Software Component Allocation on Heterogeneous Embedded Systems Using Coloured Petri Nets

Issam Al-Azzoni

Department of Software Engineering
College of Computer and Information Sciences
King Saud University
Riyadh, Saudi Arabia
Email: ialazzoni@ksu.edu.sa

Abstract—**In this paper, we present a new approach to component allocation in heterogeneous embedded systems using Coloured Petri Nets (CPNs). While several techniques in optimization exist to solve the component allocation problem, this is the first paper to develop a corresponding CPN model and outline a technique to find an optimal and feasible allocation. The CPN model represents an advancement towards a model-driven engineering view of the problem allowing to subject the model to other types of non-functional analysis. We also exploit the use of CPN Tools, a well-known tool for analyzing CPNs, in generating the state spaces and finding optimal allocations.**

Keywords–Component allocation; Coloured Petri Nets; Model-driven engineering; Embedded systems; Heterogeneous systems.

I. INTRODUCTION

Embedded systems have recently become ubiquitous. These systems contain multiple integrated software components and hardware computational units. An embedded system is a computer system, and its associated software, built into some piece of equipment [1]. There are numerous examples where embedded systems are being exploited, including telecommunication systems, household appliances, robots, automobiles, and airlines. These systems are also becoming more heterogeneous with the advent of several types of processors including Central Processing units (CPUs), Graphical Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs).

This heterogeneity has created new challenges for software architects and designers who decide the placement of the different software components on top of the hardware computational units. While there are many ways to place the components while meeting the functional requirements, the problem becomes much more complex when considering the non-functional (quality) aspects of the placement. For example, a particular mapping of the components may result in better performance than other mappings. The component allocation problem aims to find an allocation (mapping) of the software components such that a certain cost function is optimized. Strategies to solve the problem provide software architects with the necessary tools to make decisions on the allocation of components for heterogeneous embedded systems.

Model-driven engineering (MDE) advocates the use of models in systems analysis and design [2]. The use of models permits various types of analysis to be performed on the models before the actual system is implemented. This can be done at a high level of abstraction and in an automated fashion. While there exist several techniques for solving the component allocation as an optimization problem, this paper presents a new model-based approach for modeling and solving the component allocation problem. The new approach uses Coloured Petri Nets (CPNs) as the modeling language. CPNs have a very rich set of supporting theory and automated tools for model analysis [3]. By modeling the component allocation problem in CPNs, we not only can find an optimal allocation that optimizes a cost function, but also can subject the optimal allocation for other types of non-functional analysis including security and dependability analysis. CPNs have been applied extensively in analyzing non-functional aspects of systems [4][5]. In addition, different approaches exist to transfer other standard software modeling language models into Petri Net models (see the work of [6]).

The contributions of the paper are summarized as follows:

1) We describe a new approach to model the component allocation problem in CPNs.
2) We describe the use of CPN Tools [7] in analyzing the CPN model and solving the component allocation problem.

The organization of the paper is as follows. In Section II, we define the component allocation problem more formally. We illustrate our approach in Section III. In Section IV, we evaluate our CPN based approach using a realistic case study. The related work is discussed in Section V. Section VI concludes the paper and outlines future work.

II. PROBLEM DEFINITION

Consider a software system consisting of n components. Every component needs to be assigned to a computational unit on a hardware platform consisting of m computational units. The computational units offer a number of resources l (for example, computation, memory, and energy resources).

The Component Resource Consumption Matrix $T = [t_{ijk}]_{(n \times m \times l)}$ defines the amount of resources each component requires. The element t_{ijk} represents the necessary amount of the $k-th$ resource required by the $i-th$ software component when allocated on the $j-th$ computational unit.

The Computational Unit Resource Capacity Matrix $R = [r_{jk}]_{(m \times l)}$ defines the amount of resources that each computational unit can provide. The element r_{jk} represents the $k-th$ resource capacity of a $j-th$ computational unit.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-449-7

An allocation of the components maps each software component to one of the m computational units. Two or more components can be allocated on the same computational unit. From a mathematical viewpoint, an allocation represents a permutation with repetition which assigns one computational unit for each software component. Note that there are m^n possible allocations which implies that the search space increases exponentially with the number of components and computational units.

