Petri Net Based Design of Reconfigurable Embedded Real-Time Systems

Carsten Rust, Friedhelm Stappert and Reinhard Bernhardi-Grisson

University of Paderborn/C-LAB, Fuerstenallee 11, D-33102 Paderborn, Germany

Tel.: +49 5251 606126, Fax: + 49 5251 606065, Email: {car.fst,reinhard}@c-lab.de

Abstract: During the last years we have developed a methodology for the design of complex embedded real-time systems. The methodology supports the complete design flow reaching from modeling of embedded systems on a high level of abstraction over simulation and analysis down to the implementation on target platforms. In this paper, we describe our current work, which aims at opening the methodology for dynamically reconfigurable systems. We describe the main ideas for extending our formal model of High-Level Petri Nets in order to capture these systems. Furthermore, we describe our approach for timing analysis of these systems.

1. INTRODUCTION

Embedded systems design is becoming more and more complicated. The reasons are manifold, e.g. limited resources, reliability requirements, or the fact that embedded systems usually are distributed and contain concurrent behavior. In addition, embedded systems are – under several aspects like target architecture, modeling paradigms, and demands concerning reliability – heterogeneous. A further factor raising complexity of embedded systems design is that they are to an increasing extent dynamically reconfigurable. As an example, we will consider an automotive system in Section 3. Obviously, dynamically changing system parts increase the heterogeneity of a system even more. Dynamically evolving subsystems – which imply a powerful basic model for specification – have to be consid-

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35599-3_29](https://doi.org/10.1007/978-0-387-35599-3_29)

ered together with basic controllers running under hard reliability constraints.

Several design methodologies for embedded systems based on different formal models have been developed in recent years. Examples are the projects Ptolemy [5], which is based on several models of computation, and Moses [6], which is based on High Level Petri Nets. The methodology for our approach (cf. Section 2) is also based on Petri Nets. All mentioned methodologies are devoted to the design of heterogeneous systems. However, to our knowledge existing methodologies do not support dynamic reconfiguration of systems at run-time. Hence, existing methodologies are not well-suited for the above described systems, which we would like to characterize as heterogeneous embedded real-time systems including dynamically reconfigurable components. In this paper, we describe our approach for the design of these systems. In the following sections, we will first give an overview over the existing methodology and a typical application scenario and then describe our main ideas for two subtasks of the design, namely modeling and timing analysis.

2. PREVIOUS WORK

The approach presented in this paper is based on an existing methodology for static embedded real-time systems [1]. The methodology proposes a design flow divided into the three stages *Modeling*, *Analysis and Partitioning*, and *Synthesis* (cf. Figure 1). Within the stage of modeling, a heterogeneous model of the system under construction – specified using languages from different application domains – is transformed into one unique High Level Petri Net model, namely an extended Predicate/Transition-Net (Pr/T-Net). In the second stage, Petri Net analysis and timing analysis methods are applied in order to validate functional as well as temporal requirements and furthermore in order to gather information for an effective implementation of the system. The implementation is generated in the final stage of synthesis.

Figure 1: Design Flow

3. APPLICATION SCENARIO

In Figure 2, several components of an application scenario - an automotive system - are depicted, e.g. an air conditioning system (*ClimateControl*) and a display unit. The components are interconnected via different busses. The specification is heterogeneous in several ways, e.g. in that it mixes continuous controllers with state based specifications. In addition, the specified system is dynamically reconfigurable, for instance for the handling of so called Fail-Over situations, that is in error situations, where functionality has to be relocated. Another potential reason for dynamic reconfiguration is an altering functionality of a component, e.g. for diagnosis purposes. In the future applications are conceivable, where certain tasks are assigned to a component at run-time. An example may be a part of a decentralized intersection management, which aims at guiding several convoys of vehicles through an intersection [2].

Figure 2. Application Example

4. MODELING

Instead of a monolithic High-Level Petri Net model as before, our methodology now uses a set of Petri Net languages as formal model. For each part of the system under construction, an appropriate Petri Net model may be chosen for modeling it. For example, simple models like Place/Transition Nets may be chosen for the specification of small safety critical systems, that do not require high modeling power, but must support exhaustive analysis possibilities. On the other hand, for switching components between different modes, high-level self modifying Petri Net models are provided. Despite their different properties with respect to modeling power and analysis capa-

bilities, all Petri Net models are compatible with each other and hence can be integrated into one model enabling global investigations of the system.

