Calculating Resource Trade-offs when Mapping Component Services to Real-Time Tasks

Johan Fredriksson, Mikael Åkerholm and Kristian Sandström
Mälardalen Real-Time Research Centre
Department of Computer Science and Engineering
Mälardalen University, Västerås, Sweden
johan.fredriksson@mdh.se
http://www.mrtc.mdh.se

Abstract

The research on real-time systems has produced algorithms for effective scheduling of system resources while guaranteeing the real-time properties. However, the issue of allocating component services to schedulable task entities has gained little focus, even though component based development has attracted an increasingly interest, also in the real-time community. Trade-offs when allocating component services to tasks, are, e.g., cpu-overhead, footprint and integrity.

In this paper we present a general framework for calculating properties, such as memory consumption and cpu-overhead, of a given mapping of component services to tasks, while utilizing existing real-time analysis.

1 Introduction

The embedded systems domain represents a class of real-time systems where the requirements on safety, reliability, resource usage, and cost leaven all through development. Historically, the development of such systems has been done using only low level programming languages, to guarantee full control over the system behaviour. As the complexity and the amount of functionality implemented by software increase, so does the cost for software development. Also, since product lines are common within the domain, issues of commonality and reuse are central for reducing cost as well as increasing reliability. Therefore component-based development has shown to be an efficient and promising approach for software development, enabling well defined software architectures as well as reuse.

Typically, embedded systems react on the environment and have to respond within a bounded interval in time, i.e., they are real-time systems; hence timing and scheduling are central concepts. Furthermore, these systems are often resource constrained; consequently memory footprint and CPU load are desired to be as low as possible.

A problem in current component based embedded software development practices is the mapping of component services to run-time threads (tasks) [8]. Because of the real-time requirements on most embedded systems, it is vital that the mapping considers temporal attributes, such as *worst case execution time* (WCET), deadline (D) and period time (T). In a system with many small component services, the overhead from context switches will be quite high. Embedded real-time systems consist of periodic and sporadic events and they usually have end-to-end timing requirements. Periodic events can often be coordinated and executed by the same task, while preserving temporal constraints. Hence, it is easy to understand that there can be profits from grouping several component services into one task. Some of the benefits can be less memory in form of stacks and task control blocks or lower CPU utilization due to less overhead for context switches. There are many trade-offs to be made when allocating component services to tasks. Different properties can be accentuated depending on how component services are allocated to tasks, e.g., footprint, performance or integrity.

Allocating component services to tasks, and scheduling tasks are both complex problems and different approaches are used. Simulated annealing and genetic algorithms are examples of algorithms that are frequently used for optimization problems. However, to be able to use such algorithms, a framework to calculate properties, such as memory consumption and overhead, is needed. The work described in this paper presents a general model for reasoning about trade-offs concerning allocating component services to tasks, while preserving extra-functional requirements. A framework is developed to help transit from component services, to a run-time model while enabling verification of temporal constraints, and optimization for low footprint and overhead.

The problem of allocating tasks to different nodes is a problem that has been studied by researchers using different methods [6,18]. There are also methods proposed for transforming structural models to run-time models [3,5,10], but extra-functional properties are usually ignored or considered as non-critical [8]. However, allocating components to tasks is a different problem. In [16], an architecture for embedded systems is proposed, and it is identified that components has to be grouped into tasks, however there is no focus on the allocation of components to tasks. In [8] the authors propose a model transformation where all components with the same priority are allocated to the same task. The idea of assigning components to tasks considering extra-functional properties for embedded systems is a relatively uncovered area. However, similar approaches to this work have been formulated by Shin et. al [16] and Kodase et. al [8]. In [4], the authors discuss how to minimize memory consumption in real-time task sets. Shin et. al [15] are discussing the code size, and how it can be minimized.

The outline for the rest of the paper is as follows; section 2 gives an overview of the component service to task mappings, and describes the structure of the components and tasks. In section 3 a framework for calculating the properties of components allocated to tasks. Section 4 discusses allocation and scheduling approaches, while an illustrative example is given in section 5. Finally in section 6, future work is discussed and the paper is concluded.