The component allocation problem is to find an allocation $(p_1, \cdots, p_n)$, where component i is assigned to computational unit p_i, such that it is both feasible and optimal. A feasible allocation means that the resources consumed by the software components allocated on any computational unit do not exceed the resource capacities that the computational unit provides. Thus, the feasibility condition can be stated as follows: given an allocation $(p_1, \cdots, p_n)$, for any computational unit j:

$$\sum_{i, p_i = j} (t_{ip_ik}) \leq r_{jk} \tag{1}$$

for all resources k.

In addition to satisfying (1), we might consider additional constraints that need to be satisfied by a feasible allocation. In this paper, we consider the system architectural constraint that in a feasible allocation a particular component should be (or should not) be allocated on a particular computational unit. There could be several of such architectural constraints that a feasible allocation needs to satisfy.

Given an allocation $(p_1, \cdots, p_n)$, its cost can be computed using the following cost function:

$$w = \sum_{k=1}^{l} f_k \sum_{i=1}^{n} t_{ip_ik} \tag{2}$$

Here, f_k represents a trade-off factor whose purpose is to specify the weights of each resource in the cost function. This allows to differentiate the importance of different resources. An optimal allocation has the smallest w (greater than 0). The component allocation problem is to find an optimal and feasible allocation. Thus, the chosen allocation needs to satisfy (1) (in addition to possibly additional constraints) and has the smallest cost w (greater than 0) which is defined by (2).

The component allocation problem can be formulated as a 0-1 integer linear programming problem which is NP-complete [8]. For exact solutions and small problem sizes (the problem size is based on the number of components and computational units), one can use traditional integer programming techniques. However, for large problem sizes, one needs to resort to heuristics which find good approximations through large space search methods.

III. Approach

In this section, we apply the CPN based approach to solve a component allocation problem using parameters of a realistic system borrowed from [9]. Section III-A gives a brief description of the system. In Section III-B, we develop a CPN model of the system and in Section III-C we describe the generation and analysis of the state space using CPN Tools.

$$\begin{bmatrix} 10 & 90 & 90 & 55 \\ 50 & 20 & 20 & 72 \\ 30 & 20 & 20 & 72 \\ 10 & 40 & 40 & 72 \\ 20 & 40 & 40 & 72 \\ 20 & 50 & 50 & 55 \\ 90 & 20 & 20 & 15 \\ 20 & 10 & 10 & 70 \\ 20 & 10 & 10 & 70 \\ 20 & 15 & 15 & 70 \\ 90 & 10 & 10 & 33 \end{bmatrix}$$

(a)

$$\begin{bmatrix} 48 & 256 & 256 & 128 \\ 128 & 256 & 256 & 148 \\ 64 & 256 & 256 & 148 \\ 48 & 168 & 168 & 148 \\ 64 & 168 & 168 & 148 \\ 64 & 168 & 168 & 64 \\ 168 & 128 & 128 & 64 \\ 148 & 96 & 96 & 148 \\ 48 & 32 & 32 & 148 \\ 48 & 32 & 32 & 148 \\ 168 & 64 & 64 & 96 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 2 & 18 & 18 & 11 \\ 10 & 4 & 4 & 14 \\ 6 & 4 & 4 & 14 \\ 2 & 8 & 8 & 14 \\ 4 & 8 & 8 & 14 \\ 4 & 10 & 10 & 11 \\ 18 & 4 & 4 & 3 \\ 4 & 2 & 2 & 14 \\ 4 & 2 & 2 & 14 \\ 4 & 3 & 3 & 14 \\ 18 & 2 & 2 & 7 \end{bmatrix}$$

(c)

Figure 1. The component resource consumptions.

A. Case Study

To demonstrate our approach, we borrow the same parameters used to develop a component allocation problem from [9]. The system considered is a software system that handles and interprets vision data on an autonomous underwater vehicle (AUV) while simultaneously interacting with them in real time. That system is being developed as a part of RALF3 project [10].

The system consists of $n = 11$ components. These are: **1-UI** User Interface, **2-CH** Communication Handler, **3-MP** Message Parser, **4-MD** Manual Drive, **5-MM** Mission Manager, **6-MC** Movement Control, **7-V** Vision, **8-AC** Actuator Control, **9-SI** Sensors Layer 1, **10-S2** Sensors Layer 2, and **11-SF** Stream Filtering components. The hardware platform consists of $m = 4$ computational units. These are: **1-mCPU** Mulicore CPU, **2-FPGA** FPGA I, **3-FPGA** FPGA II, and **4-GPU** GPU. There are $l = 3$ resources: average execution time (measured in milliseconds), memory (measured in megabytes), and average energy consumption (measured in milliamperes per hour).

