CSP with Synthesisable SystemC™ and OSSS

Claus Brunzema
OFFIS e.V. Research Institute
26121 Oldenburg

Wolfgang Nebel
Carl v. Ossietzky University
26111 Oldenburg

Abstract

C. Hoare's *Communicating Sequential Processes* (CSP) notation [12] to describe communication patterns of highly parallel systems is based on a well defined semantic and is accompanied by a wealth of research. This allows automatic checks for important properties like deadlock-freedom, livelock-freedom etc.. In order to obtain an executable system for simulation and later synthesis the algebraic CSP notation has to be translated and augmented with behaviour. There are several library implementations of CSP concepts for various programming languages. Other languages have built-in CSP elements or are based on CSP. This paper presents an implementation of CSP concepts in synthesisable SystemC[1]. The key features of CSP are available to the programmer allowing a high-level approach to system design. The CSP constructs are modelled after the CSP-based programming language occam[2], all the advanced C++ features like object-orientation etc. are still usable. By adhering to the *Oldenburg System Synthesis Subset* (OSSS, [10, 11]) the resulting designs will be set to be synthesised down to hardware automatically by the tools developed in the ICODES project [10]. An 3x3 image filter example is used to demonstrate the design flow from a CSP specification to an OSSS implementation.

1 Introduction

A classical approach to describe the communication in parallel systems is the CSP notation of C. Hoare [12]. There are methods and tools like FDR2 [4] that can prove various properties like deadlock/livelock-freedom of the specification. Since CSP is just a mathematical notation, the specification must be implemented in a programming language in order to arrive at an executable system for simulation and/or synthesis.

Several concurrent programming languages like Ada95 [8] and Handel-C [6] are more or less strictly based on CSP, the most prominent one being occam [5]. In order to ease the transformation of CSP notation in a SystemC [14] based design flow, we implement occam-inspired language constructs in SystemC. Key elements of CSP like channels, explicit parallel/sequential execution and choice are realised as synthesisable template classes, preprocessor macros or code transformations. By relying on OSSS for all implementation details the models will be synthesisable with the FOSSY synthesis tool developed in the ICODES project [10].

It is now possible to design a system using communication patterns described by algebraic CSP and to integrate these patterns seamlessly with additional behaviour in a synthesisable SystemC resp. OSSS model, see Figure 1.

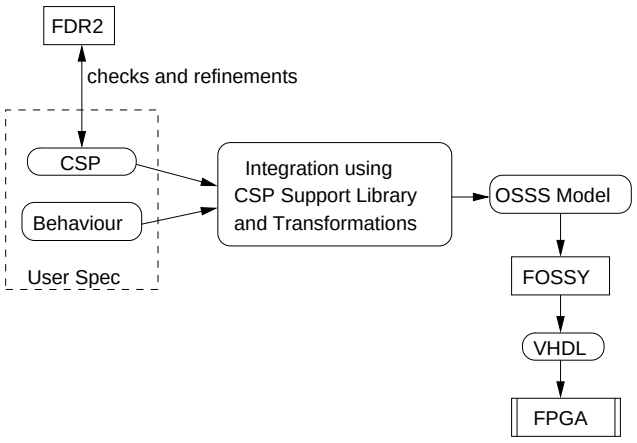

Figure 1: The Design Flow

The design flow using CSP elements in OSSS is demonstrated with an image filter example. The initial sketch is a graphical representation of processes connected with communication channels. The communication patterns on the channels are detailed with a CSP description. The CSP is then used for two implementations of the actual filter system. The occam implementation is the 'close to the source' reference model. It is not needed in the normal design flow, but the occam implementation is used here to test the design and to compare the details of the CSP constructs. The OSSS version can be simulated and is set for synthesis.

