Proceedings of the
47th IEEE Conference on Decision and Control
Cancun, Mexico, Dec. 9-11, 2008

Self Configuration of Dependent Tasks for Dynamically Reconfigurable Automotive Embedded Systems

Lei Feng, DeJiu Chen, Martin Törngren

Abstract— **The configurations of an automotive embedded system are normally fixed in production and remain static over the vehicle lifetime. Future scenarios, however, call for more flexible configuration support. DySCAS (Dynamically Self-Configuring Automotive Systems) project aims to introduce context-awareness and self-management features into automotive embedded systems via middleware technologies. Contributing to online configuration decisions, this paper formalizes a fundamental self-configuration problem. It forms a basis for managing the cross interdependencies of configurational items, assessing the system-wide impacts of changes, and making dynamic decisions about new configurations.**

I. INTRODUCTION

A modern automobile normally contains dozens of *electronic control units* (ECUs) [1], [2], which communicate and collaborate through networks. Automotive manufactures currently configure the functions of the ECUs statically at design time. AUTOSAR (AUTomotive Open System ARchitecture) [3], for instance, is an emerging standard to address the software integration challenge via promoting the platform transparencies, interface standardization, and code portability of automotive software [4], but the support is delimited to static configuration.

The DySCAS (*Dynamically Self-Configuring Automotive Systems*) [5], [6] project aims to develop a middleware architecture and technologies that allow context-awareness and self-adaptiveness of automotive embedded systems, in particular in the vehicular infotainment domain. For example, a vehicle should be able to automatically detect various external devices attached to it, e.g., mobile phones and PDAs, and integrate them into the vehicular network. Exploiting their computational resources and services, the vehicle can then provide additional personalized services.

The main research objectives of the DySCAS project include a middleware architecture that supports context-awareness and dynamic configuration management using policies [6], [7], and a set of algorithms that determine new system configurations in the events of, e.g., attaching/detaching new devices [6], hardware failures, and anomalous resource usage. Contributing to the latter objective, this paper formalizes a *self-configuration* problem, which focuses on the interdependencies of (application) tasks, the system-wide impacts of changes, and the configuration decisions on tasks. The result will support *Autonomic Configuration Management* [7] service of the DySCAS middleware.

As a part of DySCAS project, this work is funded by the 6th framework program "Information Society Technologies" of the European Commission. Project number: FP6-IST-2006-034904.

The authors are with the Department of Machine Design, Royal Institute of Technology - KTH, Stockholm, Sweden. {leifeng, chen, martin}@md.kth.se

The self-configuration problem is as follows. Suppose a set of ECUs, with limited memory and CPU resources, connected by a local area network (LAN). Each ECU is assigned with a group of tasks with specified resource usage and timing requirements, and whose executing status depends on the availability of required signals in the ECU system. Find an algorithm to determine the tasks that may successfully run and meet their timing requirements. Here a task is a concurrent thread of an application program running on top of the real-time operating system (RTOS) and middleware at an ECU. We assume that the task-to-ECU allocation [8], [9] has been decided, and do not consider in this paper run-time task-migration [10]. The only action on the tasks is to switch them on or off and only the selected tasks will be scheduled in RTOS.

While many research projects and publications on reconfigurable software emerged recently [11]–[15], most of them propose new software architectures and focus on implementation techniques. To the best of our knowledge, there is no report on any *distributed* algorithmic solution of the self-configuration problem formalized above.

Roman and Islam [13] developed a middleware infrastructure that allows the user to specify the execution order of micro building blocks (MBBs) as a directed graph and manipulate it for system reconfiguration. Similar to our work, Kinnebrew et al. [15] also address the task selection problem based on the data dependencies of tasks. The most relevant tasks for a mission goal are selected using the spreading activation task network. The fundamental difference of our work is that we do not assume a global model of all tasks in the system. The decision is based on distributed computation.

Section II presents the model of task execution process. Section III introduces a simple recursive algorithm to solve the task self-configuration problem. The algorithm is further verified through a Matlab simulation in Section IV. Finally Section V draws the conclusion and outlines future works.

II. TASK EXECUTION MODEL

In a networked embedded system, software tasks communicate via input and output signals. Thanks to communication transparency support [7], tasks access signals and I/O ports by directly referring to their *names* without any knowledge of their actual locations in the network. At one ECU, the names and signals have a one to one mapping. In the sequel, we do not distinguish a signal and its name.

