A Formal Approach to Specify and Synthesize at the System Level*

Christian Blumenröhr

Institute for Circuit Design and Fault Tolerance (Prof. Dr.-Ing. D. Schmid),
University of Karlsruhe, Germany e–mail: blumen@ira.uka.de

Abstract

In this paper, a new and formal methodology for specifying and synthesizing systems is presented. Systems are modeled as structures of concurrent processes. The way the processes communicate realizes a hand-shake protocol. The specification at the system level is part of our hardware description language Gropius, which ranges from the gate to the system level. Gropius was designed for a formal synthesis scenario, where synthesis is performed by applying basic mathematical rules within a theorem prover, thus guaranteeing correctness of designs implicitly.

1 Introduction

The synthesis of hardware systems is heading towards more and more abstract design levels. At the algorithmic level, the specification for a single process is given as a piece of software program. For modeling complex systems, it is often not sufficient to start from a single process specification. At the system level, therefore, systems are modeled as a structure of concurrent processes, each of them represented by a piece of software program. Starting from this abstraction level, the system need not necessarily be realized in hardware. The descriptions at the system level can as well be a starting point for a pure software or a mixed hardware/software implementation (HW/SW-codesign, embedded systems). However, depending on whether the processes are to be implemented as software or hardware, different cost functions have to be considered during the synthesis and optimization process [1]. In this paper, we restrict ourselves on the synthesis of hardware components.

Due to the fact that the complexity of today's systems is increasing as well as the synthesis process for deriving them, the correctness of hardware and software components has become an important matter — especially in safety critical domains. Since simulation is normally (i.e. for large designs) not exhaustive in reasonable time and an automated

*This work has been partly financed by the Deutsche Forschungsgemeinschaft, Project SCHM 623/6-1.

post-synthesis verification is extremely costly, it is our objective to perform synthesis via logical transformations thus guaranteeing "correctness by construction".

There are mainly two concepts to fulfill this paradigm: *transformational design* and *formal synthesis*. In both approaches, synthesis is based on correctness-preserving transformations. To guarantee this, the correctness of the transformations is proven. However, the transformations are implemented within a complex synthesis program. In this step, many failures may occur. Therefore, it would be necessary to prove the correctness of the implementations of the transformations. However, such proofs are not provided in transformational design [2, 3, 4].

In formal synthesis, the synthesis process is performed within a theorem prover. The circuit descriptions are formalized in a mathematical manner and the transformations, which are derived from elementary mathematical rules, are directly performed in this representation style within the theorem prover. Thus, not only the correctness of the transformations is proven, but also the correctness of their implementations is guaranteed. In our approach we use the theorem prover HOL [5]. Due to the fact, that deriving theorems in HOL is restricted to a small core of rules and axioms, our approach is extremely safe as to correctness. Other approaches in the formal synthesis domain are restricted to lower levels of abstraction [6, 7, 8] or to pure data flow graphs at the algorithmic level [9]. See [10] for a survey on formal synthesis.

In this paper, we address formal synthesis at the system level. It extends our recent work, which was dedicated to synthesis of pure data flow graphs [11, 12] and mixed control/data flow graphs [13, 14] at the algorithmic level. In the next section, we introduce, how systems can be specified in our approach and in section 3 it is shown, how systems can be mapped on implementations at the RT-level.

2 Specifying systems

Our work is based on a formal hardware description language called Gropius[1] ranging from the gate level to the system level.

Gropius is functional, strongly-typed, polymorphic, higher-order and has a precise semantics, since every construct has been defined in the theorem prover HOL [5].

Unlike most standard hardware description languages such as VHDL and Verilog, Gropius is strictly divided into sublanguages each corresponding to a specific abstraction level. However, due to the common core of the sublanguages, Gropius is not just a set of hardware description languages. The elementary constructs and the user defined functions are a common starting point for all circuit descriptions. Figure 1 shows the structure of Gropius. In the following, only the part of figure 1 that corresponds to the system level (Gropius-3) is introduced. For the description of Gropius-0 (gate-level) and Gropius-1 (RT-level) see [15], for Gropius-2 (algorithmic level) see [13, 14].

