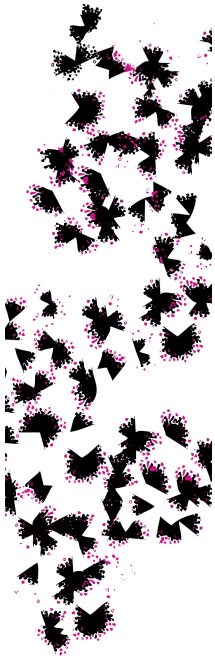


FPGA design support using CλaSH and LUNA

F.P. (Frits) Kuipers

MSc Report

Committee:
Dr.ir. J.F. Broenink
Dr.ir. J. Kuper
Dr. R. Wester
Z. Lu, MSc

May 2017

009RAM2017
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

Modern software development for embedded systems has an increasing amount of requirements, constantly increasing the complexity of the design process. An often used approach to simplify the design process of embedded systems is Model-driven design. *gCSP (graphical Communicating Sequential processes)* is such a model. It is a graphical way of displaying CSP models, which conforms to a precise syntax and has external tool support.

Traditionally embedded systems consist of an embedded processor running real-time software on a real-time operating system. Due to higher demands, more design effort is needed to meet these requirements. Since an embedded processor is often used for other purposes it is difficult to meet these real-time requirements. Offloading these real-time processes to an FPGA should resolve this problem. Due to the parallel nature of CSP, the FGPA platform is extremely suitable for CSP execution.

The goal of this project is twofold. The first part is to move (hard real-time) functionality from the embedded processor to the FPGA. The second part of this project is to integrate this functionality in the already existing design flow used in the TERRA tool chain.

This starts with mapping from CSP to hardware using the functional language CλaSH. As a proof of concept, several producer-consumer examples are implemented and simulated using this mapping. Next the design flow is changed to incorporate the code generation from gCSP models to CλaSH code. Since this code generation process is not yet complete some additional steps by the user are needed. The CSP structure is completely generated. The function calls for user-definable code has to be added manually.

The mapping of the CSP elements is tested using some producer-consumer examples. To test the complete design flow a demonstrator is shown. This test aims to demonstrate the complete design flow of FPGA hardware within the TERRA tool suite, as well as an overall test case of the CλaSH CSP mapping presented in this paper. The demonstrator shows it is possible to use the mapping and workflow in the design of robotic systems.

The move of functionality is achieved by implementing a mapping of CSP to CλaSH. Furthermore, the conversion from CSP diagrams is partially automated within the TERRA Tool suite. The FPGA design support in CλaSH is suitable to usable in robotic applications, but at this point it is necessary for the user to have intricate knowledge of computer engineering.

In the current implementation of the design flow it is only possible to choose between the FPGA or the embedded processor. In future work it is desired that on an architecture level it is possible to differ between CSP on the FPGA and executed on the embedded processor.

Contents

1 Introduction

Modern software development for embedded systems has a constantly increasing amount of requirements: more sensors and more actuators to better interact with the environment. This means simplifying the design process is of the essence. More and more people work on a single embedded software solution. So, making communication between individuals and teams simple and consistent is another important aspect of modern embedded software design. Achieving this requires a standardisation of the terminology, automatic consistency checking, and quality control. An often used approach to meet these requirements is MDD (Model-Driven Design). *gCSP (graphical Communicating Sequential processes)* is such a model. It is a graphical way of displaying CSP (Hoare, 1978) models, which conform to a precise syntax and has external tool support.

The *Twente Embedded Real-time Robotic Application* (TERRA) is a MDD tool suite simplifying the design process of embedded systems (Bezemer, 2013). TERRA uses models to design embedded systems on a higher abstraction level. The model structure is formalised using CSP. This way live- and deadlock checks can be performed easily.

All currently used embedded targets are based on the Von Neuman Architecture (Von Neumann, 1993). An alternative to this architecture is the FPGA (Brown et al., 2012). In current embedded control systems a embedded processor is often combined with an FPGA, where the embedded processor is used for the control loop and the FPGA for I/O purposes. Since a embedded processor is often used for other computing purposes, such as computer vision, it is difficult to accomplish hard real-time guarantees. Offloading these real-time processes to an FPGA should resolve this problem. Due to its parallel nature, an FPGA is an ideal platform for the execution of CSP models. CSP constructs can be executed in true parallel in stead of concurrently on a embedded processor. This makes execution of these models faster and above all more predictable, making the accomplishment of hard real-time guarantees easier.

The *CλaSH* (Baaij et al., 2010) compiler provides a way to generate hardware descriptions in an efficient way. CλaSH is a hardware-description language, it borrows its syntax and semantics from the functional programming language Haskell. Conventional HDLs, such as VHDL or Verilog, allow specifying detailed hardware properties, which can be cumbersome for larger projects. It allows for quick development of both combinational and synchronous circuits.

1.1 Project Goals and Approach

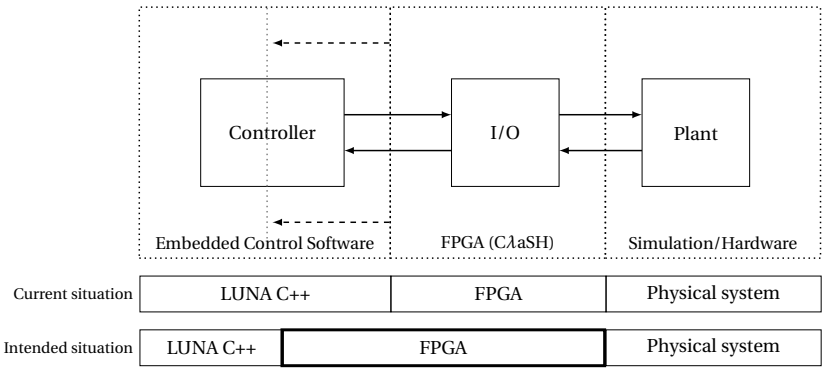


Figure 1.1: Use case of the CλaSH CSP mapping in embedded control. The dashed arrows denote the move of more functionality to the FPGA.

The goal of this project is twofold. The first part is to move (hard real-time) functionality from the embedded processor to the FPGA. The move of this functionality is displayed in Figure 1.1 and is denoted by two arrows. Below the controller, I/O and plant diagram, the *current* and the *intended* situation are displayed. The intended situation has more functionality on the FPGA. This moved functionality can be only the safety layer, but can be extended by also moving loop control or even more to the FPGA. The choice of this split should be made by the designer of the system in question. The second part of the project is to integrate this functionality in the TERRA tool suite. The TERRA tool is currently used to generate code for the LUNA execution platform. In a similar manner code can be generated for the FPGA platform. This requires a model-to-text transformation.

The TERRA tool uses gCSP diagrams to describe processes and their communication. These diagrams are then used to generate LUNA C++ code, which is able to execute these diagrams. Keeping the workflow identical for functionality on the FPGA is desirable. So, the process from a user point of view starts with designing the system in gCSP. Subsequently, this gCSP can be used to generate a hardware description. The first step in making this possible is to create a mapping of CSP to the FPGA. In this work this mapping is created using CλaSH.

The CSP description only describes the relations between processes and the communication between them. The next step is to give functionality to these processes. Starting with CλaSH implementations of standard I/O Blocks, like PWM generators or encoder readers.

1.2 Proposed Workflow

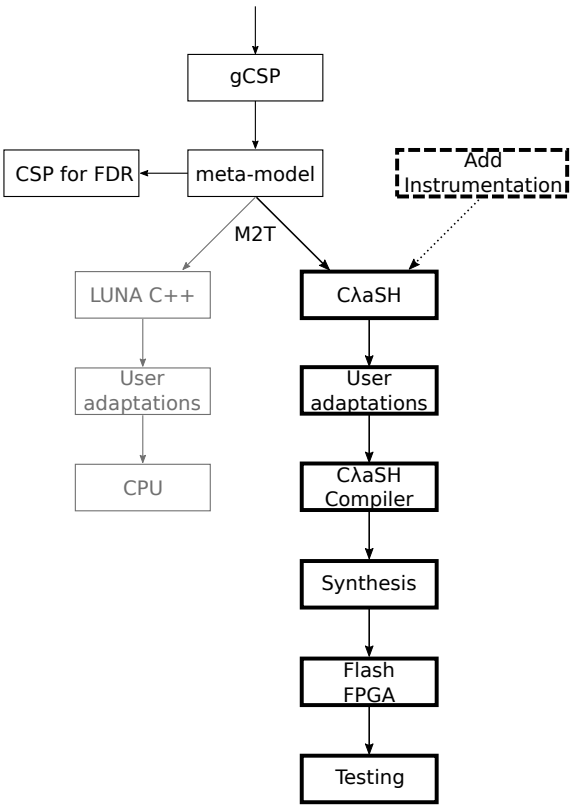


Figure 1.2: Intended workflow. The added work is emphasised with bold lines. The already existing LUNA C++ workflow is greyed out.

The intended workflow is shown in Figure 1.2. The workflow starts with a *gCSP* (graphical CSP) diagram in TERRA. This model is used to construct a CSP meta model in TERRA. From this

meta-model three code generation options are possible; CSPm, LUNA C++ and CλaSH. CSPm can then be formally checked by the tool FDR for live- and deadlocks.

The TERRA gCSP editor should provide some way to distinguish between gCSP intended for an embedded processor and gCSP intended for an FPGA.

The generated CλaSH code can be extended by the user. These extensions should also be written in CλaSH. When the user is satisfied with the added functionality the code can be interpreted and tested by the CλaSH compiler. Subsequently, the CλaSH compiler can generate VHDL which can be synthesised and flashed using for instance Quartus.

In this workflow it should also be possible to integrate instrumentation on the FPGA, making testing easier. This instrumentation should be able to output or log user selected signals from within the FPGA.

1.3 Project Layout and Organisation

This report describes the mapping of CSP to CλaSH and the generation thereof. The main body of this report is formed by the paper in Chapter 2 written for the CPA conference of 2016(Kuipers et al., 2016). The paper explains the Mapping of CSP to CλaSH, combined with some signal-level testing. The mapping of some more CSP constructs to CλaSH is described in Chapter 3. The following Chapter 4 is about the parts of the design flow implemented in this work. In Chapter 5 CλaSH code generation from TERRA CSP models is explained. Chapter 6 shows a 'proof of concept' test of the CλaSH CSP mapping, by using a test setup. Finally Chapter 7 gives some conclusions and recommendations additional to the ones in the paper.

Further details about the design are presented in Appendix A. In Appendix B some additional appendices are listed, including a list of requirements from the project proposal and some practical information about the produced software.

For readers who are not familiar with CSP, TERRA or LUNA it is advised to read Appendix A.1 first. The preferred reading order is as follows; Start with the paper in Chapter 2. Next, read Chapter 3 and 4. Read Appendix A.5 when interested in some additional design details. Read Chapter 5 when interested in code generation. Finally read Chapter 6 and Chapter 7.

For an overview of the requirements and how they were achieved, refer to Appendix B.1. For a a manual on how to use the CSP mapping and CλaSH code generation, refer to Appendices B.2 through B.4.

2 Paper: “Mapping CSP Models to Hardware Using CλaSH”

The next pages contain the paper "Mapping CSP Models to Hardware Using CλaSH". This paper is written for the CPA conference (Communicating Process Architectures) [1]

[1] http://www.wotug.org

Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

Mapping CSP Models to Hardware Using CλaSH

Frits P. KUIPERS [a], Rinse WESTER [b], Jan KUPER [b] and Jan F. BROENINK [a]
[a] *Robotics and Mechatronics,*
[b] *Computer Architecture of Embedded Systems,*
CTIT Institute, Faculty EEMCS, University of Twente, The Netherlands.

Abstract. Current robotic systems are becoming more and more complex. This is due to an increase in the number of subsystems that have to be controlled from a central processing unit as well as more stringent requirements on stability, reliability and timing. A possible solution is to offload computationally demanding parts to an FPGA connected to the main processor. The parallel nature of FPGAs makes achieving hard real-time guarantees more easy. Additionally, due its parallel and sequential constructs, CSP matches structurally with an FPGA. In this paper, a CSP to hardware mapping is proposed where key CSP structures are translated to hardware using the functional language CλaSH. The CSP structures can be designed using the TERRA tool chain while CλaSH code is generated for implementing hardware. The functionality of the CSP mapping is illustrated using some producer-consumer examples. In this paper, the design, implementation and tests are presented. Future work is to implement the ALT construct, generate token diagrams for user understanding.

Keywords. CSP process algebra, CλaSH, FPGA, TERRA, Embedded Systems

Introduction

Software for embedded systems has an increasing amount of requirements, constantly increasing the complexity of the design process. Additionally, quality control and automatic consistency checking are of essence in a design with an increasing amount of requirements. An often used approach to meet these requirements and simplify the design process is MDD (Model-Driven Design). CSP (Communicating Sequential processes) is such a model and is often used to verify timing of embedded control systems.

Embedded control system often consist of a central embedded processor combined with an FPGA. The central processor is often used for the control loop while the FPGA is mostly used for I/O purposes. Hard real-time guarantees are often difficult to accomplish on a embedded processor that also used for other computing purposes. Offloading these real-time processes to the FPGA should make this easier.

Due to their parallel nature, FPGAs are extremely suitable for CSP execution. CSP constructs can be executed in parallel in stead of concurrently on a embedded processor. This does not only make execution faster, but also makes the execution more predictable.

For FPGA code generation, we use CλaSH [1,2]. CλaSH is a hardware description language borrowing syntax and semantics from the functional programming language Haskell [3]. Additionally, the code can be simulated by the interpreter. One of the Goals of MDD is designing a system that is first-time right, simulation before actual testing on hard-

ware brings this one step closer. To make the process even less error prone it is desirable that the CλaSH code is also auto-generated using MDD with the TERRA tool chain [4].

In this paper, a mapping from CSP to hardware using the functional language CλaSH is presented. As a proof of concept, several producer/consumer examples are implemented and simulated using the aforementioned mapping.

Outline

The remainder of this paper is structured as follows. First, background information is given on CλaSH, TERRA and other related work. In Section 2, the design and design choices of the CSP to CλaSH mapping are illustrated. In Section 3, CλaSH code generation and model-driven design using the TERRA tool is explained. The CSP mapping and tested by means of some simple producer-consumer examples are covered in Section 4. Finally, conclusions are drawn and directions for future work are presented in Section 5 and 6 respectively.

1. Background

The background section first starts with a short introduction in CλaSH. This work makes extensive use of Finite State Machines structured as Mealy machines [5], which are explained using a small example. Furthermore, some background information is given about the TERRA tool and other related work.

1.1. CλaSH

CλaSH is a functional hardware description language (HDL), whose descriptions are translated to VHDL or Verilog by the CλaSH compiler. Conventional HDLs, such as VHDL or Verilog, allow specifying detailed hardware properties, which can be cumbersome for larger projects. CλaSH allows for quick development of both combinational and synchronous circuits [1,2].

Since CλaSH is a functional language, each of the CSP constructs can be defined in a function. The functionality of these structures can be checked using CλaSH simulation, even before synthesis is necessary.

Hardware components in this work have a *state* which is achieved using registers. In CλaSH a state can be achieved by instantiating register components directly or using Mealy machines, i.e. every output and new state is a function of the current state and the input. A register is a component like any other component in CλaSH and simply delays the input signal by one clock cycle. A Mealy machine is constructed by using a function in the form shown in Listing 1 where the state variable *s* contains state information. The input variable *i* is the input of the mealy machine. The output of the function is a tuple that contains both the new state *s´* and the output *o*. A function in this form can be used to construct a Mealy machine by using the function *mealy*. This *mealy* function also requires the initial value of the state. The CλaSH compiler recognizes the mealy structure and translates the use of the current and next state into a register.

```
-- Mealy machine function format
func :: State -> Input -> (State, output)
func s i = (s', o)
        where
                s' = .....
                o  = ....

-- Construction of a mealy machine using a function called func.
machine = mealy func initialState
```

Algorithm 1. Mealy machine function structure in CλaSH.

Listing 2 shows an example of a discrete integrator to demonstrate the usage of the Mealy-machine function format. The new state of the Mealy machine is the current state incremented by the input while the output is the new state [6]. The last line shows how the final architecture is created using the Mealy-machine function that assigns the initial state 0 to the circuit.

```
integrator s inp = (s', out)
        where
            s'  = s + inp
            out = s'

-- Construction of a mealy machine for integrator
machine = mealy integrator 0
```

Algorithm 2. Integrator example in CλaSH.

Every CλaSH description is a valid Haskell description and can be simulated by a Haskell compiler or simulator such as GHC. This does not work the other way around, i.e. not every Haskell description is a CλaSH program. For instance, CλaSH does not support recursive functions and recursive datatypes (yet).

1.2. TERRA

The *Twente Embedded Real-time Robotic Application* (TERRA) tool chain is a Model-Driven Design (MDD) tool chain for the design process of embedded systems [4]. TERRA supports designing using CSP models and integrates models from other tools, such as 20-sim[1] models and co-simulation. Properties of TERRA models can be formally verified by exporting to machine-readable CSP and using a tool like FDR3 [7]. TERRA allows easy use of the CSP-execution engine of LUNA [8], allowing the CSP structure to be drawn instead of written by hand.

CSP allows an easy decomposition of the structure of a program into a set of sequential and parallel tasks. Support for more advanced structures (e.g. timed channels, (guarded) alternatives) is present, allowing also complex structures to be decomposed. Adding blocks with custom C++ code allows the user to add the functionality of the program to the structure defined with the CSP constructs. Furthermore, embedding converted 20-sim models is supported, allowing for easy implementation of digital controllers.