In Figure 3, the Petri Net specification for a component of our application scenario is depicted, namely the unit for controlling the air conditioning system (*ClimateControl*). The specification includes one input place (*Event*) as well as an output place (*Ready*). By means of the input place, messages are received from other components like the air conditioning operating unit, the central display unit, or by other control units. The Power Management may for instance send a message *OFF* when the battery's state of charge is critical causing the Climate Control to switch off the air conditioning. The output place *Ready* is marked in order to indicate when a reaction to a specific input has been computed. To control the air conditioning system, calls to API functions of the system are associated with the firing of transitions. A firing of transition T_3 results for instance in a deactivation of the system.

Figure 3: Specification of Climate Control

The depicted Petri Net results from an incremental design process starting with a simple model which is developed to a complete specification by means of refinement steps. An example for refining a specification is the replacement of the node *Active* in *ClimateControl* by the net depicted in Figure 4. Another example is a so called Type Refinement, that replaces the type of *Event* and its associated places like *On* and *Ready* with an enumeration of all possible event types. Besides extending the type, the refinement assigns guards like ($e = ON$) to the transitions $T_1, \dots, T_3$.

For the incremental modeling, we basically use the refinements summarized by Lakos in [3]. These are type refinement, node refinement (like that of node *Active*) and in addition a refinement adding a subnet to an existing net. Since we are dealing with real-time systems, we additionally need refinements introducing a notion of time. They allow for instance to annotate a transition with the execution time of its corresponding code in the implementation. Technically, a refinement introducing transition delays is similar

to a type refinement. The types of places related to the timed transition are extended with a time-stamp. Transitions on the other hand are annotated in order to evaluate and manipulate the time-stamps.

Figure 4: Refinement of node Active

For realizing dynamic reconfiguration, we propose to apply the mechanism of net refinement – usually only used during the specification of a system – at run-time. Technically, the refinement of components is associated to the firing of certain transitions.

As an example, consider a Fail-Over situation in our application scenario: The system is notified of a failure of the Power Management, which must be handled by relocating functionality from the Power Management to other components. One step of this error handling is to modify the above described *ClimateControl*. After the modification, the Climate Control itself includes the Power Management component, that observes the battery's state of charge and generates an event *OFF*, when the state becomes critical.

The modification is realized by a series of refinements, e.g. a subnet refinement for adding a node, which is further refined to the battery supervision. The refinements are triggered by the firing of transitions, which get concession to fire when a failure of the Power Management is signaled. Obviously, a couple of technical features in the target system is presumed for the described reconfiguration to work, e.g. the possibility to redirect values of battery sensors to the Climate Control. However, providing a sound formal framework for specifying the reconfiguration is a prerequisite for a proper design process.

The main advantage of using refinements is that they allow a well-structured development of models. The refinement mechanism can be used for creating a heterogeneous hierarchical Petri Net (heterogeneous in the sense of comprising different model classes), whose single components are classified rigorously. The Petri Net class of each subnet comprises for instance information over the datatypes used, whether the subnet is timed, and whether the subnets incorporate dynamic reconfiguration at run-time.

The classification is implicitly done during refinement: In addition to the usual mapping from an abstract net to a refined one, each refinement step determines the class of the refined net. The class is determined depending on the class of the abstract net as well as of the components added by the refinement step. As described above, the mechanism of net refinement is also used for specifying dynamical changes at run-time. Hence, at run-time the same classification is possible for dynamically evolving subsystems as for static subsystems at design time.

The result of the modeling process, a hierarchical Petri Net specification with classified subnets, is finally handed over to the further design steps, e.g. evaluation of the complete model by simulation or verification of single components in terms of functionality. The Petri Net class of each component determines which tools may be chosen for its analysis and synthesis. In this paper however, we will not discuss the entire analysis and synthesis stages, but concentrate on timing analysis, which is treated in the following section.

5. TIMING ANALYSIS

When the final implementation of a model (or parts of it) is generated, it must be assured that it will run fast enough on the given target processor, i.e. that it will meet the given real-time constraints at run-time. Therefore, accurate estimates of the Worst-Case Execution Time (*WCET*) of the given Petri Nets are calculated. These estimates serve as input for a subsequent schedulability analysis. In this context, executing a net means that, starting from a specified initial marking, a defined end-marking of the given net is reached. For example, in *Figure 3* an end marking is reached when the place *Ready* is marked.