2 Mapping component services to real-time tasks

Component based software engineering is a promising approach for efficient software development, enabling well defined software architectures as well as reuse. Temporal constraints are of great importance in embedded real-time systems; hence we need an efficient mapping from component services to tasks that enables verification of temporal behaviour. End-to-end deadlines are denoted transactions, and are defined by a sequence of component services and a timing requirement. Given a mapping from component services and transactions to tasks we can determine if the mapping is valid and schedulable, and calculate the properties memory consumption and overhead. The verification is performed with a framework during compile-time. The work in this paper has two main concerns:

1. Verification of mappings from component services to tasks.
2. Calculating system properties for a mapping

This paper is a refinement of previous work, autocomp [14], which is a component technology. An overview of the autocomp technology can be seen in Figure 1. The different steps in the figure is divided into design time, compile time, and run-time to display at which point in time during development they are addressed or used. The component model is used during design time for describing an application. The compile time steps, illustrated in Figure 1, incorporate a mapping from the component based design, to a real-time model and mapping to a real-time operating system (RTOS). During this step the component services are replaced by real-time tasks and the component service requirements are mapped to task-level attributes.

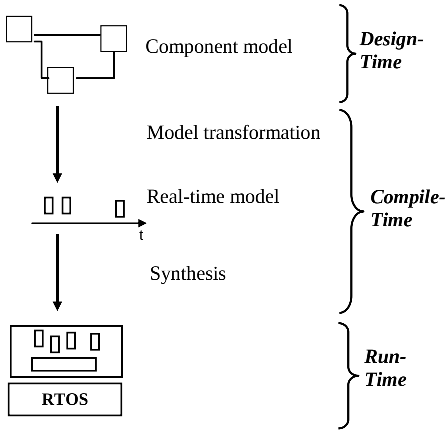

Figure 1 System description.

Our general component and task model combined with the notion of transactions and a pipe-and-filter model constitutes a general approach that should be easy to implement for a large set of component technologies for embedded systems such as Autocomp [14], Rubus [1], Koala [19] and Port-based objects [17].

The component and transaction characteristics are described in the sections 2.1 and the tasks characteristics are described in section 2.2.

2.1 Component characteristics

In this paper we will describe characteristics for a general component model that should be applicable to a large set of commercial or research embedded component models. The component and task models are meta-models for modelling the most important attributes of a mapping between component services and tasks. The models are used for evaluation considering a set of requirements, e.g., memory consumption and overhead. The component structure used throughout this paper is a pipe-and-filter model with transactions. In Figure 2 a component assembly with six component services and two transactions is described. Each component service has a trigger; a time trigger, an event trigger or a trigger from a preceding component. A component transaction describes an order of component services and defines an end-to-end timing requirement. Each primitive is graphically denoted in Figure 2.

Many component models do not have the notion of transactions built in, however, if there are possibilities to model end-to-end timing requirements and precedence, then that can be seen as transactions at a higher abstraction level. A system is described with components, component services and transactions defining the temporal requirements on the component services.

- **A component c_i** is described with the tuple *<S, I >* where S is a set of component services provided by the component. The isolation set (I) defines a relation between components that need to be isolated for guaranteeing integrity (memory protection). This is often required for safety critical components.

- **A component service c_i^j** is described with the tuple *<G, wcet, mem >*, *G* is a trigger which is described with the tuple *<S,T>* where S is a signal from another component, an external event or a timed event. T represents the minimum inter arrival time (MINT) in the case of an external event. It represents the period in the case of a timed trigger and it is unused if the signal is from another component. The parameter *wcet* is the worst case execution time, and *mem*, is the amount of memory required by the service. A component can have an arbitrary set of services. A component *i* with a service *j* is denoted as c_i^j. A component service can only trigger one subsequent component service. However, a subsequent component service can be triggered by several proceeding component services, i.e., a component service can be part of several component transactions.