1.3. Related Work

Groothuis *et al* [9] use gCSP extended with automated Handel-C code generation to FPGAs. Loop controllers are converted from floating point to integer-based calculations, be-

[1] http://www.20sim.com/

cause Handel-C does not support floating point operations. Development using this approach has stopped since Handel-C is not supported anymore.

Coyle *et al* [10] use UML diagrams to describe hardware, the models are transformed to hardware using MODCO, a transformation tool which takes UML state diagrams as input and generates a HDL description for an FPGA. This research focuses on the translation of state diagrams and does not exploit the parallel nature of the FPGA.

Basten *et al* [11] present the GASPARD design framework for massively parallel embedded systems. This framework allows design using a model-driven design approach using MARTE [12]. These models are then refined to lower abstraction levels. Subsequently, code can be generated for formal verification, simulation and hardware synthesis.

Brown [13] has a different approach to translating CSP into Haskell. Monads are used to specify sequence and monadic combinators allow for composition of monadic actions. This is however only a translation to Haskell, not to hardware. CλaSH has limited support for monads therefore this approach cannot be used.

2. CSP Constructs in CλaSH

2.1. CSP Compositions

The Haskell CSP structures have to be designed in a way that conforms to the way FPGA hardware operates. Haskell functions realized on a FPGA can be executed immediately, and in parallel. CSP defines parallel structures, sequential structures, alternative structures with deterministic choice, and without. The order of execution of these structures has to be accomplished within the FPGA. Structures have to be stopped and started accordingly.

In this work, tokens are used to enforce the execution order of CSP structures. This similar to the use of tokens data-flow graphs except that there is no data stored inside of them. A token is used to activate a CSP structure. A CSP process is designed as a structure that can receive and return a token. The token is returned by the structure when it is finished. So, when a reader "contains" a token, it is ready to receive a value. Tokens work in the same way for writers, and structures of other CSP constructs. A CSP process can be a reader or writer, or a composition of readers and writers. A composition itself is also a CSP process, and can have a relation with another structure, e.g two parallel structures can be sequential.

Table 1 lists all the functions explained in the subsections below. Each of the structures are first introduced shortly followed by a data-flow diagram displaying the token-flow. Finally, the CλaSH code of each function is listed and explained.

Table 1. List of CSP constructs and their CλaSH functions.

CSPm	Haskell function
p ||| q	parallel
p ; q	sequential
p [] q	alternative (future work)
c ! variable	writer
c ? variable	reader
channel c	channel

2.2. The Parallel and the Sequential Operator

The interleaving-parallel operator, see Figure 1, is one that maps very well to the FPGA platform. The operator stands for independent concurrent activity. The process behaves as process P and Q simultaneously. On a single-core embedded processor P and Q would be

arbitrarily interleaved in time while on an FPGA, both processes can be executed completely parallel.

$$P|||Q$$

Figure 1. Interleaving operator. The process behaves as process P and Q simultaneously.

CSP also has a sequential operator for sequencing two processes denoted by a semicolon, shown in Figure 2. The process initially behaves as P, after P has finished it behaves as Q.

$$P;Q$$

Figure 2. Sequential operator. This process behaves first as process P. When P is finished it behaves as Q.

The sequential and parallel structure data flow diagrams are shown in Figure 3. The sequential operation is achieved by pipelining processes. When a sequential block receives a token, the token is forwarded to process P thereby activating it. When process P is finished it forwards the token to the next process in sequence, process Q. Finally, the last process returns its token to the sequential structure. The sequential structure then returns its token to its parent.

The parallel operator produces as much tokens as the amount of processes in parallel. This way all processes are activated simultaneously. After all processes in parallel have finished the parallel structure returns its token. This means the parallel structure has to collect all the tokens and return its own token only when all internal tokens are received.

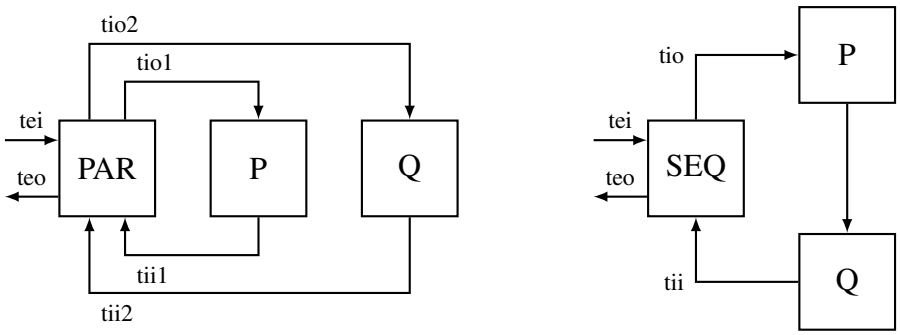


Figure 3. Data flow graphs of the parallel and sequential composition. Lines carry tokens. Processes are denoted as boxes.

The Haskell description of the parallel structure is shown in Listing 3. It conforms to the Mealy function format and has three state variables, `(te,ti1,ti2)`. These state variables store respectively the input token, the returned token of process P (tii1), and the returned token of process Q (tii2). The structure updates the states and the outputs. Tokens are sent immediately to P and Q when the parallel structure receives a token. These structures return their token when finished. The parallel structure returns its token to the outside when both tokens have been received. Analogously, both tokens are removed from the state when the token is returned from the structure.

```
parallel' (te, ti1, ti2) (tei, tii1, tii2) = ((tei, ti1r, ti2r), (teo, tio1, tio2))
  where
    -- Return token when both are received
    teo = ti1 && ti2

    -- Only consume token one if both are received
    ti1r = ti1 && ti2

    -- Only consume token two if both are received
    ti2r = ti1 && ti2

    -- Return token to both structures in parallel
    tio1 = te
    tio2 = te

parallel tei tii1 tii2 = mealy parallel' (False, False, False) (tei, tii1, tii2)
```

Algorithm 3. Parallel construct in CλaSH. The behaviour is described in *parallel'* in the format according to Listing 1. The function is transformed to a mealy machine in *parallel*.

As shown in Listing 3, the parallel construct has three inputs: *tei*, *tii1* and *tii2*. *tei* is a token input that triggers the execution of the parallel construct. *tii1* and *tii2* are the signals for the tokens from the parallel processes. Similarly, the outputs *teo*, *tio1 tio2* are used indicate to the parent process whether the processing is finished. The other variables on the first line indicate the current and next state. The two statements in the middle of the code compute the value for registers *ti1r* and *ti2r* which indicate to the two parallel process weather the trigger tokens have been received. The vertical bar symbols are used to check for the completion condition of the child processes, i.e., both processes have to be finished before the parallel construct is finished.

The description of the sequential operator is shown in Figure 3. The sequential operator just passes its input token to the first construct in sequence. When it receives the token from the last construct in the sequence, it passes the token to its parent. The *register* in the construct is used to store the token of both processes.

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

Algorithm 4. Sequential function. The tokens are returned with one clock cycle delay from the inputs (tei,tii) to the outputs (teo,tio).

2.3. Multiple CSP Structures in Parallel

The sequential composition can easily be extended to three or more processes by adding more processes in the token passing chain. The extension of the parallel composition is a little bit more complicated, since the parallel function only has ports for two processes. It would possible to construct a parallel component for every number of structures necessary, but this requires a large amount of functions which are hard to maintain. So, it is chosen to compose four parallel structures by parallelising two parallel structures essentially parallelising four CSP structures. The resulting composition for four and three CSP structures is shown in Figure 3. The downside of this approach is that it takes one clock tick longer to activate the CSP components in this structure.

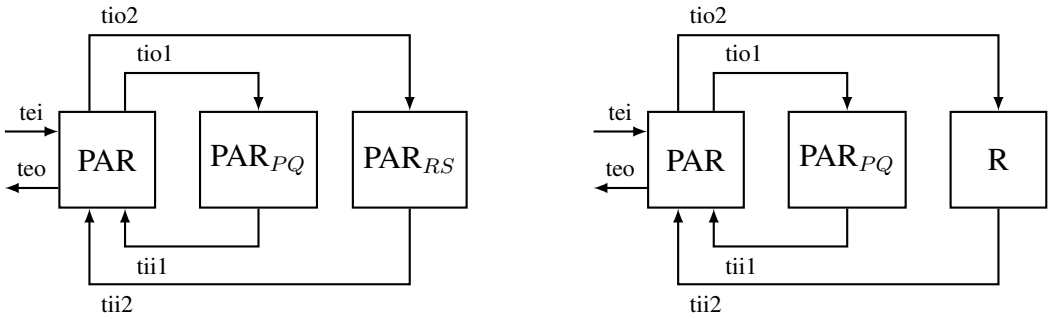


Figure 4. Three or more parallel CSP structures can be parallelised by using compositions of parallel structures and processes.

2.4. Channel Communication

Communication between processes works through channels. A process can output its data using a writer, while another process can input data using a reader. These operations are denoted in CSP by respectively an exclamation mark and question mark. Transfer of data can not proceed until the other end is ready to offer or accept data. Handshake signals are introduced to facilitate the communication. The order of execution in CSP is therefore not only determined by CSP relational structures, but also by (rendezvous) channels.

Although channels have one-way data communication, their synchronisation is bi-directional. A channel has bi-directional communication to ensure proper functionality. For example, a writer block may only finish (return its token) when its value is received. A channel "block" in this description is always active and does not need a token.

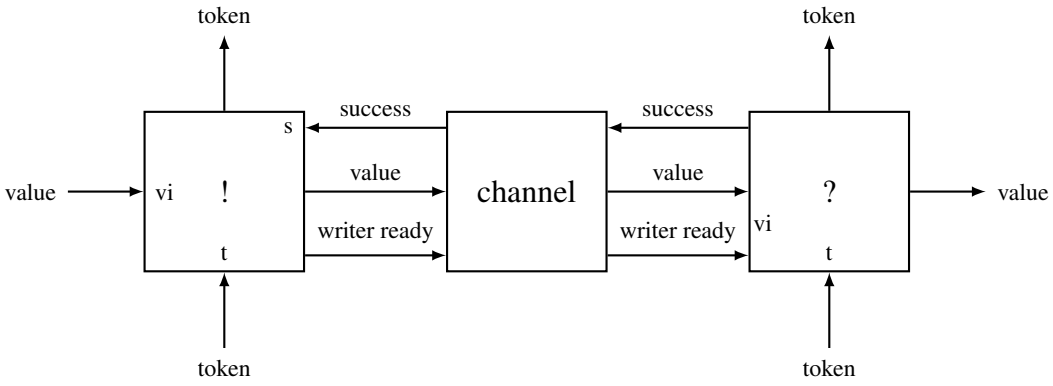


Figure 5. Channel communication and synchronisation.

In Figure 5, the communication and synchronisation of a channel in a producer-consumer example is shown using three signals. One of them, *value*, contains the value written by the writer, denoting the data communication. The *writer ready* signal indicates the writer is active and the reader is receiving valid data. This signal is combined with the value signal using the *Maybe* type. A Maybe type can be in state *Nothing* or *Just* with a corresponding value. As soon as the reader has accepted the data it returns a *success* signal. This way the writer knows the communication has finished and it can return its token.

The reader and writer functions are displayed in Listing 5 and 6. Both are implemented using pattern matching and conform to the Mealy function structure (see Listing 1).

The writer has three state variables: $(haveToken, success, value)$. *haveToken* stores the token of the writer and will be returned when channel communication has finished. *success* stores the success value returned from the reader. *value* stores the value the writer intends to send. When the token is available and there is no success, the writer component reads a new

value from its input, and outputs the current value from its memory. When the writer component is active, it is assumed the input is stable. When the reader has successfully received the value from the writer component, the success signal is set. When the success signal is received by the writer component the token is returned to its parent. In all other cases the writer component outputs Nothing.

```
writer' (haveToken,success,value) (t,s,vi) = case (haveToken,success,value) of
  -- When Token is available and no success (yet) get new value from
  -- input and output current value.
  (True, False, v) -> ((True, s, vi),      (False, v))
  -- When Token is available and success return the token and output Nothing.
  (True, True, v)  -> ((False, False, vi), (True, Nothing))
  -- In all other cases output nothing.
  (_, _, v)        -> ((t, s, vi),         (False, Nothing))
```

Algorithm 5. Haskell code for the Reader.

The reader has two state variables: $(haveToken, value)$. *haveToken* is the token of the reader and will be returned when channel communication has finished. *value* is the value of the reader, received from the writer. When no token is available, the reader component keeps it current states. When the token is available and Nothing is on the reader components channel input, the writer component is apparently not active and the reader keeps its current states. When the token is available and there is a value on the channel, communication takes place. The reader saves the new value to its *value* state and sets the *success* flag.

```
reader' (haveToken,value) (t,vi) = case (haveToken,value,vi) of
  -- When no token is available keep the current value. Success is false.
  (False,v,vi)     -> ((t,v),       (v,False))
  -- Token is available, nothing on input -> Keep current value. Success is false.
  (True,v,Nothing) -> ((True,v),    (v,False))
  -- Token is available, new value on input -> take new value. Success is true.
  (True,v,vi)      -> ((False,vi),  (v,True))
```

Algorithm 6. Haskell code for Reader.

The channel used in this example is the standard *rendezvous channel*. The implementation of this channel is straightforward. It simply connects the signals from the writer and the reader. Essentially, the function just describes some wires, as the synchronisation is implemented in the reader and writer. The channel function is shown in Listing 7.

The channel function will be removed by synthesising the generated VHDL code. It can be removed by just connecting the writer and the reader directly. It is chosen to keep the channel function separate to support buffered channels later on in the development process. This way the channel function can be easily swapped out for a buffered version. This also simplifies code generation earlier in the design process.

```
-- | Unbuffered Channel (Rendezvous channel)
channel valueIn valueReady = (valueIn, valueReady)
```

Algorithm 7. Haskell code of the channel.

3. MDD Work-flow and Code Generation

The TERRA tool chain is a MDD tool suite simplifying the design process of embedded systems [4]. Based on models in TERRA LUNA C++ descriptions can generated. In this work,

LUNA is extended with CλaSH 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 CλaSH 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 CλaSH description. Subsequently, this CλaSH description can be simulated by using the techniques presented in Section 1. This simulation shows the output of the defined structures per clock cycle. A *test input* and *expected output* can be defined to test the CSP model, using the functions: *testInput* and *expectedOutput*.

The CλaSH description can be transformed to a HDL description (either VHDL or Verilog) using the CλaSH compiler. The CλaSH compiler uses the previously defined *testInput* and *expectedOutput* to generate a test-bench. This test bench inputs the values defined in *testInput* and asserts the *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].

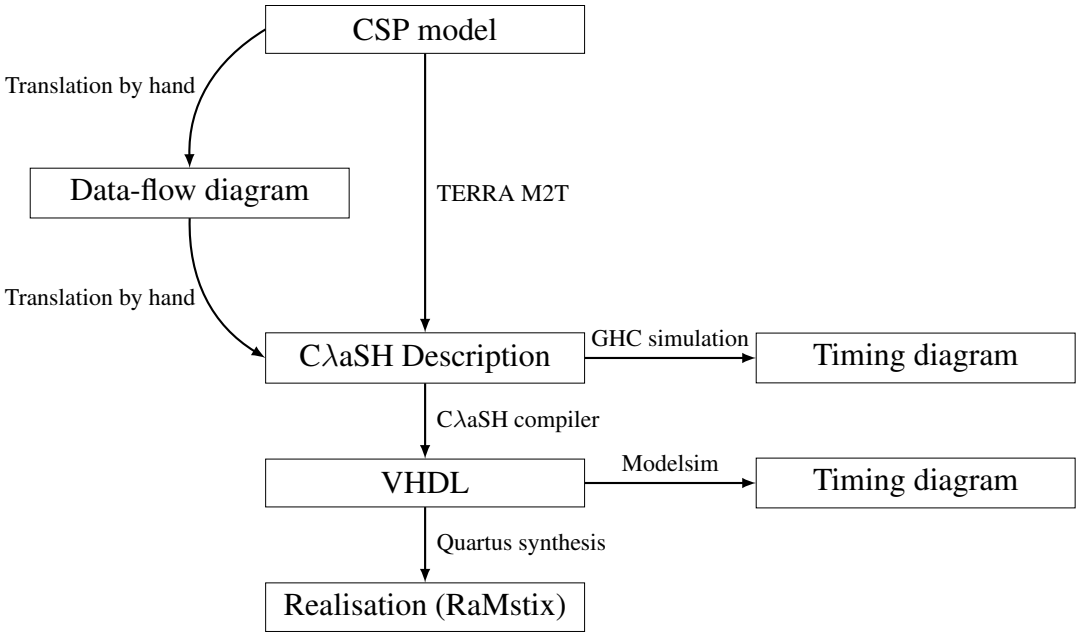


Figure 6. The current MDD work-flow from CSP models to hardware realization.

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 CλaSH compiler.

[2] `https://www.altera.com/products/design-software/`

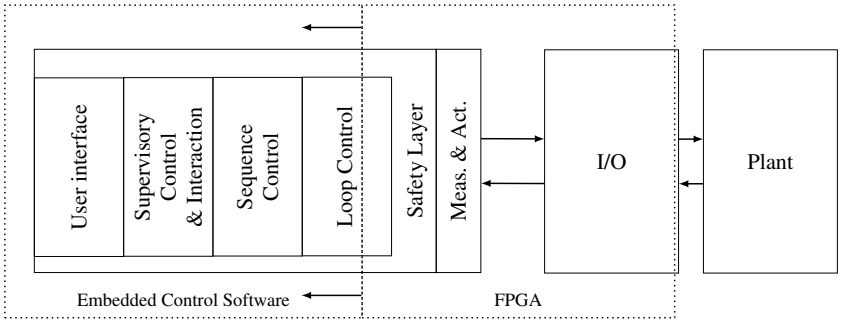


Figure 7. Use case of the CλaSH CSP mapping in embedded control.

