Integrating Load Balancing into Petri-Net based Embedded System Design *

Carsten Rust and Friedhelm Stappert
C-LAB, Universität Paderborn
Fürstenallee 11, D-33102 Paderborn
{fst|car}@c-lab.de

Stefan Schamberger
Universität Paderborn
Fürstenallee 11, D-33102 Paderborn
schaum@uni-paderborn.de

Abstract

This paper introduces a Petri net model of a load balancing component for dynamically modifiable embedded real-time systems. The presented work contributes to the extension of an existing Petri net based design methodology towards the handling of dynamically modifiable systems. Modeling the load balancing algorithm in the same formal model as the other components of the system under construction enables the engineer to simulate and analyze the overall system. This paper presents our load balancing model, gives a brief overview on diffusive load balancing algorithms and describes their integration into the application. Furthermore, some first results with our approach are presented.

1. Introduction

Embedded systems have evolved from small systems realized with very limited hardware devices to complex heterogeneous systems realized on a set of interconnected computing devices. Furthermore, there is a trend towards an increasing degree of dynamics in embedded systems. In several application areas, scenarios including dynamically modifiable components are discussed. As an example, consider an adaptive robot control, where components of the control software are changed at run-time due to results of online learning algorithms. Also in traditional application domains like automotive systems, scenarios are discussed where software components are replaced by other components during run-time, for instance within the infotainment system. Another typical example for a system with dynamically changing structure is a mobile ad hoc network (MANET). Possible examples for hosts building a MANET are mobile phones, PDA's, or robots that cooperatively solve a task.

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

An essential step in the design of embedded systems is the allocation of tasks to computing devices. Usually, the allocation is computed statically in advance. For dynamic systems however, an online component for load balancing has to be added, which is responsible for modifying the intial allocation if necessary. In this paper we present our Petri net realization of a load balancing algorithm for dynamically modifiable embedded systems. The presented approach is integrated into an existing design methodology for distributed embedded real-time systems including dynamically evolving components. The methodology covers the whole design flow, reaching from modeling of a system on an abstract level via analysis and partitioning down to the synthesis of an implementation on a given target architecture. The main characteristic of the methodology is that during analysis and synthesis the whole system under construction is present in one uniform formal model, a High-Level Petri net model. This enables evaluation and analysis of all components including their interaction. When it comes to the realization of a dynamic system, we have to take into account that load balancing – opposed to static design steps like scheduling and allocation – is a part of the implementation. Hence, it should be subject to analysis methods like the other parts. In order to integrate load balancing into the overall design, we have to build a model in our Petri Net formalism, which is the topic of this paper.

Many design methodologies and modeling means for embedded real-time systems exist, based on a variety of formal models. Several methodologies originate from the area of hardware/software-codesign, e. g. Ptolemy [2] and SPI [4]. These systems usually provide only limited support for dynamic systems. Ptolemy for instance allows for creating tasks at run-time, but they cannot properly be connected to existing tasks. Consequently, other design approaches do – to our knowledge – not consider the integration of load balancing into the overall system design as

proposed in this paper.

In the remaining sections of the paper, we first provide some background concerning the existing design methodology (Section 2) and the applied load balancing strategy (Section 3). Our Petri Net-model for load balancing and its integration into an application is presented in Section 4. In Section 5, first results with our approach are described.

2. Design Methodology

In this section we give a brief overview of the methodology to which the presented approach contributes and the underlying Petri netformalism. The methodology proposes the design flow depicted in Figure 1. It is divided into the three stages *Modeling*, *Analysis and Partitioning*, and *Synthesis*. 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. 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.

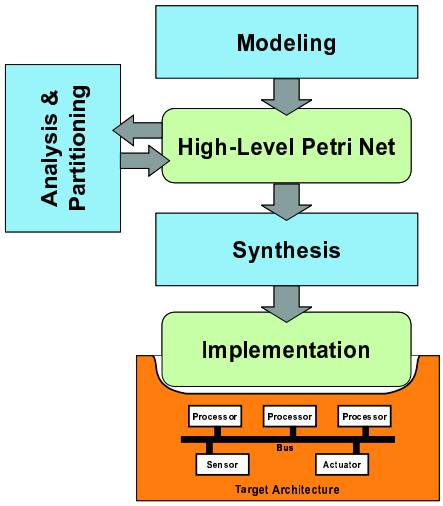

Figure 1. Design Flow

A more detailed overview of the methodology for the design of static systems without dynmically changing components is for instance given in [8]. In [7] ideas for the extension to dynamic systems are presented. This extension has an impact on all stages of the design flow. First, the underlying Petri Netformalism – which is described below – has to be extended with constructs for self modification in order to obtain an adequate modeling means. For the enhanced model, new tools for simulation, analysis and synthesis are needed. As to synthesis the main task is to extend existing tools by run-time services for the management of dynamic reconfigurations. One of these services is the component for load balancing described in this paper.