Consider a network of m ECUs, where each ECU $k \in \mathbf{m} := \{1, \cdots, m\}$ runs an instance of the DySCAS middleware. Every ECU works as a node in the network. Supported by middleware services, an ECU directly manages analog and digital I/O ports. Let IP_k $(k \in \mathbf{m})$ be the name set of the input ports at ECU k and OP_k the set of the output ports.

The UML state machine diagram in Fig. 1 shows the execution procedure of one task. To avoid clutter, we ignore the indexes of the events and the event labels within state "Active" as they are well known. At the initial state, the task is not loaded. Then it is deployed to an ECU via event *load*. The event may occur not only at the startup process, but also be invoked by the middleware for dynamic configuration, e.g., attaching new devices and task migration. Upon the occurrence of event *load*, the task must allocate memory space of the ECU for code and data entries. In case of inadequate memory space, the middleware must disable event *load*.

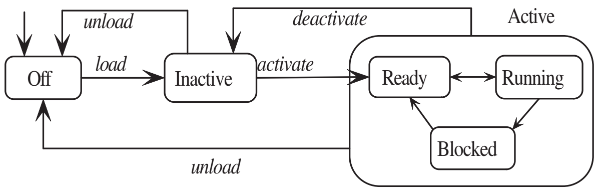

Fig. 1. Task Execution Model

Suppose that there are n_k tasks loaded at ECU $k \in \mathbf{m}$. Denote each task τ_{ik} $(i \in \mathbf{n_k}, k \in \mathbf{m})$. After loaded, every task registers to the middleware at ECU k its input and output names, I_{ik}, O_{ik} $(i \in \mathbf{n_k}, k \in \mathbf{m})$. Let

$$I_k := \bigcup_{i=1}^{n_k} I_{ik} \text{ and } O_k := \bigcup_{i=1}^{n_k} O_{ik} \tag{1}$$

be the collections of input and output names managed at ECU k. Since these tasks within ECU k may communicate, a name can be the output of one task and the input of itself or another, i.e., it may be true that $I_k \cap O_k \neq \emptyset$.

At state "Inactive", if the middleware can locate all the required inputs of the task, event *activate* may happen. The occurrence of this event, however, also relies on the CPU utilization of the ECU. If the CPU is too busy to execute the task, the middleware must disable the event and choose the right response according to the designer's preference: The task may stay at state "Inactive", still occupy the memory, and wait for required inputs or CPU slots. Alternatively, it may be unloaded from the memory via event *unload*.

At state "Active", the task has secured the necessary inputs and resources, and operates under the scheduling of an RTOS, as illustrated by the sub machine within "Active". Event *deactivate* switches off the task in the CPU but does not affect the memory. Event *unload* removes the task from the ECU. At ECU $k \in \mathbf{m}$, we denote all the active tasks as a set TA_k. Then the outputs of these active tasks are available to the whole system. They form the set

$$OA_k := \bigcup \{O_{ik} | i \in \mathbf{n_k}, \tau_{ik} \in TA_k\} \tag{2}$$

These available outputs will be used as inputs by the tasks in the system. Hence the available inputs for ECU k form the set

$$IA_k := [I_k \cap (\bigcup_{j=1}^{m} OA_j)] \cup IP_k \tag{3}$$

Based on the available inputs at ECU k, the set of active tasks at the ECU is

$$TA_k \subseteq \{\tau_{ik} | I_{ik} \subseteq IA_k\} \tag{4}$$

which must be *schedulable* [16], namely all members in the set meet deadlines. The necessary condition to activate τ_{ik} is that all its inputs are available, i.e., $I_{ik} \subseteq IA_k$.

Equations (2) to (4) are closely coupled. We must carefully choose the active task sets TA_k $(k \in \mathbf{m})$ to satisfy the three equations. Moreover, it is NP-hard to find an "optimal" solution that maximizes, for example, the total number of active tasks. The startup and reconfiguration processes of the middleware are essentially iterative procedures to find a solution of this problem. The solution need not be optimal owing to the high complexity.