4. Examples

As a proof of concept, two producer-consumer examples are implemented using the mapping methodology presented in Section 2. The first example shows a parallel composition of a single writer and a single reader. The second example contains two writers and two readers showing a more complicated ordering of execution. Additionally, an alteration of the second example is shown containing a deadlock.

4.1. Producer Consumer

The first example is shown in Figure 8. A writer and a reader are connected by a channel using a parallel construct. Since both the reader and writer are active in a parallel constructs, channel communication can take place. Note that the parallel structure is not recursive, because it is activated manually.

In this example, trigger tokens are injected externally from a test bench. This trigger token is is sent to the parallel construct which activates both the reader and writer. Execution of the parallel construct finishes when both the reader and writer are finished, sending a *finished* trigger back to the parallel construct.

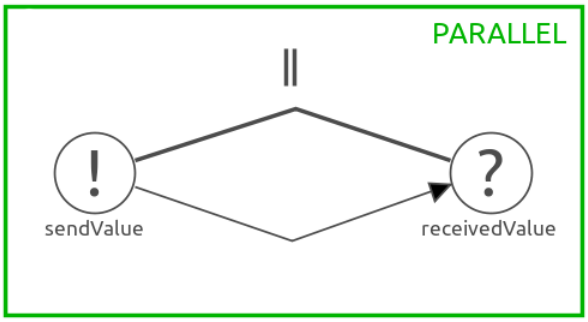


Figure 8. Producer consumer example. A writer and a reader in parallel relation connected by a channel.

The execution order of the producer consumer is shown in Figure 9. First, the parallel construct is activated, by a trigger token. The parallel construct then activates both the reader and writer in parallel by sending them a trigger token. The writer outputs the ready signal and its value. When the reader receives the ready signal, it reads the value and sets the success signal. Afterwards, both the writer and the reader return their trigger token to indicate to the parallel construct that both processes are finished.

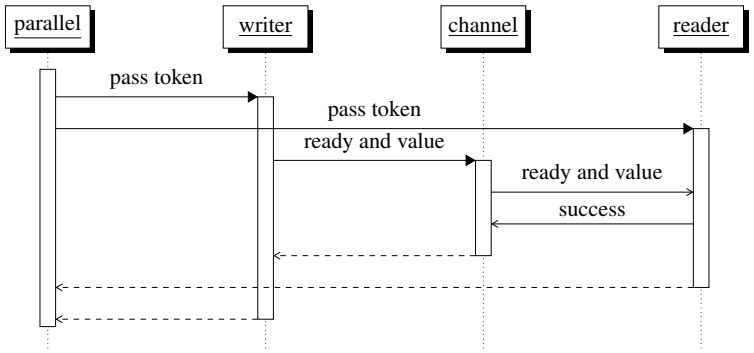


Figure 9. Sequence diagram of a producer-consumer example.

Figure 10 shows how the CSP constructs are mapped to an FPGA using CλaSH components. The ordering and dependencies in timing among constructs are made explicit with wires. Additionally, data communication using a channel is also made explicit using an instantiation of a channel component. Note that every component in the CλaSH definition is mapped to a different location on the FPGA. The implementation is therefore completely parallel.

As shown in Figure 10, the execution of the parallel construct is triggered by a token in input *ti*. Both the writer and reader are triggered by a token on *tio1* and *tio2* respectively. Since channel communication requires acknowledgements to ensure that transmissions are finished completely, status signals *s* and *rr* are connected to the channel. Using *rr*, the reader indicates to the channel that the value is read while *s* indicates to the the writer that the value is successfully sent through the channel and that a new value can be sent. When both the writer and the reader finished their operation, both send a token back to the parallel construct to indicate their completion using the wired *tii1* and *tii2* respectively. Finally, when both tokens are received by the parallel construct, a token is put on the *discard* output thereby indicating the completion of the whole computation.

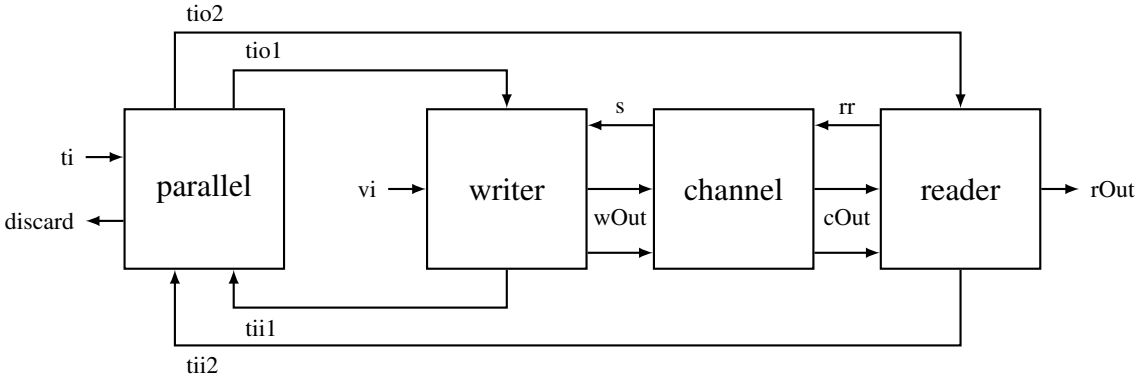


Figure 10. Data-flow diagram of the producer-consumer example.

The CλaSH code of the producer consumer example of Figure 10 is shown in Listing 8. On the first line, *prod_cons* is the function representing the whole circuit. As argument, the function *prod_cons* accepts a singe token containing a trigger input *ti* and value for the writer *vi*. On the output, a tuple is produced containing the value produced by the reader *rOut* and the *discard* signal. All instantiations of the components are described in the where-clause. For each component, the all incoming signals are connected on the right hand side while the output signals can be found left of the equal-sign. Note that the ordering in the where-clause has no impact on the execution, the code is a completely structural description of the circuit. The code is therefore structurally equivalent to the circuit shown in Figure 10.

```
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. CλaSH code of producer consumer example.

Using the CλaSH 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.

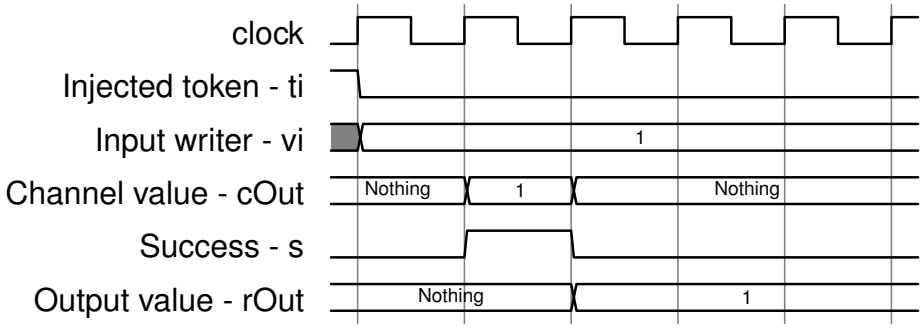


Figure 11. Timing diagram of the producer consumer example.

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.

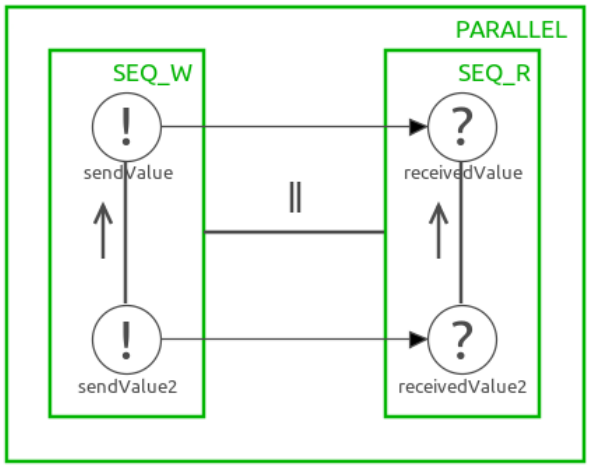


Figure 12. Multiple producer consumer example. Two writers sequential in parallel with two readers sequential communicating over separate channels. The orderings within the sequential constructs are indicated by the thick vertical arrows.

Listing 9 shows the CλaSH code for the doubling producer consumer example. Similar to the first example, the first argument for *double_prod_cons* is a tuple with the input data for the channels (*vi0* and *vi1*) and a trigger input *ti* to start the process. Also the output has a similar structure with two outputs from the readers (*rOut0* and *rOut1*) and the *discard* output to indicate completion of the whole process. In the where-clause, all readers, writers and channels are instantiated and connected. To control the execution order, one parallel and two sequential constructs are instantiated as well.

```
double_prod_cons (ti, vi0, vi1) = bundle (rOut0, rOut1, discard)
  where
    -- Two writers sequential
    (wT0, wOut0)        = writer vi0 s0 tio0
    (wT1, wOut1)        = writer vi1 s1 wT0
    (teo0, tio0)        = sequential pT1 wT1

    -- Channels
    (cOut0, s0)         = channel wOut0 rr0
    (cOut1, s1)         = channel wOut1 rr1

    -- Two readers sequential
    (rT0, rOut0, rr0)   = reader cOut0 tio1
    (rT1, rOut1, rr1)   = reader cOut1 rT0
    (teo1, tio1)        = sequential pT2 rT1

    -- The two structures above in parallel
    (eT, pT1,pT2)       = parallel ti teo0 teo1
```

Algorithm 9. Code for the double producer consumer example.

Again, the CλaSH code is compiled and simulated after which the timing diagram of Figure 13 is extracted. Similar to the first example, the whole process is started by injecting the trigger token at the parallel construct. Consequently, both sequential constructs are triggered. The sequential structures pass their tokens to the first reader and writer triggering the communication over the first channel. The active writer and reader pass their token to the second reader and writer such that the communication over the second channel is triggered. Finally, when the second reader and writer are finished the whole process is completed and the channels are back into the *Nothing* state.

4.3. Multiple Producer Consumer with Dead-Lock

By reversing the ordering of the sequential construct containing the readers, a deadlock can be created. This is due to the fact that the first writer to be activated cannot complete because the second reader has to wait on the completion of the first reader. Similarly, the first reader cannot complete its operation because it will never receive a message from the channel. Figure 14 shows the CSP schematic of the double reader-writes with deadlock.

After the CλaSH code has been compiled, simulated and a timing diagram has been derived, Figure 15 emerges. As expected, the first channel communication will not finish due the fact that the reader will never become active. The second channel is never activated. In the timing diagram, this is shown by the channel and reader outputs: the output remains a stable *Nothing*.

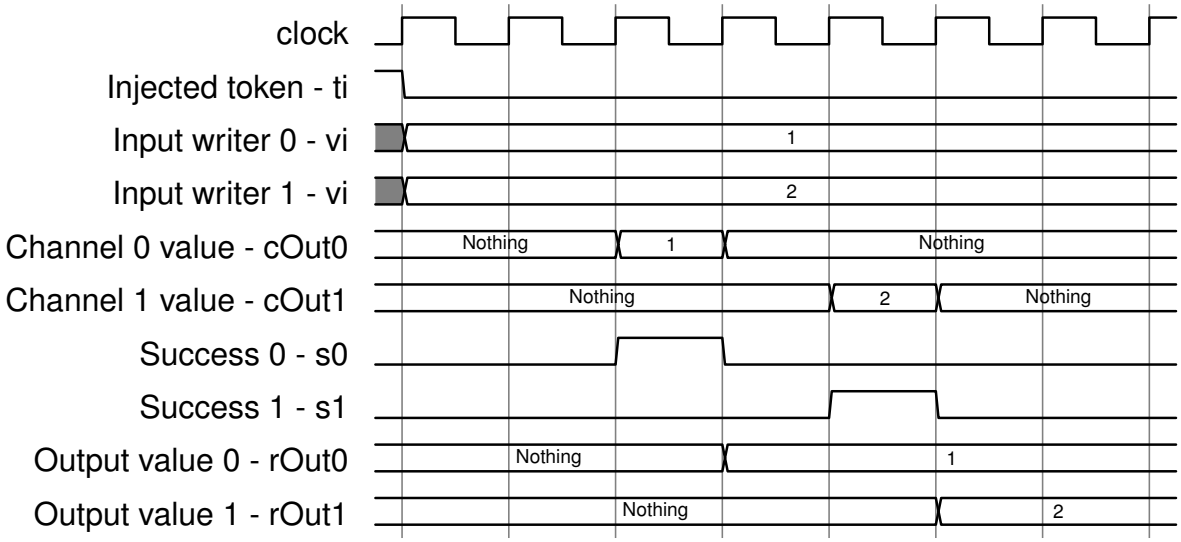


Figure 13. Timing diagram of the multiple producer consumer example.

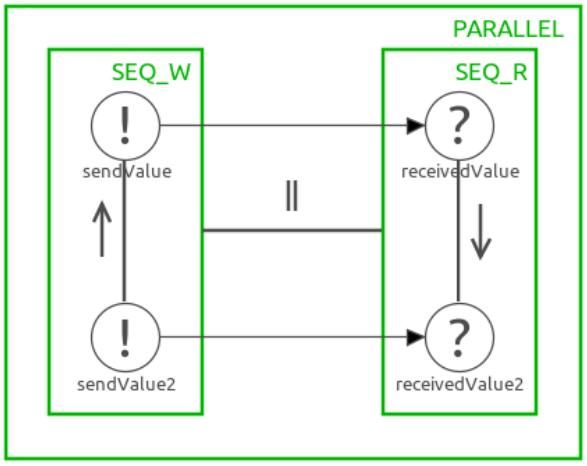


Figure 14. Multiple producer consumer example in a dead-locking configuration.

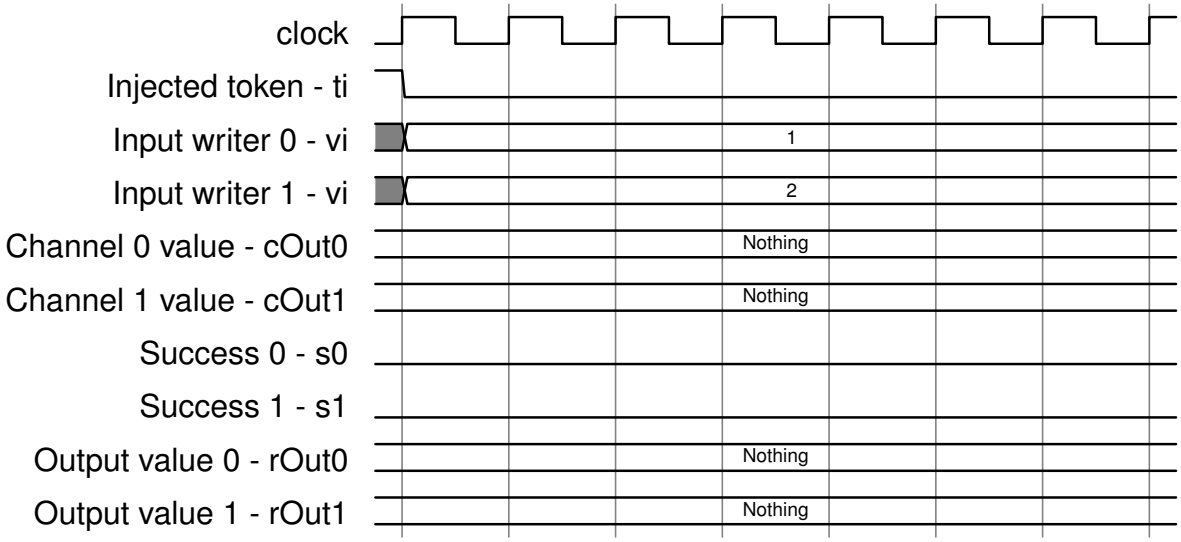


Figure 15. Timing diagram of the deadlocking multiple producer consumer example.

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.

Table 2. Logic element usage of the different examples.

Example	Logic Elements
Producer consumer	23
Double producer consumer	37
Double producer consumer deadlock	37

5. Conclusions

In this paper, a way to map CSP to hardware using CλaSH 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.

CλaSH code can be generated from the CSP model in TERRA, which can be used to generate hardware description code. This code can then can be synthesized and realized on a FPGA.

The generated code can be simulated at two levels. The first being a interpreted CλaSH 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 CλaSH 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. These parameters are used to make a generic block application specific. 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

is shown in Figure 16. The alternative relation can, optionally, be prioritised. Either way, the ALT in Figure 16 must wait for a signal on either 'g1' or 'g2' to arrive. If only one of them arrives, it accepts it and triggers the process guarded by that signal ('P' for 'g1' or 'Q' for 'g2'). If they arrive together or were already present when the ALT was activated, what happens next depends on whether the ALT was prioritised. If it was, the priority order defines which signal to take - say 'g1'. If it was not prioritised, the choice can be made *arbitrarily*. An acceptable resolution is to make the same choice as if it were prioritised (i.e. 'g1'), so that only a prioritised version of ALT need be implemented. A *random* choice could be made but that is computationally expensive and unnecessary. We expect that the implementation of the deterministic alternative, i.e. CλaSH code generation from TERRA diagrams is a matter of careful development. A non-deterministic alt will not be implemented since it is rarely used in physical applications.

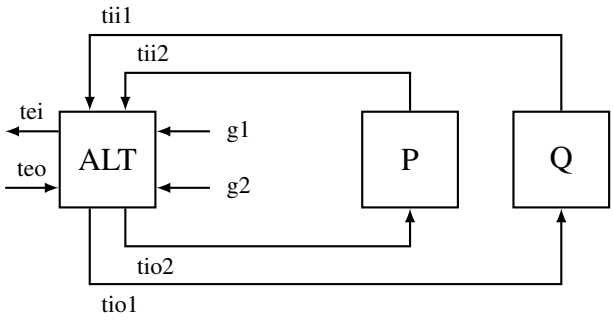


Figure 16. Data flow graph of the alternative composition. Lines carry tokens. Processes are denoted by the letters P and Q. The guards are denoted by g1 and g2.

References

[1] C. Baaij. CλasH : from haskell to hardware. Master's thesis, University of Twente, December 2009.

[2] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. CλaSH: Structural descriptions of synchronous hardware using Haskell. In *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, pages 714–721. IEEE Computer Society, September 2010.

[3] Simon Marlow. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[4] M. M. Bezemer. *Cyber-physical systems software development: way of working and tool suite*. PhD thesis, University of Twente, November 2013.

[5] George H Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[6] Rinse Wester, Christiaan Baaij, and Jan Kuper. A two step hardware design method using CλaSH. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 181–188. IEEE, 2012.

[7] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3: a parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*, 2015.

[8] M. M. Bezemer, R. J. W. Wilterdink, and J. F. Broenink. LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework. In *Proceedings of the Communicating Process Architectures 2011*, pages 157–175. IOS Press BV, June 2011.

[9] M. A. Groothuis, J. J. P. van Zuijlen, and J. F. Broenink. FPGA based control of a production cell system. In *Communicating Process Architectures 2008,*, volume 66 of *Concurrent Systems Engineering Series*, pages 135–148, Amsterdam, September 2008. IOS Press.

[10] Frank P. Coyle and Mitchell A. Thornton. From UML to HDL: a model driven architectural approach to hardware-software co-design. In *Information systems: new generations conference (ISNG)*, volume 1, pages 88–93, 2005.

[11] Twan Basten, Emiel van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian de Smet, Lou Somers, Egbert Teeselink, Nikola Trčka, Frits Vaandrager, Jacques Ver-

riet, Marc Voorhoeve, and Yang Yang. *Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset*, pages 90–105. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[12] Imran Rafiq Quadri. *MARTE based model driven design methodology for targeting dynamically reconfigurable FPGA based SoCs*. Theses, Université des Sciences et Technologie de Lille - Lille I, April 2010.

[13] Neil C.C. Brown. Communicating Haskell Processes: Composable explicit concurrency using monads. In *CPA*, pages 67–83, 2008.

[14] M. A. Groothuis and J. F. Broenink. HW/SW Design Space Exploration on the Production Cell Setup. In P.H. Welch, H. W. Roebbers, J. F. Broenink, and F. R. M. Barnes, editors, *Communicating Process Architectures 2009, Eindhoven, The Netherlands*, volume 67 of *Concurrent Systems Engineering Series*, pages 387–402, Amsterdam, November 2009. IOS Press. bibtex: groothuis2009cpa.

3 More Constructs in CλaSH

The paper listed in the previous chapter contains a mapping of the most basic attributes of CSP. This chapter gives an additional mappings and also solves the bootstrapping problem the mapping in the paper has. The first section (3.1) deals with the bootstrapping problem by injecting a token into the top structure. After further testing it appeared the CλaSH implementation of the PAR operator only worked for one cycle. A small addendum to the implementation is given in Section 3.2. The previous chapter also states a possible implementation for an alternative operator. In section 3.3 an implementation in CλaSH is explained. Up till now all given CSP implementations way to incorporate user-definable content. Therefore, it is needed to integrate user functionality into the structure. This is done by defining a user-definable code block which can be integrated as a process in the token structure proposed.

3.1 Initialisation

Until now all the CSP structures defined are not initialised with a starting token. The building blocks defined in the paper have to be started manually by injecting a token. For instance, the top structure in the producer-consumer example needs to be started to make the example execute. When a token is injected this top structure starts its children recursively.

The top construct can also be recursive. In CSP a recursion or loop is written as follows (Bezemer, 2013):

