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,1], 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

[1]Corresponding Author: *Frits Kuipers, Robotics and Mechatronics, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.*Tel.: +31534894419; E-mail: `f.p.kuipers@student.utwente.nl`

MDD is designing a system that is first-time right, simulation before actual testing on hardware 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 Algorithm 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.

Algorithm 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.

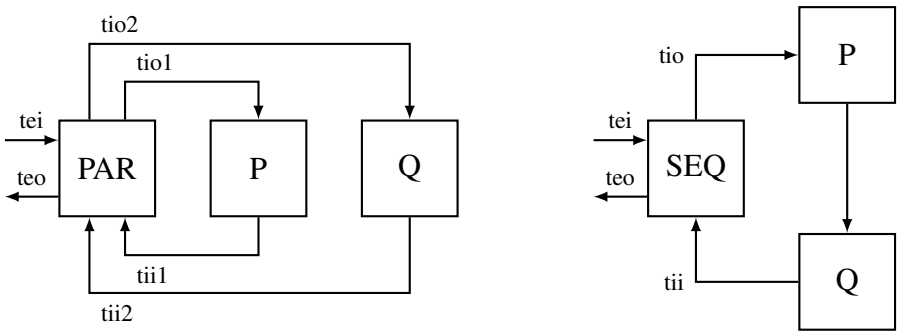

Figure 3. Data flow graphs of the parallel and sequential composition. Lines carry tokens. Processes are denoted as boxes.

The Haskell description of the parallel structure is shown in Algorithm 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 Algorithm 1. The function is transformed to a mealy machine in *parallel*.

As shown in Algorithm 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.

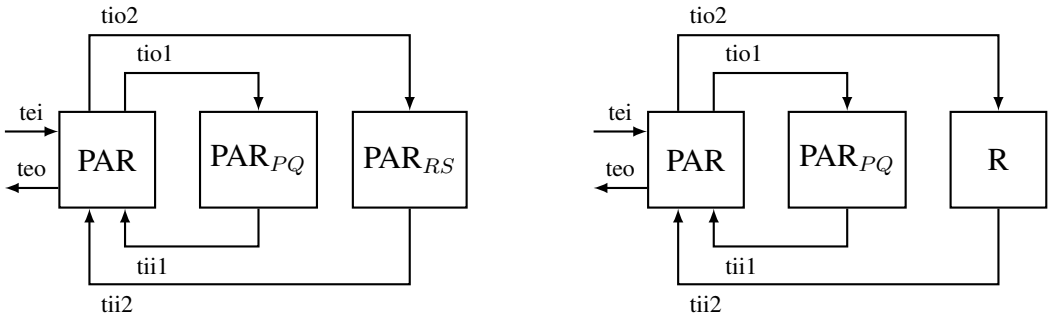

Figure 4. Three or more parallel CSP structures can be parallelised by using compositions of parallel structures and processes.

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.

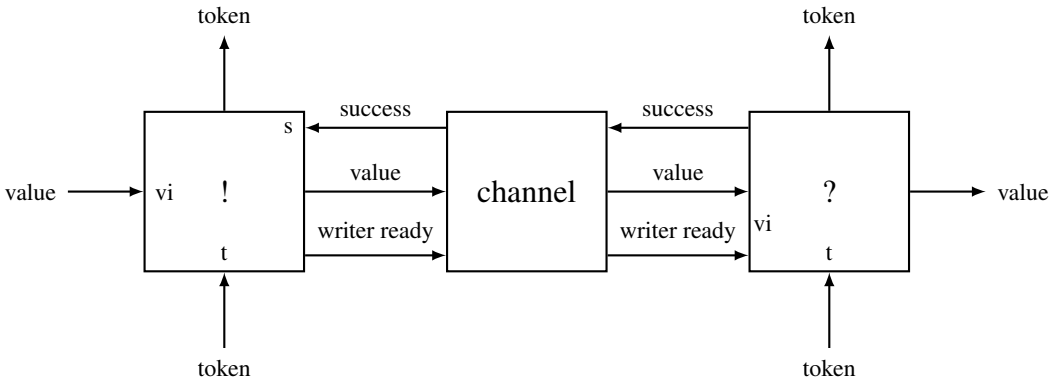

Figure 5. Channel communication and synchronisation.

