CIT. Journal of Computing and Information Technology, Vol. 26, No. 2, June 2018, 85–97
doi: 10.20532/cit.2018.1003892

An Improved Coloured Petri Net Model for Software Component Allocation on Heterogeneous Embedded Systems

Issam Al-Azzoni

College of Engineering, Al Ain University of Science and Technology, Al Ain, United Arab Emirates

We extend an approach to component allocation on heterogeneous embedded systems using Coloured Petri Nets (CPNs). We improve the CPN model for the embedded systems and outline a technique that exploits CPN Tools, a well-known CPN tool, to efficiently analyze embedded system's state space and find optimal allocations. The approach is model-based and represents an advancement towards a model-driven engineering view of the component allocation problem. We incorporate communication costs between components by extending the CPN formalism with a non-trivial technique to analyze the generated state space. We also suggest a technique to improve the state space generation time by using the branching options supported in CPN Tools. In the evaluation, we demonstrate that this technique significantly cuts down the size of the generated state space and thereby reduces the runtime of state space generation and thus the time to find an optimal allocation.

ACM CCS (2012) Classification: Software and its engineering → Software notations and tools → Context specific languages → Domain specific languages

Computer systems organization → Embedded and cyber-physical systems → Embedded systems → Embedded software

Keywords: component allocation, coloured Petri Nets, model-driven engineering, embedded systems, heterogeneous systems

1. Introduction and Related Work

Designers of embedded systems today face new challenges due to the high heterogeneity that characterizes such systems [1]. An embedded system may consist of several types of processors (or computational units) varying in terms of their performance, including Central Processing units (CPUs), Graphical Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs). Also, the software components which need to be allocated on top of the hardware computational units may differ in terms of their resource usage. Such kind of heterogeneity presents a challenge to the designers when deciding about the placement of software components on top of the computational units [2].

The component allocation problem finds optimal allocations or mappings of the software components to the computational units [3]. While several allocations can be functionally correct in terms of being feasible, one or more of these allocations can have better non-functional (quality) aspects than the remaining allocations. Finding the allocation that is functionally correct and that maximizes a certain quality metric is at the heart of solution methods to the component allocation problem. The component allocation problem can be formulated as an integer linear programming problem and thus there are several solution methods in mathematical optimization to the component allocation problem [4], [5].

In the conference version of this paper [6], we proposed an approach to address the component allocation problem by using Coloured Petri Nets (CPNs) to model the embedded system. The CPN model can be dynamically analyzed by searching through its state space to determine an optimal allocation. CPNs have a very

rich set of supporting theory and automated tools for model analysis [7], [8]. In this paper, we extend the work presented in [6]. The new contributions of this paper are summarized as follows:

1. We incorporate the communication costs between software components in the problem definition and the CPN model. In [6], the communication costs were not included.
2. We make several changes to the CPN model in order to improve the state space generation time.
3. We modify the CPN model and the CPN ML queries to incorporate the communication costs. This is not trivial, since it cannot be directly accounted for in the CPN body similar to the non-communication resource costs.
4. We run several more experiments to verify the optimal allocations found using our CPN approach.
5. We suggest a technique to scale the CPN approach to larger systems by using the branching options in CPN ML state space generation tool. This is presented in Subsection 3.3.

The use of a model-based approach has several benefits. For example, the same CPN model can be used to address the optimal allocations for other types of non-functional analysis, including security and dependability. CPNs have been applied extensively in analyzing non-functional aspects of systems [9], [10]. Furthermore, models of the embedded system in different notations can be automatically transformed into equivalent CPN models. Thus, the main contribution of devising a CPN model to address the component allocation problem in heterogeneous embedded systems is the application of a model-driven engineering (MDE) approach to the problem.

MDE advocates the use of models in systems analysis and design [11], [12], [13]. 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. In [14], the authors propose an approach for the automatic transformation from an Ecore-based model of a component allocation problem into an equivalent CPN model. The resulting CPN model can be analyzed using the method presented in this paper. This allows to identify a component allocation problem and to solve it without having to know about CPNs.