```
p = if (<expression>) then <process> ; p else SKIP
```

In the CSP meta-models the < $expression$ > is implemented as a property. If < $expression$ > is True the < $process$ >; p part is executed, p is activated and the loop continues.

The CSP top construct in hardware has to be activated at least once. When recursive, it has to start after it has finished. To achieve this behaviour a *starter* building block is introduced. This block injects at least one token into the top structure to activate it. When the structures recursive property is *True* the return token line is connected to the starter structure and it is activates the top structure again. So, the loop continues. This block in both configurations is displayed in Figure 3.1.

Figure 3.1: Starter structure in recursive and non-recursive configuration.

The starter block is implemented simply as a register initialised with *True*, as displayed in Listing 3.2. When started it will pass the initial token (since it is initialised as True). When the input of the block is connected, it will receive a *True* when the Top structure is ready and passes the token.

```
-- Start function
-- Generates one time token. Next token is generated when input is true.
starter = register True
```

Figure 3.2: CλaSH code of the starter block.

3.2 Parallel addendum

The parallel structure defined in the paper was only tested for one cycle. The passing of its token after it is finished was not tested. Two registers were used to receive the incoming tokens, namely $ti1$ and $ti2$. Those registers were never set when a token was received.

The new version of the parallel function is listed in Listing 1, the changed lines are highlighted. The new value for $ti1$ is *False* when both tokens are received, the value is *True* when its token is received on the input ($tii1$), and otherwise it should keep its previous value.

```
parallel' (te, ti1, ti2) (tei, tii1, tii2) = ((tei, ti1r, ti2r), (teo, tio1, tio2))
                where
                    -- Pass token when both are received
                    teo = ti1 && ti2

                      -- Only consume token one if both are received
                      ti1r | ti1 && ti2 = False
                           | tii1       = True
                           | otherwise  = ti1

                      -- Only consume token two if both are received
                      ti2r | ti1 && ti2 = False
                           | tii2       = True
                           | otherwise  = ti2

                    -- Pass token to both structures in parallel
                    tio1 = te
                    tio2 = te

parallel tei tii1 tii2 = mealy parallel' (False, False, False) (tei, tii1, tii2)
```

Listing 1: Parallel construct in CλaSH. The behaviour is described in *parallel'* in the format according to listing 1 in the paper. The function is transformed into a Mealy machine in *parallel*. The changed lines are highlighted

3.3 Alternative

The alternative is another compositional CSP relation type. The relation ensures one and only one of the child processes can become active. The rest of the child processes are skipped. An example of alternative relationships in TERRA is shown in 3.3. The alternative can either be guarded or unguarded. In the case of a guarded alternative, each of the child processes has a guard which determines whether a child process may become active. In the case of the unguarded alternative the first child that can establish communication first becomes active. When the unguarded alternative structure has two children that could be active at the same time, one should be picked at random. At this moment this random function is not implemented.

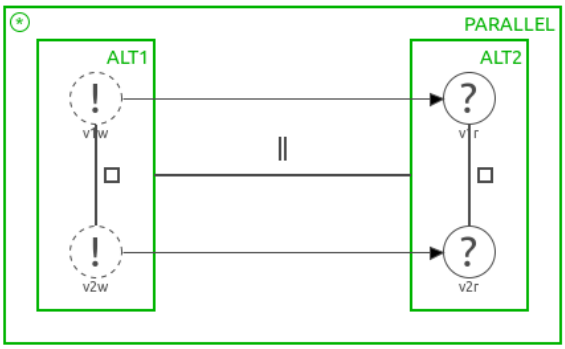


Figure 3.3: Two alternative structures in parallel. The *ALT* on the left hand side is *guarded*, the *ALT* on the right hand side is *uguarded*

The CλaSH implementation of the alternative relationship is shown in Listing 2. The Data flow graph is the same as shown in Figure 16 in the paper. The *Alternative* function has token inputs and outputs and guard input and outputs. The guards are labeled *g1* end *g2*. When one of the alternative guards is *true* the token is passed to that specific structure.

```
alternative' (te,ti1,ti2) (tei, tii1, tii2, g1, g2) = ((tei,tii1,tii2),(teo,tio1,tio2))
                where
                   -- Output the new token to one of the alt structure.
                   tio1 | g1 = te
                        | otherwise = False

                   tio2 | g2 = te
                        | otherwise = False

                   -- Pass token when finished
                   teo = ti1 || ti2
```

Listing 2: Haskell code for the alternative relationship

The guards can be provided as Haskell expressions. There is no random implemented. When no guard is set the activation depends on the possibility of communication of the alternatives children. For instance when two writers are in an alternative structure (e.g. Figure 3.3) and it is possible for the first writer to start communication, that child is activated. In this case the output token of the construct on the other side of the communication channel is used as guard. This construct can be a reader or a writer. When this construct is active, it is ready for communication so this alt child is chosen.

3.4 User-definable code block

The only CSP processes described so far are writers, readers and compositions of these. To give the TERRA application usability the *User-definable code block* or *CλaSH block* is introduced.

```
module TEMPLATE where

import CLaSH.Prelude

template ti = mealy template' <state init> ti

template' <state> (ti,<inputs>) = ((<state'>),(to,<outputs>))
    where
        <user defined implementation>
```

Listing 3: Template of a user-definable block. The user has to define the parts denoted with the angled brackets.

This function has to conform to some rules to fit in the CSP structure. This function needs at least a token input and output. This way it can be used as a CSP child. When the function has accepted the input token it may become active and accept or send data from channels, after passing the token is should become inactive.

The block defines one CλaSH function. The template for a user definable code block is given in Listing 3. In this template the user definable parts are denoted by angled brackets. The <*state*> is the current state of the *udb* Mealy machine, analogous the <*state'*> is the next state. Next to the token input (*ti*) and the token output (*to*) more inputs and outputs can be added. The user can define the body of the function completely as long as the tokens are handled correctly. The user can define the contents of the function.

Listing 4 is an example of a user definable block, a counter block. This counter, when active, counts to *cnt_max* and releases its token.

```
module TIMER where

import CLaSH.Prelude

timer ti = mealy timer' (0::Signed 18) ti

timer' cntr ti = ((cntr'),(to,cnt))
    where
        -- 1Khz @ 50Mhz clock
        cnt_max = 500000

