Mapping CSP Models to Hardware using CLaSH

Frist Kuipers, Rinse Wester, Jan Kuper & Jan Broenink
University of Twente, Enschede
Contents

- Introduction
- CLaSH
- CSP constructs
- Results
- Conclusions
Introduction

* Embedded system design more complicated
* Increase in number of requirements
* Model-Driven Design (MDD)
Introduction
CLaSH

- Functional Hardware Description Language (Haskell)
- Structural description of hardware
- Components based on Mealy-machine
CLaSH

mealy s i = (s’, o)
where
(o, s’) = f s i
mac s (a, b) = (s’, out)
where
  output = s’
  s’    = s + a * b

The result, the output and the new state, are shown at the end of the line in contrast to the mathematical definition of the Mealy machine.

This is because the result is defined last. Line shows that mac accepts two arguments, one for the current state s and a tuple containing the inputs (a, b).

The resulting tuple contains the new state s’ and the output of which the values are determined in the where-clause. In the where-clause, the actual MAC operation is performed and the result is assigned to the output (line and ). Finally, the initial state is assigned to the MAC circuit using the <^> operator resulting in the component macL.

For a reset of the circuit, the initial state of s is . Note that the reset circuitry is generated by the CλaSH compiler but not used during simulation.

The circuit corresponding to listing is shown in .

To verify the functionality, the MAC circuit can be simulated using the predefined CλaSH function simulateP. Note that simulation can be performed in an interactive CλaSH environment similar to GHCI. simulateP takes two arguments: a listed function representing the circuit (in this case macL) and a list of values acting as inputs.

Since CλaSH code is valid Haskell code, simulating the architecture is equivalent to executing a Haskell program. This is also advantageous for simulation speed since no separate simulator is needed. Listing shows the syntax to simulate the MAC circuit and the result of simulation. Note that take is added to stop the simulation after clock-cycles since simulateP runs indeﬁnitely.

res :: [Value]
res = take 3 (simulateP macL [(1, 2), (1,3), (2,2)])

[2,5,9]
CSP constructs in CLaSH

- Components with trigger tokens
- Three basic CSP constructs
  - Parallel
  - Sequential
  - Channels
Parallel construct

\[
\text{parallel'} (\text{te}, \text{ti}_1, \text{ti}_2) (\text{te}_i, \text{ti}_i1, \text{ti}_i2) =
((\text{te}_i, \text{ti}_1r, \text{ti}_2r), (\text{te}_o, \text{ti}_o1, \text{ti}_o2))
\]

where

-- Return token when both are received
\[
\text{te}_o = \text{ti}_1 && \text{ti}_2
\]

-- Only consume token one when both received
\[
\text{ti}_1r = \text{ti}_1 && \text{ti}_2
\]
\[
\text{ti}_2r = \text{ti}_1 && \text{ti}_2
\]

-- Return token to both structures in parallel
\[
\text{ti}_o1 = \text{te}
\]
\[
\text{ti}_o2 = \text{te}
\]
Sequential construct

sequential tei tii = (teo, tio)
   where
      teo = register False tii
      tio = register False tei
Channels

\[
\text{circuit } (\text{val}_\text{in}, \text{tkn}_\text{in}\_\text{writer}, \text{tkn}_\text{in}\_\text{reader}) = (\text{val}_\text{out}, \text{tkn}_\text{out}\_\text{writer}, \text{tkn}_\text{out}\_\text{reader})
\]

where

\[
(\text{value}, \text{writer\_ready}, \text{tkn}_\text{out\_}) = \text{writer } (\text{val}_\text{in}, \text{success}, \text{tkn}_\text{in}\_\text{writer})
\]

\[
(\text{success}, \text{value}, \text{writer\_ready}) = \text{channel } (\text{success}, \text{value}, \text{writer\_ready})
\]