The authors of [3] apply a genetic algorithm to find optimal solutions to the component allocation problem. Our model that defines the component allocation problem is based on the model presented in [3]. The authors also apply analytical hierarchical process to calculate the trade-off vector. Genetic algorithms usually find good solutions; however, generally speaking, there is no guarantee that these solutions are the optimal solutions. A prototype tool that implements the genetic algorithm is presented in [15]. The tool is named SCALL (Software Component ALLocator for Heterogeneous Embedded Systems). SCALL is developed as an Eclipse plugin utilizing Eclipse Modeling Framework (EMF) and Graphical Modeling Project (GMP). SCALL is based on using a metamodel for the software component allocation problem specified in Ecore notation. The user of SCALL can create a model for a software component allocation problem in a drag-and-drop fashion from a palette. SCALL then returns an optimal allocation computed using the genetic algorithm presented in [3].

A model-driven engineering approach for component allocation is presented in [16]. The approach allows to specify allocation constrains in ASL (Allocation Specification Language) that uses OCL operations. Then, the feasible allocations can be derived automatically without having to know how to encode and solve the allocation problem as an integer linear program (ILP). This is achieved by using model-to-model transformation that generates models for ILPs solvable by an ILP solver.

Another method for solving the component allocation problem is presented in [17]. The method uses branch-and-bound and forward checking mechanisms. The method was implemented in the Automatic Integration of Reusable Embedded Software (AIRS) toolkit [18].

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 [2]. The framework formally defines the

component 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 thought of as a particular instantiation of the framework. In addition, the CPN based approach can supplement the solution algorithms presented in [2].

Another framework for modeling and analyzing the component allocation problem in heterogeneous computing systems is presented in [19]. The framework is called LOSECO (an al**lo**cation of parallel **s**oftware to h**e**terogeneous **co**mputing platform framework). In LOSECO, the software execution units, which correspond to components in our model, can have precedence relationships amongst each other. Our component model assumes that the components are independent. The authors propose a partitioning based allocation heuristic which partitions the graph representing the software execution units and their dependencies into multiple smaller subgraphs. Subsequently, each subgraph is solved using heuristics such as genetic algorithms.

The authors of [1] present a formal model for allocation optimization of embedded systems which contain 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 [11], [20], [21], [22]. In real-time embedded systems, components (tasks) have additional attributes such as completion time, period, and deadline. The allocation problem for real-time embedded systems needs to ensure that tasks are completed before their deadlines. Our CPN based approach uses a different component model which does not take these timing properties into account.

In [23], graph theory is used to address the software component allocation problem in automotive networked embedded systems. The components communicate with each other via signals that can be periodic or sporadic. The presented algorithms minimize the total communication cost only and are based on graph partitioning theory. Our component model does not include signals. The CPN approach presented in this paper minimizes an objective cost function that includes multiple resources, including communication.

The organization of the paper is as follows. In Section 2, we define the component allocation problem more formally. We illustrate our approach in Section 3. In Section 4, we evaluate our CPN based approach. Section 5 concludes the paper and outlines future work.

2. 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). Our model for the component allocation problem is based on [3].

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.

To incorporate the cost of communication between the software components, we define two matrices. The first is the Communication Intensity Matrix $K = [k_{ij}]_{(n \times n)}$, where k_{ij} represents the communication intensity between the i-th and j-th components. If the components i and j are not communicating, then $k_{ij} = 0$. Also, notice that the matrix K is symmetric since the direction of communication is assumed to be not relevant. In addition, the diagonal elements of k are all equal to zero. The second matrix is the Platform Communication Cost Matrix

$C = [c_{ij}]_{(m \times m)}$, where c_{ij} represents the communication cost between the i-th and j-th computational units. For $i = j$, $c_{ij} = 0$. The inclusion of both matrices is necessary since the total communication cost depends on the communication intensity between the components in addition to the platform characteristics of the communication channels connecting the computational units.

An allocation to the components maps each software component to one of the m computational units. One 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 to 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.

Consider an allocation $(p_1, ..., p_n)$, where component i is assigned to computational unit p_i. An allocation is called feasible if the resources consumed by the software components allocated to any computational unit do not exceed the resource capacities that the computational unit provides. More formally, for any computational unit j, a feasible allocation satisfies the condition:

$$\sum_{i, p_i = j} \left(t_{ip_{ik}} \right) \leq r_{jk} \qquad (1)$$

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 (or should not) be allocated to a set of computational units. There could be several of such architectural constraints that a feasible allocation needs to satisfy.

