2006-03

An Admission Control Method for Dynamic Software Reconfiguration in Complex Embedded Systems

Qiao, Y.; Wang, H.; Luqi; Berzins, V.

https://hdl.handle.net/10945/65709

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun
AN ADMISSION CONTROL METHOD FOR DYNAMIC SOFTWARE RECONFIGURATION IN COMPLEX EMBEDDED SYSTEMS

Y. Qiao, Luqi

Software Engineering Automation Center, US Naval Postgraduate School
833 Dyer Road
Monterey, CA 93943, USA
yqiao@nps.navy.mil

Abstract – High confidence is critical for complex embedded systems, known as systems of embedded systems (SoES) since this kind of systems are widely used in the field where consequences of failures are very serious. Although many methods have been proposed to deal with dynamic software reconfiguration in embedded systems, they ignore the potential risk for high confidence of SoES caused by configuration changes. In this paper we take this situation into consideration and present an admission control method to prevent dynamic software reconfiguration from damaging high confidence of SoES.

KEYWORDS: Embedded software, Systems of embedded systems, Dynamic software reconfiguration, High confidence, Schedulability

1. Introduction

As investments for individual embedded systems obtained great success, requirements for combining isolated embedded systems towards common and broad objectives have been receiving a great amount of attention. Systems of embedded systems (SoES) emerge as effective solutions to meet these requirements. A typical example of SoES is complex control systems for autonomous vehicles. In this system, each individual vehicle interacts in a distributed environment, e.g., in communication with a ground-station, mother-ship, or other
individual vehicles [10]. SoES have several heterogeneous component systems developed by different organizations with different tools and run on different platforms. They must rapidly accommodate frequent changes in requirements, mission, environment and technology. To make SoES have the ability to quickly and gracefully react to a changing environment or to changes in their own configuration, dynamic software reconfiguration becomes an important issue in SoES development.

Many works have been done to handle software reconfiguration in complex embedded systems. [9] presents an open software architecture, known as Open Control Platform (OCP), for integrating control technologies and resources. The OCP allows systems of components with standardized interfaces to be connected while abstracting away implementation details of lower level components. This enables component’s “plug and play” at run-time to support dynamic reconfiguration. [8] proposes a software architecture to support reconfigurable software for machine control systems. This software architecture is based on a combination of object-oriented models and executable formal specifications. Furthermore, it supports the separation of structural specification from behavioral specification which enables the controller software structure to be reconfigured independently of application, and software behavior to be reconfigured independently of controller software structure. [7] uses a port-based object abstraction, based on combining the notion of an object with the port-automaton algebraic model of concurrent processes to support dynamically reconfigurable real-time software. [1] presents a rapid prototyping technique that focuses on transition management for reconfigurable control systems. It allows hybrid systems and their reconfiguration strategies to be prototyped and validated at an early stage in development and for the strategies to be
directly transferred into the control systems software without a separate, manual reimplemention step.

All above methods concentrate on providing methods or frameworks to support the adjustment of system configurations and hold the assumption that the reconfiguration is safe to happen. However, this assumption is not true in practice. Dynamic software reconfiguration may induce the failure of whole SoES since configuration changes may have negative impacts on satisfaction of some key properties such as timing properties in SoES. Apparently, all above methods ignore this potential risk and the arbitrary acceptance of reconfiguration may result in two serious consequences: (a) damaging high confidence of the whole system and (b) increasing development costs caused by extra efforts for system recovery and redesign.

The contribution of this paper is to tackle above problem by presenting an admission control method for dynamic software reconfiguration in SoES. The core of this method is the dynamic scheduling analysis based on an integrated dynamic scheduling algorithm for heterogeneous embedded systems. In this case, only new component systems whose addition will not result in unschedulability of SoES can be allowed to enter the system. Furthermore, if the scheduling analysis shows the addition of new component systems violates schedulability of whole system, those new component systems will be put into a 'training process', which will find 'more suitable' parameters of new component systems according to the suggestion given by the scheduling analysis. This admission control method will improve the confidence of SoES by preventing whole system from the possible failure caused by dynamic software reconfiguration. At the same time, the proposed approach will reduce development costs by
avoiding the extra efforts for system recovery and redesign in case the system fails after the reconfiguration.

The rest of this paper is organized as follows: Section 2 proposes the framework of admission control method; Section 3 briefly introduces the computational model for SoES which is presented in our previous research; Section 4 addresses the dynamic scheduling analysis in SoES which is the core of the admission control method; Section 5 addresses an example to show how the proposed admission control method is applied to dynamic software reconfiguration in case of adding new component systems to SoES; Section 6 discusses how to apply the proposed admission control method to dynamic software reconfiguration in case of removing exiting component systems and software component migration; Section 7 is the conclusion.

