AUTOMATIC SYNTHESIS OF SYSTEMC-CODE FROM FORMAL SPECIFICATIONS*

Carsten Rust, Achim Rettberg
University of Paderborn, Germany
Carsten.Rust@c-lab.de, Achim.Rettberg@c-lab.de

Abstract: The paper presents an approach for realizing high-level Petri net models in SystemC. The approach contributes to an existing methodology for the Petri net based design of distributed embedded real-time systems. It is intended to be a vehicle for realizing Petri net components in hardware. The paper describes the use of standard SystemC language constructs to realize the execution of a high-level Petri net, which is assumed to be separated into partitions. Besides techniques for realizing the mechanisms of Petri net execution, the integration of the code generation into the overall design flow is discussed. To demonstrate the effectiveness of our approach we use the inverse discrete cosine transformation (IDCT) that is part of the MPEG-2 algorithm.

Keywords: Petri nets, SystemC, Embedded Systems, System Synthesis.

1. INTRODUCTION

We describe a method for synthesizing SystemC-Code from high-level Petri nets. The synthesis method is part of a Petri net based methodology for the design of distributed embedded real-time systems [13]. The methodology leads through the complete design process from modelling on an abstract level using high-level Petri nets via analysis and partitioning of the model down to automatic synthesis of an implementation. Within the synthesis stage, we currently are able to automatically generate target code for different microcontrollers that are interconnected by a communication media, for instance a CAN bus. Such a software implementation is sufficient for many applications. In certain cases, however, it is necessary to realize components of an embedded system in hardware in order to meet real-time constraints.

For realizing a component in hardware, a specification in a Hardware Description Language (HDL) has to be generated. Appropriate languages are

*This work was supported by the German Science Foundation (DFG) project SFB-376

for instance Verilog and VHDL. In addition to these languages, C/C++ based languages for hardware design and hardware software codesign respectively emerged in recent years. Prominent examples are SystemC [14] and SpecC [3]. Especially SystemC has been developed towards a standardized modelling language, that is intended to enable system level design and IP exchange at multiple levels of abstraction for hardware and software components in embedded systems. Developed for today's System-on-Chip (SoC) design with increasing complexity, it offers executable specifications with high performance in early design phases. Hence, SystemC can be a particularly suited vehicle for realizing Petri net components in hardware. In order to accomplish that, techniques for realizing the basic operations of Petri net execution in SystemC are needed. A description of these techniques is the main topic of this paper.

In the remaining sections of the paper, we first give an overview of approaches related to our work (Section 2) and provide some background concerning SystemC (Section 3). Section 4 gives an overview of the entire synthesis process and its integration into our design methodology, while Section 5 presents several details of the realization of high-level Petri net execution in SystemC. Finally, in Section 6 an application example is described.

2. RELATED WORK

Many applications of Petri nets to various aspects of hardware design can be found in literature (cf. for instance [16]). One reason for the usage of Petri nets in this area is their ability to express concurrency and parallelism. Furthermore, Petri nets are a well-investigated formal model offering a variety of analysis methods. They are used e.g. for behavioural modelling, analysis and verification, synthesis, and performance analysis. There are even complete Petri net based systems for hardware design, e.g. the CAMAD high-level synthesis system [9]. A canonical method for interfacing suchlike Petri net based methodologies to standard hardware design tools is to generate VHDL- or Verilog-code. For generating HDLs from Petri nets, again several approaches can be found in literature. In [10] behavioural VHDL code is generated from Petri nets. The generated code is not synthesizable, but it can be used for simulation of the model. Another approach is described in [11]. Here, structural – fully synthesizable – code is generated from so called Hardware Petri nets, an extension of Place/Transition nets.

In order to develop similar approaches for the relatively new C-based HDL SystemC, some basic techniques for realizing Petri nets in SystemC should be provided.