Figure 1 shows the component resource consumptions (*i.e.*, the elements of the matrix T). Since T is three-dimensional (components, computational units, resources), we use three matrices to display three different resources (*i.e.*, the third dimension): (a) average execution time, (b) memory, and (c) average energy consumption. The computational unit resource capacity matrix is given by:

$$R = \begin{bmatrix} 100 & 256 & 50 \\ 150 & 640 & 25 \\ 150 & 640 & 25 \\ 100 & 256 & 15 \end{bmatrix}$$

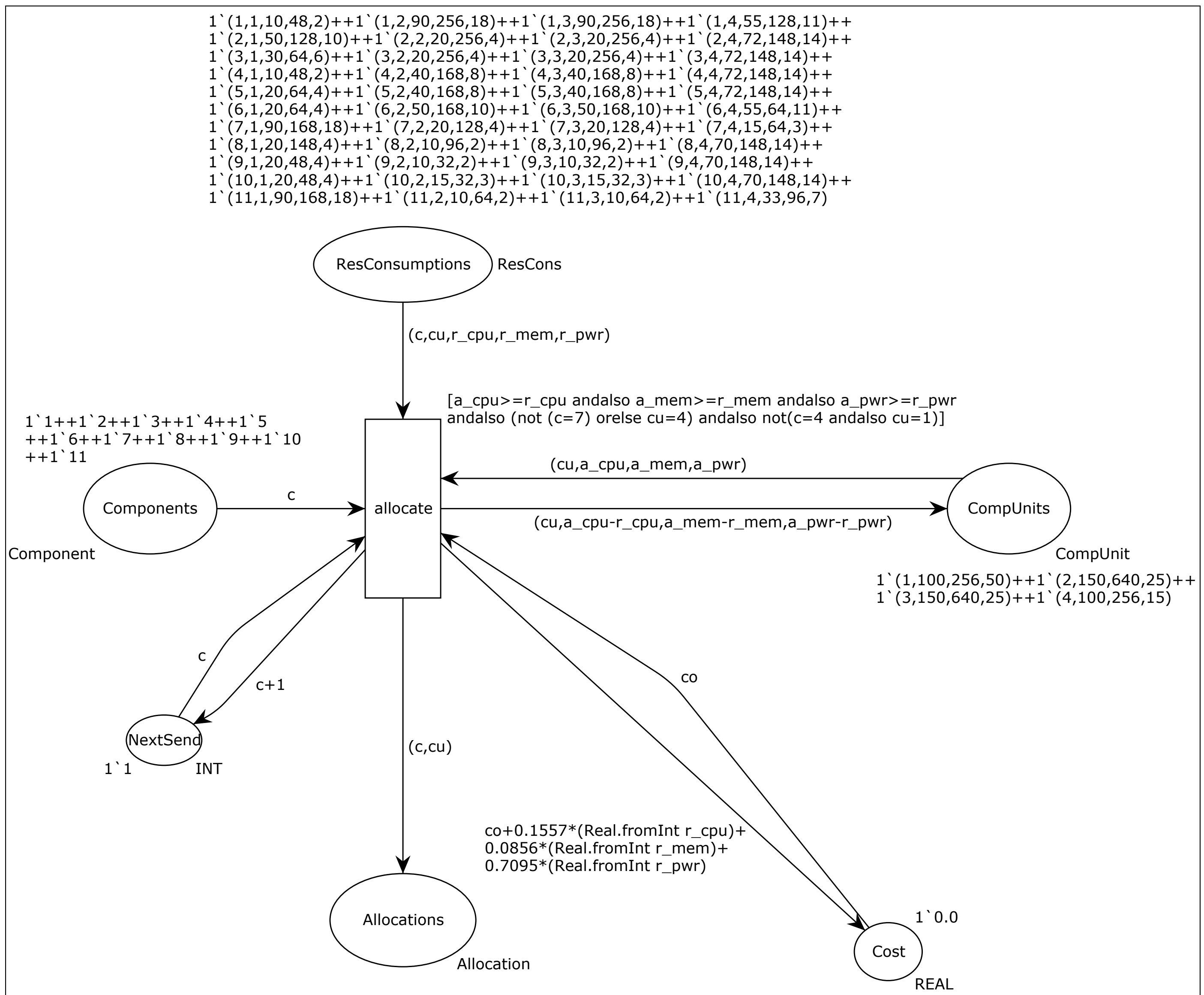

Figure 2. The CPN model for the system of the case study.