The rest of this paper is structured as follows: Section 2 gives a very short overview on the FOSSY synthesis tool that will be used to synthesise OSSS models. Section 3 describes how CSP concepts are available in various libraries and programming languages. Section 4 links CSP to the occam programming language, Section 5 describes the corresponding representations in OSSS. The image filter example is presented in Section 6, with implementations in occam and OSSS. Finally, Section 7 contains the conclusion to this paper and directions for future work.

[1] SystemC is a registered trademark of Open SystemC Initiative, Inc.

[2] occam is a registered trademark of ST Microelectronics

2 The FOSSY synthesis tool

The FOSSY synthesis tool transforms a subset of SystemC called OSSS into synthesisable VHDL [7]. Standard RTL synthesis tools can be used to transform the VHDL code into hardware. FOSSY is developed in the ICODES project. Since many high-level constructs like inheritance or object-orientation do not exist in standard VHDL, these constructs are lowered by a series of sophisticated parse tree transformations. The final version of FOSSY will be able to process the complete OSSS allowing synthesis of high-level OSSS channels [11]. Currently FOSSY is only able to convert OSSS without channels, but the OSSS software simulation library contains complete channel support. In order to implement CSP concepts in OSSS, we use the simulation library to test our designs. The final version of the FOSSY synthesis tool will be able to turn these designs into hardware automatically.

3 Programming with CSP

CSP describes systems with independent processes that interact only by synchronising on message-passing communication events. Algebraic operators are used to construct complex processes from simpler ones. CSP is not a programming language, it is mainly concerned with the communication between processes, not the internal behavior of the processes. A significant amount of the complexity of large parallel systems lies in the communication between the parts of the system. Thus using CSP to help describe a system is useful even if it does not capture all aspects of the system. The formal CSP can be automatically checked for properties like deadlock-freedom, livelock-freedom with tools like [4]. The CSP description is then used to implement the system with a programming language for simulation and synthesis.

There are several libraries for various programming languages that implement the semantics of CSP elements [3, 2, 1, 15, 9]. These libraries use concurrency features like threads and forking available in the host language to implement CSP processes. Message-passing over blocking point-to-point communication channels is usually implemented with hidden shared variables that are protected by a locking protocol. To model the internal behavior of the processes all features of the host language like object-orientation and/or (higher-order) functions are still available. Although the systems described using these libraries can be simulated by running the compiled programs, automatic synthesis to hardware is not possible.

Even the approaches targeted at SystemC [15, 9] are not intended to be synthesisable with standard SystemC synthesis tools.

Several programming languages have built-in support for CSP-inspired constructs. In Ada95 parallel processes are described with tasks, blocking point-to-point communication channels take the form of entries and accept statements. There is no widely available tool for synthesising Ada to hardware.

Handel-C is a C-based programming language designed for hardware synthesis. Parallelism is achieved with the `par` keyword, blocking point-to-point communication channels can be declared with the `chan` keyword. Since Handel-C is based on conventional C, it lacks modern programming language features like object-orientation or templates.

The programming language most closely following CSP ideas is occam. Every occam program is composed solely of processes that can only communicate via unidirectional blocking point-to-point channels[3]. There is ongoing work on an occam to FPGA compiler [16], but there is no industrial support for this design flow. occam has no support for object-orientation, function overloading or any template mechanism.

Our approach builds on the OSSS synthesisable SystemC subset. This allows the designer to use the advanced features of C++ in a model that can be synthesised and turned into hardware automatically with the tools developed in the ICODES project. In order to allow a smooth implementation of a design with CSP-specified communication patterns we show how to realise CSP constructs in OSSS. These constructs are modelled after occam, since occam represents CSP most clearly.

4 CSP and occam

There are many extensions and variants on the original CSP, we will use the basic syntax and semantics as described in [13].

CSP is described with twelve grammar elements, some of them have a more or less direct representation in occam, some are not really needed for real-world systems, see Table 1.

The primitive processes `STOP` rejecting any event and `SKIP` representing successful termination are directly available as keywords.