Concerning SystemC, currently no approach for coupling it with a formal graphical modelling language like Petri nets is known to us. Methodologies based on SystemC use either this language directly (as for instance [4]) or

blockdiagrams for representing the entire system (e.g. [8]). Blockdiagrams however are rather a means for visualization than a formal model. Hence, our work is a contribution to interface a formal implementation independent high-level Petri net-model to the implementation oriented language SystemC.

3. BACKGROUND

In this section, we give a brief overview of our target language SystemC. We pass on a detailed description of our Petri net model used for specification. A small example net is depicted in Figure 2 on page 191. Basically, our formal model is a form of high-level Petri nets. An overview of several high-level Petri net-models is given in [5]. A general introduction into Petri nets can be found for instance in [7]. Beyond standard constructs of high-level Petri nets, our formal model includes a hierarchy concept in order to support easy modelling of complex systems. Furthermore, we support delay specifications.

As regards SystemC, introductions can be found in [14, 6]. Similar to other HDLs, users can construct structural descriptions of designs in SystemC using modules, ports and signals. To enable structural design hierarchies, modules (SC_MODULE) can be instantiated within other modules. Communication between different modules is enabled by ports (single directional or bi-directional) and signals. All ports and signals are declared by the user to have a specific data type. Typical data types include single bits, bit vectors, characters, integers, floating point numbers, vectors of integers, etc. Four-state logic signals (i.e. signals that model 0, 1, X, and Z) are also supported by SystemC. For the behavioural part of a hardware specification, concurrent behaviours can be modeled using processes. Such a process can be thought of as an independent thread of control which resumes execution when some set of events occur or some signals change, and suspends execution after performing some action. Generally SystemC process instances have their own independent execution stack. Code within processes is executed sequentially. Certain processes in SystemC that suspend at restricted points in their execution do actually not require an independent execution stack - these process types are termed SC_METHODs. Optimizing SystemC designs to take advantage of SC_METHODs leads to great improvements of simulation performance when the number of process instances in a design is large, see [14]. In SystemC a set of features for generalized modelling of communication and synchronization is available. These are channels, interfaces and events. They enable designers to model the wide range of communication and synchronization found in system designs. Examples include HW signals, queues (FIFO, LIFO, message queues, etc.), semaphores, memories and busses (both as RTL and transaction-based models), see [14].

4. SYNTHESIS PROCESS

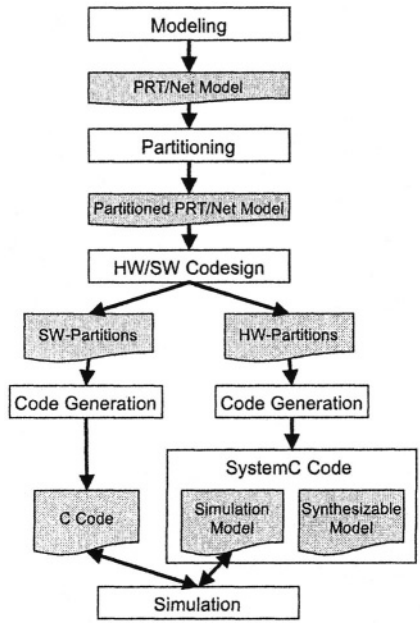


Figure 1. Overview of entire synthesis process

The synthesis method presented in this paper is integrated into our methodology [13] as shown in Figure 1. We first give an overview of the first three steps, which are not in the scope of this paper. The step *modelling* leads to a hierarchical high-level Petri net of a system under construction. The next step is to flatten the net specification, which is straightforward, and to partition it. For *Partitioning,* we use a method introduced in [15]. The basic characteristics of this method are on the one hand that very small partitions are produced and on the other hand that the connections between different partitions have a simple structure. The latter is reached by encapsulating conflicts into single partitions. Due to the simple connection structure, communication between different partitions can be realized using a simple 'send and forget' mechanism. For more details concerning modelling and partitioning, we refer to [13].