        -- Set output
        to | cntr==0 = True
           | otherwise = False

        -- Increment counter
        cntr' | cntr == cnt_max = 0
              | otherwise       = cntr + 1
```

Listing 4: Timer example of a user definable block.

Since all blocks in the CSP structure should conform to this template it is also a basis for the standard I/O blocks. Appendix A.5.1 lists a set of these standard I/O blocks.

4 Design flow

The proposed design flow in Chapter 1 was partially accomplished. The general workflow stays the same, but some parts have to be done by hand. After code generation some parts have to be extended, this is explained in the Chapter below. Furthermore, instrumentation has to be added by hand.

The main program used in the design flow is the TERRA Tool suite. It is used to manage files, design models and generate code. Development of software and hardware design is done in specialised tools.

Next to TERRA, two other programs can be distinguished in the development for FPGA. The first is CλaSH, the second is Quartus. CλaSH is used to test CλaSH code and generate HDL (Hardware Description Language Files).

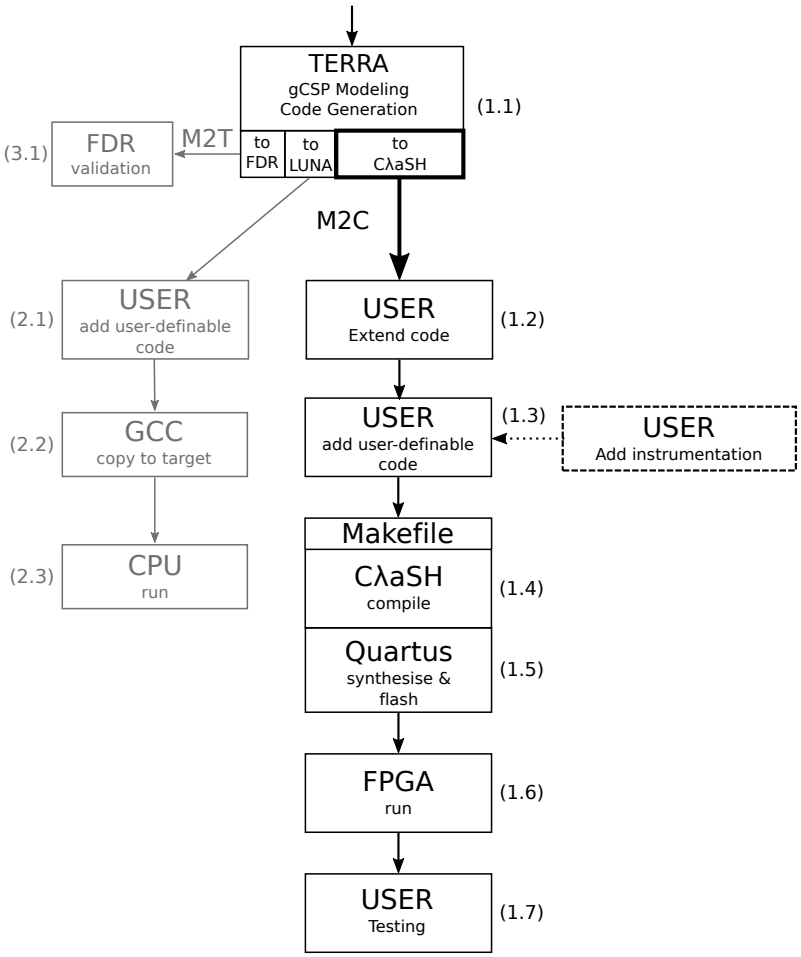


Figure 4.1: Implemented workflow. Each block depicts a tool and an action or several actions.

- TERRA (1.1)

  The design flow starts with creating a new project in TERRA. This is in essence a eclipse project, containing some models and some generated code. The next step is creating a CSP model in the new TERRA project. In the future this should be an architecture model where one can define which submodel is in CλaSH and which is in LUNA C++. The current implementation does not support this split. The creation of a gCSP diagram in TERRA automatically results in a corresponding model.

TERRA can generate code for FDR, LUNA and CλaSH from this meta-model. This is a M2T (model to text) transformation, where the model is the meta-model and the text is either CλaSH (2.1), C++ (3.1) or CSPm (4.1).

- Complete code (1.2)

  The code generation process is not yet finished. Several parts of the code have to be extended. For a complete explanation see the next Chapter. The function calls of the CλaSH code blocks have to be completed. The function call is there, but the inputs and outputs have to be added by hand.

  The input and the output of the generated *topEntity* have to be connected to the needed input and output pins. These pins differ per application, but mostly consists of actuators and sensors. These outputs can be connected to specific output pins by using the *ANN* notation.

  Furthermore, the instrumentation is not yet autogenerated. So, when the user wants to have some instrumentation included. It has to be done by hand.

- Fill in CλaSH user-definable blocks (1.3)

  Both C++ and CλaSH code has parts that have to be filled in by the user. For C++ code it is the C++ blocks, for CλaSH code it is the user-definable blocks.

- CλaSH Compiler (1.4)

  The next step is to invoke the CλaSH compiler to generate VHDL files for synthesis. This can be done by calling `make vhdl` from the command line.

- Quartus (1.5)

  The generated VHDL files can be synthesised using Quartus. This process can be simplified by making use of an Makefile. This Makefile creates the Quartus project, adds the VHDL files and Synthesises them.

- Flash FPGA (1.5)

  Quartus can also be used to flash the FPGA. This can also be done from the command line by calling `make program`

- Run on FPGA (1.5)

  Now the FPGA is flashed with the new hardware-description and can be run.

- Testing (1.7)

  The project can now be tested by using the instrumentation added before.

The workflow used to generate code for LUNA remains unchanged. The steps used are listed below:

- User adaptations (2.1)

  The generated C++ has some dedicated places where the user can add there own functionality.

- GCC (2.2)

  The final step is to compile the generated C++ and copy it to the target environment.

5 Code generation

Code generation tools transform models into source code, using model-to-code transformations. This generated source code can then be compiled or synthesised (in case of CλaSH), together with their libraries into the actual application. In a way the transformed model is transformed in a way that it can be executed.

In this work the TERRA tool chain is used for code generation. The tool chain is extended with CλaSH code generation. The generated CλaSH can be tested using a GHC (Glasgow Haskell Compiler)(Jones et al., 1993) simulation. GHC is an open-source Haskell compiler and interpreter. This generated CλaSH code is then transformed into a HDL description (in this case VHDL) using the similarly named CλaSH compiler. The generated HDL description can then either be tested by Modelsim or directly deployed on the FPGA using Quartus synthesis. This process is also described in the paper an visualised in Figure 6.

In the paper code generation was introduced. Only *readers*, *writers*, two-child *parallel* and *sequential* could be generated. These blocks can only be used to test simple CSP applications. In the following work the code generation has been extended by n-child *parallel* and *sequential* generation. The *alternative* structure can also be generated. This generation required an implementation of the guards. Finally, the user definable block can also be generated. The code generation is described in detail in the sections below. The explanation of the generation of the writer and the reader functions is skipped because their description and generation is unchanged since the paper.

Eclipse and the Eclipse Model Framework (EMF) are used in TERRA for modelling and code generation. EMF is used for the meta-models in TERRA and is also used for the meta-models in TERRA itself. The *Graphical Eclipse framework* (GEF) is used for the graphical CSP editor in TERRA. This CSP editor is unchanged

For these projects the graphical editor for CSP diagrams in TERRA is used as a base to design CSP structures. The model designed in this editor can be used to generate CSP, which subsequently could be analysed in FDR. LUNA code can also be generated directly from these models. The CSP Terra tool suite overview is shown in Figure 5.1, the bold part is the work added in this project.

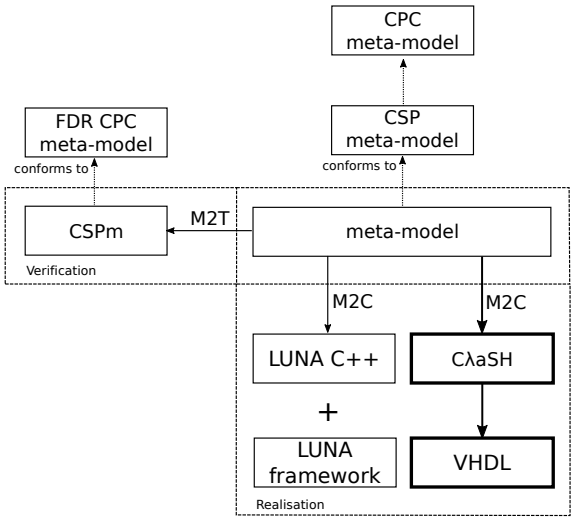


Figure 5.1: Partial TERRA tool suite overview (excluding simulation). The bold part is the extension in this project.

In this work the code generation is extended with Epsilon Object Language (EOL) functions and templates in the Epsilon Generation Language (EGL) to generate the CλaSH CSP library and the use of these functions, as well as some auxiliary files such as Makefiles and Altera Quartus configuration files.

The CSP models in TERRA are described in EOL. Each CSP element is a EOL object. These objects can be extended with functions to generate a piece of code corresponding to that specific object. To generate a piece of code for every block in the CSP diagram, the *process* function of the diagram object is called, which subsequently calls the process function of all its children. This way piece of code is generated for all the CSP elements in the diagram.

5.1 Levels of code generation

The code-generation support in the TERRA tool suite is work in progress. Some parts of the diagrams in TERRA are completely generated and some have to be edited by hand.

To illustrate the level of code generation implemented, four different levels of code generation are distinguished. The levels describe what of the code is generated. The first level says only the function definition is generated. The user has to adapt this line in the generated code to add the input and output variables, and add the function description. The second level says also the in and output variables are generated, the third level says also the structure of the body is generated, the fourth says even the body is generated. Figure 5.2 shows the different levels of code generation with small examples on the right hand side.

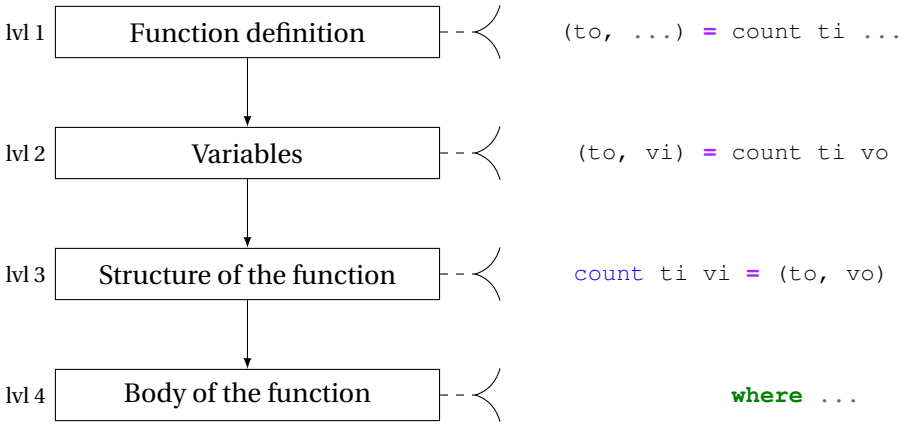


Figure 5.2: Levels of code generation

Table 5.1 contains a list of the functions that can be generated in the current state of the TERRA tool suite. Each of the functions has a corresponding code generation level. Most of the functions can be generated entirely within TERRA. The CλaSH code block (UDB) has only level one support, only the function name is generated in the code.

Table 5.1: Level of code generation per functionality.

Name	Level	Comment
Writer	4	
Reader	4	
Parallel	4	More than 2 leafs possible
Sequential	4	More than 2 leafs possible
Alternative	4	Only 2 leaf
Code block	1	

5.2 Implementation details of CSP Operators

5.2.1 Sequential

Using the *sequential* operator, the sequential child nodes are activated in order and one at a time. The sequential construct children can be daisy chained. The token output of each child can be connected to the token input of the next node. The sequential construct itself is only used to inject the token and pass it to its parents. This way the injected token propagates through all the children until it is returned to the sequential construct. This process is illustrated in Figure 5.3.

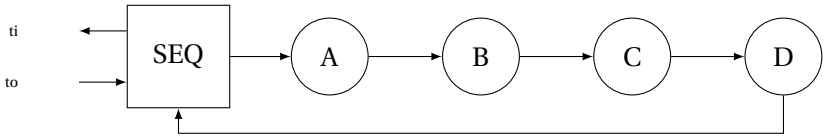


Figure 5.3: Propagation of tokens through child nodes of a sequential construct.

5.2.2 Parallel

Using the *parallel* operator all the child nodes are activated simultaneously, when all child nodes are finished the structure as a whole is finished. The Mapping CSP Models to Hardware Using CÎżaSH paper listed in Chapter 2 only describes a parallel function that can parallelise two child nodes. To parallelise more than two child nodes a separate function is required. This method would require a lot of different functions. Therefore, a different method is proposed. Parallelising parallel structures to parallelise more than two child nodes. This method is displayed in Figure 5.4. This figure displays the parallelisation of five child nodes, denoted with circles. First two sets of processes are parallelised (*P0* and *P1*). Thereafter, those two parallel processes are also parallelised (*P2*). Finally *P2* is parallelised with the last child process *E*.

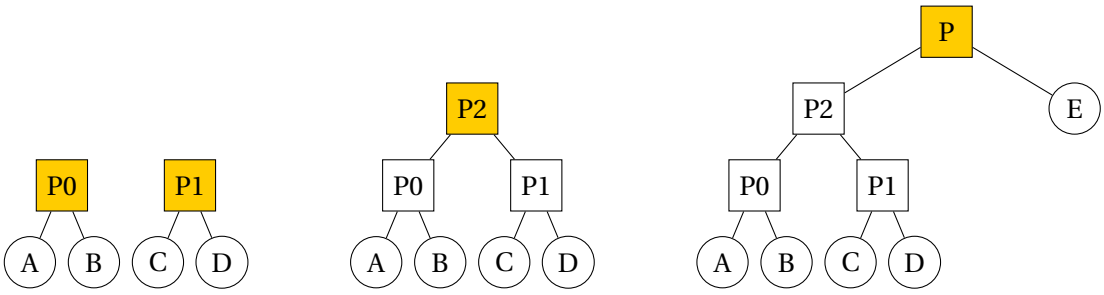


Figure 5.4: Process of parallel code generation

5.2.3 Alternative

The *alternative* operator described in the previous chapter only works for an *alt* with two children. To create an alt with more than two children there are two possible implementations. One implementation is to write an new alt function for the amount of children needed. This approach is displayed in Figure 5.5. This approach required to generate a new function for every amount of leaves.

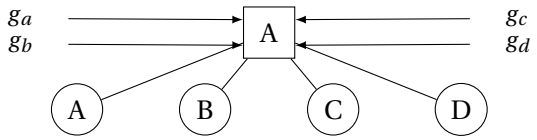


Figure 5.5: *ALT* guards by extending the *ALT* function.

The second implementation is to compose ALTs out of two-child ALT functions. For instance, with four children by using three two-child alt structures as displayed in Figure 5.6. The guards for each child are used to determine which leaf needs to be activated. These guards are combined by an 'OR', so when one of the two guards evaluates as true, this leaf is activated.

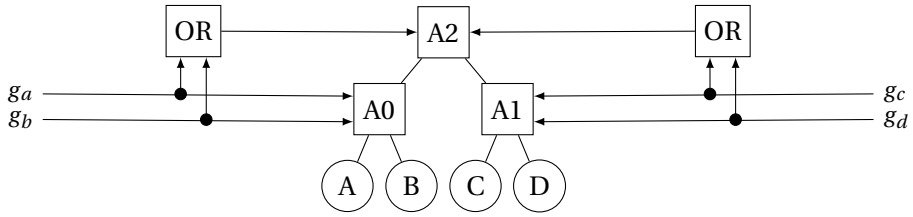


Figure 5.6: *ALT* guards by using a tree with leaves.

Both methods are complicated to generate. The CSP library was designed to exist of small re-usable blocks which can be interconnected. The first method requires for the library to be extended with functions for the amount of child nodes needed. This makes the setup less modular. Also the function calls become larger and less readable, since the amount of arguments can become very large.

The second approach does consist of the pre-defined CSP blocks. In this case the tree approach makes the code less readable. Another drawback is the tokens have to propagate through the leaf structure. This only introduces a very minor delay, according to the following equation: $2^n = a$. Where n is the amount of clock cycles in delay and a is the amount of leafs. Although this delay is very minor, it is not introduced while using a re-generated CSP library as described above.

5.3 Auxiliary files

Next to the CλaSH files some auxiliary files are generated.

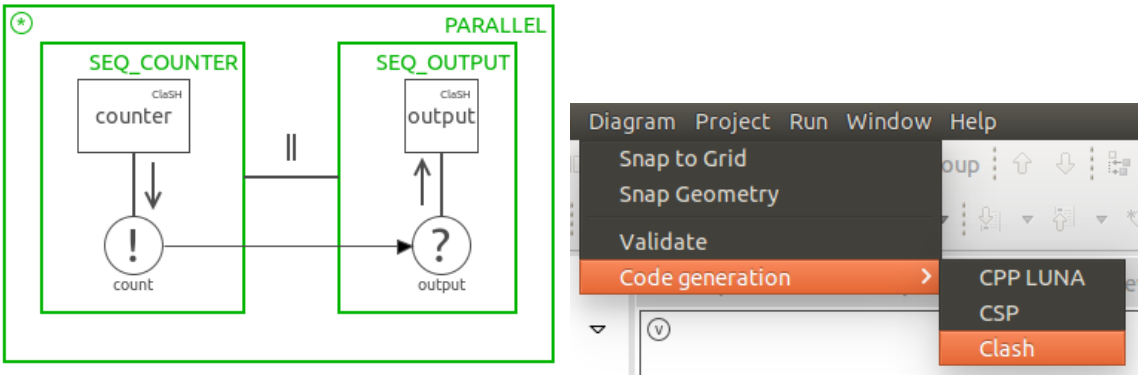
First of all a GNU Makefile is generated. This makefile can start the compilation of the CλaSH files, start synthesising the VHDL files and flash the FPGA. The makefile is located in the CλaSH root folder of the current project. The command *make vhdl* calls the CλaSH compiler and generates the VHDL files. The command *make quartus* start synthesis with the Quartus toolchain. A new project is started and the generated VHDL files and pin definitions are automatically added. *make program* flashes the generated *sof* file to the FPGA.

The *Quartus II Settings File* (QSF) file is a static file containing the pin definitions, in the current state this file contains the pin definitions of the RaMstix.

5.4 Code generation example

To explain the changes necessary in the generated code an example is given below. First a simple CSP diagram is drawn in TERRA, subsequently the CλaSH code is generated followed by the necessary changes.

The diagram used is shown in Figure 5.7a. Two parallel structures can be distinguished. The first contains a *counter* UDB in sequence with a writer, the second contains a reader in sequence with an *output* UDB. The diagram uses two defined variables. The first is the *count* variable, which is the counter value. The second variable is the *output* variable, which is the output value. Both variables have the type *Unsigned 3*. This is a 3-bit unsigned data-type. The counter UDB sets the *count* variable and the output UDB uses the *output* variable.



(a) The CSP diagram for the "counter" example. **(b)** CλaSH code generation button in the menu.

Figure 5.7

The next step is to generate code CλaSH code within TERRA. The generation can be started by clicking CλaSH in de code generation menu, as displayed in Figure 5.7b. The resulting generated code is listed in Appendix A.2 in Listing A.1.

In the generated code the two UDB function calls can be found at line 38 and 51. Only the adaptations for the function *counter* are discussed below. In the generated code the line contains one output, a token *in* and a token *out*. The counter function will only have one output. The adapted version of Line 38 is listed in Listing 5.

```
(to_output, count) = counter ti_output
```

Listing 5: Counter function call at line 38 (see Appendix A.2).

The next step is to define the function and the body of the function, the code is listed in Listing 6. The counter is realised by using a *Mealy* machine with one internal state. This state is the counter value and is called `cntr`. The counter value is only updated when the token is available. The counter counts from 0 to 7 and then starts over.

```
-- Counter function definition
counter ti = mealyB counter' (0::Unsigned 3) ti

counter' cntr ti = ((cntr'),(ti, Just cntr))
        where
            cnt_max = 7

            -- Increment counter only when active
            cntr' | cntr == cnt_max    = 0
                  | ti                 = cntr + 1
                  | otherwise          = cntr
```

Listing 6: Example of the "counter" function

The next step is to define the output. This code is listed in Listing 7 For this example the output is the variable *output*, this is already default in the generated code. So, no changes are necessary here. The data type *Maybe* is used in this example. The input and output can be either *Nothing* or a *Unsigned 3* value.