  A component transaction ctr_i is an ordered relation between component services and an end-to-end deadline. A component transaction can stretch over one or several component services and it is described with the tuple < *N*, *d* >, where N is a set of component services $\{c_i^x, c_j^y, c_k^z\}$ and d is a relative deadline. The deadline is relative to the event that triggered the component transaction. A component transaction describes a precedence order, i.e., the component services defined by the N-set are executed in the order they appear in the component transaction (in the N-set). The same component service can participate in several component transactions. The precedence order is loose, meaning that the component services can be executed several times within the same component transaction, i.e., the exact order 1-2-3-4 is fulfilled also if the component services execute in the order 1-3-2-3-4 which is also formalized below:

[1] $$\{c_i^a, c_j^b, c_k^c\}: c_i^{a^+}\left[c_k^{c^*}\right]c_j^b\left[c_i^{a^*}c_j^{b^*}\right]c_k^c.$$

Because of the loose precedence order no consistency or atomicity models are needed. There are requirements and restrictions on component transactions; the first service in a component transaction has to be triggered by an event, a time trigger or another component. An event trigger may only trigger the first service in a component transaction.

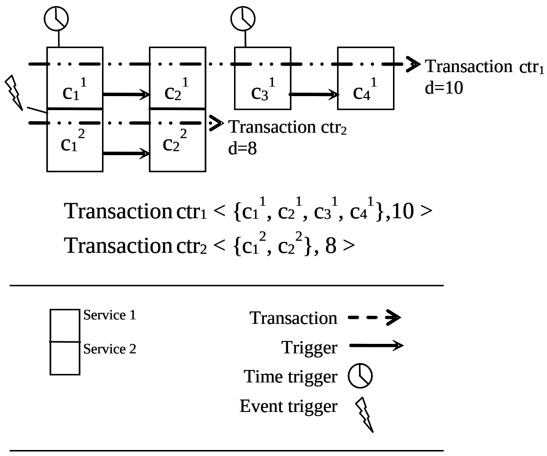

Figure 2 Component transaction model.

2.2 Task characteristics

The run-time model specifies the organization of entities in the component model into tasks and task transactions. During the transformation from component model to run-time model, extra-functional properties like schedulability and response-time constraints must be considered in order to ensure the correctness of the final system. Actions within a task are executed at the same priority as the task, and a high priority task pre-empts a low priority task. Component services only interact through explicit interfaces, hence tasks do not synchronize outside the component model. The task model is for evaluating schedulability and other properties of a system.

- **A system K** is described with the tuple *<A, tcbsize, ρ >* where A is a task set scheduled by the system, *tcbsize* is the size of each task control block, and can be considered constant, and the same for all tasks. The constant ρ is the time associated with a task switch. The system kernel is the only explicit shared resource between tasks; hence we do not consider blocking. Also blocking is not the focus of this paper.

[1] "*" means zero or more occurenses; "+" menas one or more occurenses

- **A task τ_i** is described with the tuple < *S, T, wcet, stack*> where S is an ordered set of component services. Component services within a task are executed in sequence. T is the period or minimum inter arrival time of the task. The parameters *wcet* and *stack* are worst case execution time and stack size respectively. The *wcet*, *stack* and period (T) are deduced from the component services in S. Hence, for a task τ_i:
    - $wcet = \sum_{\forall_i \forall_j (c_i^j \in S)} c_i^j.wcet$
    - $stack = \forall_i \forall_j (c_i^j \in S) \max(c_i^j.mem)$
    - $T = \forall_i \forall_j (c_i^j \in S) \min(\langle c_i^j.G \rangle.T)$
- **A task transaction ttr_i** The timing requirements of a task transaction ttr_i are deduced from the timing requirements of the component transactions ctr_i. A task transaction ttr_i is described with the tuple < *M, d* >, where M is a set of tasks { τ_i, τ_j, τ_k } and d is a relative deadline. The task transaction ttr_i is a direct mapping from the component transaction ctr_i. The task transaction ttr_i has the same parameters as the component transactions ctr_i but τ_i, τ_j and τ_k are the tasks that map the services c_i^j, c_j^k and c_k^l respectively, see Figure 3. The task transaction defines a loose precedence order between tasks, meaning that a task transaction is realized when the tasks have executed in the order they appear in the task transaction, i.e. even if the component services do not execute in the exact order *1-2-3-4* they can execute in the order *1-3-2-3-4*, which is still a valid task transaction. This is due to the pipe-and-filter model where the data flow through the component services defines the task transaction. The same restrictions applied on the component transactions ctr_i, apply on the task transactions ttr_i.