III. DYNAMIC CONFIGURATION PROCESS

When dynamic configuration is wanted, e.g., in the cases of system startup or new device attachment, the middleware services at the ECUs communicate through the network and find active tasks at all ECUs. Our underlying assumption is that the network need not have a master ECU, so that a global description of the entire system configuration may be unavailable.

A. *Ignoring CPU Limitation*

Starting from a simple question, we temporarily ignore the CPU limitation; hence (4) is simplified as

$$TA_k := \{\tau_{ik} | i \in \mathbf{n_k}, I_{ik} \subseteq IA_k\} \tag{5}$$

We propose an iterative selecting procedure to find the active tasks at every ECU. The procedure starts by supposing all the loaded tasks to be active, and then verifies if these active task sets satisfy (5). We remove the tasks that do not have necessary inputs and then update the sets of available outputs and inputs at all ECUs. This change may affect the active task sets again. The procedure is formalized as follows.

Let T_k be all the tasks already loaded at ECU $k \in \mathbf{m}$. Let I_k, O_k, computed by (1), be the input and output name sets at ECU k. Because we initially assume that all tasks are active, for all $k \in \mathbf{m}$ we have

$$\begin{aligned} TA_k(0) :=& T_k \\ OA_k(0) :=& \bigcup \{O_{ik} | \tau_{ik} \in TA_k(0)\} = O_k \\ IA_k(0) :=& [I_k \cap (\bigcup_{j=1}^{m} OA_j(0))] \cup IP_k \end{aligned}$$

Let integer $r = 0, 1, \cdots$. From $TA_k(r), OA_k(r), IA_k(r)$, we can derive

$$TA_k(r+1) := \{\tau_{ik} | I_{ik} \subseteq IA_k(r)\}$$
$$OA_k(r+1) := \bigcup \{O_{ik} | \tau_{ik} \in TA_k(r+1)\}$$
$$IA_k(r+1) := [I_k \cap (\bigcup_{j=1}^{m} OA_j(r+1))] \cup IP_k$$

If the sequence $\{TA_k(r)\}$ converges, its final limit is then the set of possible active tasks at ECU k.

Proposition 1: Sequences $\{TA_k(r)\}$ for all $k \in \mathbf{m}$ as defined above converge.

Proof: Using mathematical induction, we prove that for all $k \in \mathbf{m}$ the three sequences $\{TA_k(r)\}$, $\{OA_k(r)\}$, $\{IA_k(r)\}$ are all nonincreasing.

When $r = 0$, we immediately have $TA_k(1) \subseteq TA_k(0)$ and $OA_k(1) \subseteq OA_k(0)$, as $TA_k(0) = T_k$ and $OA_k(0) = O_k$. Consequently, $\bigcup_{k=1}^{m} OA_k(1) \subseteq \bigcup_{k=1}^{m} OA_k(0)$ and $IA_k(1) \subseteq IA_k(0)$.

Suppose that for any integer r, it is true that $TA_k(r+1) \subseteq TA_k(r)$, $OA_k(r+1) \subseteq OA_k(r)$, $IA_k(r+1) \subseteq IA_k(r)$. By the definition of the sequences and the induction assumption, we can show

$$TA_k(r+2) = \{\tau_{ik} | I_{ik} \subseteq IA_k(r+1)\}$$
$$\subseteq \{\tau_{ik} | I_{ik} \subseteq IA_k(r)\} = TA_k(r+1)$$

Using the same argument, we further have $OA_k(r+2) \subseteq OA_k(r+1)$ and $IA_k(r+2) \subseteq IA_k(r+1)$. Therefore sequences $\{TA_k(r)\}$, $\{OA_k(r)\}$, and $\{IA_k(r)\}$ are all nonincreasing.

Since $TA_k(0)$ are finite for all $k \in \mathbf{m}$, there must be an index such that $(\forall r \geq r^*) TA_k(r) = TA_k(r^*)$ for all $k \in \mathbf{m}$. Sequences $\{TA_k(r)\}$ converge to sets $TA_k(r^*)$ for all $k \in \mathbf{m}$. ■

Alternatively, one may propose an incremental configuration process that begins with empty task sets and gradually adds the tasks whose inputs are all available. This process, unfortunately, does not work for tasks forming loops in the network. For example, Fig. 2 shows two tasks forming a loop. Suppose Task 1 is deployed at ECU1 and Task 2 at ECU2. If we start from the empty sets at both ECUs, i.e., $TA_1(0) = TA_2(0) = \emptyset$, then $OA_1(0) = OA_2(0) = \emptyset$, $IA_1(0) = \{inPort\}$, and $IA_2(0) = \emptyset$. Consequently, $TA_1(1) = TA_2(1) = \emptyset$ and the two sequences both converge to $\emptyset$. Evidently, this is incorrect. The method proved in Proposition 1, however, can achieve the right answer.