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 Algorithms 5 and 6. Both are implemented using pattern matching and conform to the Mealy function structure (see Algorithm 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 Algorithm 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].

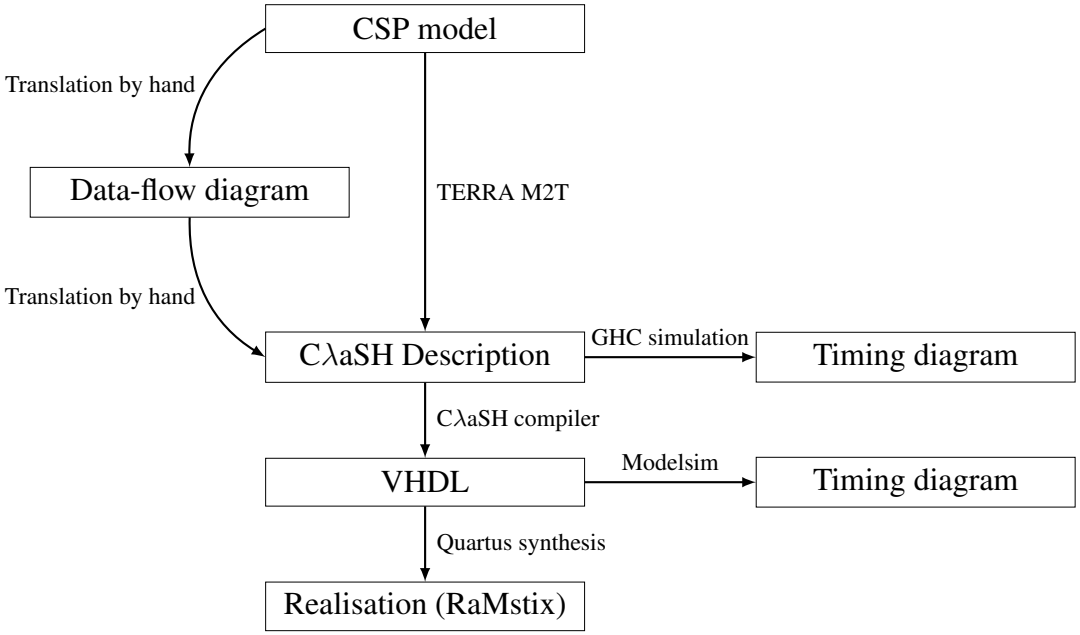

Figure 6. The current MDD work-flow from CSP models to hardware realization.

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 re-

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

quires some challenges to be overcome. For instance, most controllers require floating point operations, which are not (yet) supported in the CλaSH compiler.

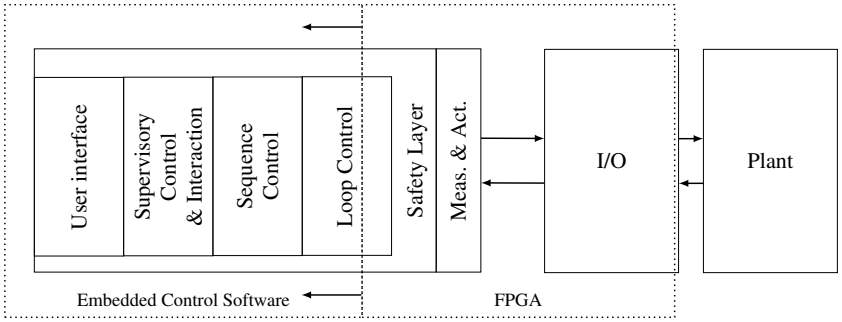

Figure 7. Use case of the CλaSH CSP mapping in embedded control.

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.