In occam all events must be transferred via channels (`CHAN`), synchronisation on events takes place by sending or receiving data on a channel. The event prefix turns into a blocking channel read or write operation denoted $c?x \rightarrow P(x)$ resp. $c!v \rightarrow P$, with c being the channel, x being a variable carrying a transmitted value into P and v being a value to be sent over c.

The `SEQ` construct corresponds to sequential process construction directly.

Both variants of parallel process execution are implemented with the `PAR` construct. If the processes want to synchronise on events (interface parallel), they need to install channels between them, when there is no synchronisation needed (interleaving parallel), the processes are just listed under the `PAR`.

The nondeterministic choice is not needed in real-world applications, since it would be redundant to sup-

[3] Actually, there are functions as well, but since those are not allowed to have any side-effects, one can always inline them completely.

CSP		occam
STOP	primitive process	STOP
SKIP	primitive process	SKIP
$e \rightarrow P$	event prefix	CHAN I/O, *see text*
$P;Q$	sequence	SEQ
$P\|\|Q$	interface parallel	PAR, *see text*
$P\|\|\|Q$	interleaving parallel	PAR, *see text*
$P \sqcap Q$	nondeterministic choice	not needed
$P \Box Q$	deterministic choice	ALT
$P\ e$	hiding	local CHANs
$f(P)$	relabelling	translator PROCs
name	process reference	PROC call
$\mu\ name \bullet P$	recursion	transform into iteration

Table 1: CSP and occam

ply two implementations of the same algorithm without external influence on which one is used.

The deterministic choice is realised with the ALT construct. Every ALT clause watches for input on a different channel, the one that receives data first will be executed.

When sub-processes are combined to form a new process, the events used between the sub-processes are not interesting to the outside world and can be hidden. Since occam always uses point-to-point channels to transmit events, there is no need to explicitly hide anything, the events are only visible to the processes connected by the channel anyway.

Relabelling is a transformation of the alphabet of accepted events for a process. This can be realised by instantiating transformation PROCs that map incoming data to the alphabet accepted by the original process resp. outgoing data to the alphabet needed by the environment.

All occam processes are named and can be referenced anywhere.

Recursion is not available in standard occam, but it can always be substituted by iteration.

5 CSP and OSSS

In order to use CSP in OSSS we use a library containing constructs modelled after their occam counterparts, see Table 2. The FOSSY synthesis tool accepting OSSS will be able to turn the design into actual hardware.

SC_CTHREADs are used to model occams processes.

Unidirectional blocking point-to-point channels are implemented with the help of the OSSS synthesis library [11]. The csp_channel template class contains transactors that transform read and write method calls on the connected osss_ports into a low-level signal protocol, see Figure 2.

User data is transmitted via the data signals, the OSSS library contains support for serialisation, so even

occam	OSSS
PROC	SC_CTHREAD
CHAN of *datatype*	csp_channel<*datatype*, *datawidth*>
STOP	while(true) wait();
SKIP	; (empty statement)
SEQ	SEQ-transformation
PAR	PAR-transformation
ALT	ALT and ALT_CLAUSE macros

Table 2: occam constructs in SystemC

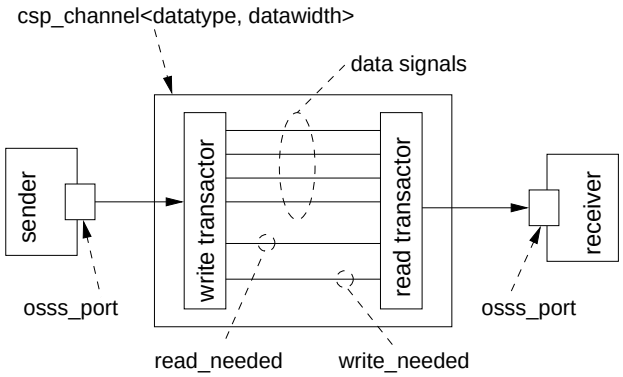

Figure 2: The csp_channel

large datatypes can be transmitted without requiring excessively many signals. The channels can be customised to use any number of signals for the internal data transfer. The additional read_needed and write_needed signals are used to implement the blocking behaviour. The programmer using csp_channels doesn't need to care about signals and serialisation, he just instantiates the channel template with the user datatype (and optionally datawidth) and binds to corresponding osss_ports.