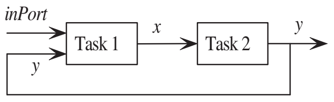

Fig. 2. Tasks Forming a Loop

In the configuration process, network communications are required to obtain sets $IA_k(r+1)$, which use $OA_k(r+1)$ from all ECUs in the system. To reduce the communication overhead and avoid the synchronization of all ECUs, we hope to update $IA_k(r)$ to $IA_k(r+1)$ by considering only the output names that have been changed in $OA_k(r+1)$, namely, those in sets $OA_k(r) - OA_k(r+1)$.

To this end, we require the system to satisfy the condition that the output name sets of all ECUs are pairwise disjoint, namely,

$$O_{ik} \cap O_{i'k'} \neq \emptyset \Rightarrow k = k' \quad (6)$$

The limitation is not mandatory, but a helpful suggestion for implementation. According to (6), if two tasks output an identical name, then they must reside at the same ECU. In real implementation, if two ECUs do output an identical name, we simply add ECU stamps to it. Define the following sequences:

$$TA'_k(0) := T_k,\ OA'_k(0) := \bigcup \{O_{ik} | \tau_{ik} \in TA'_k(0)\}$$

$$IAO'_k(0) := I_k \cap (\bigcup_{j=1}^{m} OA'_j(0)),\ IA'_k(0) := IAO'_k(0) \cup IP_k$$

Let integer $r = 0, 1, \cdots$. We further derive

$$TA'_k(r+1) := \{\tau_{ik} | I_{ik} \subseteq IA'_k(r)\}$$
$$OA'_k(r+1) := \bigcup \{O_{ik} | \tau_{ik} \in TA'_k(r+1)\}$$
$$\Delta OA'_k(r+1) := OA'_k(r) - OA'_k(r+1)$$
$$IAO'_k(r+1) := IAO'_k(r) - \bigcup_{j=1}^{m} \Delta OA'_j(r+1)$$
$$IA'_k(r+1) := IAO'_k(r+1) \cup IP_k$$

Two new sequences are introduced at each ECU to reduce network communication. Sequence $\{IAO'_k(r)\ | r = 0, 1, \cdots\}$ represents the available inputs at ECU k that are provided by the outputs of all tasks in the system. Sequence $\{\Delta OA'_k(r) | r = 1, 2, \cdots\}$ contains all the outputs that become unavailable at ECU k at step r. Since we have proved in Proposition 1 that sequence $\{OA_k(r)\}$ is monotonically nonincreasing and can show $OA'_k(r) = OA_k(r)$, no new outputs may become available at step r and set $\Delta OA'_k(r)$ indeed contains all the changed outputs. Only these changed ones need to be communicated over the network. The result is formalized in Proposition 2, whose proof relies on (6).

Proposition 2: For all $k \in \mathbf{m}$ and all possible integers r, $TA_k(r) = TA'_k(r)$, $OA_k(r) = OA'_k(r)$, and $IA_k(r) = IA'_k(r)$.

B. Considering CPU Limitation

Next we consider the CPU limitation in the configuration process. The major difference is that the active task set TA_k at ECU k is determined by (4) and TA_k must be schedulable.

Following the same procedure developed in Section III-A, we define the following sequences for all $k \in \mathbf{m}$,

$$TA_k(0) := T_k,\ OA_k(0) := O_k$$
$$IA_k(0) := [I_k \cap (\bigcup_{j=1}^{m} OA_j(0))] \cup IP_k$$

and for integer $r = 0, 1, \cdots$, $TA_k(r+1)$ is a schedulable subset of $\{\tau_{ik} | I_{ik} \subseteq IA_k(r)\}$,

$$OA_k(r+1) := \bigcup \{O_{ik} | \tau_{ik} \in TA_k(r+1)\}$$
$$IA_k(r+1) := [I_k \cap (\bigcup_{j=1}^{m} OA_j(r+1))] \cup IP_k$$

There exist two problems for this straightforward approach. First, the result of $TA_k(r+1)$ is not unique. Second, the more important problem is that the sequences $\{TA_k(r)\}$ may not be monotonic. During the configuration process, when an active task is deactivated because of insufficient input, more CPU time is available and then tasks previously disabled owing to heavy CPU workload may run. Sequences $\{TA_k(r)\}$, therefore, may not converge. This situation is illustrated by the example in Fig. 3.