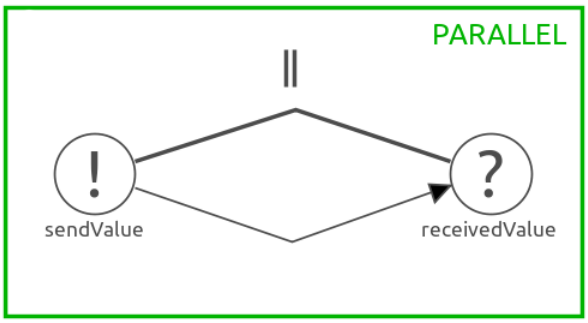

Figure 8. Producer consumer example. A writer and a reader in parallel relation connected by a channel.

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.

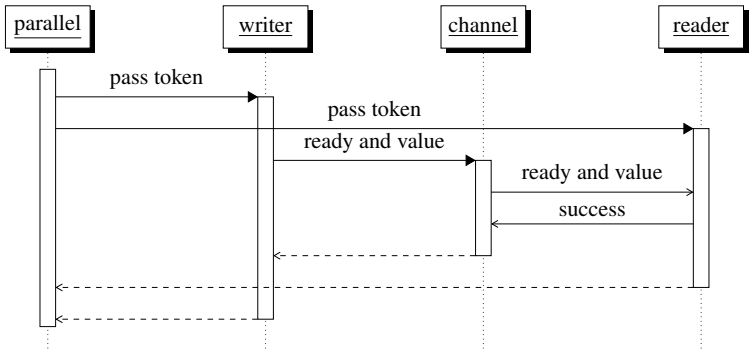

Figure 9. Sequence diagram of a producer-consumer example.

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.

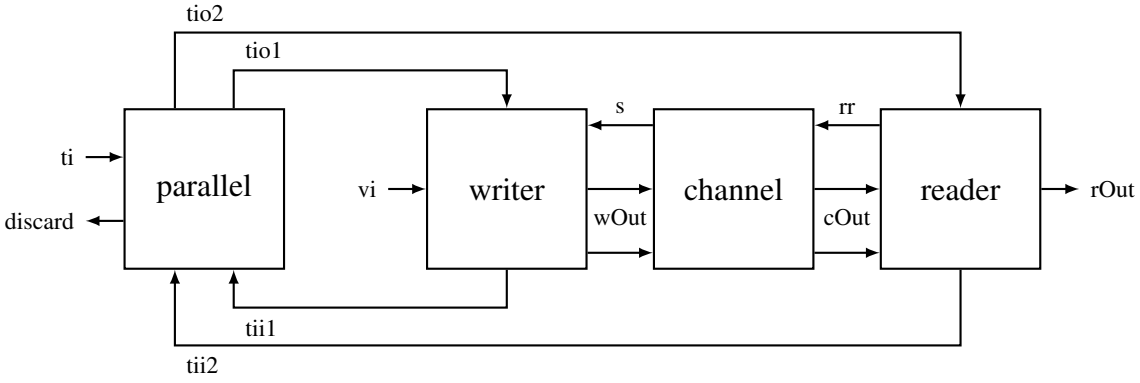

Figure 10. Data-flow diagram of the producer-consumer example.

The CλaSH code of the producer consumer example of Figure 10 is shown in Algorithm 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 Algorithm 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.

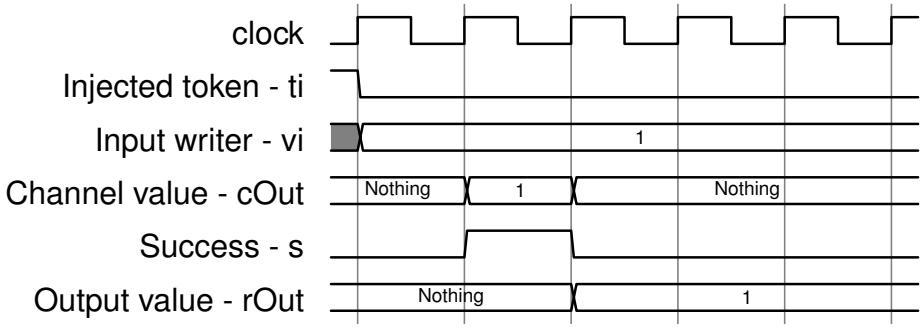

Figure 11. Timing diagram of the producer consumer example.