The STOP process that doesn't take part in any communication resp. synchronisation is modelled with an infinite loop.

Successful termination denoted with SKIP needs no special handling in SystemC, a behaviour process just ends.

SEQ is implemented with a simple source code transformation. A wait() statement is inserted after each statement inside the SEQ-block. This forces a sequential schedule of the original statements.

The PAR construct requires a more sophisticated source transformation, since the parallel process invocations and the synchronisation at the end of the PAR must be written out separately.

The parallel execution of the processes (actually module methods with return type void) is realised by creating wrapper SC_CTHREADs. Those wrappers listen on an argument channel with a type that matches the formal parameter list of the wrapped method. When a complete set of arguments is received, the original method is called with these arguments. After the wrapped routine terminates, the wrapper sends an

acknowledge signal on an additional synchronisation channel.

The synchronisation channel is a special `csp_channel` that doesn't transmit any data, only the blocking behaviour is used to synchronize at the end of the `PAR` construct.

The `PAR` source transformation creates the wrappers, the argument and synchronisation channels together with the required ports. Every process call under a `PAR` construct is turned into a channel output operation that sends the arguments to the wrapper `SC_CTHREAD` corresponding to the process method. Additional synchronisation statements on all wrapper synchronisation channels are inserted at the end of the `PAR` (see Listing 3).

The result of this transformation is not entirely parallel, since the argument transmission at the start of the `PAR` and the synchronisation at the end of the `PAR` are still handled sequentially. But the wrapped methods run in parallel after the arguments are received.

In order to be able to implement occam's `ALT`, a process has to be able to listen to more than one input channel simultaneously, a nonblocking access method for the `csp_channel`s is needed. An additional `can_read()` boolean valued method is available at the reading end of a `csp_channel`. The two preprocessor macros `ALT` and `ALT_CLAUSE` hide the handling of the low-level `can_read()` from the programmer (see Listing 2).

6 Image Filter Example

To demonstrate the CSP-assisted design flow, we implement a simple streaming image filtering system (see Figure 3) that applies a 3x3 filter to a 256-level grayscale image.

For every column of the image there is a buffer process and a filter process. The buffer processes receive the original image pixels sent by the testbench process. Every buffer process stores three consecutive pixels of an image column. The filter processes gather packets of three pixels from three buffer processes, collecting a complete 3x3 matrix. The filter kernel is applied to the 3x3 matrix and the resulting pixel value is sent from the filter processes to the trace process. The trace process displays the filtered image.

Special care has to be taken at the edges of the image. Since every buffer process sends its data to three receivers, dummy data sinks are installed at the left and right edge of the filter system. These data sinks simply discard all data they receive. Similarly, every filter expects pixel data from three sources. Data sources sending an endless stream of zeros are installed at the left and right edge of the system.

The buffer processes have an additional command input channel, allowing commands to be sent to the system. A 'clear' command is used to clear the internal pixel memory of a buffer process.

CSP notation of the communications of the buffer, filter, zero source and dummy data sink processes are presented in Figure 4. The testbench and trace processes representing the external environment are omitted for brevity. Although it is possible to encode assignments, expressions and loops etc. in CSP, here the inner behaviour of the buffer and filter processes is not represented in the CSP formulas.