After the step of partitioning, the system under consideration is given as a set of small Petri net-units with a simple connection structure. For each unit, it now has to be decided whether to realize it as hardware or as software. This non-trivial and usually iterative process is represented by the item *HW/SW Codesign.* Without going into the details of this process we can state that it leads to a separation of the Petri net-model into Hardware- and Software-

partitions. The software parts are typically implemented in C or C++. To the hardware parts, the code generation presented in this paper is applied. In general, each Petri net can be mapped to an equivalent SystemC-specification, since SystemC is a superset of C. However, with the aim of realizing a Petri net component in hardware, the approach may be applied only to subnets that use a certain set of transition types. Petri nets that breach this condition, e.g. because a transition is annotated with a complex C function, have to be realized in software or to be modified by the engineer. With standard tools already available, the SystemC-code can be simulated together with the software parts, that were realized in C or C++. Tools for automatic synthesis of hardware from SystemC-Code are not available yet. However, these tools are under development. Guidelines for specifying synthesizable SystemC-models are given in [2].

5. PETRI NET REALIZATION

We first describe the basic concepts for execution of high-level Petri nets in SystemC using the example depicted in Figure 2. Afterwards, the coupling of subnets will be described. The SystemC-Code for the net in Figure 2 is (partly) depicted in Figure 3. The entire net is realized as a SystemC-module. Hence, it can define several methods to be executed on activation of the module. In our SystemC-realization for high-level Petri nets, the constructor of the module (SC_CTOR, line 58) determines the method for Petri net execution (`P02_main`, line 16) to be executed on activation. Therefore, this routine is executed – concurrently to the routines of other modules – each time the module is triggered. The routine itself is executed sequentially. The constructor also includes – besides the definition of the main method – the initialization of the module: For all inner places (in the example just one), the capacity and the initial marking is set (line 59) by means of methods that are not depicted in Figure 3. The mechanisms for triggering in line 64 are related to the interface specification, which has – as usual in hardware specification – to be defined for

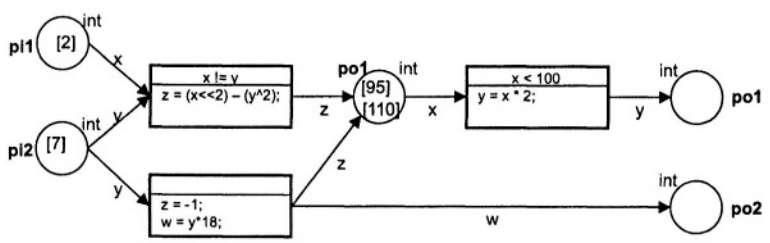
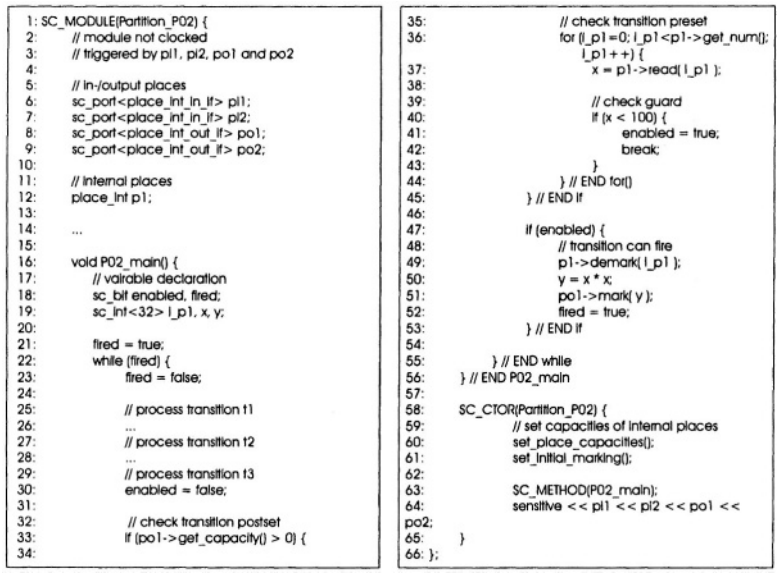