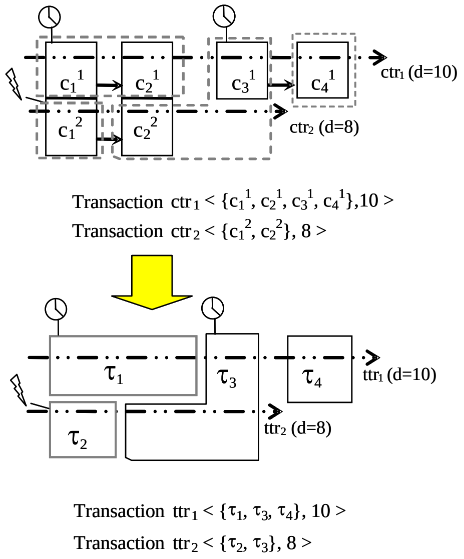

Figure 3, translation of component transactions, to task transaction.

2.3 Constraints on transactions

If several task transactions ttr_i span over the exact same tasks, the task transaction with the shortest deadline is the only valid.

A component transaction defines precedence between component services. When component services are allocated to tasks, the precedence defined by the component transactions must never be broken. In other words, a set of component services c_1^1, c_2^1 and c_3^1 with the precedence 1-2-3, may be allocated to tasks in any way that do not break the precedence. Hence, only services c_1^1 and c_3^1 may not be allocated to the same task. However, services c_1^1 and c_2^1, or c_2^1 and c_3^1 may be allocated (Figure 4), thus the component service precedence is preserved.

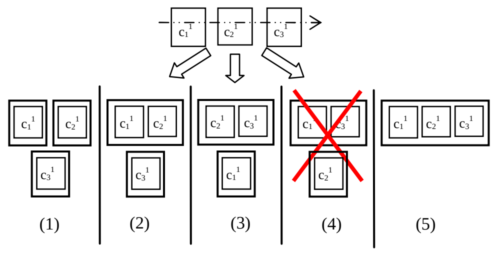

Figure 4 Three different allocations

Allocations (1), (2), (3) and (5) do not violate the precedence S1-S2-S3. Allocation (4) has violated the precedence (S1-S3).

If component transactions intersect, there are different strategies for how to allocate the component where the transactions intersect. The component in the intersection (c_{int}) has to be allocated to a separate task when a transaction that has an event trigger intersects another transaction. The task c_{int} has to be triggered by both transactions. This is done to increase the schedulability by increasing responsiveness.

In Figure 5 (1) the component c_{int} is allocated to a separate task because an event triggered transaction is intersecting an other transaction. In Figure 5 (2) there is no event triggered transaction, and c_{int} can be allocated to any task. Two intersecting time triggered transactions can be handled in any way that does not violate the precedence relations.

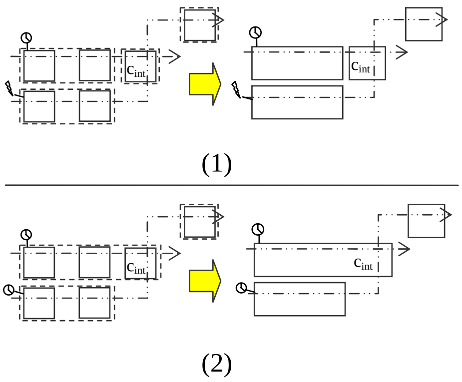

Figure 5 Intersecting transactions, event-triggered vs. time-triggered.

3 Evaluation framework

The evaluation framework is a set of models for calculating properties of allocations of component services to tasks. The properties calculated with the framework are used for optimization algorithms, to find a feasible allocation that fulfils given requirements on memory consumption and performance overhead.

For a task set *A* that has been mapped from component services to tasks in a one-to-one fashion, it is trivial to calculate the system memory consumption and performance overhead since each task has the same properties as the basic component service. When several component services are grouped to one task we need to calculate the tasks properties. For a set of component services, $c_1^1 \ldots c_n^n$, mapped to a set of tasks A the following properties are considered.