In this very simple example deadlock-freedom is already evident from the graphical representation, since there are no loops in the data flow. But for more complex models an automated check of the CSP notation will give more confidence than visual inspection.

$$
\begin{aligned}
&\texttt{BUFFER}_n = \\
&\quad (\texttt{cmd?cmd\_code} \rightarrow \texttt{BUFFER}_n) \\
&\qquad \Box \\
&\quad (\texttt{in?pixel} \rightarrow \\
&\qquad ((\texttt{buf\_to\_filter}_{n-1}\texttt{!threepixel} \rightarrow \texttt{SKIP}) \\
&\qquad\quad ||| \\
&\qquad (\texttt{buf\_to\_filter}_{n}\texttt{!threepixel} \rightarrow \texttt{SKIP}) \\
&\qquad\quad ||| \\
&\qquad (\texttt{buf\_to\_filter}_{n+1}\texttt{!threepixel} \rightarrow \texttt{SKIP})) \\
&\qquad\quad ; \\
&\quad \texttt{BUFFER}_n) \\
&\\
&\texttt{FILTER}_n = \\
&\quad ((\texttt{buf\_to\_filter}_{n-1}\texttt{?threepixel} \rightarrow \texttt{SKIP}) \\
&\qquad ||| \\
&\quad (\texttt{buf\_to\_filter}_{n}\texttt{?threepixel} \rightarrow \texttt{SKIP}) \\
&\qquad ||| \\
&\quad (\texttt{buf\_to\_filter}_{n+1}\texttt{?threepixel} \rightarrow \texttt{SKIP})) \\
&\qquad ; \\
&\quad \texttt{out}_n\texttt{!pixel} \rightarrow \texttt{FILTER}_n) \\
&\\
&\texttt{ZERO\_SOURCE\_L} = \\
&\quad \texttt{buf\_to\_filter}_0\texttt{!0} \rightarrow \texttt{ZERO\_SOURCE\_L} \\
&\\
&\texttt{ZERO\_SOURCE\_R} = \\
&\quad \texttt{buf\_to\_filter}_{width}\texttt{!0} \rightarrow \texttt{ZERO\_SOURCE\_R} \\
&\\
&\texttt{DATA\_SINK\_L} = \\
&\quad \texttt{buf\_to\_filter}_0\texttt{?dummy} \rightarrow \texttt{DATA\_SINK\_L} \\
&\\
&\texttt{DATA\_SINK\_R} = \\
&\quad \texttt{buf\_to\_filter}_{width}\texttt{?dummy} \rightarrow \texttt{DATA\_SINK\_R}
\end{aligned}
$$

Figure 4: CSP notation

6.1 occam Implementation

Expressing the elements of the image filter system in occam is straightforward. All processes are modelled with standard occam `PROC`s, all data transfer is done via occam channels.

The buffer process (Listing 1) uses the alternation construct (`ALT`) to simultaneously listen to the image pixel channel and the command channel. When a 'clear' command arrives, the internal pixel memory is cleared. When pixel input arrives, the pixels in the internal memory are shifted by one position. The oldest pixel value is discarded. The three pixels in the internal memory are sent to the three connected filters using the `PAR` construct. All procedures under a `PAR` are run in parallel. The `PAR` terminates after the last procedure terminates.

The filter process receives the three parts of the 3x3 filter region in parallel. The process then applies the

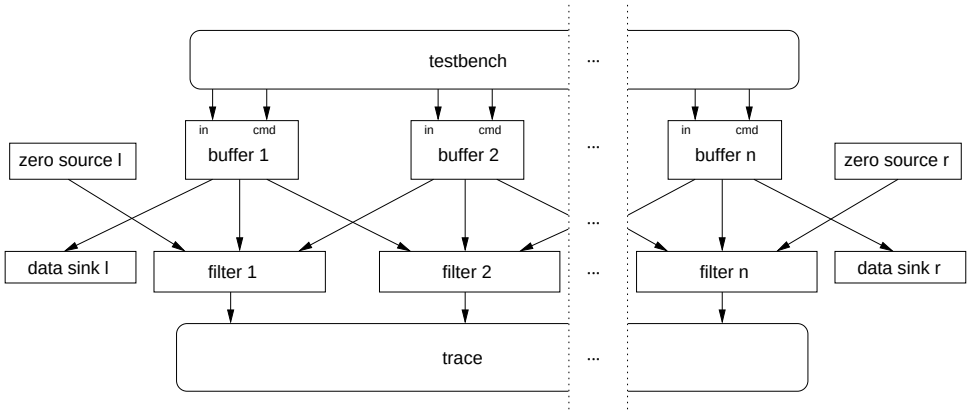

Figure 3: The image filtering system

3x3 kernel matrix and sends the resulting pixel value to the trace process.