Traditional WCET analysis approaches (e.g. [4]) only analyze the code generated for a given specification as well as the corresponding processor-specific assembler code, but they have no knowledge of the original Petri Net. Thus, we propose to perform an additional analysis on the Petri Net Level.

For a motivation of the analysis on Petri Net Level, consider the code typically generated for the execution of a Petri Net in *Figure 5*. Analyzing source code only usually cannot yield an upper bound on the number of iterations of the while-loop. Furthermore, without any information about the run-time behavior of the code, a WCET analysis would assume the worst-case for each iteration of the loop, i.e. that each *if*-branch will be executed, although typically only a few transitions will fire during one iteration. An analysis of the Petri Net can yield both, a tight upper bound on the number of iterations and the required information about run-time behavior. In gen-

eral, by means of the analysis on the Petri Net level we want to answer questions like ‘How many steps will the net need at most to reach the end-marking?’, ‘How often will transition T_i fire at most during all steps?’, or ‘Which transition(s) will fire/not fire during the i -th step?’. This information is handed over to a traditional source code based WCET analysis.

```

while net is alive
  get enabled transitions
  if T1 is enabled
    fire T1
  if T2 is enabled
    fire T2
  ...
end while

```

Figure 5: Petri Net Code

The analysis on Petri Net level is divided into the two phases *reachability analysis* and *behavioral analysis*. The reachability graph yielded by the reachability analysis is needed in order to find longest execution paths using standard graph algorithms (like Dijkstra’s Algorithm). The structure of the reachability graph used in our analysis is different from graphs computed by common Petri Net analysis algorithms, as described in the following.

As we are primarily interested in the start- and end-states only, it is not important to investigate all possible intermediate states a net can reach. Therefore, we assume that the net is executed according to the *step-semantics*: For a given marking (i.e. state) of the net first the set of enabled transitions is determined and then all these transitions are assumed to fire in parallel. For example, in Figure 6 c) we do not explicitly represent the states that are reached by firing t_1 or t_2 alone, but only the state that is reached after firing both t_1 and t_2 regardless of which transition fires first. Hence, the edges of the reachability graph are not labelled with a single transition only, but instead with the *firing set* of transitions firing concurrently in the corresponding step. This significantly reduces the size of the reachability graph.

Figure 6: Step Semantics

In addition, we not only need information about *which* states a net can reach, but also *how often* these states will be reached during net execution. States that are reached more than once are indicated by cycles in the reachability graph. Hence, additional information about how often these cycles will be taken must be provided, especially when these figures depend on the input data of the net. To bound the number of iterations of such a cycle, it is sufficient to specify an upper bound on the maximum number of executions of one or more transitions in it. This number can either be provided manually by the designer, or derived automatically by analyzing the net, e.g. using techniques known from program flow analysis [10].

In order to be able to apply Dijkstra's Algorithm for finding longest paths, the reachability graph must be a DAG (*Directed Acyclic Graph*), since the longest-path problem is *NP*-complete for graphs that contain cycles. To achieve this property, we use the information about the maximum number of executions of transitions described above to 'unroll' cycles in the reachability graph. Each transition with a maximum execution count gets an additional *count place* as input place that contains as many tokens as the execution count, and one of these tokens is consumed each time the transition fires. Due to these additional places, each execution of a cycle is identified with unique markings that differ only in the number of tokens on the corresponding count places. It is however not necessary to unroll all cycles completely. Instead, all edges inside a cycle can be multiplied with the number of times this cycle is taken in the worst case.

After computing the reachability graph, the behavioral analysis computes longest paths, whereby according weights are assigned to the edges so that the length of the path indicates the desired result. Some examples are:

Maximum number of steps: To bound the number of steps of the given net, all edges in the reachability graph are assigned the weight 1. The length of the longest path from the initial to the final state then is the maximum number of steps the net can execute.

Maximum number of firings: The total number of possible transition firings can be found by assigning n to each edge of the reachability graph, where n denotes the number of transitions that fire in the corresponding step. Then the length of the longest path indicates the desired number.

Maximum number of firings of a certain transition: Similar to the above, the number of times a certain transition T will fire at most can be calculated by giving the weight 1 to all edges whose firing set contains T , and 0 to all others. Again, the length of the longest path represents the sought value.