At the algorithmic level, only single processes can be modeled. To describe more

[1]WALTER GROPIUS (1883-1969), founder of the BAUHAUS (*form follows function*).

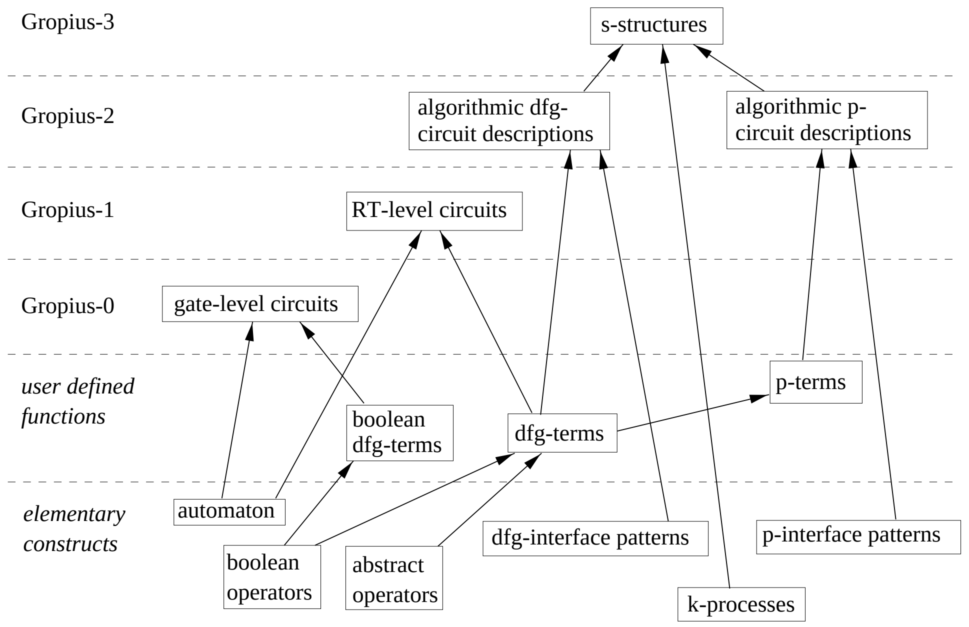


Figure 1: The language Gropius

complex systems, this is not sufficient. Therefore, at the system level, a circuit is described by a set of processes.

2.1 Algorithmic circuit descriptions

The description of algorithmic circuits at the system level as well at the algorithmic level consists of two components. An algorithmic description only defines, how some input value is mapped to some output value. Time is not yet considered. In [13, 14] the algorithmic description of processes is introduced. To bridge the gap between an algorithmic description and a hardware implementation one has to additionally define an interface behavior. The interface behavior describes, how the circuit communicates with the environment via its interface.

In most approaches of the high-level and system level synthesis domain, algorithmic and interface descriptions are mingled in an arbitrary manner. In our approach, algorithmic description and interface behavior are strictly separated.

At the algorithmic level a fixed set of interface patterns is provided. The circuit designer can first write some ordinary, time independent algorithm and can then select one of the interface patterns, thus defining the way the circuit communicates with the environment. The designer can easily switch from one interface behavior to another without changing the program.

This is what we call “true abstraction”. The design starts with some general purpose program just as in software design. At this point, the design is not at all hardware specific: no signals, no timing, no clock, no registers. The designer may then switch from software to hardware by selecting an interface pattern, but he may as well remain in the software domain and have his piece of code being executed on a computer.

The syntax for algorithmic circuit descriptions (see figure 1) is as follows:

algorithmic dfg-circuit description ::=
dfg-interface pattern "(" *dfg-term* "," *number of cycles* ")"
algorithmic p-circuit description ::=
p-interface pattern "(" *program* ")"

The algorithm given as a dfg-term or a p-term is handled to the interface pattern as a parameter. There is one major difference between the hardware implementation of dfg-terms and p-terms. Dfg-terms correspond to acyclic data flow graphs and can be executed in a fixed number of clock ticks. P-terms on the other hand correspond to arbitrary computable programs and therefore one cannot guarantee that they are executed in a fixed number of clock ticks and it may even be, that execution does not terminate at all. Dfg-based algorithmic circuit descriptions have therefore an extra parameter specifying the number of clock cycles for executing the dfg-term. This parameter may be given by the designer or the designer may use a variable which is then instantiated during synthesis.