Figure 2. High-Level Petri Net Example

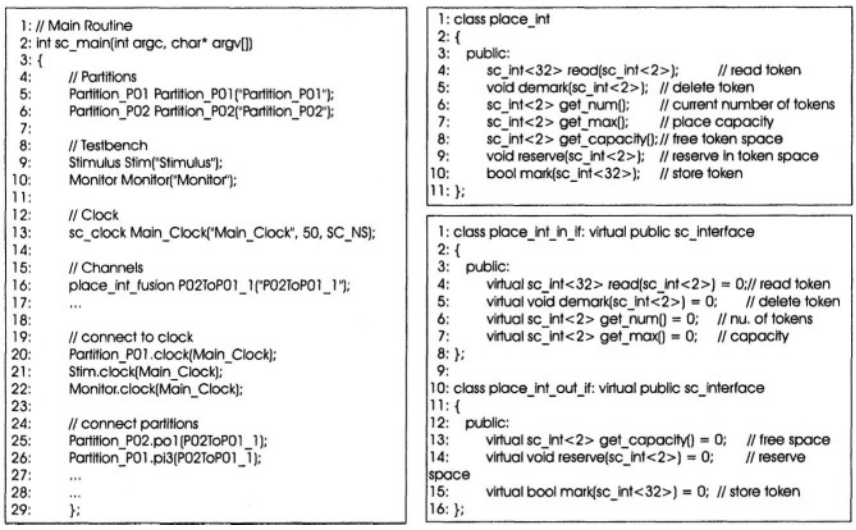


```
 1: SC_MODULE(Partition_P02) {
 2:     // module not clocked
 3:     // triggered by pi1, pi2, po1 and po2
 4:
 5:     // in-/output places
 6:     sc_port<place_int_in_if> pi1;
 7:     sc_port<place_int_in_if> pi2;
 8:     sc_port<place_int_out_if> po1;
 9:     sc_port<place_int_out_if> po2;
10:
11:     // internal places
12:     place_int p1;
13:
14:     ...
15:
16:     void P02_main() {
17:         // variable declaration
18:         sc_bit enabled, fired;
19:         sc_int<32> i_p1, x, y;
20:
21:         fired = true;
22:         while (fired) {
23:             fired = false;
24:
25:             // process transition t1
26:             ...
27:             // process transition t2
28:             ...
29:             // process transition t3
30:             enabled = false;
31:
32:             // check transition postset
33:             if (po1->get_capacity() > 0) {
34:
35:                 // check transition preset
36:                 for (i_p1=0; i_p1<p1->get_num();
                        i_p1++) {
37:                     x = p1->read( i_p1 );
38:
39:                     // check guard
40:                     if (x < 100) {
41:                         enabled = true;
42:                         break;
43:                     }
44:                 } // END for()
45:             } // END if
46:
47:             if (enabled) {
48:                 // transition can fire
49:                 p1->demark( i_p1 );
50:                 y = x * x;
51:                 po1->mark( y );
52:                 fired = true;
53:             } // END if
54:
55:         } // END while
56:     } // END P02_main
57:
58:     SC_CTOR(Partition_P02) {
59:         // set capacities of internal places
60:         set_place_capacities();
61:         set_initial_marking();
62:
63:         SC_METHOD(P02_main);
64:         sensitive << pi1 << pi2 << po1 <<
po2;
65:     }
66: };
```

Figure 3. SystemC-Code for Example in Figure 2

each module in addition to the behavioural specification. The interface of the module for subnet `P02` is specified in lines 5 to 9 of Figure 3. For each place of the net that is to be connected to other components, an instance of SC_PORT is defined. When instanciating the subnet, an object has to be provided for each port that implements the interface specified for the port (`place_int_in_if` and `place_int_out_if` respectively). The specification of the triggering mechanism in line 64 has the effect that the module is activated each time the value of one of the ports changes. Since the module is triggered only by changes on in- and outports, it is independent from the system clock.