- Performance overhead p_A
- Memory consumption m_A

The performance overhead is not dependent on how many services are allocated to a task. Each service c_i^j has a memory consumption *stack*. The Stack of the task is the maximum size of all services stacks allocated to the task since all services will use the same stack.

The CPU overhead *p*, the memory consumption *m* for a task set A in a system K is described below:

- $p_A = \sum_{\forall_i (\tau_i \in A)} \frac{K.\rho}{\tau_i.T}$
- $m_A = \sum_{\forall_i (\tau_i \in A)} \left(\tau_i.m + K.tcbsize\right)$

3.1 Constraints on allocations

It is not realistic to expect that components can be grouped in an arbitrary way. There may be explicit dependencies that prohibits that certain components are grouped together. Therefore each component has an isolation set I that defines with which components it may not be grouped.

A component c_i may have defined an isolation set $I_i\{c_k,\ldots,c_n\}$, with components which it may not be allocated to ensure integrity between components. Hence it must be assured that two components that are defined to be isolated do not reside in the same task. The isolation is a restriction on which components may not be allocated to the same task. The isolation of a task set *A* can be validated and confirmed with:

$$I_A = \forall_i \forall_j \forall_k \begin{pmatrix} j \neq k \wedge \tau_i \in A \wedge C_j^? \in \tau_i.S \wedge C_k^? \in \tau_i.S \rightarrow \\ C_k \notin C_j.I \wedge C_j \notin C_k.I \end{pmatrix}^2$$

Some grouping of component services to tasks can be performed without impacting the schedulability negatively. Component services with a precedence order can be grouped into a task if they have no other explicit dependencies, thereby lowering the overhead generated by context switches and lowering the memory usage by using one stack, see (1) in Figure 6. Component services with the same period time can be grouped if they do not have any other explicit dependencies, (2) Figure 6.

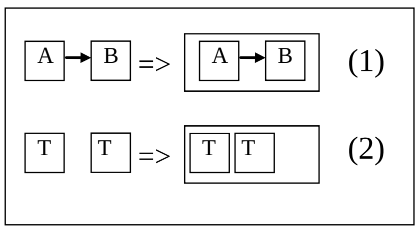

Figure 6 Component service grouping.

Schedulability analysis is highly dependent on the scheduling policy chosen. Depending on the system design, different analyses approaches have to be considered. The task and task transaction meta-models are constructed to fit different scheduling analyses.

[2] Question mark $C_i^?$ indicates component C_i independent of which service is handled

Furthermore, only allocations with several components to one task are considered, hence we leave out *multiple-to-multiple* allocations. *Multiple-to-multiple* allocations make the system less analyzable and increase the complexity. Also It is assumed that if the system is schedulable in a one-to-one mapping fashion, it is also schedulable after the component service to task allocation.

4 Using the framework

An allocation can be performed in several different ways. In a small system all possible allocations can be evaluated and the best chosen. For a larger system, however, this is not possible due to the combinatorial explosion. Different algorithms can be used to find a feasible allocation and scheduling of tasks. For any algorithm to work there must be some way to evaluate an allocation or real-time schedule. The proposed evaluation framework can be used to calculate schedulability, performance overhead and total memory load.

Simulated annealing, genetic algorithms and bin packing are well known algorithms often used for optimization problems. These algorithms have been used for problems similar to those described in this paper; bin packing, e.g., has been proposed in [13]. Here we discuss how theses algorithms can be used with the described framework, to perform component to task allocations.

Bin Packing is a method well suited for our framework. In [7] a bin packing model that handles arbitrary conflicts (BPAC) is presented. The BPAC model constrains certain elements from being packed into the same bin. which directly can be used in our model as the isolation set I. The bin-packing feasibility function is the schedulability, and the performance and memory overhead constitute the optimization function.

Genetic algorithms can solve, roughly, any problem as long as there is some way of comparing two solutions. The framework proposed in this paper give the possibility to use the properties memory consumption, performance overhead and schedulability as grades for an allocation, in order to evolve new allocation specimen. Similar work with genetic algorithms has been made in, e.g., [12] and [11].