To compute the cost of an allocation in (2), we use the trade-off vector:

$$F = [\ 0.1557 \quad 0.0856 \quad 0.7095\ ]$$

Here, the k-th element in vector F represents the trade-off factor f_k. The trade-off factors are computed using Analytic Hierarchy Process (AHP) [11]. The details are given in [9].

We will consider two additional constraints:

- **Constraint I:** Component **7-V** should be allocated on **4-GPU**.
- **Constraint II:** Component **4-MD** should not be allocated on **1-mCPU**.

B. The CPN Model

The CPN model is shown in Figure 2. The CPN contains six places. The place $Components$ holds tokens which represent the components. The place $CompUnits$ holds tokens representing the computational units. Each token records the available resources that the corresponding computational unit currently has. The place $ResConsumptions$ holds tokens which encode the component resource consumption matrix T. The place $Allocations$ holds tokens which represent the allocations of components to computational units. The place $NextSend$ is used to control which component is to be allocated next. The place $Cost$ holds a single token which records the total cost of the allocated components. There is only one transition in the CPN. Firing the transition $allocate$ corresponds to assigning a component to one of the computational units.

The colour sets are defined as follows:

```
colset UNIT = unit;
colset INT = int;
colset REAL = real;
colset BOOL = bool;
colset STRING = string;
```

```
val max_val: real = 2000.0;


fun tot_cost n =
let
val accCostsToken = Mark.model'Cost 1 n;
in
hd(accCostsToken)
end;


fun DesiredTerminal1 n = (Mark.model'Components 1 n == empty);
val x = SearchNodes(EntireGraph, DesiredTerminal1, NoLimit, tot_cost, max_val, Real.min);
fun DesiredTerminal2 n = DesiredTerminal1(n) andalso (Mark.model'Cost 1 n == 1`x);
val y = SearchNodes(EntireGraph, DesiredTerminal2, NoLimit, fn n => n,[],op ::);



CalculateOccGraph();
```

Figure 3. The CPN ML queries used to generate and search through the state space for the CPN model of Figure 2.

```
colset Component = int;
colset CompUnit = product INT * INT * INT * INT;
colset Allocation = product INT * INT;
colset ResCons = product INT * INT * INT * INT * INT;
```

The variables are declared as follows:

```
var c,cu: INT;
var co:REAL;
var a_cpu,a_mem,a_pwr: INT;
var r_cpu,r_mem,r_pwr: INT;
```

The place $NextSend$ is used to reduce the state space by allocating the components in order of their numbers. This is valid since the order of assigning components to computational units does not matter with respect to the feasibility condition (see (1)).

The constraints are included in the CPN model by using the guard of transition $allocate$. For example, in Constraint I, Component **7-V** should be allocated on **4-GPU**. Thus, a feasible allocation of components should satisfy the condition that $(c = 7) \rightarrow (cu = 4)$ which is logically equivalent to $\neg(c = 7) \vee (cu = 4)$. For Constraint II, Component **4-MD** should not be allocated on **1-mCPU**. Thus, a feasible allocation of components should also satisfy the condition that $\neg((c = 4) \wedge (cu = 1))$. Both conditions are added to the guard of transition $allocate$.

When a component is allocated to a computational unit, the corresponding cost needs to be added to the total cost (the colour of the token in place $Cost$). This is modeled by using the arc from transition $allocate$ to place $Cost$. Note the trade-off factors f_k in the arc expression.

C. State Space Generation and Analysis

We use the state space tool of CPN Tools Version 4.0 to find an optimal and feasible component allocation. CPN Tools Version 4.0 adds the support for real colorsets. Figure 3 shows the query functions used to generate and search through the state space. These queries are written in the CPN ML programming language (presented in Chapter 3 in [3]). For a given marking represented by n, the function tot_cost returns the total cost of the assigned components which is equal to the value (colour) of the token in place $Cost$.