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.

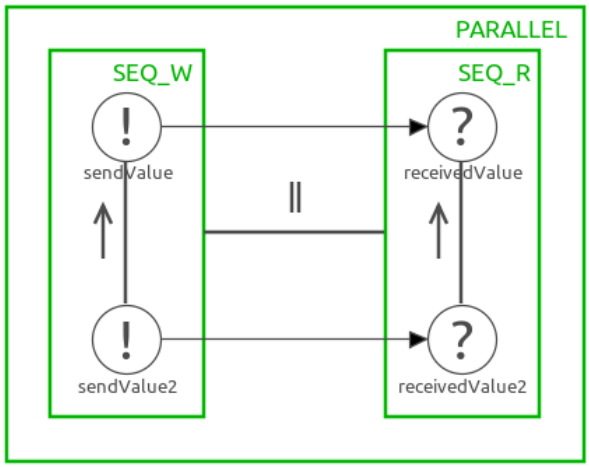

Figure 12. Multiple producer consumer example. Two writers sequential in parallel with two readers sequential communicating over separate channels. The orderings within the sequential constructs are indicated by the thick vertical arrows.

Algorithm 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*.

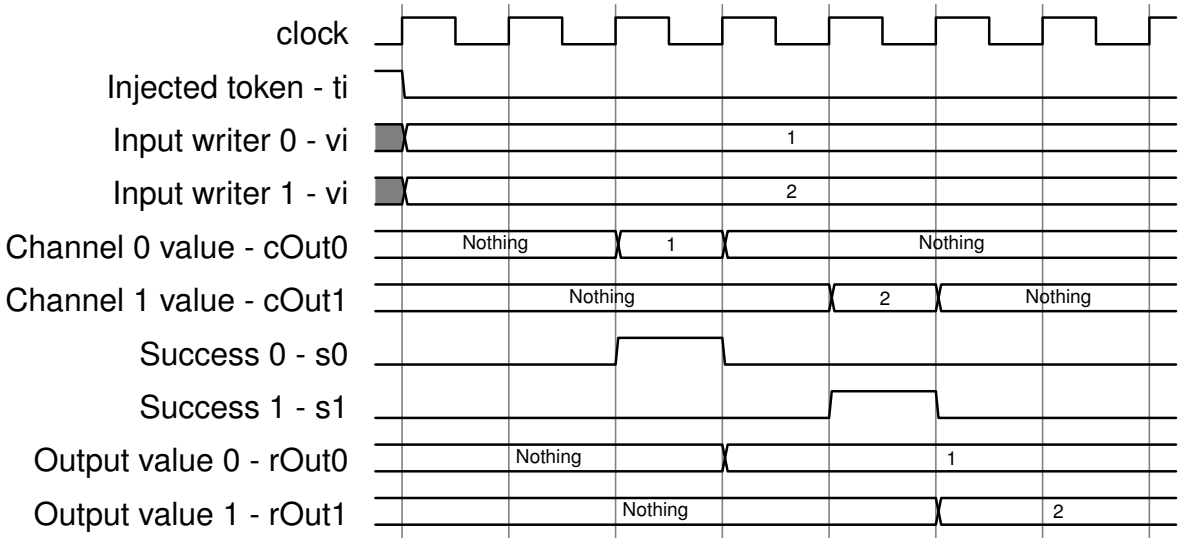

Figure 13. Timing diagram of the multiple producer consumer example.