```
topEntity:: Signal (Maybe (Unsigned 3)) -> Signal (Maybe (Unsigned 3))
topEntity input = output
```

Listing 7: Example of the "counter" function

6 Testing

In Chapter 3 the Alternative operator is added and a user-definable block. In Section 6.1 a signal level test of the Alternative is shown. In Section 6.2 a test of the UDB block is shown. (note: The paper in Chapter 2 already has some tests for some CSP operators in CλaSH.)

In section 6.3 a demonstrator is shown. This test aims to demonstrate the design flow of FPGA within the TERRA tool suite, as well as a overall test case of the CλaSH CSP mapping presented in this paper. Firstly the setup is described, secondly the implementation using TERRA. Finally some resulting measurements are shown.

6.1 Alternative

In the paper in Chapter 2 the Parallel and Sequential operator were tested by using a simple producer-consumer example. The Alternative operator, also introduced in Chapter 2, is explained in Section 3.3. The following example is done in the same manner as the examples in the paper.

The following example is composed of two writers, two readers and two channels for communication. Figure 6.1 shows the structure and the relations of the processes. The two writers and the two readers are in alternative relation. The guards of the two writers are hard-coded for this example. The first writer, *w0*, has the signal *True* as guard. The second writer has the signal *False* as guard. The readers are unguarded. The code for this example is generated within the TERRA tool suite. Since the autogenerated code is substantial in size it is shown in Appendix A.2 in Listing A.2.

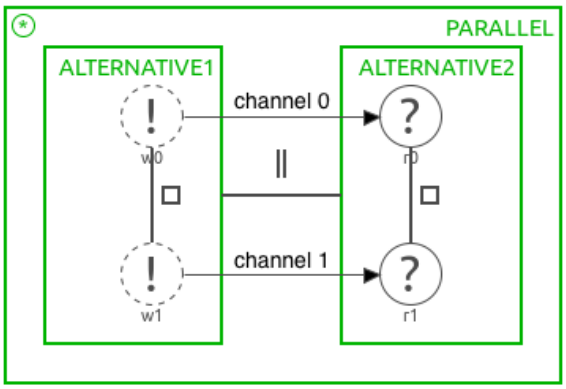


Figure 6.1: Alternative test CSP diagram within the TERRA tool suite.

Using the CλaSH compiler, the description of Listing A.2 is compiled and simulated. The simulation results are converted to a timing diagram as shown in Figure 6.2.

First, the start token is injected to trigger the execution of the parallel construct. Subsequently, the writer and reader are activated. Since the guard of writer *w0* is *True*, this process is activated. The readers in *ALTERNATIVE2* are unguarded. The alternative structure checks if one of the readers is able to perform communication without blocking. When it is possible for one of the readers to communicate, it is activated. In this case the first reader, *r0*, can be activated and communication takes place. As expected, only the communication takes place.

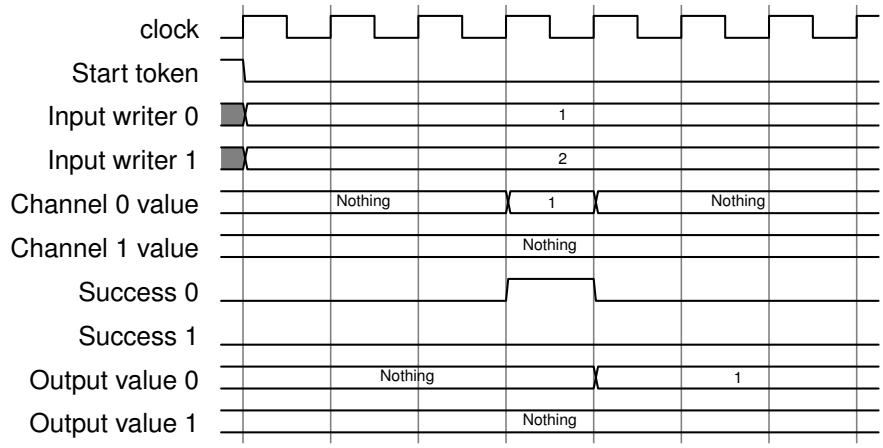


Figure 6.2: Timing diagram of the alternative operator example.

6.2 UDB - "Counter" example.

The code generation example explained in Section 5.4 is also compiled and simulated using the CλaSH compiler. The resulting signals are converted to a timing diagram and are shown in Figure 6.3. The counter example counts from 0 to 7, the output token of reader is also displayed. The *output* variable is first Nothing. The clock cycle after the readers returns its token, its output value is valid.

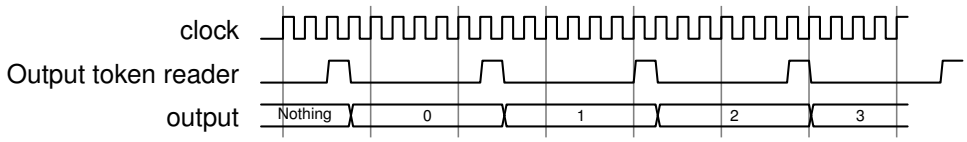


Figure 6.3: Timing diagram of the alternative example.

As can be seen in the figure the counter starts counting at zero and increases every cycle of the diagram. It takes 7 clock cycles for one cycle of the diagram to complete.

6.3 Demonstrator

6.3.1 Setup

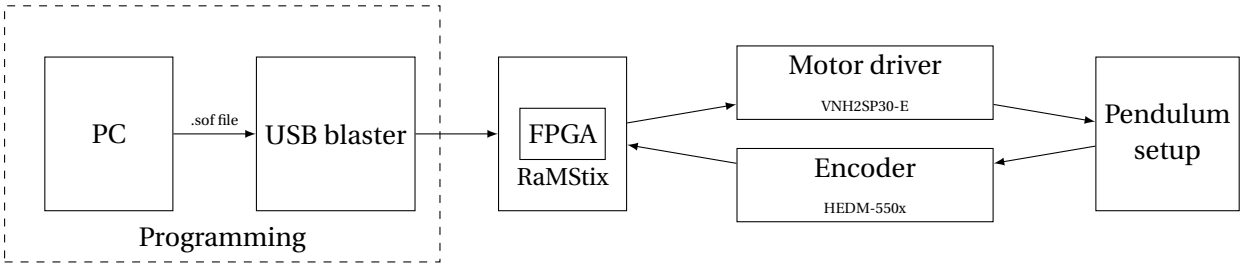


Figure 6.4: Setup overview, the PC and blaster are used to program the FPGA. The FPGA is connected to a motor driver and and encoder. Which are connected to the setup.

The CSP mapping and VHDL is tested using a “proof of concept" approach. The setup consists of a Ramstix board and a Pendulum setup, see Figure 6.4. The Ramstix board is used only for its FPGA; an Altera Cyclone III. In this proof of concept CSP is used for control, as well as I/O and safety. The FPGA is programmed using a PC combined with an USB Blaster. The FPGA is programmed from a PC using Quartus and a USB Blaster. The aim of the setup is to control the position of the pendulum.

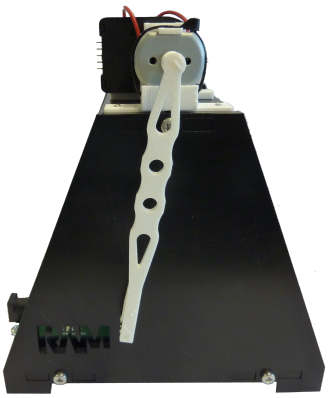


Figure 6.5: Pendulum setup (T.G. Broenink, 2015).

The pendulum setup displayed in Figure 6.5 is developed within RaM. The setup has already been used for other projects. For this reason the setup is also used in this project as a demonstrator. The pendulum setup consists of a safety/power board, a motor driver board, a DC motor and an encoder. The DC motor drives an inverted pendulum. The pendulum can be stabilised at the top, or it can be used as a real pendulum: swinging back and forth. This proof of concept’s focus is to prove the method proposed in this report is suitable for control applications.

The controller used for the course ESL is implemented in this proof of concept. The original controller consisted of a PD controller including gravity compensation. For gravity compensation a sinus function is necessary. This is difficult on an FPGA. Therefore, for simplicity reasons the gravity compensation is left out, leaving a PD controller.

The PD controller has one input, which is is the setpoint of the pendulum, there is one output which is the PWM signal for the DC motor.

6.3.2 Implementation

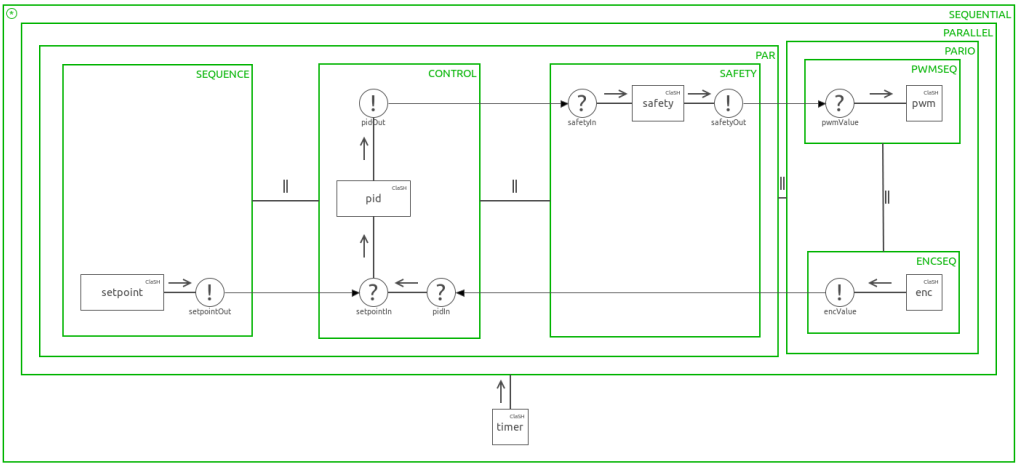


Figure 6.6: CSP used for controlling the pendulum setup.

The gCSP structure is designed in TERRA. The used diagram is displayed in Figure 6.6. Four parallel blocks can be distinguished. A sequence generator, a controller, a safety layer and I/O. Each of these blocks contains user defined blocks. These user defined blocks are listed in Appendix A.5.

The sequence generator generates the pendulum set-points. These set-points can either be a fixed position or a square wave. In this test the set-point is switched between -0.6 rad and 0.6 rad every second to make a pendulum-like movement. The controller part accepts the set point and the position measured by the encoder. A PD controller is used to determine the PWM output value. The PD controller uses *SFixed* values for the multiplications in the control loop.

The safety layer only checks the outgoing PWM value. The safety is basically a clamp, limiting the PWM value. The I/O part contains the PWM controller and the Quadrature encoder. The PWM output is also generated in VHDL hardware. These I/O blocks are displayed and explained in Appendix A.5.1.

Below the four parallel blocks a timer is used. This block starts the four top structures every 0.01 second to make sure the control loop runs at 100Hz.

Instrumentation

The system described above is run completely on an FPGA. Since it is not yet possible to test using co-simulation. For instance co-simulating using the CλaSH interpreter and 20-sim. Therefore the system was only tested using the physical system. To test and measure the system it was necessary to add some instrumentation to the generated code.

The CλaSH code is flashed to the FPGA on the RaMStix. The FPGA on the RaMStix is connected to the Gumstix using a GPMC (General Purpose Memory Controller) bus. Gumstix runs Linux and is connected using an Ethernet cable. Since, the connections were already in place it was decided to use the GPMC bus to extract values from the FPGA. This requires an implementation of the GPMC protocol in CλaSH. The FPGA acts as a block of memory which can be read or written to via the GPMC bus. The code for the GPMC block can be found in Appendix A.6.1.

The GPMC communication requires a bi-directional port. This is not yet supported in CλaSH. Therefore a VHDL wrapper is written. This wrapper outputs values when the nOE bit is low, otherwise the bus is kept in high-impedance state. In Figure 6.7 an overview is shown of the blocks in the setup combined with the added instrumentation. Dashed blocks are used to indicate which parts are written in CλaSH and which in VHDL.

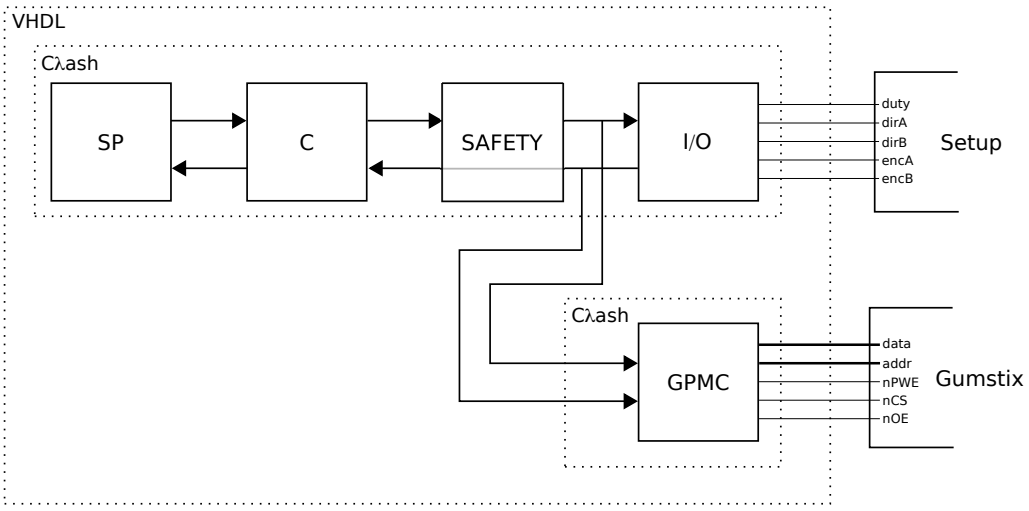


Figure 6.7: Added instrumentation to the setup.

6.3.3 Results

The model in TERRA was used to generate CλaSH code. The code for the user-definable blocks was added, in the same way as explained in section 5.4. Furthermore, the instrumentation was added so the PWM value and the encoder counter can be retrieved. Next two tests are performed. The first one is one without the dynamics of the plant, the second is a measurement of the pendulum while in operation.

The test without dynamics is achieved as follows. First the system is simulated in the CλaSH compiler, for this test the setpoint is kept at zero. The angle is used as an input, the measured angle is varied from -0.6 radians to 0.6 radians. The PWM output is measured and logged. In Figure 6.9a the duty-cycle is shown versus the Angle.

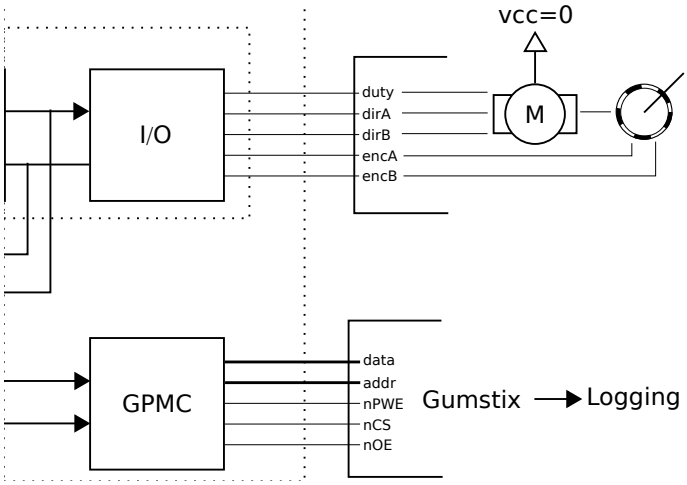
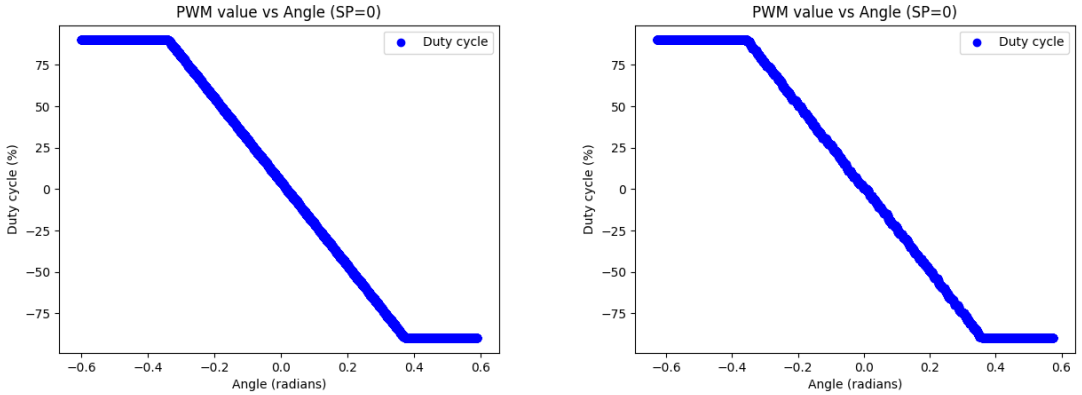


Figure 6.8: Test without dynamics. The motor power is disabled and the pendulum is rotated by hand.

Next, the measurement was performed on the physical setup, as displayed in Figure 6.8. The duty-cycle and the encoder angle were measured by logging via the GPMC bus and the angle of the pendulum was varied by moving the pendulum by hand. The resulting measurements are shown in Figure 6.9b. For this test the power to the motor was disabled.

As can be seen from the plots, the behaviour of the simulation and the measurements on the setup give the same results.



(a) Simulation in CλaSH of the PWM value vs the Angle of the encoder.

(b) Measurement of the setup of the PWM value vs the Angle of the encoder.

Figure 6.9: Comparison of measurement and simulation of the setup, without the dynamics of the plant.

The next test is a measurement on the setup while the pendulum is operational. The setpoint is switched every second between −0.6 radians and 0.6 radians. During the test the encoder position and the PWM signal are logged via the GPMC bus, as displayed in Figure 6.7.

The resulting measurements are shown in Figure 6.10. The pendulum does swing around ever second. As can be seen in the plot the pendulum does not completely reach its setpoint. A steady-state error is expected since this is a PD-controller. The steady-state error in this test is quite large and is quite possibly caused by another factor. Due to a lack of time this is not investigated.

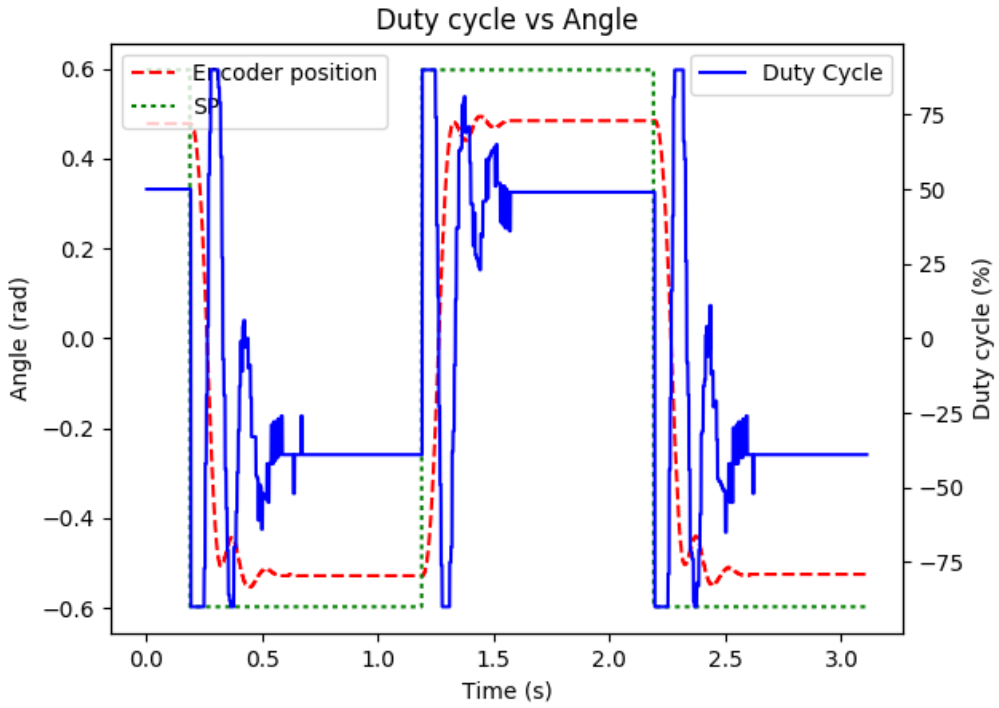


Figure 6.10: Measurement of the pendulum setup.

The resource usage of the setup is shown in Table 6.1. The Table shows the used amount of Logic cells, registers and DSP elements. In this table all the used DSP elements are multipliers. The resource table has been split in two parts. The first part is the resource usage of the actual setup; the encoder, controller, PWM generator, safety layer and setpoint generator. The second part is the added instrumentation: the GPMC memory block.

Most of the parts of the system do not use a lot of resources. Only the controller uses a large amount of logic cells and registers. This is due to several fixed-point multiplications. The controller has to store previous values to calculate the D action. All these values are in the large fixed-point format and take a lot of space. As can be seen from the table, the GPMC controller takes an enormous amount of space in both logic cells and registers. This is because the GPMC is basically a large memory block.

The other category in the table is a sum of all the logic involved in the CSP diagram. This includes token distribution and readers and writers. Readers and writers both require a register size of the data type they communicate, combined with registers for keeping the token value.

Table 6.1: FPGA resource consumption

Block	FPGA Resources		
	Logic cells	Registers	DSP Elements
Encoder	25	20	0
Controller	699	61	36
PWM	84	36	0
Safety	30	0	0
Setpoint	49	27	0
Other	103	153	0
Total setup	990	297	36
GPMC	2526	2074	0
Total	3503	2371	36
Percentage of FPGA	9%	6%	14%

7 Conclusions and Recommendations

The main goal of this assignment is to move hard real-time functionality to an FPGA in embedded control solutions. This is achieved by implementing a mapping of CSP to CλaSH. Furthermore, the conversion from CSP diagrams is partially automated within the TERRA Tool suite.

The mapping of CSP diagrams to CλaSH is achieved by defining a CSP library written in CλaSH. Each of the operators used in TERRA have been mapped to CλaSH functions. Each of these functions have been tested on a signal level, to verify their behaviour.

The design flow used within TERRA has been extended with an FPGA design flow. This starts with code generation in TERRA, this works but has some limitations. The generated code still has to be adapted by the user to make it usable. Furthermore, instrumentation has to be added by hand to make the setup testable. The adapted code can be compiled by using the CλaSH compiler to generate VHDL. The generated VHDL can subsequently be used for synthesis.

The workflow has been tested using a proof of concept approach. A simple robotic demonstrator is completely controlled by the FPGA. The demonstrator works, but does still have a steady-state error which has to be investigated further.

The amount of computation offered by the FPGA is suitable for relatively large control problems. The amount of logic elements used for the demonstrator was relatively low. The current way of adding instrumentation adds a relatively large amount of resource consumption. Since this is a static amount, this should not be a problem.

To summarise, the FPGA design support in CλaSH is suitable to usable in robotic applications, but at this point it is necessary for the user to have intricate knowledge of computer engineering. Furthermore, testing using co-simulation is not possible at this point.

7.1 Recommendations

In Figure 7.1 the parts that still need work or need to be added are emphasised. In the current implementation of the workflow it is only possible to choose between the FPGA and the embedded processor. Combining the two would require a significant amount of input from the user. The desired situation is that on an architecture level it is possible to differ between CSP executed on the FPGA and executed on the embedded processor. The architecture editor uses ports to communicate between the different blocks. This communication can for instance be achieved by using the GPMC bus on the RaMStix. The proposed FPGA design flow is a promising way to develop robotic systems. At this point the workflow is incomplete due to some missing parts in code generation and missing co-simulation.

Another use-case of the ports is the one between the CSP on the FPGA and the plant. When code is generated this should result in the correct connections in the setup. During testing the port should be able to connect to a model of the plant in for instance 20-sim.

It is desirable that it is possible to add this instrumentation directly from within the TERRA tool suite. By giving the user the option to select which signals they want to log. Currently the instrumentation needs to be added by hand.

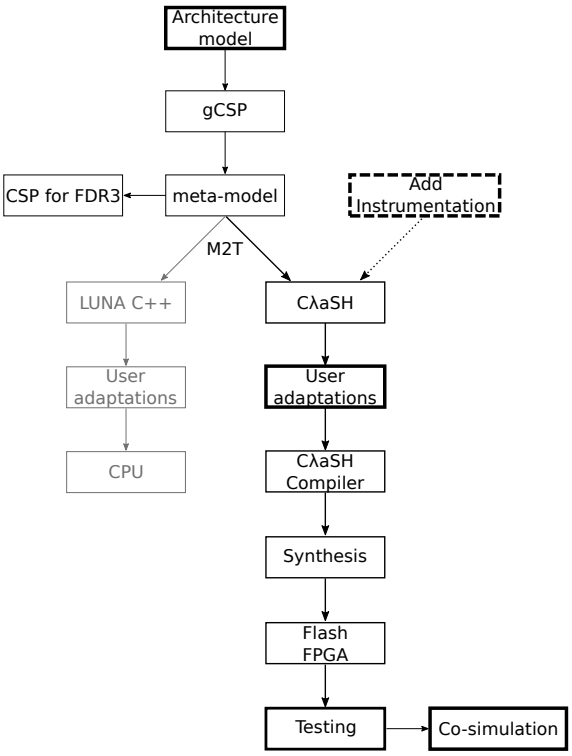


Figure 7.1: Todo in workflow. The parts that need work are emphasised with dark lines. The already existing LUNA C++ workflow is greyed out.

To simplify the workflow within the TERRA tool suite, it would also be necessary to extend the graphical editor with a code editor for the user-definable blocks. This way the user can directly edit the user-definable blocks with in the tool suite.

Finally, there are still some parts that need to be added in the CλaSH CSP library. The first is that currently the alternative is only possible for two child processes. It is possible to solve this using a tree like the approach for the parallel structure. This would require a lot of glue logic, it might be easier to just define an alternative block for each amount of child processes. These definitions could also be generated.

A Appendix

In this report two appendices are added. The first appendix elaborates on the design

In appendix A.1 an introduction to CSP, LUNA and TERRA is added for the novice reader in these subjects.

Appendix A.5 elaborates further about the design of the CSP mapping and generation thereof.

Appendix B consists of a set of additional appendices. The design requirements, the manual of the software, an overview of the added plugins to the TERRA toolchain. Furthermore, a step by step manual is given that explains how to create a new plugin in TERRA, to speedup further development.

A.1 Introduction to CSP, LUNA and TERRA

Communicating Sequential Processes (CSP) (Hoare, 1985) is a modelling language describing the interaction between several CSP processes. The language describes processes that execute independently and communicate through channels. CSP provides several different relationships between processes, such as sequential and parallel.

These processes can be synchronised using communication channels. There is a buffered and an unbuffered variant of the channel. The unbuffered channel is also known as *rendezvous-channel*. Only the rendez-vous channel makes synchronising processes possible. For example, when a process A requires a value from a process B, process A is blocked until process B produces the value. The Buffered communication does not give synchronisation. This type of channel can be used to link components with different real-time guarantees (Bezemer, 2013).

Using CSP it is possible to construct live- or deadlock structures. Deadlocks are when two or more processes are infinitely waiting on eachother. Live lock occurs when two or more processes respond to eachother indefinitely. The processes appear not to be blocked, but make no progress because they are to busy responding to each other. This can easily be missed by the designer. Therefore it is useful that CSP models can be formally checked. This gives information about for example live- and deadlocks. The *FDR3 CSP Refinement Checker* Gibson-Robinson et al. (2014) makes it possible to formally check CSP models.

The *TERRA tool chain* is a model driven design (MDD) tool chain for the design process of embedded systems. All models in TERRA are based on a CPC and derived meta-models. It supports designing in CSP models. It also integrates other tools, such as 20-sim models and co-simulation. The models in TERRA can be formally verified by exporting to machine readable CSP.

The CPC model defines basic structural elements consisting of components and ports. The CSP meta-model used within TERRA extends this model. It defines CSP compositions, writers, and readers.

The LUNA execution framework is made to reduce the complexity of the generated code within TERRA. This execution framework provides a static implementation of the CSP meta-models as described before. The execution framework is made for cyber-physical systems which require real-time guarantees. The execution framework originally used the CTC++ library, which provides hard real-time execution for CSP-based applications. This library has become outdated, and LUNA has support for the CTC++ features in its core (Bezemer et al., 2011).

A.2 Code-generation Example

A.3 Counter

The code listed in this section is to support the code generation example given in Chapter 5.

