Abstract — The paper introduces an approach for realizing high-level Petri net-models in SystemC. The presented 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 firing of a high-level Petri net, which is assumed to be separated into partitions. The execution includes for instance the execution of single transitions, the handling of conflicting transitions, the realization of transition delays and the communication between Petri net-components realized in different partitions. Besides techniques for realizing these mechanisms, the integration of the code generation into the overall design flow is discussed.

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 [10]. The methodology leads through the complete design process from modeling on an abstract level using high-level Petri nets via analysis and partitioning of the model down to automatic synthesis of an implementation. In the analysis stage, Petri net analysis and worst-case execution time analysis methods are used in order to ensure functional and temporal requirements of the system under construction. Partitioning is done in order to separate the model into units that can be realized on different elements of a given target platform. 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 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 [12] and SpecC [2]. These languages and their accompanying tools support the specification of software components as well as of hardware components. Hence, they help designers of a mixed hardware/software system to overcome the drawback from using different design languages, incompatible tools and fragmented tool flows. In the last years, especially SystemC has been developed towards a standardized modeling 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 Petri nets and SystemC (Section 3). Section 4 gives an overview of the entire synthesis process and its integration into our design methodology. The main part of the paper is constituted by Section 5. Here, the realization of high-level Petri net-execution in SystemC is presented.

2 Related Work

Many applications of Petri nets to various aspects of hardware design can be found in literature (cf. for instance [14]). 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 behavioral modeling, 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 [6]. A canonical method for interfacing such-like 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 [7] behavioral 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 [8]. 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 modeling language like Petri nets is known to us. Methodologies based on SystemC use either this language directly (as for instance [9]) or blockdiagrams for representing the entire system (e. g. [5, 11]). 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

3.1 High-Level Petri Nets

In this section, we give a brief overview of our Petri net model. Basically, we use a form of high-level Petri nets. An overview of several high-level Petri net-models is given in [3]. A general introduction into Petri nets can be found for instance in [4]. Petri nets are bipartite directed graphs augmented by a marking. The Petri net graph consists of a finite set of places, a finite set of transitions, and directed edges from places to transitions and from transitions to places respectively. Places model conditions. For this purpose they may be marked by tokens. Driven by specific firing rules, a transition can fire based on the local marking of those places it is directly connected with. By firing, the marking of these places is modified.

In the case of high-level nets the tokens are typed individuals. The other net components are annotated accordingly: places with data types, edges with variable expressions and transitions with a guard and a set of variable assignments. Now a transition can fire only if the formal edge expressions can be unified with actually available tokens and this unification passes the guard expression of the transition. By firing the input tokens are consumed and calculations associated with the transition are executed. That way new tokens are produced that are routed to output places according to variable expressions annotating the output edges of the transition.

Figure 1: High-Level Petri Net Example

Figure 1 shows an example net. It is not part of a concrete application example, but of a net constructed for testing and evaluating our synthesis approach. The example net contains five places, each of which has the datatype int, i.e. the token of these places carry an integer value. Places may also be annotated with a SystemC datatype, e. g. with an integer of limited bit-width in order to enable an efficient realization. Places may also be annotated with a tuple of types. Furthermore, place capacities are specified. Transitions fire with respect to these capacities. Hence, in the situation depicted in Figure 1 only transition t3 is enabled, although for all transitions enough input token are available. Firing of t3 will remove one token from place p1 and thereby lead to a situation where one of the other transitions can fire.

Beyond standard constructs of high-level Petri nets, our formal model includes a hierarchy concept in order to support easy modeling of complex systems. Furthermore, we support delay specifications. While hierarchical specifications are not considered in this paper (the nets are flattened before synthesis), the realization of delays will be discussed in Section 5.3.
3.2 SystemC

In this section, a brief introduction to SystemC will be given. 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 behavioral part of a hardware specification, concurrent behaviors 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. The set of signals to which the process is sensitive must be specified before simulation starts [12]. Hence processes execute concurrently and may suspend and resume their execution at user-specified points. 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 not actually require an independent execution stack - these process types are termed SC_METHODs. Optimizing SystemC designs to take advantage of SC_METHODs provides dramatic simulation performance improvements when the number of process instances in a design is large, see [12].

In SystemC a set of features for generalized modeling of communication and synchronization is available. These are channels, interfaces and events. A SystemC channel is a container object for communication and synchronization between different modules. It is possible to implement one or more interfaces with a Channel. Such an interface specifies a set of different access methods that are implemented within a channel. Another synchronization method is an event. An event is a low-level synchronization primitive. Channels, interfaces and events 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 [12].

4 Synthesis Process

The synthesis method presented in this paper is integrated into our methodology [10] as shown in Figure 2. We first give an overview of the first three steps, which are not in the scope of this paper. The step Modeling 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 [13]. 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 Modeling and Partitioning, we refer to [10].

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 [1].

5 Petri Net Realization

5.1 Behavior of Basic Subnets

We first describe the basic concepts for execution of high-level Petri nets in SystemC using the example depicted in Figure 1. The SystemC-Code for this net – which was already described in Section 3.1 – is (partly) depicted in Figure 3. The entire net is realized as a SystemC-module (SC_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 (SCCTOR, 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 will be described in Section 5.2 together with the specification of the interface.

The behavioral 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.