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

$$w = \sum_{k=1}^{l} f_k \sum_{i=1}^{n} t_{ip_i k} + f_c \sum_{i \leq j} k_{ij} c_{p_i p_j} \qquad (2)$$

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. Similarly, f_c is the communication trade-off factor.

The component allocation problem is to find an optimal allocation. An optimal allocation is a feasible allocation that has the smallest w amongst all feasible allocations. Thus, the chosen allocation needs to satisfy (1) (in addition to possibly additional constraints) and has the smallest cost w which is defined by (2).

The component allocation problem can be formulated as a $0 - 1$ integer linear programming problem which is NP-complete [24]. 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.

3. Approach

In this section, we apply the CPN based approach to solve a component allocation problem using parameters of a realistic system borrowed from [3]. Subsection 3.1 gives a brief description of the system. In Subsection 3.2, we develop a CPN model of the system and in Subsection 3.3, we describe the generation and analysis of the state space using CPN Tools. Subsection 3.4 summarizes the approach.

3.1. Case Study

To demonstrate our approach, we borrow the same parameters used to develop a component allocation problem from [3]. 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 [25].

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 compo-

nents. The hardware platform consists of $m = 4$ computational units. These are: **1-mCPU** Multicore 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).

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

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

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

Figure 1. The component resource consumptions.

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:

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

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 5 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 5 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 1 & 7 & 3 & 0 & 0 & 0 \\ 0 & 3 & 3 & 0 & 0 & 9 & 9 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 9 & 0 & 0 & 0 & 7 & 7 & 0 \\ 0 & 0 & 0 & 7 & 9 & 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 3 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2. The communication intensity matrix K.

Figure 2 shows the communication intensity matrix. The platform communication cost matrix is given by:

$$C = \begin{bmatrix} 1 & 5 & 5 & 4 \\ 5 & 1 & 2 & 3 \\ 5 & 2 & 1 & 3 \\ 4 & 3 & 3 & 1 \end{bmatrix}$$

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

$$F = \begin{bmatrix} 0.1557 & 0.0856 & 0.7095 & 0.0491 \end{bmatrix}$$

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) [26]. The last element in F is the communication trade-off factor f_c. The details are given in [3].

We will consider two additional constraints:

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

3.2. The CPN Model

The CPN model is shown in Figure 3. The CPN contains four places. Here, we briefly describe each place. 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 *Allocations* holds tokens which represent the allocations of components to computational units. The place *Cost* holds a single token which records the total cost of the allocated components, excluding the communication costs. 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:

```
acolset UNIT = unit;
colset INT = int;
colset REAL = real;
colset BOOL = bool;
colset STRING = string;
colset Component = int;
colset CompUnit = product INT * INT
                * INT * INT;
colset Allocation = product INT * INT;
```

The colour set *CompUnit* is defined as the product of four integer colour sets. This is the colour set for the place *CompUnits* holding tokens that record the available resources in each computational unit. In each such token, the colours are ordered as follows: the computational unit id, the available average execution time resource, the available memory resource, and the available average energy consumption resource.

The variables are declared as follows:

```
avar c,cu: INT;
var co:REAL;
var a_cpu,a_mem,a_pwr: INT;
```

The variables *c* and *cu* hold the component and computational unit ids, respectively. The variable *co* holds the total cost of the allocated components, excluding the communication costs. The variables *a_cpu*, *a_mem*, and *a_pwr* hold