2.2 Communication scheme at the system level

At the system level processes interact via a fixed communication scheme. There are only two interface patterns at the system level: one for dfg-terms and others for p-terms. The communication scheme is label based and closely related to higher order petri nets [16]. Every process corresponds to a transition with some places at its output. In Gropius-3, firing means removing the input labels, performing some calculation and delivering the produced result in terms of new labels to the output.

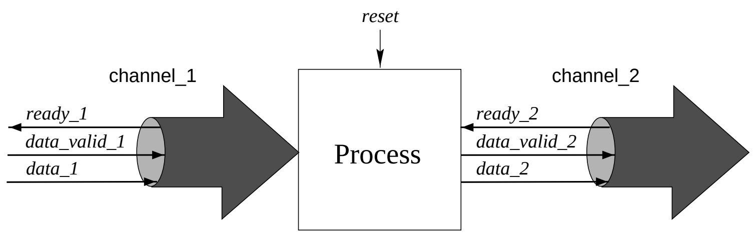


Figure 2: Interface of a single process

Circuits communicate via channels (see figure 2). A process may have several input and output channels. Furthermore, every process has a global *reset*-input. Channels are used to transport some label with some specific data package from some circuit A to some circuit B. Each channel consists of two boolean control signals *ready* and *data_valid* and a data signal *data* of arbitrary type. Channels are always one-to-one connections and they always have a fixed direction, which is defined by the direction of the *data*-signal. The *data_valid*-signal goes to the same direction whereas the *ready*-signal goes to the opposite direction.

At the system level the designer will always describe structures of circuits communicating via channels, but due to the communication scheme it is not possible to explicitly address the basic signals belonging to some channel.

In channels communication is performed via handshake. Given some processes A and B and a channel from A to B. Process A signalizes via *data_valid* that there is a label with some data being on *data*. Process B signalizes via *ready* that it is ready to read the next label and that it will read the next label as soon as process A will provide it. Whenever both *data_valid* and *ready* become true, the communication happens, i.e. the label is moved from A to B.

Figure 3 gives the formal definition of the p-term based interface pattern that is defined for the system level. Figure 4 illustrates it in a schematic manner.

