of state variables and handler functions was needed. The model forces us to think very explicitly about our hardware's state which in most cases is not that desirable for the programmer as it will quickly become confusing in larger hardware components.

After removing the boilerplate code the event driven model allows us to describe the keyboard with very few lines of code. This small code base saves the programmer effort.

6. MESSAGE PASSING

6.1 An introduction
Erlang has a special syntax for message passing. These are normally for communication between parallel Erlang processes. We will not be using parallel processes, but since different hardware components that are communicating with each other can be seen as such, message passing might be very suitable for communication between them. In order to pass a message from one component to another one of them should executing a send command and the other should be listening for this event. The receive operation blocks until a message that is listened for is received. [12]

6.2 Implementation
The goal is to communicate to other hardware by passing messages instead of reading and writing signals. The easiest way of doing this is directly sending to and receiving from other hardware components. However, a hardware design that uses direct communication is not independent of the hardware it communicates with; the components must know of each others existence, sacrificing the modularity of the system. On the border of the system one does not even have a possibility for direct communication, since the hardware can not know what’s outside. It can merely address a sending or receiving component as 'whoever is sending/receiving this message'. This is basically the same as reading or writing a signal.

So when sending and receiving messages we have no choice but to send and receive to and from signals. We now have two possible ways of implementing the indirect communication via signals. In one of those a signal acts as a server and the components interested in its value act as their clients. The other possibility reverses these roles. The server always acts as an event handling process and keeps track of the state of its inputs and/or outputs. It listens for incoming messages and acts in response to these.

First we consider the signal as a server. To read a signal that acts as a server we send a read request to it and then wait for a response. These two steps can off course be abstracted in a single read command. We can thus read (and logically also write) signals by simple calling a function that does this for us. There is no semantic difference to the way this is done in our Haskell and Clean implementation other than that we are not restricted to operation inside a monad or by uniqueness typing. This similarity extends to the nextUP and nextDown functions.

Below the start an send functions are displayed. Due to the similarities with the Haskell and Clean code no further clarification is provided. One more syntactic property must noted, namely the sequential execution of function bodies. This Erlang feature is not part of the message passing I/O model discussed here, but is an interesting feature to look at nevertheless as it benefits I/O operations. (See the next paragraph)

\begin{verbatim}
start () ->
C = test( 'KBD_CLOCK' ),
D = test( 'KBD_DATA' ),
BNE = test( 'BUFFER_NOT_EMPTY' ),
if 
C == low -> receive(8);
D == high , BNE == high -> send(8);
true -> nextDown() , start()
end.
\end{verbatim}
send (0) -> start();
send (NrOfBits) ->
  set ('KBD_CLOCK', low),
  nextUp(),
  set ('KBD_CLOCK', high),
  Val = test ("BUFFER_OUT["++NrOfBits++"]") ,
  set ('KBD_DATA', Val),
  nextDown(),
  send (NrOfBits – 1).

Secondly we consider the component as a server. Signals act as clients and can send messages when their state gets altered. The interested components can listen for these state change messages and keep track of a signals state. This, again, sounds familiar. It resembles the way we modelled our keyboard PS/2 implementation in Amanda. We can also in the same way hide the boilerplate code required to listen to the signals. Implementing this model does not seem very useful then. There is one difference though that makes it worthwhile, namely the sequential execution mechanism. It can be used in the (small) function bodies. Here it allows us, different from Amanda, to issue write (send) commands. It is thus no longer required to return the output events. This difference is very apparent in the part of the event handler that handles sending, shown below.

do (# state {m = sending, i = NrOfBits, bo = BO},
  {clock, high}) ->
  sendMsg ('KBD_CLOCK', high),
  sendMsg ('KBD_DATA', lists: nth (NrOfBits, BO)),
  state;

do (# state {m = sending, i = NrOfBits},
  {clock, low}) ->
  if NrOfBits < 7 -> sendMsg ('KBD_CLOCK', low),
  state#state {i = NrOfBits + 1};
  true -> state#state {m = start}
end;

6.3 Evaluation
The two Erlang hardware descriptions form an interesting test case. One has high resemblance to Haskell and Clean the other is similar to the Amanda implementation. For both implementations the application of the message passing I/O model does not provide a meaningful difference to with their similar ones. The distinction only relates to the sequential execution of code.