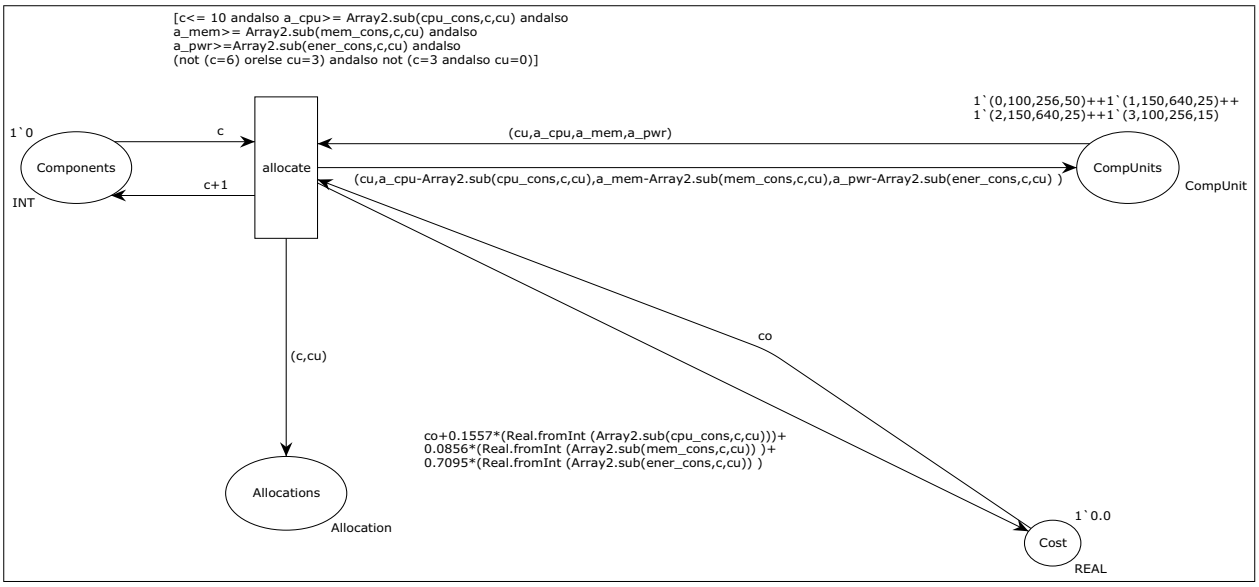

Figure 3. The CPN model for the system of the case study.

the available average execution time, memory, and average energy consumption resources, respectively.

To encode the component resource consumption matrix *T*, we define three two-dimensional arrays: *cp_cons*, *mem_cons*, and *ener_cons*. This is done by using the function *fromList* defined on *Array*2 structures in SML library. For example, the array *cpu_cons* is defined using the following:

```
aval cpu_cons = Array2.fromList(
[[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]]);
```

Components are allocated one by one, in order of their ids. This is valid, since the order of assigning components to computational units does not matter with respect to the feasibility condition (see (1)). The assignment of components is controlled by the value of the token residing in place *Components*. Note that the component ids and the computational unit ids start from zero. Thus, for example, the component with id = 0 corresponds to the component **1-UI** and the computational unit with id = 0 corresponds to the computational unit **1-mCPU**.

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 $(c = 6) \rightarrow (cu = 3)$ which is logically equivalent to $\neg (c = 6) \vee (cu = 3)$. For Constraint II, Component **4-MD** should not be allocated to **1-mCPU**. Thus, a feasible allocation of components should also satisfy the condition that $\neg ((c = 3) \wedge (cu = 0))$. 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.

3.3. State Space Generation and Analysis

We use the state space tool of CPN Tools Version 4.0 to find an optimal component allocation. CPN Tools Version 4.0 adds the support for real colorsets. Figure 4 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 [8]). For a given marking repre-

```
val max_val: real = 2000.0;

fun alloc (x,y) = y;

fun comm_cost n =
let
val allocation = ext_col alloc (Mark.model'Allocations 1 n);
val allocation_list = ms_to_list(allocation);
val comm_cost = Array2.array(11,11,0);
val reg = {base=comm_cost, row=0, col=0, nrows=NONE, ncols=NONE};
fun c(i,j,k) = Array2.sub(comp_comm2,i,j)*Array2.sub(unit_comm_cost, List.nth(allocation_list,i),  List.nth(allocation_list,j) );
val u = Array2.modifyi Array2.RowMajor c reg;
fun s(a,b) = a+b;
in
Array2.fold Array2.RowMajor s 0 comm_cost
end;


fun tot_cost n =
let
val accCostsToken = Mark.model'Cost 1 n;
in
hd(accCostsToken) + 0.0491*Real.fromInt(comm_cost(n))
end;


fun DesiredTerminal1 n = (Mark.model'Components 1 n == 1`11);
val x = SearchNodes(EntireGraph, DesiredTerminal1, NoLimit, tot_cost,max_val,Real.min);


fun DesiredTerminal2 n = DesiredTerminal1(n) andalso tot_cost(n) = x;
val y = SearchNodes(EntireGraph, DesiredTerminal2, NoLimit, fn n => n,[],op ::);
```

Figure 4. The CPN ML queries used to generate and search through the state space for the CPN model in Figure 3.

sented 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* plus the total communication cost multiplied by the communication trade-off factor $f_c = 0.0491$.

