VHDL-based Communication- and Synchronization Synthesis

Wolfgang Ecker – Manfred Huber
Siemens AG
Corporate Research and Development
Munich, Germany, 81730

Abstract

This paper describes an approach for VHDL-based communication and synchronization synthesis. This design step transforms a system level VHDL description into an RT-level description. The idea is, not to synthesize system level implementations of communication and synchronization mechanisms but to perform the synthesis step as a mapping step of an abstract communication or synchronization mechanism to one of a set of RT-level implementations.

The major sub-problem, which needed to be solved for the synthesis algorithm was the topology dependent mapping of implementations.

1 Introduction

Today, automated HW synthesis starting from an RT-level VHDL description is well established. For system-level specification and design, however, other formal description techniques than VHDL have been considered. These include SDL in the telecom domain [2], Grapes [10], StateCharts for reactive systems [9], SpecCharts [18], Structured Petri Nets [3], or Concurrently Structured Flowgraphs (CSF) [16]. While specification methods for HW design or HW/SW codesign do considerably differ in their application domains, many of these methods provide a link to simulation, RT-level synthesis, or formal HW verification through appropriate VHDL interfaces. Examples are the SDL-to-VHDL compilers [12, 15], tools based on StateCharts [8], the SpecCharts-to-VHDL translator presented in [14], and the VHDL back-end of the CSF approach [17].

These specification methods certainly provide features that are not or not directly supported by VHDL. We believe, however, that it is important to use VHDL also for the early design activities covering system-level modeling, system-level simulation and analysis, system-level partitioning, synchronization and communication synthesis. Using VHDL at system level allows for the integration of these design activities with RT-level HW design in a unique environment. The most obvious advantages are early validation with existing components both on gate and on RT level, code re-usability, and re-usability of test frames and test vectors.

In order to avoid confusion with the many different associations to what is meant by a "system", the term "system-level specification" has to be considered in more detail. System-level specification is used here to indicate a high degree of abstraction with respect to data, functionality, and time.

The design cube (see Figures 1, 2) introduced in [5] identifies time abstraction as the most important criterion for classifying design levels. With respect to time, the current level of abstraction accepted by synthesis tools is the RT level (=clock-related). The causal level is a further consequent abstraction of time. We postulate that "system-level specification" corresponds to a specification on the next level of time abstraction which we call the level of (time) causality.

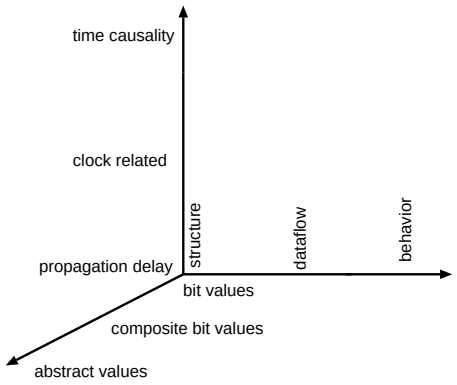

Figure 1: Coordinates of the Design Cube

An RT level (clock-related or clock-cycle based) description abstracts from propagation delays which in turn are specified as design constraints. In a similar way a specification at the causal level abstracts from a clock-cycle based view which in turn allows for the specification of clock-cycle based timing constraints (see [4]).

The timing specification at the causal level is based on communication and synchronization operations well known from operating systems. These operations specify synchronization points which provide a hook for clock-cycle based timing constraints.

We showed in [6] the applicability of VHDL at system level and the advantages in doing so. There, all transformations from system level to RT-level were performed manually. In this paper, we describe an approach towards the automation of these transfor-

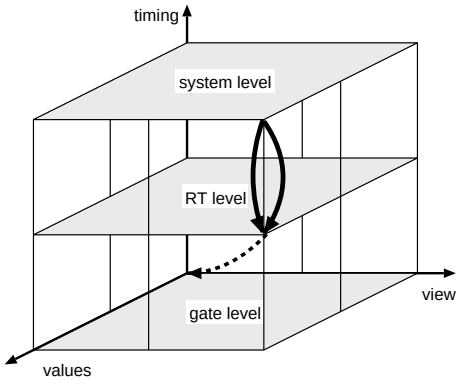

Figure 2: The Design Levels in the Design Cube

mations.