Zero data sources and dummy data sinks are implemented with infinite loops around communication statements.

The occam main processes just sets up the work processes and connects the communication channels.

```
PROC buffer(CHAN OF BYTE in,
            CHAN OF BYTE cmd,
            CHAN OF BYTE3 buf.to.filter1,
            CHAN OF BYTE3 buf.to.filter2,
            CHAN OF BYTE3 buf.to.filter3)
  [3]BYTE buffer :   -- internal memory
  BYTE pixel :
  BYTE cmd.code :
  SEQ
    buffer := [ 0, 0, 0 ]
    WHILE TRUE
      ALT
        cmd ? cmd.code
          IF
            cmd.code = clear.cmd
              buffer := [ 0, 0, 0 ]
            TRUE
              SKIP

        in ? pixel
          SEQ
            -- shift in by parallel
            -- assignment
            buffer := [ buffer[1],
                        buffer[2],
                        pixel ]
            PAR
              buf.to.filter1 ! buffer
              buf.to.filter2 ! buffer
              buf.to.filter3 ! buffer
:
```

Listing 1: occam buffer process

6.2 OSSS Implementation

The OSSS implementation uses SC_CTHREADs to model all processes. csp_channels and osss_ports are used for the communication channels. The buffer process (Listing 2) uses the ALT and ALT_CLAUSE macros to listen to the command channel and the image data channel. The internal memory is cleared when a 'clear' command arrives on the command channel.

Image data is shifted into the internal memory by an overloaded shift operator.

In the buffer process the transmission of pixel data (original form commented out in Listing 2) to the three filter processes is handled by the three methods send1(), send2() and send3(). The PAR transformation must create a wrapper SC_CTHREAD for each of these methods (see Listing 3). Argument channels and ports (send1_wrapper_arg_{in|out}_port) together with synchronisation channels and ports (send1_wrapper_sync_{in|out}_port) must be added to the parent SC_MODULE. The original process invocations turn into sending the arguments to the wrappers (*port*->write()) and waiting for synchronisation (*port*->sync_req(), see Listing 2). At the time of this writing the PAR transformation must be performed manually.

The filter process receives the pixel data from three channels, calculates the result pixel value and sends it to the trace process. The vector operations and the element-wise matrix multiplication are implemented using the operator overloading mechanism of C++.

Zero data sources and dummy data sinks are implemented like their occam counterparts.

The simulation of this OSSS model yields the same result as the occam implementation.

7 Conclusions and Future Work

We have refined a simple example from an initial graphical sketch and informal description to an OSSS model that can be simulated and – when the FOSSY synthesis tool is completely implemented – synthesised to hardware. Formal CSP notation is used to describe communication between processes. This notation allows automatic checks for important properties of the system and guides the implementation of the design in OSSS. The building blocks of the initial description can be easily and straightforwardly realised with the new constructs presented in this work.