The behavioural specification for executing the transitions of a net is like a standard software implementation using C or C++. In a loop (lines 22 to 55), which is executed until no more transition-firing can occur, each transition is evaluated. As an example, the evaluation code for transition `t3` is included in lines 29 to 53. First, it is checked whether the transition is enabled (lines 30 to 45). The check starts with testing whether the output places provide enough capacity for the token produced by a transition firing. Afterwards it is examined whether the input places contain enough token. Finally, the transition guard is evaluated. If the transition has concession to fire, the corresponding changes in the place marking are realized (lines 48 to 51). This includes removal of token from incoming places, evaluation of the transition annotation and creation of token for outgoing places. As shown in the source code in Figure 3, places are realized by instanciating a place class generated therefor (line 12). The decla-



```
// Main Routine
int sc_main(int argc, char* argv[])
{
    // Partitions
    Partition_P01 Partition_P01("Partition_P01");
    Partition_P02 Partition_P02("Partition_P02");

    // Testbench
    Stimulus Stim("Stimulus");
    Monitor Monitor("Monitor");

    // Clock
    sc_clock Main_Clock("Main_Clock", 50, SC_NS);

    // Channels
    place_int_fusion P02ToP01_1("P02ToP01_1");
    ...

    // connect to clock
    Partition_P01.clock(Main_Clock);
    Stim.clock(Main_Clock);
    Monitor.clock(Main_Clock);

    // connect partitions
    Partition_P02.po1(P02ToP01_1);
    Partition_P01.pi3(P02ToP01_1);
    ...
    ...
    };
```

```
class place_int
{
  public:
    sc_int<32> read(sc_int<2>);      // read token
    void demark(sc_int<2>);  // delete token
    sc_int<2> get_num();     // current number of tokens
    sc_int<2> get_max();     // place capacity
    sc_int<2> get_capacity(); // free token space
    void reserve(sc_int<2>);  // reserve in token space
    bool mark(sc_int<32>);    // store token
};
```

```
class place_int_in_if: virtual public sc_interface
{
  public:
    virtual sc_int<32> read(sc_int<2>) = 0;// read token
    virtual void demark(sc_int<2>) = 0;     // delete token
    virtual sc_int<2> get_num() = 0;   // nu. of tokens
    virtual sc_int<2> get_max() = 0;   // capacity
};

class place_int_out_if: virtual public sc_interface
{
  public:
    virtual sc_int<2> get_capacity() = 0;   // free space
    virtual void reserve(sc_int<2>) = 0;      // reserve
space
    virtual bool mark(sc_int<32>) = 0;  // store token
};
```

Figure 4. SystemC-Code for entire net and for a place / a channel

ration of the place class used for place `p1` of the example net is depicted in the upper right part of Figure 4. For the example net, only the depicted place class `place_int` is needed. In general, one class is generated for each place type occuring in a net specification.

Beyond evaluation of single transitions, a Petri net-implementation has to provide mechanisms for resolving conflicts, for instance the conflict between transition `t1` and transition `t2`, when both places `pi1` and `pi2` are marked with appropriate token. Conflict resolving is however realized implicitly when transitions are implemented in one module as indicated in Figure 3, since the code of one SystemC-module is executed sequentially. Furthermore, our partitioning method ensures that transitions realized in different modules are not conflicting (cf. Section 4).

Obviously, the simulation algorithm for a single subnet is pretty simple. The strategy of evaluating all transitions of a net in a loop would be very inefficient for large nets. We do however assume only small subnets to be realized in each partition (cf. Section 4), for which the simple strategy is sustainable. For the execution of the entire net, a less clumsy simulation strategy is realized, which avoids steady evaluation of all transitions. Instead, after each change of the net marking of one partition, only those other partitions are evaluated, whose transitions are affected from the change (since they are connected to a place with a modified marking). This is realized implicitly by using adequate primitives for communication between different partitions.