$$
\begin{array}{l}
\mathsf{P\_INTERFACE\_SYSTEM}\ (A)\ (\mathit{reset},(\mathit{data\_1}, \mathit{data\_valid\_1}, \mathit{ready\_1}), (\mathit{data\_2}, \mathit{data\_valid\_2}, \mathit{ready\_2}))\ =\\
\quad \mathit{ready\_1}\ 0\quad \wedge\\
\quad \forall t.\ \mathit{reset}\ t\ \Rightarrow\ (\mathit{ready\_1}\ (t+1)\ \wedge\ \neg(\mathit{data\_valid\_2}\ t))\quad \wedge\\
\qquad (\mathit{ready\_1}\ t\ \wedge\ \neg(\mathit{data\_valid\_1}\ t))\ \Rightarrow\\
\qquad\quad \mathit{ready\_1}\ (t+1)\ \wedge\ \neg(\mathit{data\_valid\_2}\ t)\quad \wedge\\
\qquad (\mathit{ready\_1}\ (t+1)\ \wedge\ \neg(\mathit{data\_valid\_1}\ (t+1)))\ \Rightarrow\\
\qquad\quad (\mathit{data\_2}\ (t+1)\ =\ \mathit{data\_2}\ t)\quad \wedge\\
\qquad (\mathit{ready\_1}\ t\ \wedge\ \mathit{data\_valid\_1}\ t)\ \Rightarrow\\
\qquad\quad \mathsf{CASE}\ (A\ (\mathit{data\_1}\ t))\ \mathsf{OF}\\
\qquad\qquad \mathsf{Defined}\ y\quad \exists m.\ (\forall n.\ n < m\ \Rightarrow\\
\qquad\qquad\qquad\qquad\qquad (\forall p.\ p \le n\ \Rightarrow\ \neg(\mathit{reset}\ (t+p)))\ \Rightarrow\\
\qquad\qquad\qquad\qquad\qquad\ \neg(\mathit{ready\_1}\ (t+n+1)) \wedge\\
\qquad\qquad\qquad\qquad\qquad\ \neg(\mathit{data\_valid\_2}\ (t+n))\ )\quad \wedge\\
\qquad\qquad\qquad\qquad ((\exists s.\ \mathit{ready\_2}\ (t+m+s))\ \Rightarrow\\
\qquad\qquad\qquad\qquad\quad (\exists s.\ (\forall n.\ n < s\ \Rightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad (\forall p.\ p \le m+n\ \Rightarrow\ \neg(\mathit{reset}\ (t+p)))\ \Rightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \neg(\mathit{ready\_1}\ (t+m+n+1)) \wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \neg(\mathit{data\_valid\_2}\ (t+m+n)) \wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad\ (\mathit{data\_2}\ (t+m+n)\ =\ y)\ )\quad \wedge\\
\qquad\qquad\qquad\qquad\qquad\quad ((\forall p.\ p \le m+s\ \Rightarrow\ \neg(\mathit{reset}\ (t+p)))\ \Rightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \neg(\mathit{ready\_1}\ (t+m+s+1)) \wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \neg(\mathit{data\_valid\_2}\ (t+m+s)) \wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad\ (\mathit{data\_2}\ (t+m+s)\ =\ y)\ )\ ))\quad \wedge\\
\qquad\qquad\qquad\qquad ((\forall s.\ \neg(\mathit{ready\_2}\ (t+m+s)))\ \Rightarrow\\
\qquad\qquad\qquad\qquad\quad (\forall n.\ \ (\forall p.\ p \le m+n\ \Rightarrow\ \neg(\mathit{reset}\ (t+p)))\ \Rightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \neg(\mathit{ready\_1}\ (t+m+n+1)) \wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \neg(\mathit{data\_valid\_2}\ (t+m+n)) \wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad\ (\mathit{data\_2}\ (t+m+n)\ =\ y)\ ))\\
\qquad\qquad \mathsf{Undefined}\quad \forall m.(\forall n.\ n \le m\ \Rightarrow\ \neg(\mathit{reset}\ (t+n)))\ \Rightarrow\\
\qquad\qquad\qquad\qquad \neg(\mathit{ready\_1}\ (t+m+1)) \wedge\\
\qquad\qquad\qquad\qquad \neg(\mathit{data\_valid\_2}\ (t+m))
\end{array}
$$

Figure 3: Formal definition of the p-interface pattern for the system level

The interface pattern P_INTERFACE_SYSTEM describes a relation between the interface signals $\mathit{data\_1}$, $\mathit{data\_valid\_1}$ and $\mathit{ready\_1}$, which are the basic signals of channel_1, the

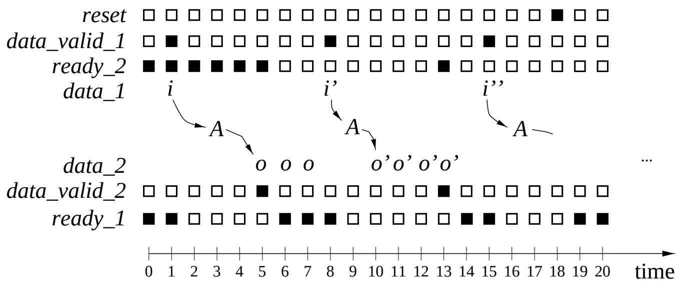


Figure 4: Schematic illustration of the p-interface pattern defined in figure 3

interface signals *data_2*, *data_valid_2* and *ready_2* , which build channel_2, and the signal *reset*. To distinguish the input and output channel, we added the suffixes *_1* and *_2* to the basic signals. The relation is with respect to some program A.

The semantics of the basic signals is as follows: the process gets the data packages from its predecessor via *data_1* and delivers data packages to its successor via *data_2*. *data_valid_1* and *data_valid_2* signal, whether the predecessor delivers new data to the process and whether the process delivers new data to its successor, respectively. With *ready_1* and *ready_2* the process and its successor notify their predecessors, whether they are able to accept new data. The global signal *reset* is used to interrupt calculations and to bring all processes into their initial state.