The underlying Petri net formalism of the methodology is a high-level model. 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 [5]. 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 and routed to output places according to variable expressions annotating the output edges of the transition. An example for a high-level Petri net is the net in Figure 2, our specification of the load balancing component. It will be explained in Section 4.

Beyond standard constructs of high-level Petri nets, our formal model includes a hierarchy concept in order to support easy modeling of complex systems as well as description means for timing to Pr/T–Nets. Furthermore, we added constructs for specifying dynamic modifications of the model. The resulting self-modifying net model allows to annotate transitions with rules for net transformations. Firing a suchlike annotated transition leads to a corresponding modification in the current net. The modification can be an instanciation of a subnet, but also a structural modification of an existing subnet. For more details we refer to [6].

3. Diffusive Load Balancing

To distribute the computational load, we apply diffusive load balancing schemes. These schemes work in it-

erations and require communication with adjacent nodes only. This is important since no additional routing mechanisms are needed. Furthermore, it has been shown that these schemes always compute the l_2-minimal flow ensuring a small number of migrations and also work in inhomogenous and asynchronous networks. Due to the space restrictions we will only give a brief overview to this topic and refer the reader to [1] and the references therein.

Let $G = (V,E)$ be the connected, undirected graph representing the network topology. We denote the nodes' computing power with $p_v \in \mathbb{R}, v \in V$ and the links' communication costs with $c_e \in \mathbb{R}, e \in E$. Let $w_v \in \mathbb{R}$ be the work load of node $v \in V$ and $\overline{w} := \sum w_v / \sum p_v \cdot (p_1, \ldots, p_{|V|})$ the vector of the propotional balanced load. The task is now to compute a balancing flow $x \in \mathbb{R}^{|E|}$, such that $Ax = w - \overline{w}$ with $A \in \{-1,0,+1\} \in \mathbb{R}^{|V|\times|E|}$ defined as the node edge incidence matrix of G.

The simplest diffusive method also called First–Order–Scheme (FOS) performs on each node $v_i \in V$ the iteration:

$$\begin{aligned} \forall e = (v_i, v_j) \in E: \quad y_e^{k-1} &= \frac{\alpha_e}{c_e}\left(\frac{w_i^{k-1}}{p_i} - \frac{w_j^{k-1}}{p_j}\right) \\ x_e^k &= x_e^{k-1} + y_e^{k-1} \\ \text{and} \quad w_i^k &= w_i^{k-1} - \sum_{e=(v_i,v_j)\in E} y_e^{k-1} \end{aligned} \tag{1}$$

where y_e^k is the amount of load sent via edge e in iteration k and the α_e are properly chosen parameters, e.g. $\alpha_{(v_i,v_j)} = 1/(1+deg(v_i))$. In matrix notation, this can be written as $w^k = Mw^{k-1}$ with the diffusion matrix $M = I - \alpha\tilde{L}P^{-1} \in \mathbb{R}^{|V|\times|V|}$. Here, $L = \tilde{A}\tilde{A}^T$ is the generalized Laplacian, meaning $\tilde{A} = AC^{-1}$ and $C = diag(\sqrt{c_e}) \in \mathbb{R}^{|E|\times|E|}$, and $P = diag(p_v) \in \mathbb{R}^{|V|\times|V|}$.

A quadratic increase in the convergence rate can be achived applying the Second-Order-Scheme (SOS). This scheme takes the work load of the previous iteration into account. In matrix notation, it can be written as $w^1 = Mw^0$; $w^k = \beta Mw^{k-1} + (1-\beta)w^{k-2}$. It converges for $\beta \in (0,2)$ and needs the least number of iterations for $\beta = 2/\left(1+\sqrt{1-(\frac{\lambda_m-\lambda_2}{\lambda_m+\lambda_2})^2}\right)$, where $0 = \lambda_1 < \lambda_2 < \ldots < \lambda_m$ are the m distinct eigenvalues of $\tilde{L}C^{-1}$.