For coupling of subnets, our code generation approach assumes partitions that communicate with each other via shared places. For building partitions, we hence assume an algorithm that – like our partitioning algorithm mentioned in Section 4 – cuts a given net at places, leaving a copy of a cut place in both partitions built through cutting. The connections between partitions are realized by means of channels. In the SystemC-Code for the partitioned net, partly provided in Figure 4, the channels needed for communication between partitions are created in lines 15 to 17. For each connection, we create a channel, in our example net an instance of class `place_int_fusion`. The channel class implements the interfaces `place_int_in_if` and `place_int_out_if`, which are depicted in the lower right part of Figure 4. Naturally, these classes have the same methods as the class for inner places of the same type `int`. Since `place_int_fusion` implements these interfaces, an instance of the class is qualified to connect for instance the outport `po1` of partition `P02` (cf. Figure 2 and 3, line 8) with the inport `pi3` of another – not further specified – partition `P01`. The – straightforward – implementation of the channel class is omitted. In general, channels are not synthesizable. Due to the simple communication structure between partitions, we suppose however, that the corresponding channels can be refined to synthesizable code using signals. In order to enable evaluation of the specified system by simulation, a testbench consisting of a stimulus (line 9) and a monitor (line 10) are created. A clock is needed for the testbench as well as for timed subnets as partition `P01`. For the realization of this timed component we refer to [12]. For the testbench, standard SystemC-classes are used.

6. APPLICATION EXAMPLE

In order to evaluate our approach, we have implemented the inverse discrete cosine transformation (IDCT) according to the approach of Chen-Wang [1] as a Petri net. The IDCT consist of 13 additions, 13 subtractions, and 14 constant coefficient multiplications. The data-flow graph of the IDCT is depicted in figure 5. The boxes indicate a partitioning into so-called butterfly components. These components each consist of one addition and one subtraction with crossed inputs. In some cases, a butterfly component has constant coefficient multiplications for each input of the addition and subtraction. Other possibilities for partitioning are to include the entire IDCT into one partition or to include into each partition one operation only.

Based on a Petri net library containing all required elements, the depicted data-flow graph could easily be transformed into a high-level Petri net and - using the approach presented in this paper - into SystemC-Code. Hence, we were able to compare the generated code with an already existing hand-written implementation. A comparison concerning simulation time has shown no sig-

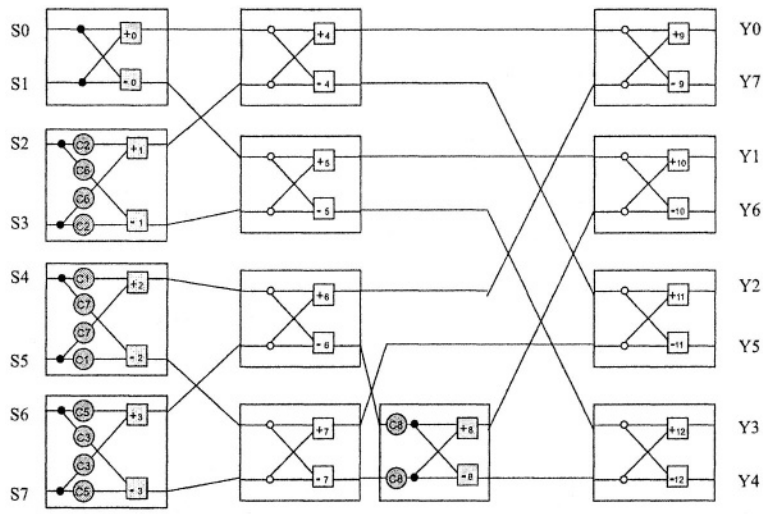


Figure 5. Data-flow graph of the inverse discrete cosine transformation (IDCT)

nificant differences between the implementations. An advantage of the transformation into Petri nets is for instance that tests with different partitionings of the model can easily be achieved. The partitioning can be influenced by the user or by design parameters in the sense of design space exploration. Furthermore, we are able to annotate timing information from previously synthesized components into the Petri net model. Starting with single partitions, we then are able to gradually analyse the timing behaviour of the specified system. This leads to a hierarchical design approach based on our SystemC code generation from Petri nets.