\[
(\text{val}_\text{out}, \text{tkn}_\text{out}\_\text{reader}, \text{success}) = \text{reader } (\text{value}, \text{writer\_ready}, \text{tkn}_\text{in}\_\text{reader})
\]
In this work, LUNA is extended with C\texttt{aSH} code generation. This section describes the MDD work-flow using this approach. The current MDD work-flow is displayed in Figure 6. The design starts by defining a CSP model in the TERRA tool suite. Currently, the diagram needs to be translated by hand by drawing a data-flow diagram and writing the C\texttt{aSH} description by hand. However, the TERRA toolchain is extended with Model-to-Text (M2T) code generation. This code generation uses the CSP model defined in TERRA and directly generates a C\texttt{aSH} description. Subsequently, this C\texttt{aSH} description can be simulated by using an Haskell interpreter, for instance, GHC [13]. This simulation shows the output of the defined structures per clock tick. A test input and expected output can be defined to test the CSP model, using the functions: \texttt{testInput} and \texttt{expectedOutput}.

The C\texttt{aSH} description can be transformed to a HDL description (either VHDL or Verilog) using the C\texttt{aSH} compiler. The C\texttt{aSH} compiler uses the previously defined \texttt{testInput} and \texttt{expectedOutput} to generate a test-bench. This test bench inputs the values defined in \texttt{testInput} and asserts the \texttt{expectedOutput}. The VHDL description including the test-bench VHDL can be tested using Modelsim [2]. During the simulation the assertions are checked, when all succeed the model works as expected. Finally, the VHDL description can be synthesized using for instance Altera Quartus [2].

In current implementations, FPGAs are mostly used as I/O boards. The FPGA description is pre-defined and not part of the model. The first goal of this work is to be able to describe I/O in CSP Models, making simulations and editing of I/O functions more simple. This opens the possibility to move more functionality from embedded control software to the FPGA platform, see Figure 7. For instance the safety layer can be moved to the FPGA hardware, which makes the system more robust and the safety layer does not rely on context switching anymore. Finally, it is possible to move the loop controller to the FPGA platform, eliminating delays and jitter between I/O and loop control, see for instance [14]. This requires some challenges to be overcome. For instance, most controllers require floating point operations, which are not (yet) supported in the C\texttt{aSH} compiler.
Results

- Two examples implemented
- Single reader and writer
- Double reader and writer
Double reader/writer

\[
\begin{align*}
\prod_{i=t_1}^{\text{prod\_cons}}(t_i, v_i) &= (r_{\text{Out}}, \text{discard}) \\
\text{where} \\
(t_{i1}, w_{\text{Out}}) &= \text{writer } v_i \rightarrow t_{i1} \\
&\quad \text{writer connected to channel} \\
(c_{\text{Out}}, s) &= w_{\text{Out}} \rightarrow r_r \quad \text{channel} \\
(t_{i2}, r_{\text{Out}}, r_r) &= \text{reader } c_{\text{Out}} \rightarrow t_{i2} \\
&\quad \text{reader connected to channel} \\
(\text{discard}, t_{i1}, t_{i2}) &= \text{parallel } t_i t_{i1} t_{i2} \\
&\quad \text{reader and writer in parallel}
\end{align*}
\]

Algorithm 8. CaSH code of producer consumer example.

Using the CaSH compiler, the description of Listing 8 is compiled and simulated. During simulation, the output is calculated for every input value. The simulation results are converted into a timing diagram as shown in Figure 11.

First, the token is injected to trigger the execution of the parallel construct. Subsequently, the writer and reader are activated in the next clock-cycle. The writer and the reader are now ready for communication. The writer sets its value on the channel followed by the reader setting the success signal. One clock-cycle later the value is set on the output of the reader.

**Figure 11.** Timing diagram of the producer consumer example.

4.2. Multiple Producer Consumer

The second example is composed of two writers, two readers and two channels for communication. Figure 12 shows the structure of and relations among processes. Both the writers and readers are in sequential relationship. Therefore, data is first sent through one channel (the lower one in the figure) followed by the second. The structure of the circuit is basically a doubling of the components from the first example and omitted.

**Figure 12.** Multiple producer consumer example. Two writers sequential in parallel with two readers sequential communicating over separate channels. The orderings within the sequential constructs are indicated by the thick vertical arrows.
Double reader/writer

\[ prod\_cons (ti, vi) = (rOut, discard) \]

where
\[ (tii1, wOut) = writer vi s tio1 \]
\[ -- writer connected to channel \]
\[ (cOut, s) = channel wOut rr \]
\[ -- channel \]
\[ (tii2, rOut, rr) = reader cOut tio2 \]
\[ -- reader connected to channel \]
\[ (discard, tio1, tio2) = parallel ti tii1 tii2 \]
\[ -- reader and writer in parallel \]

Algorithm 8. CaSH code of producer consumer example.