The function *comm_cost* returns the total communication cost for an allocation. This is implemented in three steps. First, the allocation corresponding to the marking *n* is determined. Note that the place *Allocations* contains tokens of colour set *Allocation* which is defined as the product of two integer colour sets. Thus, each token is a tuple containing two integers: one representing the component and one representing its assigned computational unit. By applying the linear extension of the function *alloc* to the marking of place *Allocations* and converting the resulting multi-set to list, *allocation_list* is determined. Second, the elements of the two-dimensional array *comm_cost* are determined. Each element $[i, j]$, where $i < j$ represents the communication cost between components *i* and *j* (all other elements are set to zero). It is calculated using the standard SML function *Array*2.*modifyi* which applies the function *c* to each element of *comm_cost*. Note that the two-dimensional array *comp_comm*2 is defined as the strictly lower triangular version of the matrix *K*, while the two-dimensional array *unit_comm_cost* is defined as the matrix *C*. Finally, communication costs are summed up using the function *Array*2.*fold* which folds the function *s* over the elements of *comm_cost* to compute the total communication cost.

```
OGSet.BranchingOptions{
TransInsts = NoLimit, Bindings =
NoLimit,
Predicate = fn n => (tot_cost(n) <=
150.88)};
```

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 *DesiredTerminal*1. The predicate *DesiredTerminal*1 returns true if and only if the marking represented by *n* satisfies the condition that the token in place *Components* has value 11 (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 for 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 *DesiredTerminal*2. The predicate *DesiredTerminal*2 returns true if and only if the marking represented by *n* satisfies *DesiredTerminal*1 and that if 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.

One technique to scale the applicability of the CPN approach is to determine an upper bound on the total cost and only generate markings having total cost less than this upper bound. This is possible in CPN ML by using the *OGSet.BranchingOptions* function as in the following example:

The branching options are used to specify the conditions under which the successors of a node (marking) are calculated. In this example, if a marking corresponds to an allocation with a total cost that exceeds 150.88, the successors of this marking are not calculated. The rationale of the use of this upper bound (150.88) is to be explained shortly. The effect is that only the allocations whose total cost does not exceed the upper bound are explored. This results in significant reduction in the size of the generated state space. Applicable heuristics can be used to determine appropriate values for the upper bound. For example, we use the genetic algorithm developed in [3]. We note that heuristics provide approximate solutions and may not converge into an optimal solution. This should not pose a problem when setting the branching options, since the upper bound needs not be the optimal solution.

3.4. Summary

The following summarizes the main steps developed in this section:

1. Creating the CPN model: The modeler can use the CPN model in Figure 3, but (only) after updating the trade-off factors in the expression of the arc from place *Cost* to transition *allocate*, the additional constraints and number of components in the guard of transition *allocate*, and the tokens in place *CompUnits* to match the computational units' resource capacities.
2. Generating the corresponding state space using CPN Tools: The modeler first needs to define the arrays *cpu_cons*, *mem_cons*, and *ener_cons* as explained earlier.
3. Running CPN ML queries to search through the state space in order to find an optimal allocation: The modeler can use the CPN ML queries presented in Figure 4, but (only) after updating the communication trade-off factor in the body of the function *tot_cost*. The modeler first needs to define the arrays *unit_comm_cost* and *comp_comm*2 as explained earlier.

4. Evaluation

In this section, we first compare the approach presented in this paper with the original approach in [6]. Then, we show the improvement in performance when using the branching options as outlined at the end of Subsection 3.3. Finally, we show the results of applying our approach on eight different component allocation problems.

First, in Table 1, we compare the original approach presented in [6] with the approach presented in this paper. Since the original approach does not consider communication cost, we exclude it when evaluating the cost of the allocations. To have a fair comparison, the branching options are not used when applying the approach described in this paper. The table also includes the cost of an optimal component allocation computed by an exhaustive search. We have implemented the exhaustive search in a Java program that computes the cost of all feasible allocations and returns one that has the minimum total allocation cost. In addition, the table shows the optimal component allocation computed using the CPN based approach, its cost *w*, the number of markings generated by CPN Tools, and the time (in seconds) it took for the CPN Tools to generate the state space (the markings). The last two results are obtained by using the CPN ML functions: *NoOfNodes*() and *NoOfSecs*(). Note that the state space generation was done on a Dell desktop computer equipped with a 3.00 GHz dual-core processor and 2 GB RAM.