The interface pattern states that the circuit has an internal state, indicated by the signal *ready_1*, which is initially T. If no calculation is performed, the *data_2*-signal holds the result of the last program execution. If a calculation is started, i.e. the signals *data_valid_1* and *ready_1* are set at the time, *ready_1* will be set to F in the next clock tick, indicating that the process is busy now and cannot accept new data from its predecessor. If *ready_2* is set when the calculation terminates, the result is immediately passed to the successor by setting the *data_valid_2*-signal. A clock tick later, *ready_1* is set again and the process is waiting for new data. If *ready_2* is not set when the calculation terminates – i.e. the successor process is busy, the result is stored in *data_2*, until *ready_2* is set. If the calculation does not terminate and/or the successor process never signals that it is able to accept new data, *ready_1* is set to F until *reset* is set thus stopping the calculation. If *reset* is set, *ready_1* is set a clock tick later indicating that the process is ready for new input.

The dfg-interface pattern, that is defined for the system level is similar to the p-interface pattern. The difference is, that calculations always terminate and that they always terminate within the same number of clock ticks. Due to lack of space the dfg-interface pattern is not shown here .

2.3 S-structures

In Gropius-3, structures of processes are named s-structures. There are three kinds of processes: dfg-term based processes, p-term based processes and k-processes. Dfg-term based processes and p-term based processes are nothing but algorithmic circuit descriptions in Gropius-2 with the specific channel-based interface pattern, which was explained in the last section.

Both dfg-term based processes and p-term based processes have a single input channel and a single output channel. K-processes are sort of a glue logic for building arbitrary s-structures. They are used for spreading, combining, buffering and delaying signals.

The syntax of s-structures is as follows:

$$\begin{array}{lcl} \textit{s-interface} & ::= & \text{"(" "reset"} \left\{ \text{","} \ \textit{channel} \right\} \text{")"} \\ \textit{s-process} & ::= & \textit{algorithmic dfg-circuit description} \ \mid \\ & & \textit{algorithmic p-circuit description} \ \mid \\ & & \textit{k-process} \\ \textit{s-structure} & ::= & \text{"}\exists\text{"} \ \textit{channel} \left\{ \text{","} \ \textit{channel} \right\} \text{"."} \\ & & \quad \textit{s-process} \ \textit{s-interface} \left\{ \text{"}\wedge\text{"} \ \textit{s-process} \ \textit{s-interface} \right\} \end{array}$$

2.4 K-processes

K-processes are channel based communication units. There are ten different k-processes, six of them are displayed in figure 5.

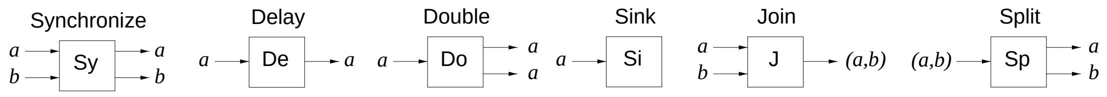


Figure 5: Some k-processes

The process Synchronize waits until both input channels hold a label and if both successor processes are ready to read new data, it moves both labels to the output at the same time. Delay delays some label by at least one clock tick. Double duplicates a label. There are two output channels — one for each copy. Join combines two labels to a single label with the new data being a pair holding the two data packages of the original labels. Split is the inverse operation to Join. Since the channels are bidirectional, there must be an extra Sink-process to represent sinks of signals.

K-processes are used for combining several algorithmic processes and to manage the communication between them. They are defined once and for all and can thus be reused in different systems.

The strict separation of functional and temporal aspects at the algorithmic level may at first glance mean a restriction in contrast to other approaches, where these descriptions can be arbitrarily mingled. However, even those descriptions can be represented in Gropius in the following way. A description containing functional as well as temporal parts is first partitioned into several purely algorithmic descriptions such that the temporal aspects only describe the communication between these processes. Then the new processes are combined at the system level in combination with k-processes to yield the original mixed functional/temporal description. In figure 6 a small but typical example is illustrated. An algorithmic circuit description contains two functional blocks A and B. After A is performed, the calculation of B is delayed until some control signal is set. In VHDL this is expressed using the "wait until"-expression. This description cannot be represented in Gropius at the algorithmic level. However, it can be represented at the system level using the k-processes Synchronize and Sink.