The knowledge of all distinct non-zero eigenvalues λ_k of $\tilde{L}C^{-1}$ is needed for the Optimal-Scheme (OPT). Replacing α_k by $1/\lambda_k$ in equation (1), only $m-1$ iterations are needed to fully balance the load. If the topology is fixed (e.g. some hardware), one can precompute the eigenvalues and store them on the nodes. Furthermore, it is also possible to look for topologies with a small number of distinct eigenvalues and thus allowing efficient load balancing.

4. Petri-Net Specification

This section presents our Petri net implementation of the FOS scheme described in the previous section. In the application scenarios we are aiming at, it is not necessary to get equal load on all nodes. Instead, it is sufficient to reach a load situation where each node can perform all its tasks fast enough, meeting all corresponding real-time constraints. The iterative process of load distribution is started when the load of one or more nodes has become inacceptably high, i.e. its load exceeds a certain threshold. The process is stopped when no node is loaded too heavily, i.e. its load does not exceed the threshold. The load of a given node may change either during load balancing when it receives load from its neighbors, or internally due to dynamic modifications in the subsystem realized on the respective node.

The FOS calculation proposed in this paper is implemented completely asynchronous. Each node decides on its own whether a load distribution should be initiated. If it does so, it reads the current load situation from all its

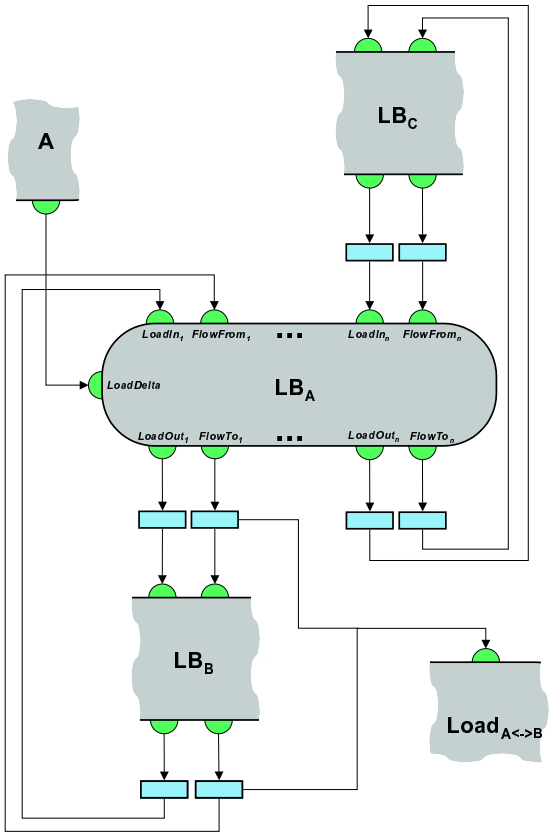

Figure 2. Top-Level Petri Net

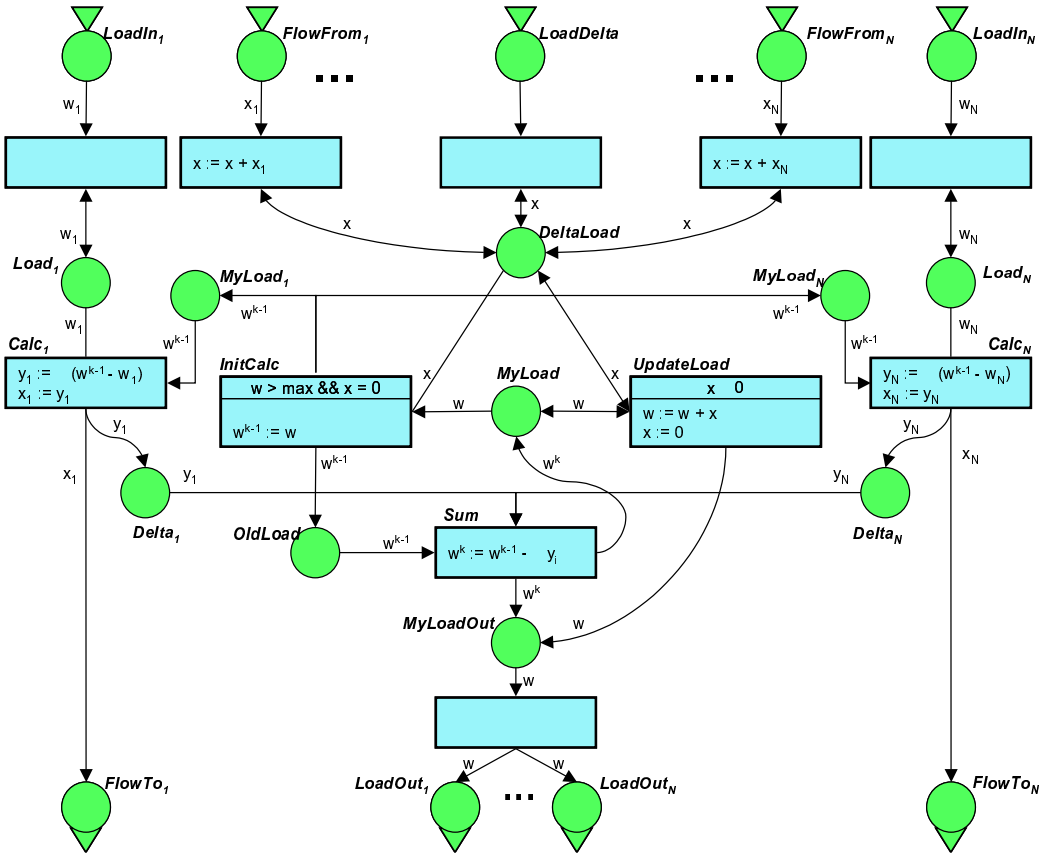

Figure 3. FOS Petri Net

neighbors and calculates the flow that should be sent to each of them. This, in turn, may trigger the distribution process in some of the neighbor nodes. This process continues until all involved nodes have reduced their load below the given threshold. Thus, no central instance is needed to coordinate the activities of the nodes.

In Figure 2 the top-level view of our Petri net model is depicted exemplarily. It shows a load balancing component LB_A of a Petri net A. The net is connected to its immediate neighbors LB_B and LB_C by means of the input-ports $LoadIn_i$ and $FlowIn_i$ from which it reads their load and flow, i.e. the amount of load the corresponding neighbor has sent. The net's own values for load and flow are sent to the corresponding neighbors via the output-ports $LoadOut_i$ and $FlowTo_i$. When the load of net A changes internally, the load-balancing component LB_A is informed via the input-port $LoadDelta$. Furthermore, for each neighbor N of A