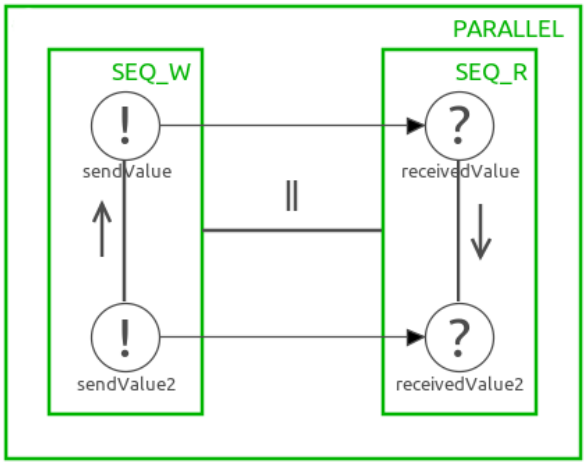

Figure 14. Multiple producer consumer example in a dead-locking configuration.

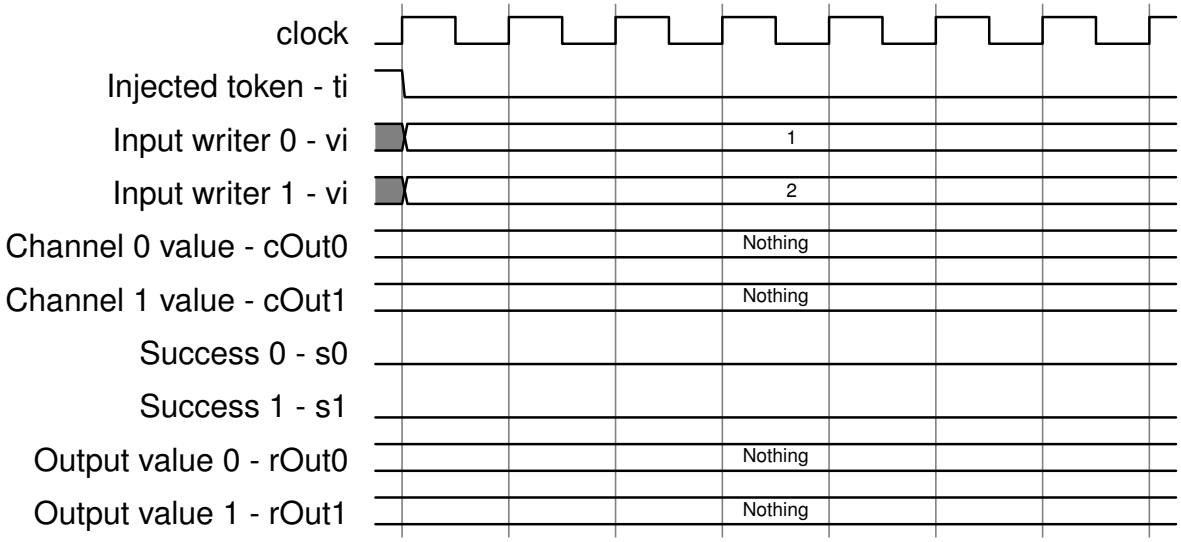

Figure 15. Timing diagram of the deadlocking multiple producer consumer example.

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 prioritised alternative, i.e. CλaSH code generation from TERRA diagrams is a matter of careful development. A non-prioritised alt will not be implemented since it is rarely used in physical applications.

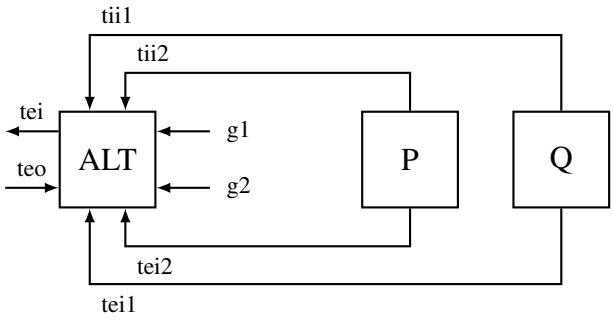

Figure 16. Data flow graph of the alternative composition. Lines carry tokens. Processes are denoted by the letters P and Q. The guards are denoted by g1 and g2.

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.