Using the CaSH compiler, the description of Listing 8 is compiled and simulated. During simulation, the output is calculated for every input value. The simulation results are converted into a timing diagram as shown in Figure 11.

First, the token is injected to trigger the execution of the parallel construct. Subsequently, the writer and reader are activated in the next clock-cycle. The writer and the reader are now ready for communication. The writer sets its value on the channel followed by the reader setting the success signal. One clock-cycle later the value is set on the output of the reader.

Figure 11. Timing diagram of the producer consumer example.

4.2. Multiple Producer Consumer

The second example is composed of two writers, two readers and two channels for communication. Figure 12 shows the structure of and relations among processes. Both the writers and readers are in sequential relationship. Therefore, data is first sent through one channel (the lower one in the figure) followed by the second. The structure of the circuit is basically a doubling of the components from the first example and omitted.

Figure 12. Multiple producer consumer example. Two writers sequential in parallel with two readers sequential communicating over separate channels. The orderings within the sequential constructs are indicated by the thick vertical arrows.

Figure 13. Timing diagram of the multiple producer consumer example.

Figure 14. Multiple producer consumer example in a dead-locking configuration.

Figure 15. Timing diagram of the deadlocking multiple producer consumer example.
## Hardware results

<table>
<thead>
<tr>
<th>Example</th>
<th>Logic Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer consumer</td>
<td>23</td>
</tr>
<tr>
<td>Double producer consumer</td>
<td>37</td>
</tr>
</tbody>
</table>

### 4.4. Resource Usage

An indication of costs of a circuit on an FPGA is expressed in logic elements (LEs), the basic building blocks on an FPGA. Obviously, more CSP components result in more logic element usage. Additionally, the number of LE is also determined by the data types used for the messages that are sent using the channels. Since these messages are first kept in a writer and then consumed by a reader, additional memory is required in both the reader and the writer. Table 2 shows how many logic elements are required when using 8-bit signed integer as datatype for the aforementioned messages.

### 5. Conclusions

In this paper, a way to map CSP to hardware using C\textit{aSH} is proposed, and tested using simulation. This mapping enables the execution of a (currently restricted set of) CSP models on an FPGA. The implementation is made scalable and reusable for future applications. The CSP mapping is a first step toward a model-driven design process to generate VHDL code. C\textit{aSH} code can be generated from the CSP model in TERRA, which can be used to generate hardware description code. This code can then be synthesized and realized on a FPGA.

The generated code can be simulated at two levels. The first being an interpreted C\textit{aSH} simulation using a Haskell interpreter, for instance, GHC. This provides a per-clock-cycle simulation, testing for functionality. The second is a simulation of the generated VHDL description in Modelsim. Next to functionality, this simulation also gives insight on the timing. The modular token-flow approach makes extending this mapping possible. Therefore, this mapping is suitable for all kinds of MDD purposes.

### 6. Future Work

This paper only provides a mapping and generation for some CSP constructs to C\textit{aSH} in a basic setting. To allow the user to create real-life control software specifications, nesting of presented structures is needed. Nesting can be a part of the CSP structure as long as it conforms to the data-flow structure proposed in this paper, i.e., it consumes and produces tokens.

Robotic systems, the target of this mapping, consists often of some reusable components, e.g. motor drivers and sensor reads. This CSP mapping could be extended in the TERRA tool with support for these building blocks. Re-using a set of blocks makes the developed software more reliable. These building blocks should have some parameters, that can be set by the user for their specific purpose. Examples are mass and length of a specific robot arm.

### 6.1. Alternative Operator

This paper only provides a mapping for the parallel and the sequential construct. The alternative operator is also often used. A possible data-flow structure for the alternative construct...
Conclusion

- Mapping for CSP to FPGA developed
- Feasibility illustrated using examples
Future work

- Integration in TERRA tool
- Support for alt construct
Questions?