the component $Load_{AN}$ (depicted for neighbor B only in Figure 2) is informed whenever load has to be moved from one node to the other. While LB_A only computes the values for the load change, $Load_{AN}$ is responsible for the actual movement of computational load, i.e. it decides which processes shall migrate from one node to the other. The movement of computational load is specified by Petri net transformations as they were described in Section 2. Their realization in the implementation is beyond the scope of this paper and not further explained here. But without going into the details of the implementation it is obvious that not every arbitrary flow value calculated by the diffusive schemes described above can be realized. Instead, the net transformation is chosen, which realizes the greatest flow smaller than the calculated value. We always realize flows less or equal than the optimal value, since the choice of a greater flow often leads to circular load balancing operations. The difference between a calculated flow value and the realized one is sent to the load balancing component via the $LoadDelta$ port.

Figure 3 shows the Petri net model specifying the calculations performed locally at each node. The variable names in the annotations of the edges and transitions correspond

to the nomenclature used in Section 3. For simplicity reasons, we assume that the computing power p_i for each node i and the communication cost c_e for each edge e are equal. Therefore, the corresponding factors are left out in the figure. As suggested in Section 3, α is set to $1/(1+deg(i))$ for each node i.

The current load situation of the node is stored in the place *MyLoad* and is also distributed to the neighbor nodes via the output-places $LoadOut_i$. If the net receives load from its neighbors via the input-places $FlowFrom_i$ or the load changes internally via the input-place *LoadDelta*, *MyLoad* is updated by the transition $UpdateLoad$.

A new iteration of FOS calculation is triggered by the transition *InitCalc* when it realizes that the current load is too high (condition '$w > max$'). *InitCalc* places a token with the current load value on each place $MyLoad_i$, thereby enabling the corresponding transitions $Calc_i$. Note that there is one place $MyLoad_i$ and one transition $Calc_i$ for each neighbor i of the given net. The transitions $Calc_i$ then compute the amount of load that should be sent to the corresponding neighbor i and store it in the place $FlowTo_i$. Furthermore, the computed values are collected and subtracted from the current load by the transition *Sum* which updates the current load value in place *MyLoad*.