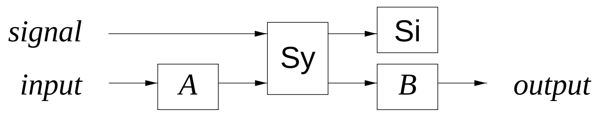


Figure 6: Example for representation of mixed functional/temporal description

3 Synthesizing systems

Each process at the system level represents a hardware component with a well-defined interface in terms of electronic input and output wires. A s-structure can be synthesized by synthesizing each process separately and then implementing the channels as bundles of wires.

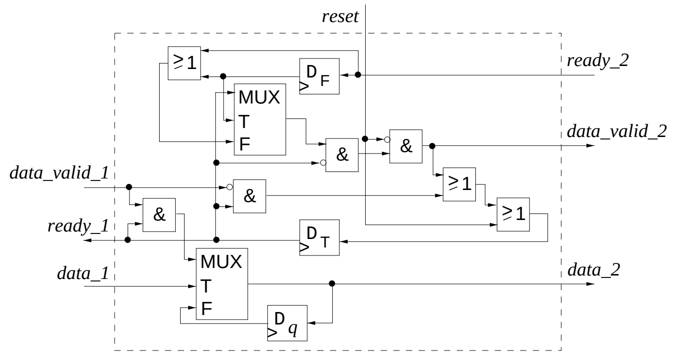
Synthesis of dfg-terms and p-terms at the algorithmic level has already been introduced in [11, 14], respectively. We here only will roughly repeat the synthesis scenario for p-terms.

In our approach, the synthesis of algorithmic p-circuit descriptions is divided into two parts: first the p-term is transformed into an equivalent p-term in the so-called single-loop-form (slf) by applying pre-proven program transformation theorems. Afterwards a pre-proven implementation theorem is applied to perform the interface synthesis thus generating a RT-level implementation. The only difference between synthesis at the system level compared to the algorithmic level is that another interface pattern is used and therefore during interface synthesis another implementation theorem is applied, which generates the RT-level implementation.

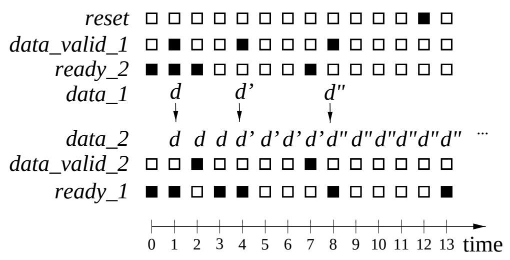
3.1 Synthesizing k-processes

When synthesizing algorithmic dfg- or rather p-circuit descriptions, several tasks like scheduling, allocation and binding have to be performed to achieve cost-minimal implementations. Since in k-processes no algorithmic calculation is performed, nothing can be optimized and thus the synthesis is restricted to applying an implementation theorem. Therefore, for every k-process an implementation at the RT-level has to be found and an implementation theorem has to be proven stating that the implementation fulfills the communication scheme for the k-process. We have proven implementation theorems for all of the ten k-processes. The following implementation theorem e.g. states that the implementation, which is sketched in figure 7(a), implies the behavior of the k-process Delay, which is illustrated in figure 7(b).

$$\begin{aligned}\vdash\ & \mathsf{IMPLEMENTATION\_DELAY} \\ & \quad (\mathit{reset},\ (\mathit{data\_1},\ \mathit{data\_valid\_1},\ \mathit{ready\_1}),\ (\mathit{data\_2},\ \mathit{data\_valid\_2},\ \mathit{ready\_2})) \\ & \Rightarrow \\ & \mathsf{INTERFACE\_DELAY} \\ & \quad (\mathit{reset},\ (\mathit{data\_1},\ \mathit{data\_valid\_1},\ \mathit{ready\_1}),\ (\mathit{data\_2},\ \mathit{data\_valid\_2},\ \mathit{ready\_2}))\end{aligned}$$



(a)



(b)

Figure 7: Schematic illustration of implementation and specification for Delay

3.2 Further optimizations

One possibility to synthesize a s-structure is to just synthesize its components independently from each other. However, the drawback is that every process, that has to be implemented in hardware, requires its own hardware resources. Since optimizations can only be performed within single processes, the resulting hardware for the complete system is normally not cost-minimal. It would be better to first combine several processes to perform optimizations over their boundaries. A second advantage is the reduction of the communication overhead. Therefore, we are on our way to develop a method to combine several processes to a single process.