```
1  -- Generated by TERRA CSPm2Clash version 0.0.1
2  -- Input file: counter.cspm
3
4  module COUNTER where
5
6  import CLaSH.Prelude
7  import CSP
8
9  -- Code to determine datatype, has to be changed to get the data type of the ports
10 -- protected region user defined area for ann on begin --
11
12 -- protected region user defined area for ann end --
13 topEntity :: Signal (Maybe (Unsigned 3)) -> Signal (Maybe (Unsigned 3))
14 topEntity input = output
15 where
16     -- Temporary
17     output = input
18     -- protected region user defined area for connecting inputs/outputs on begin --
19
20     -- protected region user defined area for connecting inputs/outputs end --
21
22     -- Starter function
23     ti_PARALLEL = starter to_PARALLEL
24
25     -- List of variables
26     --count
27     --output
28
29
30     -- Name: PARALLEL
31     (to_PARALLEL, ti_SEQ_OUTPUT, ti_SEQ_COUNTER) = parallel ti_PARALLEL to_SEQ_OUTPUT to_SEQ_COUNTER
32
33     -- Name: SEQ_COUNTER
34     ti_counter = to_countWriter
35     (to_SEQ_COUNTER, ti_countWriter) = sequential ti_SEQ_COUNTER to_counter
36
37     -- Name: counter
38     (to_counter, sendValue) = counter ti_counter
39
40     -- Name: countWriter
41     (to_countWriter, value_out_MainModel_countWriter_MainModel_countReader) = writer count
           success_MainModel_countWriter_MainModel_countReader ti_countWriter
42
43     -- Name: SEQ_OUTPUT
44     ti_countReader = to_output
45     (to_SEQ_OUTPUT, ti_output) = sequential ti_SEQ_OUTPUT to_countReader
46
47     -- Name: countReader
48     (to_countReader, output, rrMainModel_countWriter_MainModel_countReader) = reader
           value_in_MainModel_countWriter_MainModel_countReader ti_countReader
49
50     -- Name: output
51     (to_output, sendValue) = output ti_output
52
53     -- Channels
54     (value_in_MainModel_countWriter_MainModel_countReader, success_MainModel_countWriter_MainModel_countReader) = channel
           value_out_MainModel_countWriter_MainModel_countReader rrMainModel_countWriter_MainModel_countReader
55
56 -- protected region user defined area on begin --
57
58 -- protected region user defined area end --
```

Figure A.1: “counter” example generated code without edits.

A.4 Alternative operator example

The code listed in this section is to support the alternative operator example given in Chapter 6.

```
1  -- Generated by TERRA CSPm2Clash version 0.0.1
2  -- Input file: alternative.cspm
3
4  module ALTERNATIVE where
5
6  import CLaSH.Prelude
7  import CSP
8
9  -- Code to determine datatype, has to be changed to get the data type of the ports
10 -- protected region user defined area for ann on begin --
11 -- protected region user defined area for ann end --
12 topEntity:: Signal (Maybe (Signed 8),Maybe (Signed 8)) -> Signal (Maybe (Signed 8),Maybe (Signed 8),Bool,Bool,Maybe (Signed 8),
       Maybe (Signed 8))
13 topEntity input = output
14                 where
15                     -- protected region user defined area for connecting inputs/outputs on begin --
16                     output = bundle (value_in_MainModel_writer0_MainModel_reader0, value_in_MainModel_writer1_MainModel_reader1,
                              success_MainModel_writer0_MainModel_reader0, success_MainModel_writer1_MainModel_reader1,r1,r2)
17                     (w0,w1) = unbundle input
18                     -- protected region user defined area for connecting inputs/outputs end --
19
20                     -- Starter function
21                     ti_PARALLEL = starter to_PARALLEL
22
23                     -- List of variables
24                     --w0
25                     --w1
26                     --r1
27                     --r2
28
29                     -- Name: PARALLEL
30                     (to_PARALLEL, ti_ALTERNATIVE1, ti_ALTERNATIVE2) = parallel ti_PARALLEL to_ALTERNATIVE1 to_ALTERNATIVE2
31
32                     -- Name: ALTERNATIVE1
33                     (to_ALTERNATIVE1, ti_writer0, ti_writer1) = alternative ti_ALTERNATIVE1 to_writer0 to_writer1  (signal True)
                              (signal False)
34
35                     -- Name: writer0
36                     (to_writer0,value_out_MainModel_writer0_MainModel_reader0) = writer w0
                              success_MainModel_writer0_MainModel_reader0 ti_writer0
37
38                     -- Name: writer1
39                     (to_writer1,value_out_MainModel_writer1_MainModel_reader1) = writer w1
                              success_MainModel_writer1_MainModel_reader1 ti_writer1
40
41                     -- Name: ALTERNATIVE2
42                     (to_ALTERNATIVE2, ti_reader0, ti_reader1) = alternative ti_ALTERNATIVE2 to_reader0 to_reader1  ti_writer0
                              ti_writer1
43
44                     -- Name: reader0
45                     (to_reader0,r1,rrMainModel_writer0_MainModel_reader0) = reader value_in_MainModel_writer0_MainModel_reader0
                              ti_reader0
46
47                     -- Name: reader1
48                     (to_reader1,r2,rrMainModel_writer1_MainModel_reader1) = reader value_in_MainModel_writer1_MainModel_reader1
                              ti_reader1
49
50                     -- Channels
51                     (value_in_MainModel_writer0_MainModel_reader0, success_MainModel_writer0_MainModel_reader0) = channel
                              value_out_MainModel_writer0_MainModel_reader0 rrMainModel_writer0_MainModel_reader0
52                     (value_in_MainModel_writer1_MainModel_reader1, success_MainModel_writer1_MainModel_reader1) = channel
                              value_out_MainModel_writer1_MainModel_reader1 rrMainModel_writer1_MainModel_reader1
53
54
55 -- protected region user defined area on begin --
56 -- protected region user defined area end --
```

Figure A.2: "Alternative" example generated code.

A.5 Design details

This section lists a set of I/O block examples in Appendix A.5.1. Appendix **??** gives some additional details about the implementation of the CλaSH code generation in TERRA.

A.5.1 I/O Block examples

This section lists a set of I/O block examples. They both conform to the user definable block template defined in Chapter **??**. These two examples are used in the test setup described in Chapter 6. The PWM block was used to drive the motor connected to the pendulum, and the encoder block was used to measure its rotation.

PWM block

The first example is a PWM block and is displayed in Listing 8. PWM is in this case used to control the amount of power supplied to a motor. The PWM driver board used in this example expects three inputs: the PWM signal and two direction bits.

When the token is available on the input the structure is active and can accept the new duty cycle value from the input (input *duty_in*). The *pwm* function uses a counter to generate the PWM signal. This counter corresponds with the PWM frequency which is set with the variable *cnt_max*. In this case the counter is set to 2000, which corresponds with a PWM frequency of 25kHz.

```
module PWM where

import CLaSH.Prelude

pwm t dutyIn = mealyB pwm' (0,0, False) (t, dutyIn)

pwm' (cntr,duty, token) (tokenIn, duty_in) = ((cntr',duty', token'),
                                              (tokenPass, out,dira,dirb))
    where
        -- fclk = 50e6 (50 MHz) PWM @ 25kHz
        cnt_max = 2000

        -- Pass token directly
        token' = tokenIn
        tokenPass =  token

        -- Duty with sign, when input is Nothing set duty to previous value
        d = maybe duty id duty_in

        -- Next duty is new duty or previous if nothing, only when token available.
        duty' | token     = abs d
              | otherwise = duty

        -- Increase counter until max is reached
        cntr' | cntr == cnt_max = 0
              | otherwise       = cntr + 1

        -- Toggle output
        out | cntr < duty  = False
            | cntr >= duty = True

        -- Set direction bits
        dira | d<0 = False
             | otherwise = True
        dirb = not dira
```

Listing 8: PWM code block. The duty cycle and token are inputs. The PWM frequency is hardcoded (25Khz in this example).

Encoder block

The next example is listed in Listing 11. An encoder is a device that measures position. In this case the example is used for a rotary encoder. Rotary encoders measure the rotation of a shaft. The other type of encoders are linear encoders which measure distance. An encoder can also be absolute or incremental, this example uses a incremental encoder. An incremental encoder measures change in position, but does not keep track of the absolute position.

Incremental encoders produce a set of pulses per revolution of the shaft. The encoder has two channels which are out of phase by 90 degrees. This quadrature setup allows to also measure

direction. The channels are usually denoted as *a* and *b*. This encoder block implements a type x4 encoder, it measures both the rising and the falling edges of the channels.

A helper function *x4encoding* is defined. This function uses the current and previous state of both the channels. All possible orders are defined, each order comes with an increment or a decrement of the counter.

The main function, *enc*, keeps track of the position, the previous states of the channels and handles the token. Since the encoder should always count, the token is in this case directly passed. The amount of pulses is always present on the output.

```
module ENCODER where

import CLaSH.Prelude

enc ti a b  = mealyB enc' (False,False,0,False) (ti,a,b)

enc' (prevA,prevB,position, token) (tokenIn, a,b) = (nextState, (tokenPass,Just position))
     where
          nextState = (prevA',prevB',position', token')

          -- Reset is not implemented
          reset = False

          -- New position
          position' | reset     = 0
                    | otherwise = position + x4encoding a prevA b prevB

          -- Previous values
          prevA' = a
          prevB' = b

          -- Directly pass token
          token' = tokenIn
          tokenPass = token

x4encoding True  False False False = -1
x4encoding False True  True  True  = -1
x4encoding True  False True  True  = 1
x4encoding False True  False False = 1
x4encoding True  True  True  False = -1
x4encoding True  True  False True  = 1
x4encoding False False True  False = 1
x4encoding False False False True  = -1
x4encoding _ _ _ _ = 0
```

Listing 9: Encoder code block. The token and, the left and right channel are inputs. The encoder uses x4 encoding.

A.5.2 PID Block

The PID block is listed in Listing 10. It is a simple PID controller using *SFixed 18 18* datatype. The demonstrator uses a PD controller, therefore the gain of the integrator in this controller is set to zero. This controller also contains conversions.