The table validates the CPN approach in the case study, since the returned component allocation is optimal (i.e., feasible and its cost is equal to that of the optimal allocation returned by the exhaustive search). In addition, although the same number of markings are generated in both approaches, the table shows that there is almost 18% improvement in terms of the time it took to generate the state space. This is a result of reducing the memory footprint of each marking by using an optimized scheme for encoding the resource consumption matrix and the components.

Second, the next three tables show the performance improvement of using the branching options, while applying the approach presented in this paper. Table 2 shows the evaluation results when using the component allocation problem

Table 1. Evaluation results including both constraints − no communication cost and no branching options.

Optimal Cost − Exhaustive Search	141.01
Runtime in Seconds − Exhaustive Search	1.78
Optimal Allocation − CPN Approach	(1, 3, 1, 3, 1, 1, 4, 3, 3, 2, 2)
Optimal Cost − CPN Approach	141.01
Number of Markings − original CPN Approach	16813
Number of Seconds − original CPN Approach	44
Number of Markings - CPN Approach as presented in this paper	16813
Number of Seconds - CPN Approach as presented in this paper	36

presented in Subsection 3.1. Table 3 shows the evaluation results from the same allocation problem, but excluding **Constraint II**. To exclude this constraint, we remove the corresponding condition from the guard of transition *allocate*. The evaluation results, when excluding both constraints, are shown in Table 4.

In order to set the upper bound necessary when using the branching options, applicable heuristics can be used to determine appropriate values for the upper bound. We use the genetic algorithm developed in [3]. Each execution of the algorithm can have a different result. The algorithm is run five times, and we choose the smallest optimal cost as an upper bound in the setting of the branching options.

We can make three conclusions when analyzing the results. First, the optimal cost found by the CPN approach is equal to that found by the exhaustive search. This validates the CPN approach. Second, the generated state space exponentially increases when the size of the component allocation problem is increased. This is evident by comparing the different numbers of

Table 2. Evaluation results including both constraints.

Optimal Cost − Exhaustive Search	153.43
Runtime in Seconds − Exhaustive Search	1.52
Optimal Allocation − CPN Approach	(1, 3, 1, 3, 1, 1, 4, 3, 3, 2, 2)
Optimal Cost − CPN Approach	153.43
Number of Markings − CPN Approach − No Branching Options	16813
Number of Seconds − CPN Approach − No Branching Options	36
Number of Markings - CPN Approach − With Branching Options	2313
Number of Seconds - CPN Approach − With Branching Options	1

Table 3. Evaluation results excluding constraints II.

Optimal Cost − Exhaustive Search	144.50
Runtime in Seconds − Exhaustive Search	1.50
Optimal Allocation − CPN Approach	(1, 2, 1, 1, 1, 4, 4, 2, 2, 3, 2)
Optimal Cost − CPN Approach	144.50
Number of Markings − CPN Approach − No Branching Options	27745
Number of Seconds − CPN Approach − No Branching Options	109
Number of Markings - CPN Approach − With Branching Options	3703
Number of Seconds - CPN Approach − With Branching Options	1

Table 4. Evaluation results excluding both constraints.

Optimal Cost − Exhaustive Search	144.50
Runtime in Seconds − Exhaustive Search	1.54
Optimal Allocation − CPN Approach	(1, 2, 1, 1, 1, 4, 4, 2, 2, 2, 2)
Optimal Cost − CPN Approach	144.50
Number of Markings − CPN Approach − No Branching Options	103863
Number of Seconds − CPN Approach − No Branching Options	2193
Number of Markings - CPN Approach − With Branching Options	8741
Number of Seconds - CPN Approach − With Branching Options	6

markings when including both constraints, excluding a constraint, and excluding both constraints. Third, the tables show significant improvement in terms of the generated number of markings and the time to generate the state space when utilizing the branching options. For example, Table 4 shows that the time to generate the state space when using the branching options is almost 366 times quicker than when not using them for the case of excluding both constraints.