2. Framework of Admission Control for Dynamic Software Reconfiguration in SoES

In general, dynamic software reconfiguration refers to change software configuration on-the-fly, including addition of new software components, removal of existing software components and software component immigration. Specifically, in this paper, dynamic software reconfiguration refers to addition of new component systems at the run-time.

Although dynamic software reconfiguration can improve system flexibility, it may result in the negative impact on high confidence of whole system. For example, the arbitrary addition of new component systems may violate schedulability of whole system so as to violate timing properties. This will result in system failures and consequently hazard high confidence of whole system. Thus, in order to prevent dynamic software reconfiguration from
damaging high confidence of whole system, it is necessary to perform admission control to SoES once new component systems arrive. Figure 1 shows the framework of our admission control method.

Figure 1 Admission control method for dynamic software reconfiguration for SoES

Admission control method is to decide whether dynamic software reconfiguration is allowed by detecting its possibility to damage high confidence of SoES. This method includes three main parts: parameter extracting, scheduling analysis and decision making. The parameter extracting is given the computational model of whole system including new arrival component systems. It extracts the parameters for dynamic scheduling analysis as needed. These parameters include the maximum execution time, the deadline and the trigger method of certain component systems, the latency of the corresponding interactions between component systems. The scheduling analysis will be given these extracted parameters and check schedulability of SoES after adding new component systems. The core of scheduling analysis is an integrated dynamic scheduling algorithm for heterogeneous embedded systems. This algorithm is based on heuristic searching and considers precedence relationships between
component systems. The decision making is performed based on results of scheduling analysis. In this case, if the whole system is still feasible to be scheduled after adding new component systems, these component systems are allowed to be added into SoES and the reconfiguration is accepted. Otherwise, the reconfiguration is rejected.

Furthermore, if the reconfiguration is rejected, a ‘training process’ will be triggered to find 'more suitable' parameters of new component systems so as to make them allowed to be added in. Specifically, the maximum execution times of new component systems will be iteratively reduced according to the recommendation given by the scheduling analysis. In this case, besides deciding if the addition of the new component systems will violate schedulability of whole systems, the scheduling analysis will also provide suggested maximum execution times of new component systems to ensure schedulability of whole system in case of adding these component systems. After the maximum execution times of new component systems are reduced, the scheduling analysis will be applied again to existing component systems together with those new component systems. If the result of scheduling analysis still shows the addition of new component systems will cause unschedulability of whole system, maximum execution times of new component systems will be reduced again. This process will be repeated until the scheduling analysis shows the addition of new component systems will not result in unschedulability of whole system and is safe to happen. It is noted that the reduction of maximum execution time we mentioned above is virtual and the ‘training process’ is only responsible to find a proper maximum execution time to allow the new component system to be added in without violating schedulability of whole system. However, actually changing maximum execution time of a component system needs redesign of this component system and it is a design issue which is out of the scope of this paper.
3. Computational Model for SoES

In our previous research, we presented a computational model to mathematically describe the functional and non-functional aspects of SoES requirements. This model has two views: external view model and internal view model [3].

The external view model focuses on the customer’s view. It captures requirements via functional and non-functional emergent properties [3]. This model, denoted as $\zeta'$, is formally represented as follows:

$$\zeta' = (G, H)$$  \hspace{1cm} (1)

$G$ is a functional emergent property vector which represents the functional aspect of requirements in SoES, $G = (g_1, g_2, \ldots, g_l)$, $g_i (i \in [1,l])$ denotes one of functional emergent properties describing the emergent behavior of SoES. $l$ is the number of functional emergent properties. The most typical functional emergent property identified in the external view model is timing properties such as maximum response time.

$H$ denotes non-functional emergent properties related to high confidence. It is described by a high-confidence metric vector. In this context, $H = (h_1, h_2, \ldots, h_z)$, where $h_i (i \in [1,z])$ is a set of metrics for a measure of high confidence. Some typical metrics are failure rate, maximum time between two successive failures, the number of faults that can be tolerated, maximum time between safety violations and security level etc [3].

The Internal view model focuses on the designer view. It describes SoES requirements by capturing their structures and behaviors. The internal view model, denoted as $\zeta$, is formally represented as follows:
\[ \zeta = (S, E, C, D, F_1, F_2) \]  

\( S \) is a component system set, \( S = \{ s_i | i \in [1, n] \} \). \( s_i \) denotes the component system constituting SoES (\( n \) is the number of component systems in the whole SoES). \( E \) denotes the interaction sets between component systems, \( E = \{ e_{jk} | j, k \in [1, n] \} \), where \( e_{jk} \) denotes a set of interactions from component system \( s_j \) to component system \( s_k \). \( C \) denotes constraint sets on how the component systems are used in the given environment, \( C = \{ c_i | i \in [1, n] \} \). \( c_i \) is a set of constraints imposed on \( s_i \). \( D \) denotes constraint sets on interactions between component systems, \( D = \{ d_{jk} | j, k \in [1, n] \} \), where \( d_{jk} \) is a set of constraints applied to interactions in \( e_{jk} \). \( F_1 \) and \( F_2 \) are two mappings that refine emergent properties of SoES into local constraint sets imposed on component systems and interactions, i.e., \( C = F_1(G, H) \) and \( D = F_2(G, H) \).