2 Overview

The key for synthesizing system level descriptions downto RT-level descriptions is the synthesis of causal communication and synchronization mechanisms to RT-mechanisms. We call this step *communication and synchronization synthesis* or *protocol synthesis*. One possibility to perform this design step is the synthesis of causal VHDL implementations of communication and synchronization mechanisms. We believe however, that this does not give enough freedom for different implementation alternatives, due to the fact that all currently available synthesis tools produce results, which strongly depend on their input description[1]. We focussed for this reason on another solution of the problem.

Protocol synthesis, as presented in this paper, is a mapping of an abstract protocol to a protocol, which can be selected out of a set of clock related implementations. Due to the fact that we propose a pure VHDL based design flow, this synthesis step is primarily a selection step of a possible implementation for synchronization and communication operations and the replacement of the abstract protocol by the selected protocol. The selection consists of a topology check of RT-implementation versus abstract implementation, a functional comparison and a heuristic considering area and timing. The replacement can be achieved by replacement of the abstract mechanisms type, objects and subroutines. All required tasks for communication and synchronization synthesis are described in the rest of the paper.

The next section discusses implementation details of abstract and concrete protocols and derives a set of requirements for the synthesis step from causal to RT-implementation differences. Section 4 describes the synthesis step with a main focus on the topology problem. A language for specifying synchronous protocols is also shown in this section. Afterwards details of the software implementation are presented.

[1] This effect can be observed in an excellent rate by current commercially available RT-level synthesis tools.

3 Communication- and Synchronization Mechanisms

This section classifies different implementations of abstract and RT-level communication and synchronization mechanisms.

3.1 Abstract Mechanisms

Currently semaphore types are implemented for synchronization (see [7]). The implementation includes different semaphore classes derived from semaphores without, with static and with dynamic priority as well as simple semaphores, set semaphores and multiple semaphores[2]. Nevertheless, we currently plan to implement other mechanisms like concurrently structured flow-graphs [17] to allow for simplification of system level descriptions.

For our synthesis approach this requires, that it may not be specialized to a set of operations. It must allow to flexibly extend the set of synthesizable mechanisms.

Communication channels for synchronized data exchange and global memory for unsynchronized data exchange are also implemented as abstract mechanisms (see [1]). They support 1x1, 1xN, Mx1 and MxN topologies (see figure 3) as well as unidirectional, bidirectional and master-slave data exchange directions. Currently, buffered channels are under construction.

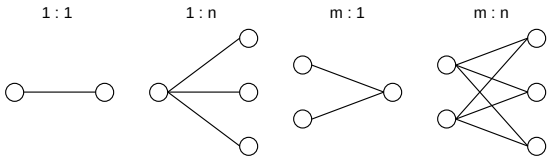

Figure 3: Regular Communication Topology

The implementation of synchronization of all abstract mechanisms is based on an abstract delta delay handshake[3]. An implementation example is shown in the following listing.

```
procedure send (
   constant data    : in    data_type;
   signal   channel : inout uni_channel_1x1 ) is
begin
   channel.source <= sender;
   channel.data   <= data;
   channel.req    <= TRUE;
   wait until channel.ack;
   channel.req    <= FALSE;
```

[2] It is important to note, that it can be proven, that semaphores can be used to describe all known synchronization problems.

[3] Delta delay is also called micro time. It describes the amount of time elapsing when executing one simulation cycle without simulation time advance.

```
    channel.source <= none;
    wait for 0 ns;
end send;
```

3.2 Clock-Related Mechanisms

Simple implementations of clock related synchronization and communication mechanisms use exactly the same handshake based implementation as abstract mechanisms. The implementation, however, is no longer delta delay but clock cycle based. The following listing shows a possible clock related implementation of the abstract send operation shown in the listing above.

```
procedure send (
    constant data    : in  data_type;
    signal   ch_data : out data_type;
    signal   ch_ok   : out ok_type;
    signal   ch_ack  : in  ack_type;
    signal   clk     : in  bit;
    signal   reset   : in  bit       ) is
begin
    ch_data <= data;
    ch_ok   <= ok;
    wait on clk until clk = '1' and ch_ack = ack;
    ch_ok   <= not_ok;
    wait until clk = '1';
end send;
```

It is important to note, that in this listing the wait-statements for abstract synchronization of listing shown in section 3.1

```
wait until channel.ack;
```

and

```
wait for 0 ns;
```

are replaced by

```
wait on clk until clk = '1' and ch_ack = ack;
```

and

```
wait until clk = '1';
```

This listing might probably be synthesized from the abstract description. But an abstract implementation may generally not be mapped onto its simple clock related implementation to obtain better results. So, different implementation alternatives like serial/parallel trade off [?], early, late or concurrent occurrence of hand-shake signals or partial removal of hand shake signals [11] should also be considered.