The next step toward a more streamlined design flow is making the SEQ-transformation (not shown in the example) and the PAR-transformation automatic. Since

```
void Buffer::buffer_proc()
{
 unsigned char pixel;
 unsigned char cmd_code;
 byte3 buf;

 buf[0] = buf[1] = buf[2] = 0;

 wait(); // for reset
 while(true)
 {
  ALT
  {
    ALT_CLAUSE(cmd, cmd_code)
    {
     if(cmd_code == CLEAR)
     {
      buf[0] = buf[1] = buf[2] = 0;
     }
    }
    ALT_CLAUSE(in, pixel)
    {
     // shift in with overloaded operator
     buf << pixel;

     {// original form:
      // PAR {
      //       send1(buf);
      //       send2(buf);
      //       send3(buf);
      //     }

      // transformed form:
      // first send the arguments
      send1_wrapper_arg_out_port->write(buf);
      send2_wrapper_arg_out_port->write(buf);
      send3_wrapper_arg_out_port->write(buf);

      // then request synchronisation
      send1_wrapper_sync_in_port->sync_req();
      send2_wrapper_sync_in_port->sync_req();
      send3_wrapper_sync_in_port->sync_req();
    }
   }
  }
 }
}
```

Listing 2: SystemC buffer process

FOSSY is built from parse tree transformations, adding `SEQ` and `PAR` support won't be difficult.

Alternatively, the implementation of occam's `PAR` could use SystemC's `SC_FORK/SC_JOIN/sc_spawn` features. However, these are no plans to extend FOSSY to support these yet.

More helper constructs should abstract further from the low-level SystemC details. For example, the first `wait()` statement in the process methods handling the reset condition should be hidden from the programmer, thus arriving at an untimed high-level design where all `wait()`s are in low-level library code.

To give the programmer more control over priorities in `ALT` blocks, this construct could be extended similar to occam's `PRI ALT`.

Consequently, this work is a first step towards a design flow that combines the simplicity and elegance of occam with the power of modern C++/SystemC/OSSS accompanied by verifiable CSP specifications.

```
void Buffer::send1_wrapper()
{
 byte3 arg;
 wait();       // for reset
 while(true)
 {
  arg = send1_wrapper_arg_in_port->read();
  send1(arg); // this is the original method
  send1_wrapper_sync_out_port->sync_ack();
 }
}
```

Listing 3: Wrapper process

References

[1] CSP (communicating sequential processes) support for Lisp. http://www.staff.ncl.ac.uk/roger.peppe/csp.tgz.

[2] CSP for Java™(JCSP). http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

[3] The Kent C++ CSP Library. http://www.twistedsquare.com/cppcsp/.

[4] *FDR2 User Manual*. Formal Systems (Europe) Ltd., 1992-2005.

[5] *occam 2.1 Reference Manual*. SGS-THOMSON Microelectronics Limited, 1995.

[6] *Handel-C Language Reference Manual*. Embedded Solutions Limited, 1998.

[7] *1076-2000 IEEE Standard VHDL Language Reference Manual*. IEEE Computer Society, 2000.

[8] John Barnes. *Programming in Ada95*. Addison-Wesley, 2nd edition, 1998.

[9] H. Fernando, P. Sánchez, and E. Villar. Modeling of csp, kpn and sr systems with systemc. In *Languages for system specification: Selected contributions on UML, systemC, system Verilog, mixed-signal systems, and property specification from FDL'03*, pages 133–148, Norwell, MA, USA, 2004. Kluwer Academic Publishers.

[10] Grabbe, Brunzema, Grüttner, Schubert, and Oppenheimer. Overview of the ICODES Project. In *Proceedings of the Ninth International Forum on Design Languages FDL 2006*, Forum on Specification and Design Languages, pages 309–310, Sep 2006.

[11] Grüttner, Grabbe, Oppenheimer, and Nebel. Modelling and Synthesis of Communication Using OSSS-Channels. In *Tagungsband: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 9. ITG/GI/GMM Workshop, February 2006.

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[13] Jeremy Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, 1996.

[14] OSCI. Open SystemC Initiative Homepage. http://www.systemc.org.

[15] Patel and Shukla. *SystemC Kernel Extensions for Heterogeneous System Modeling*. Springer, 2004.

[16] Roger M. A. Peel and Wong Han Feng Javier. Using CSP to Verify Aspects of an Occam-to-FPGA Compiler. In *Communicating Process Architectures 2004*, pages 339–352, sep 2004.