In internal view computational model, timing constraints are included in \( C \) and \( D \). Typical timing constraints include deadline and maximum execution time of the component system and latency of the interaction between two specific component systems. Furthermore, some resource constraints such as access mode and control constraints such as trigger method are also included in \( C \) and \( D \). All these constraints can be extracted as parameters used by dynamic scheduling analysis for SoES. In addition, each component system is either atomic or composite in internal view computational model. For convenience, to support the scheduling analysis, we take each atomic component system as a task to be scheduled by the scheduling algorithm. In later sections, the task refers to the atomic component system unless the specific explanation is applied.
4. Dynamic Scheduling Analysis for SoES

In our previous research, we presented an integrated dynamic scheduling algorithm for heterogeneous real-time systems [5]. In this section, we will show how this proposed scheduling algorithm can be applied to implement the admission control for dynamic software reconfiguration in SoES.

4.1 Task Model

Since systems of embedded systems are characterized by dynamic combinations of component systems, in this paper we only consider the asperiodic tasks. We have presented a task model for dynamic scheduling in heterogeneous real-time systems [5]. In this paper, we extend this task model for the use of dynamic scheduling analysis in SoES as follows:

i. Each task $T$ is described as a tuple $(a_T, r_T, D_T, v_T, E_T)$. Here, $a_T$ is $T$'s arrival time and $r_T$ is $T$'s ready time. $D_T$ denotes $T$'s deadline. $v_T$ is the number of $T$'s different logic versions. $E_T$ represents $T$'s maximum execution time. For hard real-time tasks, $E_T$ is a vector denoted by $(e_T^1, e_T^2, ..., e_T^n)$, where $e_T^j$ ($j = 1, ..., m$) is the maximum execution time of task when it executes on processor $p_j$. For soft real-time tasks, $E_T$ is a matrix denoted by $\{e_T^i\}$ ($i = 1, ..., v_T; j = 1, ..., m$), where $e_T^i$ is the maximum execution time of $T$'s logic version $i$ when it executes on processor $p_j$. Furthermore, for each $j(j = 1, ..., m)$, $e_T^1 \leq e_T^j \leq ... \leq e_T^m$.

ii. Tasks are non-preemptive, non-periodic; Tasks can not be parallelized.
iii. Besides processors, tasks might need some other resources such as data structures, variables and communication buffers for their executions. Every task can access a resource either in shared mode or in exclusive mode.

The characteristics of the task listed above can be extracted from internal view computational model addressed in section 3. For example, the deadline and maximum execution time of a task can be derived from constraint sets imposed on the corresponding component system, and the access mode of a task can be derived in the same way.

4.2 Precedence Graph

Since interactions between component systems are ubiquitous in SoES, precedence constraints should be thoroughly considered during the scheduling analysis. This constraint is to specify whether any task(s) needs to precede other task. If task $T_x$'s output is needed as input by task $T_y$, then task $T_y$ is constrained to be preceded by task $T_x$. Furthermore, there are two kinds of precedence constraints: one is AND constraint; another is OR constraint. Accordingly, there are two types of tasks: AND tasks cannot begin their computing before all their precedents have completed while OR tasks can commence after any one of their precedence complete.

**Definition 4.2.1:** $T_x \prec T_y$ represents that task $T_x$ must precede task $T_y$.

**Definition 4.2.2:** The precedent-task set of task $T_x$ is denoted by $\prec(T_x)$; that is, $\prec(T_x)$ indicates which tasks must be completed before $T_x$ can begin.

**Definition 4.2.3:** Assume $T_x$ is a task. $T_x,readytime$ denotes its ready time; $T_x,met$ denotes its maximum execution time; $T_x,starttime$ denotes its start time.
Definition 4.2.4: Assume $I_{xy}$ is the interaction between task $T_x$ and task $T_y$. $I_{xy,lat}$ denotes its latency.

The precedence constraint can be most conveniently represented by means of a precedence graph [2]. In precedence graph, each node represents a task (or a component system). The arrows indicate which task has precedence over which other task. Figure 2 shows an example of precedence graph.