For the signal as a server model it means that sequential code is not restricted by a monad or by uniqueness typing. Is this in practice really an advantage? Over uniqueness typing it is, since even the let-construct had some disadvantages. These do however not exist for Haskell. At no point during the case study we were held back by the monad restrictions. Concluding based on this that there is no significant difference between Haskell and Erlang would be a mistake though. As functional programs grow more advanced they will at some point encounter difficulties with data in a monad. An example of this is the necessity to lift non monadic functions into the monad.[10]

Discussing these problems is outside of the scope of this paper.

The component as a server approach is only a slight improvement over Amanda’s implementation. The main problem remains, namely the fragmentation of sequential code over different event handlers. Within the small function bodies, however, Erlang’s sequential execution offers a significant improvement. The sequence of actions as a result of the function call is naturally specified as a sequence of function calls. This includes the output that is also generated by function calls instead of hidden in a complex return value.

7. Conclusion
Non of the compared models had fundamental problems, each of them allowed us to specify the full spectrum of op behaviours we sought to describe. We also successfully simulated all of the behavioural descriptions in the source language. Therefore there is no useful comparison to be made concerning the expressiveness or ability to simulate between the different I/O models. The difference in suitability of the different approaches in this paper is thus completely decided by their usability aspects for the programmer.

We concluded that the signal as a server model in Erlang had a slight advantage over Haskell’s monadic model. This is however due to Erlang’s sequential body; the message passing model offers not significant advantage for the programmer. Uniqueness typing on the other hand had the slight disadvantage of being less readable due to the restrictions of the let-before construct and the cluttering of code as a result of passing the unique world. Amanda has even more disadvantages compared to the previous three and Erlang’s component as a server model offers only a small improvement over this. The sequential execution of event handler bodies and not having to return output does not solve main problems of Amanda’s I/O; the fact that the sequential code is fragmented and that the explicit use of state can become confusing and time consuming. There are two small advantages offered by the event driven model. The event listening approach is a more natural way to handle input than looping and testing for signals. Also a bit less code needs to be written. These advantages do certainly not outweigh the disadvantages though. In conclusion, Erlang’s message passing and sequential function bodies are the better choice, closely followed by a the monadic I/O model and then uniqueness typing. The event driven model as described in this paper is not found to be suitable.

7.1 Future work
Important to note is that the compared possible I/O models are not exhaustive. There might be different approaches possible such as functional reactive programming or combinations and variations of the models discussed in this paper. These possibilities might pose an interesting topic for future research.

Since this research has been largely limited to the programmers usability, approaching the comparison made in this paper from a technical point of view is much needed. Since the differences between the Haskell, Clean and Erlang’s signal as a server approach are not that big from our current viewpoint. All of these three might still turn out to be the most suitable for I/O in a functional HDL. Specific questions that need to be tackled before this can be determined are:

- How suitable are the different models for proving correctness and how useful is this?
- Are there technical difficulties while implementing these models in a compiler?
- How much processing time and memory is required for the translation to hardware?
• Can equally efficient hardware be rendered from the different models?

Also, in particular sequential function bodies need to be researched. It might very well be that generating hardware from sequential function bodies will cause difficulties for translation. Might this be the case, then Haskell’s monadic model might be the best solution. A literature study might be enough since the author of this article has not looked into this yet.

Finally asynchronous hardware has been left out of the picture. For this, a more powerful temporal behaviour specification is needed then the ones described in this paper. To the knowledge of the author, however, no asynchronous functional HDL’s exist. So whether an asynchronous functional I/O model for hardware is actually useful should first be the topic of research.

8. REFERENCES