As Table 4 shows, the CPN-based approach with branching options is slower than the exhaustive search. This is due to the overhead incurred when using the CPN-based approach. The exhaustive search is implemented directly in Java, while the CPN-based approach uses CPN Tools simulation which incurs some overhead when constructing and analyzing the state space. However, the results in terms of runtime might be different for larger problems for which the branching options severely cut down the generated state space.

Lastly, we show the results of applying our approach on several system instances. The system instances were obtained by random shuffling of the elements of the matrices T and K of the case study in Subsection 3.1. The details of the system instances can be obtained by contacting the author. Table 5 shows the optimal costs obtained using the CPN approach for eight system instances. For each instance, the table shows the optimal cost when including both constraints (case A) and when excluding **Constraint II** (case B). We verified the results by comparing them with the optimal costs obtained when using exhaustive search. Note that for the instances **3.A** and **3.B**, there is no feasible allocation. In such cases, the list of markings that are returned by the second application of *SearchNodes* (see Figure 4) is empty. For this part of the evaluation, we did not use the branching options.

Table 5. Evaluation results for different system instances.

Instance Number	Optimal Cost	Number of Markings	Number of Seconds
1.A	188.40	1853	0
1.B	188.40	2152	1
2.A	169.30	2272	1
2.B	169.30	3082	1
3.A	None	906	0
3.B	None	1297	0
4.A	198.00	1556	0
4.B	198.00	2340	0
5.A	208.54	608	0
5.B	208.53	672	0
6.A	218.43	1320	0
6.B	218.43	1945	0
7.A	195.48	2423	0
7.B	195.48	2546	1
8.A	161.72	5743	4
8.B	161.72	6565	7

5. Conclusion and Future Work

In this paper, we presented several improvements to the CPN-based approach for software component allocation on heterogeneous systems. We incorporated the costs of communication between the software components in the CPN model. Also, we explored the use of the branching options in the CPN ML state space generation tool to scale the CPN approach to larger systems.

One potential limitation of the CPN-based approach is the exponential increase in the generated state space for larger systems. In this paper, we suggested a technique to determine an upper bound on the cost and only generate the states having cost less than this upper bound. The upper bound can be determined using heuristics such as genetic algorithms. This significantly cuts down the generated state space.

However, the generated state space can become intractable for larger systems. Thus, it is of interest to explore the ways to generate and analyze the state space more intelligently. For example, the work of [27] surveys several parallel algorithms to solve discrete optimization problems such as the component allocation problem. A discrete optimization problem is often formulated as the problem of finding a path in a graph (the state space graph) from a designated initial node to one of several possible final nodes. The authors review several techniques to search the state space and discuss how these

algorithms can be parallelized. CPN Tools include limited functionality to control how the state space is generated. However, in order to scale our approach to larger systems, it is of interest to explore the use of these techniques in the context of our CPN approach.

Part of our future work should also 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) [28]. Finally, the CPN models need to be analyzed in terms of other non-functional properties such as security and dependability.

References

[1] G. Campeanu *et al*., "Component Allocation Optimization for Heterogeneous CPU-GPU Embedded Systems", in *Proceedings of the Conference on Software Engineering and Advanced Applications*, 2014, pp. 229–236. http://dx.doi.org/10.1109/SEAA.2014.29

[2] S. Malek *et al*., "An Extensible Framework for Improving a Distributed Software System's Deployment Architecture", *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 73–100, 2012. http://dx.doi.org/10.1109/TSE.2011.3

[3] I. Švogor *et al*., "An Extended Model for Multi-criteria Software Component Allocation on a Heterogeneous Embedded Platform", *Journal of Computing and Information Technology*, vol. 21, no. 4, pp. 211–222, 2013. https://doi.org/10.2498/cit.1002284

[4] H. A. Taha, "Operations Research: An Introduction", 10th ed. Prentice Hall, 2010.

[5] L. A. Wolsey, "Integer Programming", Wiley-Interscience, 1998.

[6] I. Al-Azzoni, "Software Component Allocation on Heterogeneous Embedded Systems using Coloured Petri Nets", in *Proceedings of the Conference on Advances and Trends in Software Engineering*, 2015, pp. 23 – 28.