Figure 3: SystemC-Code for Example in Figure 1

1: // triggered by pi, po, p1 and po2
2: // in output places
3: sc_port<place_int_out_if> po1;
4: sc_port<place_int_out_if> po2;
5: sc_port<place_int_in_if> pi1;
6: sc_port<place_int_in_if> pi2;
7: sc_port<place_int_in_if> p1;
8: sc_port<place_int_in_if> p2;
9: set_place_capacities();
10: set_initial_marking();
11: // internal places
12: place_int p1;
13: place_int p2;
14: ...
15: void P02_main() {
16: // variable declaration
17: sc_bit enabled, fired;
18: sc_int<32> i_p1, x, y;
19: // triggered by pi1, pi2, po1 and po2
20: if (enabled) {
21: fired = true;
22: while (fired) {
23: fired = false;
24: // process transition t1
25: ...}
26: // process transition t2
27: ...}
28: // process transition t3
29: enabled = false;
30: ...}
31: // check transition postset
32: if (p1->get_capacity() > 0) {
33: ...}
34: // check transition preset
35: for (i_p1 = 0; i_p1 < p1->get_num(); i_p1++) {
36: ...}
37: ...}
38: // check guard
39: if (x < 100) {
40: enabled = true;
41: break;
42: }
43: }
44: } // END if
45: } // END while
46: ...
47: if (enabled) {
48: // transition can fire
49: y = x*x;
50: ...
51: p1->read(i_p1);
52: fired = true;
53: } // END if
54: }
55: } // END P02_main
56: ...
57: ...
58: ...
59: set_initial_marking();
60: set_place_capacities();
61: set_initial_marking();
62: ...
63: ...
64: ...
65: }

Figure 4: SystemC-Code for a place

As shown in the source code in Figure 3, places are realized by instantiating a place class generated therefor (line 12). The declaration of the place class used for place p1 of the example net is depicted in Figure 4. It contains methods for reading a value from a place marking (read), removing a value (demark), and getting the current as well as the maximal number of elements in the place marking (get_num, get_max). Furthermore, there are methods for reading the current capacity – i.e. the difference between the maximal capacity and the current number of token (get_capacity), for reserving and freeing capacity (reserve), and for writing a token to a place (mark). reserve is needed for the realization of delays as described in Section 5.3. For the example

1: class place_int
2: {
3: public:
4: ...
5: void read(sc_int<32>); // read token
6: void reserve(sc_int<2>); // delete token
7: ...
8: ...
9: ...
10: ...
11: }

Figure 4: SystemC-Code for a place
net, only the depicted place class `place_int` is needed. In general, one class is generated for each place type occurring in a net specification. All place classes are identical with the exception of the datatype of token values.

Beyond evaluation of single transitions, a Petri net-implementation has to provide mechanisms for resolving conflicts, in fact the conflict between transition t1 and transition t2, when both places p11 and p12 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. They will be described below.

### 5.2 Interface Specification of Nets

As mentioned, the realization of the example net is very similar to a software realization in C++. Within the main method, SystemC-specific language constructs are used only for the definition of variables. Extended usage of SystemC-constructs is necessary when combining different subnets like that in Figure 1 to form a large application. In order to enable this, the implementation of a subnet has – as usual in hardware specification – to define an interface in addition to the behavioral 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 instantiating the subnet, an object has to be provided for each port that implements the interface specified for the port (`place_in_if` and `place_out_if` respectively). The specification of the triggering mechanism in line 64 is based on the interface definition. It 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.

```c++
```

Figure 5: SystemC-Code for a timed component
5.4 Communication between Subnets

The basic principles for combining several subnets into one complete net will be explained with a model instantiating the two subnets P02 and P01. The toplevel view is depicted in Figure 7. In addition to the two partitions, it includes two further elements Stimulus and Monitor. These elements form a testbench.

![Figure 7: Partitioned Net](image)

Our synthesis 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 cutted 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, provided in Figure 8, the channels needed for communication between partitions are created in lines 15 to 21. 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_in_if` and `place_out_if`, which are depicted in Figure 6.

![Figure 8: SystemC-Code for entire net](image)
with the inport pi3 of 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 stimulus (line 9), a monitor (line 10) and a trace file (line 45) are created. Therefor, standard SystemC-classes are used.

6 Conclusion and Outlook

An approach for the realization of high-level Petri net-models in SystemC was introduced. The approach aims at providing a 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 including for instance the evaluation of transitions and the realization of transition delays. While the implementation of small units is quite similar to a standard software implementation, the communication between components requires the usage of SystemC-features for modeling of communication and synchronization. Furthermore, we described, how the SystemC-code generation is integrated into our existing methodology for the design of distributed embedded real-time systems.

We are currently evaluating the presented approach. We therefor generate random nets that are similar to dataflow graphs. This will enable us to examine the efficiency of the code generated for large models. Provided that a tool for automatic synthesis of SystemC-code from dataflow graphs is available (which was for instance announced by Synopsys), we could compare the code generated from Petri nets with that generated from dataflow graphs. In particular, we want to evaluate whether our method generates synthesizable code as we suppose. Besides generating random nets, we also realize a small application example, a part of a communication protocol. We specify this example using Petri nets as well as using SystemC directly. The SystemC-Code developed manually will be compared to the generated code w.r.t. size and performance.

References