![Figure 2 Example for precedence graph](image)

In Figure 2, there are 8 tasks denoted by $T_1, T_2, ..., T_8$. We have

\[
\begin{align*}
\prec (T_1) &= \emptyset \\
\prec (T_2) &= \{T_1\} \\
\prec (T_3) &= \{T_1\} \\
\prec (T_4) &= \{T_1\} \\
\prec (T_5) &= \{T_2, T_3\} \\
\prec (T_6) &= \{T_3, T_4\} \\
\prec (T_7) &= \{T_4\} \\
\prec (T_8) &= \{T_6\}
\end{align*}
\]

The precedence graph can be constructed based on internal view computational model since precedence relationships between tasks can be derived from interactions between component systems described in this model. In this case, AND and OR tasks can be identified.
by a control constraint, i.e., trigger method imposed on each component system, which is described in internal view computational model [3]. If the trigger method of a component system is *trigger by ALL*, this component system will be taken as a AND task. If the trigger method of a component system is *trigger by SOME*, this component system will be considered as a OR task. In addition, the latency of interaction between two specific tasks can be derived from the constraint of latency imposed on corresponding interaction described in internal view computational model. According to precedence relationships, we can compute the start time for each task described in the precedence graph. The principle for this computation is described as follows:

If \( T_x \) is AND task, \( T_y \prec T_x \) and \( T_z \prec T_x \),

\[
T_x.\text{starttime} \geq \text{MAX}(T_x.\text{readytime}, \text{MAX}(T_y.\text{starttime} + T_y.\text{met} + I_{y_1}.\text{lat}, T_z.\text{starttime} + T_z.\text{met} + I_{z_1}.\text{lat}))
\]

If \( T_y \) is OR task, \( T_y \prec T_x \) and \( T_z \prec T_x \),

\[
T_y.\text{starttime} \geq \text{MAX}(T_x.\text{readytime}, \text{MIN}(T_y.\text{starttime} + T_y.\text{met} + I_{y_1}.\text{lat}, T_z.\text{starttime} + T_z.\text{met} + I_{z_1}.\text{lat}))
\]

<table>
<thead>
<tr>
<th>Interactions</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_{12} )</td>
<td>2</td>
</tr>
<tr>
<td>( I_{13} )</td>
<td>3</td>
</tr>
<tr>
<td>( I_{14} )</td>
<td>5</td>
</tr>
<tr>
<td>( I_{25} )</td>
<td>3</td>
</tr>
<tr>
<td>( I_{35} )</td>
<td>8</td>
</tr>
<tr>
<td>( I_{36} )</td>
<td>6</td>
</tr>
<tr>
<td>( I_{46} )</td>
<td>10</td>
</tr>
<tr>
<td>( I_{47} )</td>
<td>9</td>
</tr>
<tr>
<td>( I_{68} )</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1  Latency of interactions in Figure 2
Assuming the latency of interactions in Figure 2 are listed in table 1 and the ready time for each task in Figure 2 is 0, we can derive the start time for each task in Figure 2 under given maximum execution time. The computation results are listed in Table 2. Here, we assume that $T_2$, $T_3$, $T_5$ and $T_8$ are AND tasks while $T_4$, $T_6$ and $T_7$ are OR tasks.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Maximum Execution Time</th>
<th>Start Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$T_2$</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>$T_3$</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>$T_4$</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>$T_5$</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>$T_6$</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$T_7$</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>$T_8$</td>
<td>3</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 2 Computation results for start time of tasks in figure 2

4.3 Dynamic Scheduling Algorithm for SoES

In this section, we will extend our previous integrated dynamic scheduling algorithm for heterogeneous real-time systems [5] to implement dynamic scheduling analysis for SoES. For this purpose, we consider the hardware environment of SoES as a heterogeneous multi-processor system which can be described as follows: (Here, we only consider the speed difference of these processors during scheduling.)

Assume there are $m$ processors ($m > 1$), denoted as $p_1, p_2, ..., p_m$. Each processor $p_i$ ($i = 1, ..., m$) is assigned a real number $t_i$ ($i = 1, ..., m$) which is proportion to its speed, i.e., faster processor is assigned to a greater $t_i$. At the same time, $\exists i, \exists j, 1 \leq i \leq m, 1 \leq j \leq m$ and $i \neq j$, Thus, $t_i \neq t_j$. 

13
4.3.1 Definitions

In this section, we will give a set of definitions to facilitate the discussion.

**Definition 4.3.1.1** A task is feasible in a schedule if its timing constraint and resource requirements are met in the schedule. A schedule for a set of tasks is said to be a feasible schedule if all the tasks are feasible in the schedule [6].

**Definition 4.3.1.2** A partial schedule is a feasible schedule for a subset of tasks. A partial schedule is said to be strongly feasible if all the schedules obtained by extending the current schedule by any one of the remaining tasks are also feasible [4] [6].

**Definition 4.3.1.3** \( EAT_k \) (\( EAT_d \)) is the earliest time when resource \( R_k \) becomes available for shared (or exclusive) access [4].