Implementations of abstract mechanisms can store values in the signal object by using value repetition in each write access. RT-level signals, however, are not able to store values. Thus the mapping of abstract mechanisms might additionally require the instantiation of a unit[4].

[4] Examples are units storing a semaphore value or the values of a FIFO.

Especially the need for a separate unit makes the synthesis of abstract mechanisms topology dependent, due to the fact that units might be implemented in a better manner for a special topology[5]

4 Synchronization and Communication Synthesis

4.1 Topology Problem

The causal implementation of communication mechanisms supports several parameterizable implementations (`1x1`, `1xN`, `Mx1`, `MxN`). Special topologies of implementations, however, need to be handled to get a good synthesis result.

Moreover, the supported specifications are restricted to a topology underlying a channel based communication. Only the number of senders and the number of receivers can be specified. But in general, communication or synchronization mechanisms should be supported, which probably might be added later, which consist of operations not named `send` and `receive` and which may support more than two operations.

Thus a general topology of an instantiation of a communication or synchronization mechanism specifies a set of operations, a set of processes executing operations and a relation describing which operation is executed by which process.

4.2 Synchronous Protocol Description Language

A special language, called SPDL[6], was developed for the specification of synchronous protocols and their topology. The language consists of a sequence of specifiers, each classifying one implementation. An example of a classifier is shown below:

```
uni_channel_1xn
{
    Type:       "Work.communicat.all";
    Signals:    "ch_data", "ch_ok", "ch_ack";
    Operations: "send"      -> "send1_1x1",
                "broadcast" -> "send1_1x1",
                "receive"   -> "receive1_1x1";
    Topology:   "(x00)(00x)";
    Examples:   "topology_1x1";
    Unit:       "";
}
```

The name `uni_channel_1xn` specifies the name, respectively type, of the abstract mechanism, which should be mapped. The information about one possible implementation of the abstract mechanism is specified inside the braces. Entry `Type` specifies a package

[5] Eg. a semaphore operation with one up and one down operation only can be implemented more effectively than a general solution. This is, because that solution requires one line for up and one line for down only. A general solution requires a pair of up/down lines for each process, operating on the semaphore. The special solution allows in this case to save two lines and some hardware driving this lines.

[6] Synchonous Protocol Description Language

which contains all required type and subroutine declarations for the implementation. All signals which need to be declared instead of the signal representing the abstract mechanism, are enumerated in the next entry, called `Signals`. Afterwards, the mapping of abstract operations to concrete operations is enumerated. The connection of operation to signals is performed by name convention, ie. the signals must have the same name as the parameters.

The topology of the Implementation is specified in the entry `Topology`. The topology is specified by a sequence of regular expressions. Each expression is included in brackets, describing how often an expression can be repeated.

()-Brackets require exactly one occurrence of the expression.

[]-Brackets allow that the expression can occur once or that the expression may be removed.

{}-Brackets allow that the expression may be repeated on occasion.

⟨⟩-Brackets have the same semantics as {}-Brackets with the difference that ⟨⟩-Brackets require the occurrence of the expression at least once.

Each expression describes a possible execution of one or a set of operations inside a process. The expression `x00` describes that inside one process the operation `send` is executed only. With a combination of brackets with the same expression inside, there can be described a fixed number or a fixed range of processes. The complete topology specification in the example allows a topology consisting of two processes, where one process executes a `send` operation and the other process executes a `receive` operation.

`Examples` and `Unit` specify an example of the mapping and the name of a unit, which is additionally required for the clock related implementation. The empty string in this case signals, that no additional unit is needed.

We currently add the entries `Timing`, `Time` and `Area`. The last two entries should contain a value representing an abstract time and area value. This value should be used by a simple heuristic for time and area driven protocol synthesis. The entry `Timing` is reserved for later extensions, considering causal time relations in the execution of different operations.

4.3 Synthesis Steps

Communication and synchronization synthesis is performed in several steps. First, an SPDL-File and a VHDL design entity to be synthesized are read. The second step identifies all processes together with their process identification, which must be specified by the user via a VHDL attribute.