To find the optimal allocations, we use the CPN ML defined function $SearchNodes$ twice. First, we use it to find the minimum value for the total allocation cost over all markings which satisfy the predicate $DesiredTerminal1$. The predicate $DesiredTerminal1$ returns true if and only if the marking represented by n satisfies the condition that there is no token in place $Components$ (hence, all components have been assigned). Thus, the variable x stores the minimum total component allocation cost. The constant max_val is a large real number useful in the start of applying the combination function $Real.min$ of $SearchNodes$. The constant max_val can be set to any large real number, but one should ensure that it is larger than the cost of a single allocation chosen at random. Second, we use $SearchNodes$ to find the markings which satisfy $DesiredTerminal2$. The predicate $DesiredTerminal2$ returns true if and only if the marking represented by n satisfies $DesiredTerminal1$ and that the total allocation cost is equal to x. Thus, the output of the second $SearchNodes$ (stored in variable y) is the list of all markings corresponding to the optimal allocations. The optimal allocations are determined by examining the tokens in place $Allocations$ in any of such markings.

IV. EVALUATION

In this section, we show results of applying our approach on the case study presented in Section III-A. We use CPN Tools to create the corresponding CPN model as developed in Section III-B and analyze the generated state space as outlined in Section III-C.

Table I shows the evaluation results. The table includes the cost of an optimal and feasible component allocation computed by an exhaustive search. In addition, the table shows the optimal and feasible component allocation computed using the

CPN based approach, its cost w, and the time (in seconds) it took CPN Tools to generate the state space. Note that the state space generation was done on a Dell desktop computer equipped with a 3.00GHz dual-core processor and 2GB RAM. The table validates the CPN approach in the case study since the returned component allocation is optimal (its cost is equal to that of the optimal allocation returned by the exhaustive search) and feasible.

Table II shows the evaluation results for the same component model, but excluding **Constraint II**. To exclude this constraint, we remove the corresponding condition from the guard of transition $allocate$. The optimal cost found by the CPN approach is equal to that found by the exhaustive search. Note that the state space search time is almost three times worse than the previous result. This is explained by the increase in the size of the state space due to the exclusion of the constraint.

TABLE I. EVALUATION RESULTS.

Optimal Cost - Exhaustive Search	141.01
Optimal and Feasible Allocation - CPN Approach	(1,3,1,2,1,1,4,3,3,2,3)
Optimal Cost - CPN Approach	141.01
Number of Seconds - CPN Approach	44

TABLE II. EVALUATION RESULTS EXCLUDING CONSTRAINT II.

Optimal Cost - Exhaustive Search	132.23
Optimal and Feasible Allocation - CPN Approach	(1,3,1,1,1,4,4,2,2,2,3)
Optimal Cost - CPN Approach	132.23
Number of Seconds - CPN Approach	128

V. Related Work

The authors of [9] apply a genetic algorithm to find feasible, optimal solutions to the component allocation problem. Our model that defines the component allocation problem is based on the model presented in [9]. However, we do not consider communication costs between components. The authors also apply analytical hierarchical process to deal with the problem of different measurement units in calculating the trade-off factors. Genetic algorithms usually find good solutions; however, generally speaking there is no guarantee that these solutions are the optimal solutions. Compared to the CPN based approach presented in the paper, genetic algorithms scale well for large systems.

Another method for solving the component allocation problem is presented in [12]. The method uses branch-and-bound and forward checking mechanism. The method was implemented in the Automatic Integration of Reusable Embedded Software (AIRS) toolkit [13].

A generic framework aimed at finding the most appropriate deployment architecture (mapping of software components onto hardware resources) for a distributed software system is presented in [14]. The framework formally defines the allocation problem and provides a set of applicable algorithms for solving the problem. In addition, a tool suite is developed to enable the use of the proposed framework. The component allocation problem presented in this paper can be though of as a particular instantiation of the framework. In addition, the CPN based approach can supplement the solution algorithms presented in [14].

The authors of [15] present a formal model for allocation optimization of embedded systems which contains a mix of CPU and GPU processing nodes. The authors use mixed-integer nonlinear programming as the optimization model. In addition, the authors translate the model into a solver using a standard format called MPS (Mathematical Programming System) that can be interpreted using most solvers. The authors make the observation that the mixed-integer nonlinear programming solvers do not scale well for medium and large size problems.