[7] K. Jensen *et al*., "Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems", *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 3, pp. 213–254, 2007. http://dx.doi.org/10.1007/s10009-007-0038-x

[8] K. Jensen and L. M. Kristensen, "Coloured Petri Nets − Modelling and Validation of Concurrent Systems", Springer, 2009. http://dx.doi.org/10.1007/b95112

[9] I. Al-Azzoni *et al*., "Modeling and Verification of Cryptographic Protocols using Coloured Petri Nets and Design/CPN", *Nordic Journal of Computing*, vol. 12, no. 3, pp. 201–228, 2005.

[10] 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. http://dx.doi.org/10.7146/dpb.v31i563.7120

[11] J. Feljann *et al*., "Towards a Model-based Approach for Allocating Tasks to Multicore Processors", in Proceedings of the Conference on Software Engineering and Advanced Applications, 2012, pp. 117–124. http://dx.doi.org/10.1109/SEAA.2012.56

[12] P. Liggesmeyer and M. Trapp, "Trends in Embedded Software Engineering", *IEEE Software*, vol. 26, no. 3, pp. 19–25, 2009. http://dx.doi.org/10.1109/MS.2009.80

[13] 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. http://dx.doi.org/10.1109/ISORC.2006.54

[14] L. Al-Dakheel and I. Al-Azzoni, "Model-to-Model based Approach for Software Components Allocation in Embedded Systems", in *Proceedings of the International Conference on Model-Driven Engineering and Software Development*, 2017, pp. 321–328. http://dx.doi.org/10.5220/0006126903200328

[15] I. Švogor and J. Carlson, "SCALL: Software Component Allocator for Heterogeneous Embedded Systems", in *Proceedings of the European Conference on Software Architecture Workshops*, 2015, pp. 66:1–66:5. http://doi.acm.org/10.1145/2797433.2797501

[16] U. Pohlmann and M. Hüwe, "Model-driven Allocation Engineering", in *Proceedings of the International Conference on Automated Software Engineering*, 2015, pp. 374–384. http://dx.doi.org/10.1109/ASE.2015.18

[17] S. Wang *et al*., "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. http://dx.doi.org/10.1109/RTTAS.2004.1317267

[18] AIRES, [Accessed March 2017]. https://kabru.eecs.umich.edu/aires/

[19] Z. Krpić *et al*., "Towards a Common Software-to-hardware Allocation Framework for the Heterogeneous High Performance Computing", in *Proceedings of the Computers, Software and Applications Conference*, 2014, pp. 378–383. http://dx.doi.org/10.1109/COMPSACW.2014.65

[20] J. Carlson *et al*., "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. http://dx.doi.org/10.1109/SEAA.2010.43

[21] J. Fredriksson *et al*., "Optimizing Resource Usage Component-based Real-time Systems", in *Proceedings of the Symposium on Component-based Software Engineering*, 2005, pp. 49–65. http://dx.doi.org/10.1007/11424529_4

[22] 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. http://dx.doi.org/10.1109/RTAS.2006.21

[23] Y. Shoukry *et al*., "Graph-based Approach for Software Allocation in Automotive Networked Embedded Systems: A Partition-and-map Algorithm", in *Proceedings of the Forum on Specification and Design Languages*, 2013. http://dx.doi.org/10.1007/978-3-319-06317-1_11

[24] R. M. Karp, "Reducibility Among Combinatorial Problems", in *Proceedings of the Symposium on the Complexity of Computer Computations*, 1972, pp. 85–103. http://dx.doi.org/10.1007/978-1-4684-2001-2_9

[25] RALF3 Project Web, [Accessed March 2017]. http://www.mrtc.mdh.se/projects/ralf3/

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

[27] A. Grama and V. Kumar, "State of the Art in Parallel Search Techniques for Discrete Optimization Problems", *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 28–35, 1999. http://dx.doi.org/10.1109/69.755612

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

Received: November 2017
Revised: June 2018
Accepted: July 2018

Contact address:
Issam Al-Azzoni
College of Engineering
Al Ain University of Science and Technology
Al Ain
United Arab Emirates
e-mail: issam.alazzoni@aau.ac.ae

Issam Al-Azzoni received his MSc and PhD degrees in software engineering from McMaster University, Hamilton, Ontario, Canada in 2005 and 2009, respectively. He is presently an Assistant Professor in the Department of Software Engineering and Computer Science in the College of Engineering at Al Ain University of Science and Technology, United Arab Emirates. His research interests include modeling, model transformation, Coloured Petri Nets, and the application of formal methods in software engineering. He is a member of the IEEE.