The next step looks for all signals representing abstract communication or synchronization mechanisms[7]. The topology of each identified signal is then extracted. A possible implementation is selected according the SPDL-File and extracted topology. The user is interactively asked for resolution, if more than one implementation is possible. This will be replaced in the next version of the tool by a simple heuristic considering area and timing.

The major transformation is then performed according to the selection. First all signals representing the abstract mechanism are replaced by a number of signals, required for the selected implementation. In this step, the type of the signals must meet the requirements of the protocol[8].

The implementation of a protocol can be sensitive to the order of signals assigned to a composite type parameter of an operation. Moreover, signals are generated only for that number of processes, that participate in communication or synchronization. Thus a mapping of the process identification (in figure 4: `Pid`) of a process to the index of the vector signal assigned to an operation executed by that process must be established. This mapping is not performed by the synthesis algorithm directly. A VHDL mapping table is generated for this reason and included as constant (in figure 4: `AVM`) in the design entity. Using this table, the processes do not index their line directly. The index is transformed by an array access to the generated mapping table (in figure 4: `AL(AVM(Pid))`).

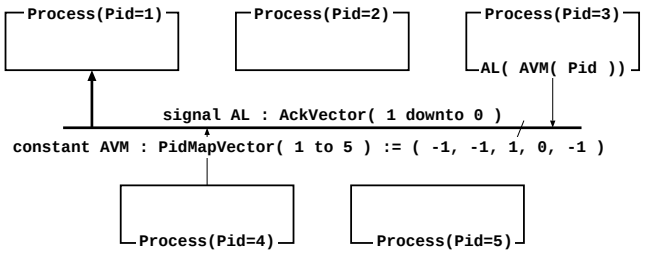

Figure 4: Mapping to an Acknowledge Vector

To allow for clock related implementations of communication and synchronization mechanisms a clock signal must be visible. A reset signal is mostly also required for initialization of hardware. Both a clock and a reset signal are included in the interface of the design entity in the subsequent step.

Afterwards all subroutine calls activating abstract protocol operations are replaced by subroutine calls activating an implementation of the abstract protocol. The mapping of the subroutines is taken from the SPDL-File, too. The replacement includes also the re-mapping of the interface signals.

If a unit is required to satisfy the behavior of the RT-implementation, A unit is included also in the synthesis step before last.

[7] The signal types, which correlate with an abstract mechanism can be extracted from the SPDL-File.

[8] Eg. The acknowledge signal of a communication mechanism allowing more than one receiver must be a one dimensional array type of the basic acknowledge type.

Finally the name(s) of the synthesized design entity is (are) modified, Use-clauses referring to VHDL implementations of protocols are replaced or added, and the modified VHDL design entity is written to the database.

4.4 Software Implementation

The implementation of the synthesis tool is based on the VTIP of COMPASS. The general design flow is shown in figure 5.

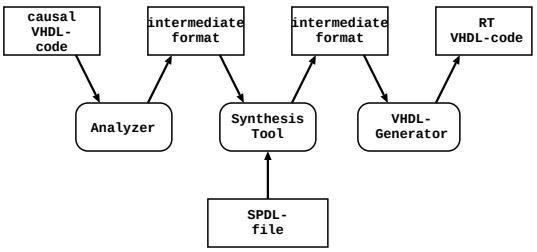

Figure 5: Software Implementation

First, a causal level VHDL description is compiled by the analyzer in a tool specific intermediate format. The synthesis tool, which consists of 1685 lines of C-code, modifies the intermediate via a procedural interface called SPI, based on the protocol specifications read from the SPDL file. Finally, a VHDL generator writes the modified intermediate and generates in this way RT-level VHDL code.