**Definition 4.3.1.4** \( IEST(T) \) is the ideal earliest start time of task \( T \). Let \( PE \) be the set of processors and \( R_T \) be the set of resources required by task \( T \). Thus, \( IEST(T) = \text{MAX}(T_{\text{starttime}}, \text{MIN}_{P \in PE} \text{avail}(P), \text{MAX}_{R \in R_T} EAT_{R}^u) \). Here, \( T_{\text{starttime}} \) is the start time for task \( T \). To derive the start time for task \( T \), we should go through the precedence graph for whole system and find out the precedent-task set of task \( T \). Based on this, we can use the principle described in section 4.2 to compute the start time for task \( T \). \( \text{avail}(P) \) denotes the earliest time at which the processor \( P \) becomes available for executing a task and the third term denotes the maximum among the earliest available times of the resources required by task \( T \) (\( u = s \) for shared mode and \( u = e \) for exclusive mode).

**Definition 4.3.1.5** \( \text{avail}(T,P) \) denotes the feasibility of task \( T \) for executing on the processor \( P \). If task \( T \)'s deadline can be met when it executes on processor \( P \), \( \text{avail}(T,P) = 1 \); otherwise, \( \text{avail}(T,P) = 0 \).
Definition 4.3.1.6 \textit{sysavailtime} is the minimum available time of processors in whole systems, i.e., \( \text{sysavailtime} = \text{MIN}_{P \in P}(\text{availtime}(P)) \).

Definition 4.3.1.7 \( \text{Spe}(P) \) reflects the speed of processor \( P \). The lower speed of processor \( P \), the larger value of \( \text{Spe}(P) \).

Definition 4.3.1.8 \( \text{Finishtime}(P,T) \) denotes the finish time of task \( T \) when it executes on processor \( P \).

4.3.2 Overview of Algorithm

This scheduling analysis algorithm is to dynamically and integratedly schedule a task set with combination of the tasks with hard and soft deadlines. It is based on heuristic searching [6]. When a set of new component systems with precedence constraints and resource constraints arrive system, these component systems together with other tasks that have already been in the system but have not been scheduled will consist the unscheduled task sets and this scheduling analysis algorithm will be triggered.

In this algorithm, the schedule starts at the root of the search tree which is an empty schedule, the algorithm tries to extend the schedule (with one more task) by moving to one of the vertices at the next level in the search tree until a full feasible schedule is derived. For this purpose, we set the feasibility check window with \( K \) size for the unscheduled task set. Then, we will check the feasibility of the current schedule by extending it with each task in this window. During this process, once the current schedule is not feasible due to certain task in the feasibility check window, we will use certain degradation policy described in section 4.3.4 to degrade this task.
If the current schedule is strongly feasible, we will choose the task with minimal value of heuristic function $H$ to extend the current schedule. The heuristic function $H$ for task selection is $H(T) = D_T + W \cdot IEST(T)$, where $D_T$ is the deadline of task $T$ and $W$ is a weight. Once a task within the feasibility check window is selected to extend current schedule, the task will be assigned to a specific processor according to the task assignment policy which will be described in section 4.3.3.

On the other hand, if the current schedule is not strongly feasible (even after all soft real-time tasks in the feasibility check window that cause infeasibility of current schedule have been degraded to the lowest service level), we will backtrack to the previous search level. Then, we extend the current schedule by another task having the next smallest $H$ value in this search level and choose a suitable processor for this task according to the same task assignment policy. Furthermore, if a task in the feasibility check window is associated with a new component system and is found to result in infeasibility of current schedule by the feasibility check, we will compute and record its ‘maximum execution time threshold’ (METT). Here, METT refers to the upper bound of the maximum execution time which is needed to make a task feasible in current schedule. Assume $E_T^*$ represents the METT of task $T$. Then, $E_T^* = D_T - IEST(T)$.

After the feasibility check for current schedule, the feasibility check window is moved by one task. Above process will be repeated until a complete feasible schedule has been found or no more backtracking is possible. In this case, if the complete feasible schedule is found, it means that adding new component systems will not violate schedulability of whole system so that the dynamic reconfiguration is accepted. Otherwise, it means that adding new component
systems will result in unschedulability of SoES and consequently damage high confidence of whole system. Thus, the dynamic reconfiguration is rejected.

In case the dynamic reconfiguration is rejected, we need to further detect if unschedulability of whole system is resulted from certain new component systems. If so, the recorded METT of tasks associated with these new component systems will be provided as the suggested maximum execution times to the ‘training process’ for virtually reducing their maximum execution times.

4.3.3 Task Assignment Policy

The task assignment policy is to select the most suitable processor to execute task T which is selected to extend the current partial schedule during heuristic searching. We will use a heuristic function $S$ developed in [5] to achieve this goal. For each processor $P$, $S(P) = availtime(P) + Spe(P)$. The basic idea of this task assignment policy is that we always select the processor having largest value of heuristic function $S$ for task $T$’s execution since earlier sysavailtime and faster speed of processor having the minimum available time after executing task $T$ can lead to higher feasibility of unscheduled tasks. Furthermore, we will check whether all successive task of task $T$ can meet their deadlines if $T$ is executed in the processor having the largest $S$ value. If the answer is negative, we should choose the processor having the next large $S$ value.