Several approaches exist for component allocation in real-time embedded systems [16][17]. In real-time embedded systems, components (tasks) have additional attributes such that completion time, period, and deadline. The allocation problem for real-time embedded systems needs to ensure that tasks complete before their deadlines. Our CPN based approach uses a different component model which does not take these timing properties into account.

VI. Conclusion and Future Work

This paper has presented a new approach to component allocation using CPNs and CPN Tools. One potential limitation that needs to be considered in the future work is the exponential increase in the generated state space for larger systems. Techniques to scale the applicability of the CPN approach are needed. One approach is to determine an upper bound on the cost and only generate states having cost less than this upper bound. The upper bound can be guessed or can be determined using other optimization methods including genetic algorithms. Also, part of our future work should concentrate on automated methods for model transformation to/from other modeling languages, including the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [18]. Finally, the CPN models need to be analyzed in terms of other non-functional properties such as security and dependability. Future work should apply the new approach and its techniques on several realistic case studies.

References

[1] J. Carlson, J. Feljan, J. Mäki-Turja, and M. Sjödin, “Deployment modelling and synthesis in a component model for distributed embedded systems,” in Proceedings of the Conference on Software Engineering and Advanced Applications, 2010, pp. 74–82.

[2] B. Selic, “Model-driven development: Its essence and opportunities,” in Proceedings of The Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2006, pp. 313–319.

[3] K. Jensen and L. M. Kristensen, Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer, 2009.

[4] I. Al-Azzoni, D. G. Down, and R. Khedri, “Modeling and verification of cryptographic protocols using coloured petri nets and Design/CPN,” Nordic Journal of Computing, vol. 12, no. 3, 2005, pp. 201–228.

[5] L. Wells, “Performance analysis using Coloured Petri Nets,” in Proceedings of the Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002, pp. 217–221.

[6] S. Distefano, M. Scarpa, and A. Puliafito, “From UML to Petri Nets: The PCM-based methodology,” IEEE Transactions on Software Engineering, vol. 37, no. 1, 2011, pp. 65–79.

[7] CPN Tools. http://cpntools.org/ [Accessed December 2014].

[8] R. M. Karp, “Reducibility among combinatorial problems,” in Proceedings of the Symposium on the Complexity of Computer Computations, 1972, pp. 85–103.

[9] I. Švogor, I. Crnković, and N. Vrček, “An extended model for multi-criteria software component allocation on a heterogeneous embedded platform,” Journal of Computing and Information Technology, vol. 21, no. 4, 2013, pp. 211–222.

[10] RALF3 Project Web. http://www.mrtc.mdh.se/projects/ralf3/ [Accessed Aug 2014].

[11] T. L. Saaty, Fundamentals of Decision Making and Priority Theory with the Analytic Hierarchy Process. RWS Publications, 1994.

[12] S. Wang, J. R. Merrick, and K. G. Shin, “Component allocation with multiple resource constraints for large embedded real-time software design,” in Proceedings of the Real-Time and Embedded Technology and Applications Symposium, 2004, pp. 219–226.

[13] AIRES. http://kabru.eecs.umich.edu/bin/view/Main/AIRES [Accessed December 2014].

[14] S. Malek, N. Medvidović, and M. Mikic-Rakic, “An extensible framework for improving a distributed software system’s deployment architecture,” IEEE Transactions on Software Engineering, vol. 38, no. 1, 2012, pp. 73–100.

[15] G. Campeanu, J. Carlson, and S. Sentilles, “Component allocation optimization for heterogeneous CPU-GPU embedded systems,” in Proceedings of the Conference on Software Engineering and Advanced Applications, 2014, pp. 229–236.

[16] J. Fredriksson, K. Sandström, and M. Åkerholm, “Optimizing resource usage in component-based real-time systems,” in Proceedings of the Symposium on Component-based software engineering, 2005, pp. 49–65.

[17] I. Bate and P. Emberson, “Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems,” in Proceedings of the Real-Time and Embedded Technology and Applications Symposium, 2006, pp. 221–230.

[18] OMG, UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems, version 1.1, formal/11-06-02; June 2011.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-449-7