5 Conclusion and Outlook

The implementation of a tool which performs VHDL based communication and synchronization synthesis was presented in this paper. The tool shows that the automatic transformation of system level VHDL to RT-level VHDL is possible. We generated a set of VHDL models (a small test example is shown in figures 6 and 7) by the tool and synthesized the result with commercial available RT-level synthesis tools to demonstrate that an automatic way from system level VHDL downto netlist exists.

```
use work.communication.all;

entity chan1 is
    port (Kanal : inout uni_channel_1x1);
end chan1;



use work.communicat.all;

entity ent is
    port (Kanal_data: in data_type := -2147483648;
          Kanal_ok: in ok_type;
          Kanal_ack: out ack_type;
          clk: BIT;
          reset: BIT);
end ent;
```

Figure 6: Synthesis Example (1)

Future work will concentrate on the insertion of wait-statements for the clock related specification of time between the communication and synchronization mechanisms and on automatic insertion of a reset mechanism. Further research will lie on causal level optimization like channel sharing or causal level operation optimization.

```
architecture senden of chan1 is
    signal channel : uni_channel_1x1;
begin
    process
        variable daten : data_type := 10;
    begin
        receive (daten, Kanal);
        send (daten, channel);
        wait;
    end process prozess1;
end senden;


architecture arch of ent is
    signal data: data_type := 0;
    signal channel_data: data_type := -2147483648;
    signal channel_ok: ok_type;
    signal channel_ack: ack_type;
begin
    process
        variable daten: data_type := 10;
    begin
        receive_1x1(daten, Kanal_data, Kanal_ok, Kanal_ack, clk, reset);
        send_1x1(daten, channel_data, channel_ok, channel_ack, clk, reset);
        wait;
    end process;
end arch;
```

Figure 7: Synthesis Example (2)

Acknowledgements

We would like to thank Michael Münch, Sabine Rössel and Bernd Wurth, for their support and for helpful discussions.

References

[1] M. Bauer and W. Ecker. Communication Mechanisms for the Specification and Design of Hardware starting at Higher Levels. In Proceedings of the Spring '93 Meeting of the VHDL-Forum for CAD in Europe, 1993.

[2] CCITT. Functional Specification and Description Language (SDL), Recommendations Z.100-Z.103, Blue Book, October 1989.

[3] L.A. Cherkasova and V.E. Kotov. Structured Nets. In Proceedings of MCSF. Springer LNCS 118, 1981.

[4] W. Ecker. Specification of Timing Constraints in the Design Process. In Proceedings of the Spring '92 Meeting of the VHDL-Forum for CAD in Europe, pages 175–183, 1992.

[5] W. Ecker and M. Hofmeister. The Design Cube–A New Model for VHDL Designflow Representation. In Proceedings of the EURO-VHDL, pages 752–757, 1992.

[6] W. Ecker and S. März. System-Level Specification & Design Using VHDL: A Case Study. In CHDL 93 - Computer Hardware Description Languages and their Application, pages 505–522, Ottawa, Canada, April 1993.

[7] W. Ecker and A. Scheuer. Semaphores in HW-Design. In Proceedings of the Spring '92 Meeting of the VHDL-Forum for CAD in Europe, pages 137–153, 1992.

[8] D. Harel. StateCharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8:231–273, 1987.

[9] D. Harel. On Visual Formalisms. Communications of the ACM, 31(5), May 1988.

[10] G. Held. Sprachbeschreibung GRAPES. Siemens AG, 1990.

[11] D.C. Ku, C. Coelho, and G. De Micheli. Interface Optimization for Concurrent Systems under Timing Constraints using Interface Matching. In High Level Synthesis Workshop 92, pages 202–213, 1992.

[12] B. Lutter, W. Glunz, and F.J. Rammig. Using VHDL for Simulation of SDL Specifications. In Proceedings of the EURO-VHDL, 1992.

[13] S. Narayan and D.D. Gajski. Synthesis of System-Level Bus Interfaces. In Proceedings of the European Design Automation Conference (EDAC), 1993.

[14] S. Narayan, F. Vahid, and D.D. Gajski. Translating System Specifications to VHDL. In Proceedings of the European Design Automation Conference (EDAC), pages 390–393, February 1991.

[15] O. Pulkkinen and K. Kronlöf. Integration of SDL and VHDL for High-Level Digital Design. In Proceedings of the EURO-VHDL, 1992.

[16] M.T.L. Schäfer. Architekturentwurf für nebenläufige Systeme mit Verifikation der Funktionssicherheit. Siemens internal Report.

[17] M.T.L. Schäfer and W.U. Klein. Correctness Verification of Concurrent Controller Specifications. In Proceedings of the EURO-DAC'92/EURO-VHDL'92, pages 706–712, 1992.

[18] F. Vahid, S. Narayan, and D.D. Gajski. SpecCharts : A Language for System Level Synthesis. In Proceedings of the IFIP Tenth International Symposium on Computer Hardware Description Languages and their Applications, pages 135–153, April 1991.