In this dynamic scheduling analysis algorithm, the new task assignment policy is denoted as function $choosep(T)$. This function returns the identification number of processor selected.
for task $T$'s execution or NULL. The detailed task assignment policy is listed below: (We assume that $ESTR_T$ is the earliest available time of the resources required by task $T$.)

1) If the resources required by task $T$ are no more than processors, we will choose processor $P$ which can meet the following constraint for task $T$. The constraint is:

$$S(P) = \text{MAX}_{pe \in \{pe|pe \in PE \text{ and } \text{avail}(T,pe)=1\}} S(pe)$$

2) If task $T$ requires some other resources besides the processors, we will employ the method listed below:

2.1) If there is no intersection between the resources required by task $T$ and the resources required by remaining tasks, we will employ the same processor selection policy as 1) for task $T$.

2.2) If there is an intersection between the resources required by task $T$ and the resources required by remaining tasks, and both task $T$ and remaining tasks have shared access to the resources in this intersection, we will employ the same processor selection policy as 1) for task $T$.

2.3) In other cases, our policy used to choose the processor for task $T$ is listed below:

(We assume that processor $P'$ meets following constraint. The constraint is:

$$\text{availtime}(P') = \text{MAX}_{pe \in \{pe|pe \in PE \text{ and } \text{avail}(T,pe)=1\}} (\text{availtime}(pe))$$

2.3.1) If $r_T \leq ESTR_T$ and $ESTR_T = \text{MAX}_{pe \in \{pe|pe \in PE \text{ and } \text{avail}(T,pe)=1\}} (\text{availtime}(pe))$ and $Spe(P') = \text{MAX}_{pe \in \{pe|pe \in PE \text{ and } \text{avail}(T,pe)=1\}} (Spe(pe))$, we will employ the same processor selection policy as 1) for task $T$.}
2.3.2) If \( r_T \geq ESTR_T \) and \( r_T \geq \max_{pe \in PE \text{ and } \text{avail}(T,pe)=1} (\text{availtime}(pe)) \), we will employ the same processor selection policy as 1) for task \( T \).

2.3.3) In other cases, we will choose processor \( P \) which can meet the following constraint for task \( T \). The constraint is:

\[
\text{Finishtime}(P,T) = \min_{pe \in PE \text{ and } \text{avail}(T,pe)=1} \text{Finishtime}(P,T)
\]

3) Check whether all successive tasks of task \( T \) can meet their deadlines under current processor selection.

3.1) If it is not true, choose another processor with the next largest \( S \) value for case 1), 2.1), 2.2), 2.3.1) and 2.3.2); choose another processor with the next minimum value of function \( \text{Finishtime} \) for case 2.3.3).

3.2) Go back to 3 until the processor is selected or no more processor can be selected

4) If a processor is selected, return the identification number of this processor; otherwise, return NULL.

4.3.4 Degradation Policy

In [5], we presented a degradation policy for soft real-time tasks to improve schedulability of hard real-time tasks by comprising on result quality of soft real-time tasks. In the dynamic scheduling analysis algorithm for SoES, we use this degradation policy to iteratively degrade quality of service (QoS) of soft real-time tasks in necessary cases so as to give maximum possible quality while meeting the deadline. During the feasibility check, at first, we use the logic version with longest execution time of each soft real-time task in the feasibility check.
window to perform feasibility check. Once the current schedule is not a strong feasible due to unschedulability of a soft real-time task in the feasibility check window, QoS of this soft real-time tasks is degraded to the next lower level. This degradation will continue till the feasibility check is successful. If the feasibility check is still not successful when QoS of this soft real-time task has already been degraded to the lowest level, a backtracking will happen.

We denote the detailed degradation policy for task $T$ as a function $\text{degrade}(T)$ and the return value of this function is the number of task $T$’s allowable logic version, i.e., the actual service level that can be provided for task $T$. The detail of function $\text{degrade}(T)$ is listed below:

1) If task $T$ which results in unfeasibility of current schedule is a soft real-time task

   (1.1) Check the current service level of task $T$.

   (1.2) If the current service level is not the lowest, we will degrade the task $T$’s service level to the next lower one, i.e., select a logic version with the next shorter execution time to the current service level for task $T$.

   (1.3) Repeat (1.1) – (1.2) until the service level of task $T$ has been degraded to the lowest one or the feasibility check is successful.

2) Else exit.

4.3.5 Algorithm Description

In summary, the dynamic scheduling analysis algorithm for SoES is given below:

1) Order the tasks in the task queue in non-decreasing order of their deadlines and then start with an empty partial schedule.

2) Check the feasibility window.

3) For $i = 1$ to $K$ (or less $K$)
3.1) If the schedule is not feasible by extending the current schedule with task $T_i$

Then degrade($T_i$);

4) Determine whether the current partial schedule is strongly feasible by performing feasibility check for $K$ or less than $K$ tasks in the feasibility check window. If the current partial schedule is strongly feasible, $\text{feasible} = \text{true}$; otherwise $\text{feasible} = \text{false}$.