In order to exploit the information compiled above on the source code level, we use the *Flow Facts Language* introduced in [7], which can be utilized by our WCET analysis tool presented in [8]. The information about the behavior of a given Petri net described above can easily be converted to flow

facts. As a simple example, consider Figure 7 a), which depicts the reachability graph of the net in Figure 4. The nodes are characterized by the places that contain a token in the corresponding state. Obviously the net can make at most two steps to reach the end marking, and therefore the corresponding loop iterates twice. A set of flow facts is shown in Figure 7 b): The first two facts specify that for the first loop iteration (i.e. the first step) none of $T4$, $T5$ and $T6$ will be executed, and that exactly one of the transitions $T1$, $T2$ and $T3$ will fire. The respective information for the second loop iteration is specified by the flow facts 3 and 4. Furthermore, the (redundant) information that all transitions can fire at most once during all iterations (denoted by '[]') is encoded in the set of facts in 5).

Figure 7: Example Analysis

Since the models we investigate in our design environment are intended to be changing at run-time, a complete static timing analysis of the model at design time is not possible. Instead we additionally need an *online* timing analysis performed at run-time. Furthermore, in our intended scenario the set of available processors is changing dynamically at run-time, hence the online timing analysis must be processor-independent.

This is achieved by replacing the hardware-dependent back-end of the WCET tool with a more generic analysis that yields an estimate of the computational complexity of the source code by e.g. counting the arithmetic operations performed. Consequently, the low-level analysis will not yield a target-specific WCET for each basic block of the given program, but instead an estimate on the computational complexity of the corresponding code. The higher level analysis, like program flow analysis, is performed as usual and not affected by the different implementation of the low-level analysis part. An approach for generic analysis is presented by Carro et al. in [9]. There, the code for a given task is first translated to a universal 3-address code for a virtual machine. This code is then analyzed according to some performance characteristics like how many jumps, arithmetic operations or memory access operations it employs. Available processors are classified according to how fast they are able to perform these operations. Based on these figures

together with the specified real-time constraints of the task (like period and deadline), an appropriate processor can be chosen for the given task.

6. CONCLUSION

We have presented our current work towards a methodology for the design of dynamically reconfigurable systems embedded systems. The main ideas for extending our formal model of High-Level Petri Nets in order to capture these systems were described. Furthermore, we presented our approach for a timing analysis of such systems.

REFERENCES

- [1] C. Rust, J. Tacke, C.Böke, "*Pr/T--Net based Seamless Design of Embedded Real-Time Systems*". In Proc. of 22nd Int. Conference on Application and Theory of Petri Nets (ICATPN'2001) Newcastle upon Tyne, UK, June, 2001, volume 2075 of Lecture Notes in Computer Science, pages 343-362.
- [2] R. Naumann, R. Rasche, and J. Tacke, "*Managing autonomous vehicles at intersections*". IEEE Intelligent Systems & Their Applications, Vol. 13, No. 3, May/June 1998.
- [3] C.A. Lakos, "*Composing Abstractions of Coloured Petri Nets*". In Proc. of 21st Int. Conference on Application and Theory of Petri Nets (ICATPN'2000) Aarhus, Denmark, June, 2000, volume 1825 of Lecture Notes in Computer Science, pages 323-342.
- [4] F. Stappert, P. Altenbernd. "Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs". *Journal of Systems Architecture*, 46(4): 339-355. Kluwer Academic Publishers 2000.
- [5] E. Lee et al., "Overview of the Ptolemy Project". Technical Memorandum UCB/ERL M01/11, Dept. EECS, University of California, Berkeley, 2001.
- [6] "The Moses Project". <http://www.tik.ee.ethz.ch/~moses/>.
- [7] Jakob Engblom, Andreas Ermedahl. "Modeling Complex Flows for Worst-Case Execution Time Analysis". *Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00)*. 2000.
- [8] Jakob Engblom, Andreas Ermedahl, Friedhelm Stappert. "A Worst-Case Execution-Time Analysis Tool Prototype for Embedded Real-Time Systems". *Workshop on Real-Time Tools*. Aalborg, Denmark. 2001.
- [9] Luigi Carro, Flavio R. Wagner, Marcio Kreutz, Marcio Oyamada. "A Design Methodology For Embedded Systems Based On Multiple Processors". *Proc. Distributed And Parallel Embedded Systems*, Paderborn, Germany. Kluwer Academic Publishers. 2001.
- [10] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000.