The simulated annealing (SA) is a global optimization technique that is regularly used for solving NP-Hard problems. The energy function consists of a schedulability test, the memory consumption and performance overhead. In [18] and [2] simulated annealing is used to place tasks on nodes in a distributed system.

5 Evaluation

In order to evaluate the performance of our allocation approach we have made an implementation of the framework. We choose to perform a set of allocations and compare the results to a basic mapping where each service is allocated to a task. We compare the allocations with respect to memory usage and cpu overhead.

The implementation is based on genetic algorithms (GA) [20]. Each gene represents a service, and contains a reference to the task it is assigned. Each chromosome represents the entire system with all services assigned to tasks. Each allocation produced by the GA is evaluated by the framework, and is given a fitness value dependent on the validity of the allocation and the memory consumption and cpu overhead.

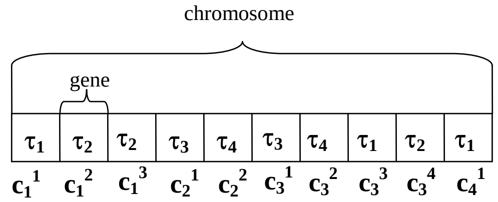

Figure 7 The genetic algorithm view of the component service to task mapping; A system with 10 services.

A simulator generates systems with a given number of services, components and transactions. The GA framework then performs an allocation and record the improvement in memory usage and cpu overhead compared to a *one-to-one* mapping. The average stack usage and the cpu overhead for one-to-one mapping and for our component service mapping is shown in Table 1. The data set consists of approximately 300 simulations.

Number of services	One-to-one mapping		Component service mapping	
	Stack	Overhead %	Stack	Overhead %
5	2898	9%	2253	5%
10	5705	18%	4516	12%
15	8250	27%	6451	17%
20	11068	35%	8369	21%
25	13516	37%	10364	23%
30	16737	41%	12431	25%

Table 1 Average stack usage and cpu overhead for one-to-one mappings and component service mappings.

Note that the improvement is almost constant independent of the number of services. Figure 8 summarizes the improvement in stack size and cpu overhead in component service mapping compared to one-to-one mapping. The number of tasks generated for different number of component services is shown in Figure 9. Hence we can see a clear improvement in both memory usage and cpu overhead when facilitating the framework for allocating component services to tasks. In an average case our studies suggest an improvement in memory usage of 35%. For cpu overhead the improvement is approximately 20%.

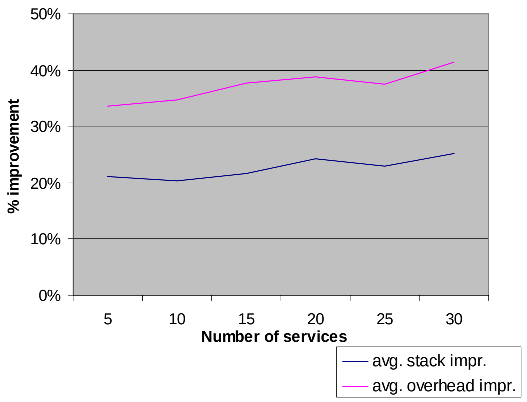

Figure 8 Average improvement of stack and overhead; comparing component service mapping to one-to-one mapping.

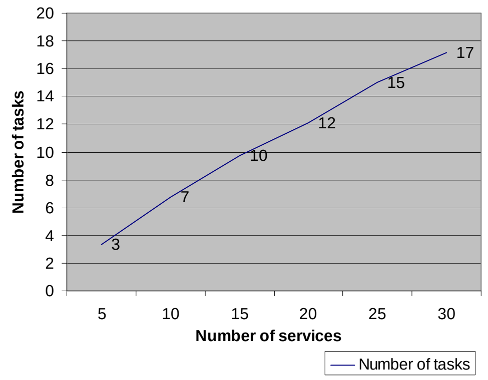

Figure 9 Number of tasks generated regarding the number of component services.