5) If ($\text{feasible} == \text{true}$)

5.1) Compute the heuristic function $H$ value for the $K$ or less than $K$ tasks in the feasibility check window, where $H(T) = D_r + W * IEST(T)$ for task $T$.

5.2) Extend the schedule by task $T$ having the smallest $H$ value in the feasibility check window and choose a suitable processor for task $T$ by using function $\text{choosep}(T)$

5.3) If $\text{choosep}(T) = \text{NULL}$, go to 5.4).

Else

5.4) For each task $T$ in the feasibility check window, if $T$ is associated with a new component system and is not feasible in the current schedule, compute and record $E^*_T$ (the maximum execution time threshold (METT) of task $T$)

5.5) Backtrack to the previous search level.

5.6) Extend the schedule by the task $T'$ having the next smallest $H$ value in this search level and choose a suitable processor for task $T'$ by using function $\text{choosep}(T')$.

6) Move the feasibility check window by one task.

7) Repeat steps 2-6 until any termination condition listed below is met:
a) a complete feasible schedule has been found;

b) no more backtracking is possible

8) If a complete feasible schedule is found

Accept the addition of new component systems and the reconfiguration

Else

Reject the addition of new component systems and the reconfiguration;

If the task associated with the new component system results in unschedulability of whole system, provide METT of the task to ‘training process’

5. Example

Example 5.1: Assume at the beginning, there are 6 component systems in the system, i.e., \( s_1, s_2, \ldots, s_6 \). We assume that at the time of 30, a dynamic reconfiguration will happen, i.e., a component system \( s_7 \) will be added into the whole system. In the later section, we will use the proposed admission control method to check whether this dynamic software reconfiguration is allowed.

5.1 Computational Model

Figure 3 is the graphical internal view computational model for the system described in example 5.1. In figure 3, the part with solid lines is to describe the system before dynamic software reconfiguration while the part with dash lines is to describe the configuration changes. The bubbles labeled with \( s_1, s_2, \ldots, s_7 \) represent component systems. The arrows labeled with \( e_{13}, e_{23}, e_{25}, e_{34}, e_{35}, e_{46}, \) and \( e_{67} \) represent interactions between component systems. DL and MET are timing constraints imposed on each component system. DL denotes the deadline of the component system while MET denotes the maximum execution
time of the component system. TRI is a control constraint imposed on the component system. It represents the trigger method for each component system. The trigger method for component system $s_3$ is ALL \textit{(Trigger by ALL)} and the trigger method for component system $s_5$ is SOME \textit{(Trigger by SOME)}. LAT is a timing constraint imposed on interactions. It denotes the latency of the interaction between two specific component systems. The internal component model for this example can be formally represented as follows:

![Diagram](image)

\textbf{Figure 3 Computational Model for Example 2}

Before dynamic reconfiguration, the component system set $S = \{s_1, s_2, \ldots, s_6\}$; after dynamic reconfiguration, the component system set $S = \{s_1, s_2, \ldots, s_7\}$.

Before dynamic reconfiguration, the interaction set $E = \{e_{12}, e_{23}, e_{25}, e_{34}, e_{35}, e_{46}\}$; after dynamic reconfiguration, the interaction set $E = \{e_{12}^{1}, e_{23}^{1}, e_{25}^{1}, e_{34}^{1}, e_{35}^{1}, e_{46}^{1}\}$.

Before dynamic reconfiguration, the constraint set imposed on component systems $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$; after dynamic reconfiguration, the constraint set imposed on component systems $C = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$.

$c_i = \{c_i, c_i^2\}$; $c_i^3$: DL = 12, $c_i^4$: MET = 10;
Before dynamic reconfiguration, the constraint set imposed on interactions

\[ D = \{d_{13}, d_{23}, d_{25}, d_{35}, d_{34}, d_{46}\} \]; after dynamic reconfiguration, the constraint set imposed on interactions \( D = \{d_{13}, d_{23}, d_{25}, d_{35}, d_{34}, d_{46}, d_{67}\} \)

\[ d_{13} = \{d_{1}\}; \quad d_{13} : \text{LAT} = 2; \]
\[ d_{23} = \{d_{23}\}; \quad d_{23} : \text{LAT} = 3; \]
\[ d_{25} = \{d_{25}\}; \quad d_{25} : \text{LAT} = 6; \]
\[ d_{35} = \{d_{35}\}; \quad d_{35} : \text{LAT} = 5; \]
\[ d_{34} = \{d_{34}\}; \quad d_{34} : \text{LAT} = 2; \]
\[ d_{46} = \{d_{46}\}; \quad d_{46} : \text{LAT} = 6; \]
\[ d_{67} = \{d_{67}\}; \quad d_{67} : \text{LAT} = 2; \]