5. Experiments

Currently we are evaluating our load balancing method by means of random graphs. Using a library with small Petri net components, a Petri net of arbitrary size is built up automatically. The generated net is organized into units whose load is estimated by the number of their transitions. The Petri nets are self-modifying, i. e. the load of each unit changes over time.

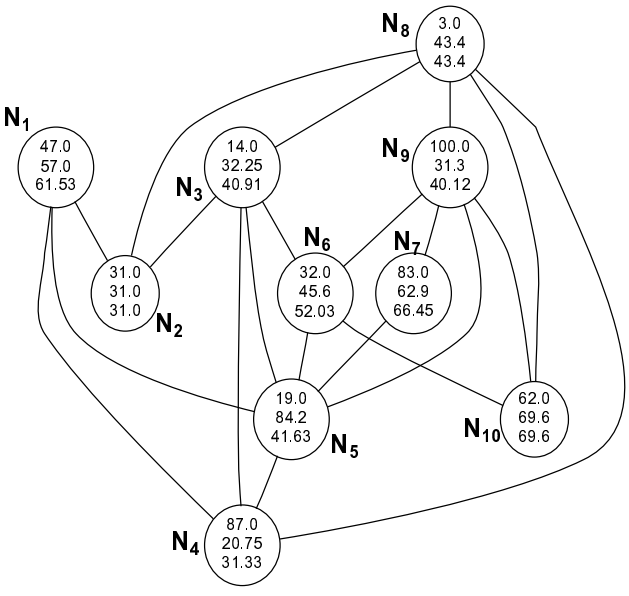

Figure 4. Example Graph

Figure 4 shows the top-level view of a generated net with ten units. Each node represents one unit and the edges of the graph represent neighbor relations between the subnets. The values annotating the graph nodes are load values for three successive situations. We assume that the maximal acceptable load value is 80. In the first situation, this value is exceeded for three nodes: N_4, N_7, and N_9. After one iteration of load balancing, i. e. after each of these nodes has sent load to its neighbours once, the second value is obtained. After a second iteration, the last values are reached and the algorithm terminates since a situation is reached where no node exeeeds the maximal load value of 80. Further iterations would lead to a situation where the load is distributed evenly over all nodes.

6. Conclusion and Future Work

We presented a Petri net model of a load balancing component for dynamically modifiable embedded real-time systems. The model itself as well as its integration into an application were described. Some first experiments with the load balancing implementation were described. As to the experimental results however, we only have shown so far that the implemented strategy works for generated random nets and leads to sufficiently balanced loads after a small number of iterations.

Future work includes performance investigations with very large applications. Such nets could easily be generated using our random net generation. Also, further investigations with real-world applications are planned, which may have additional properties not covered by the random nets we currently use.

Currently, we can only simulate top-level nets with static structure. For the applications we are aiming at, it is necessary to realize dynamic behavior – i.e. nodes can enter or leave the overall system – on this level, too. For example, when a node leaves the system, its entire load has to be moved to its neighbors at once, and the top-level net structure has to be reconfigured accordingly.

At the moment, we are working on the implementation of the $Load_{AN}$ component shown in Figure 2, i.e. the actual movement of Petri net components from one node to another.

References

[1] R. Elsässer, B. Monien, and R. Preis. Diffusion schemes for load balancing on heterogeneous networks. *Theory of Computing Systems*, 35:305–320, 2002.

[2] E. L. et al. Overview of the Ptolemy Project. Technical Memorandum M01/11, Dept. EECS, University of California, Berkeley, CA 94720, July 2001.

[3] K. Jensen and G. Rozenberg, editors. *High-Level Petri Nets*. Springer Verlag, 1991.

[4] M. Jersak, D. Ziegenbein, F. Wolf, K. Richter, R. Ernst, F. Cieslok, J. Teich, K. Strehl, and L. Thiele. Embedded System Design using the SPI Workbench. In *Proc. FDL'00, Forum on Design Languages 2000*, Tübingen, Germany, September 2000.

[5] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[6] F. Rammig and C. Rust. Modeling of Dynamically Modifiable Embedded Real-Time Systems. *accepted for* 9-th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003F), Oct. 2003.

[7] C. Rust, F. Stappert, and R. Bernhardi-Grisson. Petri Net Based Design of Reconfigurable Embedded Real-Time Systems. In *Distributed And Parallel Embedded Systems*, pages 41–50. Kluwer Academic Publishers, 2002.

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