```
module PID where
import CLaSH.Prelude

pid ti count sp = mealyB pid' (0.0,0.0,False) (ti, count,sp)

pid':: ((SFixed 18 18),(SFixed 18 18), Bool) ->
        (Bool,Maybe (Signed 18),Maybe (SFixed 18 18)) ->
        (((SFixed 18 18),(SFixed 18 18),Bool),(Bool,Maybe (Signed 18)))
pid' (error, integral,token) (ti, count, spIn) = ((err',integral',token'),(to, duty))
       where
           -- Controller gains
           kp = 17.0
           ki = 0.0
           kd = 3.0
           -- Time step
           dt = 0.1

           -- Twopi
           twopi = 2*3.1415 :: SFixed 18 18

           -- Conversions
           convDuty = 5000.0/7.5 :: SFixed 18 18
           convRad = (1/2000.0)*twopi :: SFixed 18 18

           -- If count is Nothing set it to zero
           countm = maybe 0 id count
           sp = maybe 0 id spIn

           -- Convert (Signed 18) to (SFixed 18 18)
           countSF = sf d18 (shiftL ((resize countm) :: Signed 36) 18)

           -- Convert counted steps to radians
           measured = countSF*convRad

           -- Controller
           err' | token = sp - measured
                | otherwise = err

           integral' | token = integral + err'*dt
                     | otherwise = integral

           derivative = (err'-err) -- (err - previous_error)
           output = kp*err'+ki*integral+kd*derivative

           --Pass token immediately
           token' = ti
           to = ti

           -- Calculate duty cycle
           outputDuty = output*convDuty

           -- Convert (SFixed 18 18) to (Signed 18)
           duty = Just $ unSF (resizeF (shiftR outputDuty 18) :: SFixed 0 18)
```

Listing 10: PID code block.

A.5.3 Timer block

The Timer block is used control the speed of the control loop. The token is only released when cnt_max is reached.

```
module TIMER where

import CLaSH.Prelude

timer ti = mealy timer' (0::Unsigned 3) ti


timer' cntr ti = ((cntr'),out)
        where
            -- 1Khz
            cnt_max = 50000

            -- Set output
            out | cntr==0 = True
                | otherwise = False

            -- Increment counter
            cntr' | cntr == cnt_max = 0
                  | otherwise       = cntr + 1
```

Listing 11: PID code block.

A.5.4 Safety function

The *safety layer* block used in the test setup is listed in Listing 12. This safety layer is nothing more than a simple clamp on the input value.

```
module SAFETY where

import CLaSH.Prelude

safety ti vi = mealyB safety' (False) (ti,vi)

safety' (token) (ti,vi) = ((token'),(to,vo))
    where
        clamp = 1800 :: Signed 18

        token' = ti
        to = ti

        input = maybe 0 id vi

        output | input>clamp = clamp
               | input< -clamp = -clamp
               | otherwise = input

        vo = Just output
```

Listing 12: Simple safety layer block, the input value is clamped and passed to the output.

A.5.5 Setpoint block

The *set point* block is used in the setup to set the angle of the pendulum in radians. This set point switches between 0.6 Radians and −0.6 Radians.

```
module SETPOINT where

import CLaSH.Prelude

setpoint ti = mealyB pendulum' (False,0::Unsigned 26, 0.9 :: SFixed 18 18) ti

pendulum' (token,cnt, output) ti = ((token',cnt', output'),(to,vo))
    where
        -- 1Hz
        cnt_max = 50000000

        token' = ti
        to = ti

        -- Increase counter
        cnt' | cnt == cnt_max = 0
             | otherwise = cnt + 1

        output' | cnt == cnt_max = -output
                | otherwise = output
        --
        vo = Just output
```

Listing 13: Simple setpoint generator block, every second the sign of the output is changed.

A.6 Instrumentation code

A.6.1 GPMC block

```
gpmc_block data_gpmc addr nwe ncs nre =
           mealyB gpmc_block' (replicate d128 0, False) inputs
    where
      -- registers on input
      inputs = (data_gpmc',
                signal 0,
                addr',
                signal 0,
                nwe',
                ncs',
                nre',
                signal False)
      data_gpmc' = register 0 data_gpmc
      addr' = register 0 addr
      nwe' = register True nwe
      ncs' = register True ncs
      nre' = register True nre

gpmc_block' :: (Num a, Num b, Enum b, KnownNat n) =>
               (Vec n a, Bool) -> (a,a,b,b,Bool,Bool,Bool,Bool) ->
               ((Vec n a, Bool), (Bool, a, a))
gpmc_block' (mem, have_token)
            (data_gpmc, data_csp, addr_gpmc, addr_csp, nwe, ncs, nre,token) =
            ((mem_new, keep_token), (release_token, data_out_gpmc, data_out))
    where
        mem_new = if (not ncs)&&(not nwe) then
                    -- Save data from gpmc bus
                    replace addr_gpmc data_gpmc mem
                  else if have_token then
                    -- Save data from csp structure
                    replace addr_csp data_csp mem
                  else
                    mem

        data_out_gpmc = if (not ncs)&&(not nre) then
                          -- Read data
                          mem !! addr_gpmc
                        else
                          data_gpmc

        data_out = mem !! addr_csp

        keep_token = token
        release_token = have_token
```

A.6.2 GPMC VHDL Wrapper

```
1  use work.all;
2  library IEEE;
3  use IEEE.std_logic_1164.all;
4  use IEEE.numeric_std.all;
5  use work.CONVERTERS.all;
6
7  entity RAMSTIXTOP is
8    port(system1000      : in std_logic;
9         system1000_rstn : in std_logic;
10
11       -- GPMC
12       GPMC_ADDR       : in std_logic_vector(9 downto 0);
13       GPMC_DATA       : inout std_logic_vector(15 downto 0);
14       GPMC_nPWE       : in Boolean;
15       GPMC_nCS6       : in Boolean;
16       GPMC_FPGA_IRQ   : in std_logic;
17       GPMC_CLK        : in std_logic;
18       GPMC_nOE        : in Boolean;
19
20       -- Headers on RaMstix
21       F_IN0           : in Boolean;
22       F_IN1           : in Boolean;
23
24       F_OUT0          : out Boolean;
25       F_OUT1          : out Boolean;
26       F_OUT2          : out Boolean
27       );
28 end entity RAMSTIXTOP;
29
30 architecture structural of RAMSTIXTOP is
31   signal gpmc_data_out  : std_logic_vector(15 downto 0);
32   signal gpmc_data_tmp  : signed(15 downto 0);
33   signal data_enc  : std_logic_vector(18 downto 0);
34   signal data_sp   : std_logic_vector(18 downto 0);
35   signal data_pwm  : std_logic_vector(18 downto 0);
36 begin
37   ram : entity ram_top
38     port map
39       (gpmc_data_in        => signed(GPMC_DATA),
40        gpmc_addr_in        => unsigned(GPMC_ADDR),
41        gpmc_nwe            => GPMC_nPWE,
42        gpmc_ncs            => GPMC_nCS6,
43        gpmc_nre            => GPMC_nOE,
44        system1000          => system1000,
45        system1000_rstn     => system1000_rstn,
46        value_out           => gpmc_data_tmp,
47        int_pwm             => signed(data_pwm(15 downto 0)),
48                 int_enc             => signed(data_enc(15 downto 0)),
49                 int_sp              => signed(data_sp(15 downto 0))
50                 );
51
52   controller : top
53     port map
54       (system1000          => system1000,
55        system1000_rstn     => system1000_rstn,
56        F_IN0         => F_IN0,
57        F_IN1         => F_IN1,
58        F_OUT0        => F_OUT0,
59        F_OUT1        => F_OUT1,
60        F_OUT2        => F_OUT2,
61        ENC           => data_enc,
62            SP            => data_sp,
63            PWM           => data_pwm
64       );
65
66   gpmc_data_out <= std_logic_vector(gpmc_data_tmp);
67   GPMC_DATA <= gpmc_data_out when (GPMC_nCS6 = false AND GPMC_nOE = false) else (others => 'Z');
68 end;
```

Figure A.3: VHDL wrapper used in the Proof of concept.

B Additional appendices

In this Chapter additional Appendices are added. The first Appendix lists a set of requirements made in the project proposal. Appendix B.2 describes how to use the CλaSH code generation TERRA. Since, code generation is not completely finished, some of the generated code needs to be edited.

Appendix B.3 lists all the added plugins to the TERRA tool suite. Appendix B.4 describes how to add a new code generation plugin to TERRA to make future work easier.

B.1 Requirements

This Appendix lists a set of requirements that was made as part of the project proposal. The following list defines the requirements of the usage of CλaSH in a cyber-physical system, while still supporting the way of working facilitated in the TERRA tool suite. The requirements are prioritised using the MoSCoW method. Each requirement is marked in italics, followed by a explanation and an explanation of how the requirement was achieved during this project.

Requirement 1: *The CλaSH design support must implement the CSP model of computation*

> In the current way of working with the TERRA tool suite, CSP is used in the development of cyber-physical systems. The models in TERRA mainly consists of CSP. So, the implementation of these models should be able to execute CSP.

The base blocks of CSP are defined in a CλaSH CSP library. These base blocks can be connected as the application requires. Refer to Chapter 2 and Chapter 3.

Requirement 2: *The CλaSH design support must implement I/O on an FPGA*

> The main application of CλaSH in a cyber-physical system is I/O. There must be modules that implement several standard I/O devices, such as PWM generators and encoder readers. These modules should be connected to the CSP structure as defined in Requirement 1.

Several I/O blocks are defined and explained in Appendix A.5.1. All these blocks conform to the *UDB* block template defined in Chapter 3.

Requirement 3: *The CλaSH design support must be testable using test benches*

> The design support implementation should be testable by using test benches. CλaSH provides a method to generate test benches by defining a test input and an expected output.

The design support can be tested by defining a *testInput* and a *expectedOutput* as explained in Chapter 2 Section 3. For a more detailed explanation of how to use these tests refer to 6.

Requirement 4: *The CλaSH design support should be able to communicate with LUNA on a embedded processor*

> Since not all the layers in a cyber-physical system can be implemented in TERRA, some have to be implemented in LUNA. Therefore, there needs to be some kind of communication between the implementation in CλaSH and the LUNA execution framework.

This requirement is platform specific. In case of the RaMStix there is a GPMC connection between the FPGA and the embedded processor. An example GPMC implementation is displayed in Appendix A.5.1.

Requirement 5: *The CλaSH design support should not be dependent on one FPGA platform*

> Each cyber-physical system can be built around another hardware platform. This implementation requires the platform to contain a FPGA. The CλaSH design support should be implemented in such a way that it is easy to change or add new FPGA platforms.

All the design blocks described in this work are not platform specific. Every CλaSH description in can be used to generate a VHDL description.

Requirement 6: *The CλaSH design support should be able to be simulated at a regular pc*

The workflow in generating a hardware description from CλaSH takes quite some time. First VHDL is generated, this VHDL can be synthesised using a tool. Thereafter it can be programmed onto a FGPA. The CλaSH compiler provides an interpreter for CλaSH on the desktop. The CλaSH design support should be made in such a way that it still works with using simulation.

All the design blocks and generated code in this work can either be simulated using the GHC interpreter or after synthesis with Modelsim.

Requirement 7: *The CλaSH design support should be scalable*

This implementation should be able to be used in all kinds of applications. It should be small enough to fit on a FPGA with limited resources. It should also be extendable to work with big robots using lots of resources.

The component based approach makes the application scalable from small FPGA's with little logic elements to big FPGA's with lots of resources. The code generation makes interconnecting lots of base blocks simple.

Requirement 8: *The CλaSH design support should implement the option for user definable blocks*

LUNA has support for user definable blocks. The CλaSH design support could implement a similar feature. These blocks can be defined within TERRA and connected to CSP structures in a similar fashion as C++ blocks in LUNA. This way the end user can use the design support for their own application.

User definable blocks are supported and are described in Chapter 3.

Requirement 9: *The CλaSH design support could be generated by the TERRA tool*

The CSP models can be translated into CλaSH by hand at first. Later it should be possible to perform CλaSH generation in the TERRA tool. Code generation simplifies the workflow of designing cyber-physical systems. This way structures can easily be defined in TERRA an directly exported on an embedded target. There should be some way to differentiate between the LUNA part of the system and the CλaSH part. Also, there should be a way to define the communication in the TERRA tool as discussed in Requirement 4.

Basic code generation is supported and is described in Chapter 5.

Requirement 10: *The CλaSH design support could implement hard-realtime safety and control parts*

The next layer to implement is the safety layer. The safety checks bounds of outgoing signal and could also check for possible hazardous scenarios. This layer is also defined in CSP structures and can therefore be implemented by CλaSH.

The loop control layer controls the physical system. The layer requires hard real-time guarantees. This is a timing property that guarantees timing constraints are always met. When these conditions are not met the control loop calculations might not be finished in time, which is required to keep the control loop stable.

The challenge with loop control on the FPGA is the use of floating point numbers. CλaSH does not support floating point calculations. This would require a different solution, such as control using only integer or fixed point calculations.

Hard real-time safety and loop control is supported and is tested in a proof-of-concept setup. This is described in Chapter 6. Floating point numbers are avoided for the time being by using fixed-point calculations.

Requirement 11: *The CλaSH design support won't have debugging support at runtime*

Debugging according to Requirement 6 only covers pre-defined inputs. It does not incorporate the physical system. With debugging and tracing support on the FPGA target it is easier to find unexpected behaviour. This requires the output of debugging information and/or a trace using for example the connection with LUNA, see Requirement 4.

This would require an approach to limit the data traffic, because the higher granularity of the FPGA. It also requires to add extra instrumentation between the design support blocks. Although desirable, this requires a lot of time. So, this won't be implemented in this project.

B.2 Manual of the software

The following steps describe how to use the CλaSH TERRA plugins. After code is generated some of the generated code needs to be edited in order to make it work.

- First design the desired system in CSP blocks, use User Definable Blocks (see Chapter 3) for I/O.
- Next export the CSPm and check for deadlocks in FDR3 as one normally would in TERRA.
- Generate CλaSH code by selecting: Diagram → Code Generation → ClaSH. This will generate the CλaSH, the CSP library and the required Makefile in the directory named *clash/<project name>*.
- In the just generated code open the file named *<project name>.hs* in a text editor.
- Include the required Cλash files in the protected region above the topEntity, use the *ANN* notation to connect the inputs and output to pins, like so:

```
-- protected region user defined area for ann on begin --
import ENCODER
import PWM
import PID
import SETPOINT
import SAFETY
import TIMER

{-# ANN topEntity
  (defTop
    { t_name     = "top"
    , t_inputs   = ["F_IN0","F_IN1"]
    , t_outputs  = ["F_OUT0","F_OUT1","F_OUT2"]
    }) #-}
-- protected region user defined area for ann end --
```

- Edit the top entity function to your needs, and connect the input and output variables. Make sure the topEntity inputs and outputs conform the ANN description.

```
topEntity:: Signal (Bool,Bool) -> Signal (Bool, Bool, Bool)
topEntity input = output
                where
                     -- Set i/o
                     output = bundle (pwmOut, dira, dirb)
                     (au,bu) = unbundle input
```

- Edit function calls of UDB functions, add inputs and outputs if necessary.
- Use the protected region near the bottom for the required UDB functions

B.3 Overview of added plugins to TERRA

- *nl.utwente.ce.terra.csp.codegen.clash*

  Implements code generation. CSP → clash
- *nl.utwente.ce.terra.clash.model*

  Defines the model for the CλaSH code block.
- *nl.utwente.ce.terra.clash.editor*

  Implements the editor part of the CλaSH code block.

B.4 How to create a new plugin based on a model in TERRA

When creating a new code generation plugin based on a model in TERRA the following steps have to be taken. These steps are included in this report for future work.

- File → new → project
- Eclipse modeling framework → Empty EMF project
- Give a appropriate name.
- Create an *Ecore model* in the new project.
- Right click on the model in the Ecore editor and select “Load resources” to load the CSP and CPC models (if needed).
- Edit the *Ns prefix* and *Ns URI* in de properties.
- Create an *Genmodel* based on the previously designed Ecore model.
- Edit the root package accordingly.
- Reference the CPS and CPC models (if needed).
- Generate packages code by right clicking the genmodel in the editor.

Bibliography

Baaij, C., M. Kooijman, J. Kuper, A. Boeijink and M. Gerards (2010), CλaSH: Structural Descriptions of Synchronous Hardware using Haskell, in *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, IEEE Computer Society, pp. 714–721.
http://doc.utwente.nl/73124/

Bezemer, M. M. (2013), *Cyber-Physical Systems Software Development - way of working and tool suite*, Ph.D. thesis, University of Twente, doi:10.3990/1.9789036518796.

Bezemer, M. M., R. J. Wilterdink and J. F. Broenink (2011), Luna: Hard real-time, multi-threaded, csp-capable execution framework.

Brown, S. D., R. J. Francis, J. Rose and Z. G. Vranesic (2012), *Field-programmable gate arrays*, volume 180, Springer Science & Business Media.

Gibson-Robinson, T., P. Armstrong, A. Boulgakov and A. Roscoe (2014), FDR3 âĂŤ A Modern Refinement Checker for CSP, in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 187–201, doi:10.1007/978-3-642-54862-8_13.

Hoare, C. A. R. (1978), Communicating sequential processes, in *The origin of concurrent programming*, Springer, pp. 413–443.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice Hall International.
http://www.usingcsp.com/cspbook.pdf

Jones, S. P., C. Hall, K. Hammond, W. Partain and P. Wadler (1993), The Glasgow Haskell compiler: a technical overview, in *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, Citeseer.

Kuipers, F. P., R. Wester, J. Kuper and J. F. Broenink (2016), Mapping CSP Models to Hardware Using CλaSH.

T.G. Broenink, F.P. Kuipers, M. V. M. D. . J. B. (2015), Embedded Control Systems Implementation Manual.

Von Neumann, J. (1993), First Draft of a Report on the EDVAC, , no.4, pp. 27–75.