5.2 Precedence Relationship

According to the interactions between component systems and the trigger method imposed on each component system which is described in the computational model in figure 3, the precedence relationship can be derived as follows:

\[ \prec (s_1) = \emptyset \]
\[ \prec (s_2) = \emptyset \]
\[ \prec (s_3) = \{s_1, s_2\}, \quad s_3 \text{ is a AND task} \]
\[ \prec (s_4) = \{s_3\} \]
\[ \prec (s_5) = \{s_2, s_3\}, \quad s_5 \text{ is a OR task} \]
\[ \prec (s_6) = \{s_4\} \]
\[ \prec (s_7) = \{s_6\} \]

5.3 Computation Results
We assume that there are 3 processors $P_1, P_2, P_3$ in the system. Here, $P_1$ is the baseline processor. $P_2$ is two times faster than $P_1$ while $P_3$ is three times faster than $P_1$. There are two resources $R_1, R_2$, and each resource has one instance. Task/component system $s_1$ and $s_6$ access resource $R_1$ in a shared mode while task/component system $s_2$ accesses resource $R_2$ in an exclusive mode. Furthermore, $s_4$ is a soft real-time task and other tasks are hard real-time tasks. $s_4$ has 2 logic versions. The parameters for each task/component system are listed in Table 3. These parameters are derived from the computational model described in Figure 3.

<table>
<thead>
<tr>
<th>Tasks/Component Systems</th>
<th>Ready Time</th>
<th>Execution Time</th>
<th>Deadline</th>
<th>Resource Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>0</td>
<td>(10, 5, 3.3)</td>
<td>12</td>
<td>Shared</td>
</tr>
<tr>
<td>$s_2$</td>
<td>0</td>
<td>(15, 7, 5)</td>
<td>18</td>
<td>Exclusive</td>
</tr>
<tr>
<td>$s_3$</td>
<td>0</td>
<td>(15, 7, 5)</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>$s_4$</td>
<td>0</td>
<td>$\begin{pmatrix} 12,6,4 \ 6,3,2 \end{pmatrix}$</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>$s_5$</td>
<td>0</td>
<td>(15, 7, 5)</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>$s_6$</td>
<td>0</td>
<td>(10, 5, 3.3)</td>
<td>35</td>
<td>Shared</td>
</tr>
<tr>
<td>$s_7$</td>
<td>0</td>
<td>(5, 2.5, 1.6)</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3 Tasks/Component Systems Parameters**

Furthermore, we assume $W = 1$ and $K = 2$. The results of applying proposed admission control method to the system are described in Figure 4. Here, each box represents a node in the schedule. Three numbers in each box represent the earliest available times for these three processors $P_1, P_2, P_3$. $s_a(b)$ denotes task/component system $s_a$ executes on processor $P_b$.

In Figure 4, at the beginning, the scheduling queue is empty. In each step, we choose the task to be executed and the processor executes this selected task. By using proposed admission control method, we detect that addition of component system $s_7$ will eventually result in unschedulability of whole system so that this dynamic software reconfiguration is
rejected. Furthermore, the METT of $s_7$ is given as the suggested maximum execution time of $s_7$ to the ‘training process’.

![Diagram](image)

**Figure 4 Admission Control for Example 5.1**

6. Discussion of the Proposed Approach

Although this paper only addresses admission control for dynamic reconfiguration in term of adding new component systems, the proposed framework in Figure 1 can be also applied to admission control for dynamic reconfiguration in case of removing existing component systems and software component immigration. In this case, we will construct the
computational model of whole system after component systems are removed or software components are immigrated and extract the corresponding task model from this computational model. After that, we will use the same dynamic scheduling analysis algorithm described in section 4.3 to detect schedulability of whole system after certain existing component systems are removed or after certain software components are immigrated. If whole system is still feasible to be scheduled, the reconfiguration is allowed, otherwise it is rejected.

7. Conclusions

Systems of embedded systems (SoES) usually have frequent changes in requirements, mission, environment and technologies. This trait makes dynamic software reconfiguration become an important issue in SoES development. Although many works have been done on this aspect, they do not realize potential negative impacts on high confidence of whole system caused by dynamic software reconfiguration. This paper takes this situation into consideration and presents an admission control method to prevent dynamic software reconfiguration from damaging the confidence of SoES. The core of this method is the dynamic scheduling analysis for SoES. Once new component systems arrives SoES, dynamic scheduling analysis is applied to check whether addition of these component systems will result in unschedulability of SoES. If SoES is still feasible to be scheduled after adding new component systems, the reconfiguration will be accepted; otherwise it will be rejected. Furthermore, a ‘training process’ will be triggered to find ‘more suitable’ maximum execution times of new component systems in case addition of these new component systems with current maximum execution times violates schedulability of whole system. An example is
also addressed to show how the proposed admission control method is applied to dynamic software reconfiguration in SoES.

References