7. CONCLUSION

An approach for the realization of high-level Petri net-models in SystemC was introduced. The approach aims at providing a code generation component for realizing Petri net-models of embedded real-time systems in hardware. We presented SystemC-implementations for several aspects of high-level Petri net execution. Furthermore, we described how the SystemC-code generation is integrated into our existing methodology for the design of distributed embedded real-time systems. As an application example, we implemented the IDCT that is part of MPEG-2. We specified this example using Petri nets as well as using SystemC directly. The comparison between the manually and the generated SystemC code for simulation have shown no significant differences. With the presented approach, we are able to perform a design space exploration based on a partitioning of the system.

REFERENCES

[1] Chen-Wang, *Inverse two dimensional DCT, in Proceedings of the IEEE ASSP-32,* pp. 803-816, August 1984

[2] Doulos Ltd., *SystemC Golden Reference Guide,* 2002.

[3] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, January 2000.

[4] J. Gerlach and W. Rosenstiel, "System Level Design Using the SystemC Modeling Platform," in *Proc. of Workshop on System Design Automation (SDA'00),* pp. 185-189, Rathen, Germany, March 2000.

[5] K. Jensen, G. Rozenberg, Eds., *High-Level Petri Nets.* Springer Verlag, 1991.

[6] W. Müller, W. Rosenstiel, and J. Ruf, editors. *SystemC - Methodologies and Applications.* Kluwer Academic Publishers, Dordrecht, June 2003.

[7] T. Murata, "Petri Nets: Properties, Analysis and Applications," in *Proceedings of the IEEE,* Vol.77, No.4 pp.541-580, April, 1989.

[8] S. Pasricha, "Transaction Level Modeling of SoC using SystemC 2.0," in *Synopsys User Group Conference (SNUG 2002),* Bangalore, May 2002.

[9] Z. Peng and K. Kuchcinski, "Automated Transformation of Algorithms into Register-Transfer Level Implementations," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 13(2): 150-166., Feb. 1994.

[10] D. Prothero, "Modelling and Implementation of Petri Nets Using VHDL," in *Hardware Design and Petri Nets,* Yakovlev, A.; Gomes, L.; Lavagno, L., Eds., Boston: Kluwer Academic Publishers, 2000, pp. 223-236.

[11] P. Rokyta, W. Fengler, and T. Hummel, "Electronic System Design Automation using High Level Petri Nets,", in *Workshop for Hardware Design and Petri Nets,* Lisboa, June 22-26, 1998, pp. 129-138. 1998

[12] C. Rust and A. Rettberg. Generating systemc code for the execution of high-level petri nets.
URL: http://wwwhni.uni-paderborn.de/eps/uni/publications/, May 2004.

[13] C. Rust, J. Tacken, and C. Böke, "Pr/T-Net Based Seamless Design of Embedded Real-Time Systems," in *LNCS 2075; International Conference in Application and Theory of Petri Nets (ICATPN),* J.-M. Colom and M. Kounty, Eds., vol. 2075. Newcastle upon Tyne, U.K.: Springer Verlag, 2001, pp. 343–362.

[14] S. Swan, "An Introduction to System Level Modeling in SystemC 2.0," published on technical paper section of Open SystemC Initiative (OSCI) website (www.systemc.org), 2001.

[15] J. Tacken, C. Rust und B. Kleinjohann. A method for prepartitioning of Petri net models for parallel embedded real-time systems, in *Proceedings of the 6th Annual Australian Conference on Parallel and Real-Time Systems, (PART'99),* Melbourne, Australia, 1999.

[16] A. Yakovlev, L. Gomes, and L. Lavagno, Eds., *Hardware Design and Petri Nets.* Kluwer Academic Publishers, March 2000.