6 Conclusion and Future Work

For embedded real-time systems resource efficiency, both performance and memory wise, is very important. Schedulability, considering resource efficiency, has gained much focus, however the mapping between component services to tasks has gained little focus. Hence, in this paper we have described an evaluation framework for allocating component services to tasks, to facilitate existing scheduling and optimization algorithms such as genetic algorithms, bin packing or simulated annealing. The framework can be extended to support other optimizations, besides performance and memory overhead. We also show that the framework can give substantial improvements both in terms of memory usage and cpu overhead. In future work, the framework will be extended with jitter and blocking requirements. We will also look into how different cpu load will affect the mapping of a system.

7 References

[1] Arcticus, *Arcticus Systems Home Page*, http://www.arcticus.se, 3-29-2004.

[2] Cheng S.-T. and Agrawala A. K., "Allocation and Scheduling of Real-Time Periodic Tasks with Relative Timing Constraints", *In proceeding of Second International Workshop on Real-Time Computing Systems and Applications (RTCSA)* pp. 210-217, 1995.

[3] Douglas B. P., *Doing Hard Time* , Addison Wesely, 1999.

[4] Gai P. and Lipari G., "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip", IEEE, 2001.

[5] Gomaa H., *Designing Concurrent Distributed, and Real-Time Applications with UML*, **Addison Wesley**, 2000.

[6] Hou C. and Shin K. G., "Allocation of periodic Task Modules with precedence and deadline Constraints in Distributed Real-Time System", In *IEEE Transactions on Computers*, volume 46, No 12, 1997.

[7] Jansen K. and Öhring S. R., "Approximation algorithms for time constrained scheduling", *In proceeding of Workshop on Parallel Algorithms and Irregularly Structured Problems* pp. 143-157, 1995.

[8] Kodase S., Wang S., and Shin K. G., "Transforming structural model to runtime model of embedded software with real-time constraints", *In proceeding of Design, Automation and Test in Europe Conference and Exhibition* pp. 170-175, 2003.

[9] Liu C.L. and Layland J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", In *Journal of ACM*, volume 20, issue 1, 1973.

[10] Mills K. and Gomaa H., "Knowledge-based automation of a design method for concurrent systems", In *IEEE Transactions on Software Engineering*, volume 28, issue 3, 2002.

[11] Monnier Y., Beauvais J.-P., and Deplanche A.-M., "A genetic algorithm for scheduling tasks in a real-time distributed system", *In proceeding of 24th Euromicro Conference* ,nr 2, pp. 708-714, 1998.

[12] Montana D., Brinn M., Moore S., and Bidwell G., "Genetic algorithms for complex, real-time scheduling", *In proceeding of IEEE International Conference on Systems, Man, and Cybernetics* , nr 3, pp. 2213-2218, 1998.

[13] Oh Y. and Son S. H., *On Constrained Bin-packing Problem*, report CS-95-14, University of Virginia, 1995.

[14] Sandström K., Fredriksson J., and Åkerholm M., "Introducing a Component Technology for Safety Critical Embedded Real-Time Systems", *In proceeding of CBSE7 International Symposium on Component-based Software Engineering*, 2004.

[15] Shin I., Lee I., and Sang M., "Embedded system design framework for minimizing code size and guaranteeing real-time requirements", IEEE, 2002.

[16] Shin K. G. and Wang S., "An architecture for embedded software integration using reusable components", *In proceeding of the international conference on Compilers, architectures, and synthesis for embedded systems, San Jose, California, United States,* pp. 110-118, 2000.

[17] Stewart D.B., Volpe R.A., and Khosla P.K., Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects, *IEEE Transaction on Software Engineering*, volume 23, issue 12, 1997.

[18] Tindell K., Burns A., and Wellings A., "Allocating Hard Real-Time Tasks (An NP-Hard Problem Made Easy)", In *Real-Time Systems*, volume 4, issue 2, pp. 145-165, Kluwer Academic Publishers , 1992.

[19] van Ommering R., van der Linden F., and Kramer J., "The Koala Component Model for Consumer Electronics Software", In *IEEE* Computer, volume 33, issue 3, pp. 78-85, 2000.

[20] Fonseca C. M., Fleming P. J., “An overview of evolutionary algorithms in multiobjective optimization”. Evolutionary Computation, 3(1):1--16, 1995.