4 Conclusion

In this paper we have presented a new methodology for deriving RT-level structures from circuit descriptions at the system level. There are two aspects, in which our approach differs from conventional approaches in this domain. Firstly, this is (to our knowledge) the first formal methodology for synthesis at the system level. The result is a guaranteed correct implementation at the RT-level, which is derived by a sequence of logical transformations. Secondly, we have defined a fixed communication scheme, where every connection between two processes directly corresponds to electronic wires. Conventionally, processes at the system level communicate by abstract channels, that have to be synthesized separately.

References

[1] D.D. Gajski, F. Vahid, S. Narayan and J. Gong. *Specification and design of embedded system.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[2] J.M. Mendias, R. Hermida and M. Fernandez. Correct high-level synthesis: a formal perspective. In *Design, Automation and Test in Europe*, pages 977–978, Paris, France, 1998. IEEE Computer Society.

[3] P.F.A. Middelhoek. *Transformational Design: An Architecture Independent Interactive Design Methodology for the Synthesis of Correct and Efficient Digital Systems.* PhD thesis, Universiteit Twente, NL, 1997.

[4] J. Hallberg and Z. Peng. Synthesis under local timing constraints in the CAMAD hig-level synthesis system. In *IEEE EUROMICRO*, Como, Italy, 1995. IEEE-Press.

[5] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[6] S.D. Johnson and B. Bose. DDD: A system for mechanized digital design derivation. In *International Workshop on Formal Methods in VLSI Design*, Miami, Florida, January 1991. ACM/SIGDA. Available as Indiana University Computer Science Department Technical Report No. 323 (rev. 1997).

[7] E.M. Mayger and M.P. Fourman. Integration of formal methods with system design. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 59–70, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[8] R. Sharp and O. Rasmussen. The T-Ruby design system. In *IFIP Conference on Hardware Description Languages and their Applications*, pages 587–596, 1995.

[9] M. Larsson. An engineering approach to formal digital system design. *The Computer Journal*, 38(2):101–110, 1995.

[10] R. Kumar , C. Blumenröhr, D. Eisenbiegler, and D. Schmid . Formal synthesis in circuit design-A classification and survey. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design. First International Conference,FMCAD'96*, number 1166 in Lecture Notes in Computer Science, pages 294–309, Palo Alto, CA, USA, November 1996. Springer-Verlag.

[11] D. Eisenbiegler, C. Blumenröhr, and R. Kumar. Implementation issues about the embedding of existing high level synthesis algorithms in HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics:9th International Conference, TPHOLs'96*, number 1125 in Lecture Notes in Computer Science, pages 157–172, Turku,Finland, August 1996. Springer-Verlag.

[12] C. Blumenröhr and D. Eisenbiegler. An efficient representation for formal synthesis. In *10th International Symposium on System Synthesis*, pages 9–15, Antwerp,Belgium, September 1997. IMEC, IEEE Computer Society Press.

[13] C.Blumenröhr and D. Eisenbiegler. Deriving structural RT-implementations from algorithmic descriptions by means of logical transformations. In F.J. Rammig and W. Müller, editor, *GI/ITG/GME Workshop Methoden des Entwurfs und der Verifikation digitaler Systeme*, pages 38–49, Paderborn, Germany, March 1998. HNI-Verlagsschriftenreihe.

[14] C.Blumenröhr and D. Eisenbiegler. Performing high-level synthesis via program transformations within a theorem prover. In *Digital System Design Workshop at the 24th EUROMICRO 98 Conference*, pages 34–37, Vaesteraas, Sweden, 1998. IEEE-Press.

[15] D. Eisenbiegler. Automata — A theory dedicated towards formal circuit synthesis. Technical Report 14/97, Universität Karlsruhe, 1997. http: //goethe. ira. uka.de/fsynth/publications/postscript/Eise97b.ps.gz.

[16] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Springer, 1992.