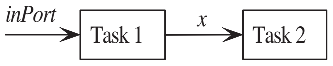

Fig. 3. Two Tasks at One ECU

The two tasks reside at a single ECU1 and Task2 has higher priority. Both tasks have 60% CPU utilization at ECU1. Task2 needs signal x to be active and x comes only from the output of low priority Task1. Evidently $TA_1(0) = \{Task1, Task2\}$, $OA_1(0) = \{x\}$, and $IA_1(0) = \{inPort, x\}$. Then either $TA_1(1) = \{Task1\}$ or $TA_1(1) = \{Task2\}$, because the two tasks cannot both properly run at ECU1. Since Task2 has higher priority, it seems more appropriate to set $TA_1(1) = \{Task2\}$. Correspondingly, $OA_1(1) = \emptyset$ and $IA_1(1) = \{inPort\}$. Now that x is not included in $IA_1(1)$, Task2 should be deactivated and Task1 can now become active, i.e., $TA_1(2) = \{Task1\}$, $OA_1(2) = \{x\}$, $IA_1(2) = \{inPort, x\}$. Notice that $IA_1(2) = IA_1(0)$. The same results for the sequences recur and this iterative process goes forever. Even if we chose $TA_1(1) = \{Task1\}$ at step 1, we shall still encounter this endless oscillation.

A simple but not necessarily optimal solution to the convergence problem is to always guarantee sequences $\{TA_k(r)\}$ ($k \in \mathbf{m}$) monotonic. Based upon this rule, the sequences $\{TA_k(r)\}$ should start from schedulable task sets $S_k \subseteq T_k$ rather than the full task sets T_k. Consequently, we can still use the sequences presented in Section III-A for system configuration under the CPU constraint, with the only modification of setting $TA_k(0) := S_k$. Since we have already shown that sequences $\{TA_k(r)\}$ are monotonically nonincreasing, the limits $TA_k(r^*)$ must be schedulable.

At ECU $k \in \mathbf{m}$ there are generally many candidates of the schedulable set S_k. If the ECU adopts the fixed priority scheduling strategy [16], [17], we choose S_k by preferring tasks with higher priorities. Therefore, we formalize the property of such an S_k.

We use priority numbers to symbolize the task priorities.

$$\mathbf{prio}_k : T_k \to \mathbb{Z}^+$$

To guarantee that every task has a unique priority, we suppose function $\mathbf{prio}_k$ to be injective. Here, a smaller priority number stands for a higher task priority, with 1 for the highest one. Let T_k be the set consisting of all tasks at ECU k and $S_k \subseteq T_k$ a task subset. We call S_k the *most important schedulable* task subset of T_k if S_k is schedulable at ECU k and for any task $t \in T_k - S_k$, the set

$$\{s \in S_k | \mathbf{prio}_k(s) \le \mathbf{prio}_k(t)\} \cup \{t\} \tag{7}$$

is not schedulable. According to the definition, even if we remove all tasks in S_k whose priorities are lower than task t, t is still not schedulable together with other higher priority tasks in S_k.

Given a task set T_k, the priority numbers of all tasks, and a criterion of deciding whether a task subset is schedulable, we propose an algorithm to compute the most important schedulable task subset S_k. Let the cardinality of T_k be n_k. Sort the tasks in T_k in the descending order on priorities, namely, $T_k = \{t_1, \cdots, t_{n_k}\}$, and

$$(\forall i, j \in \mathbf{n_k})\ i \le j \Rightarrow \mathbf{prio}_k(t_i) \le \mathbf{prio}_k(t_j)$$

Let $X_0 = \emptyset$. For all $r = 1, \cdots, n_k$,

$$X_r := \begin{cases} X_{r-1} \cup \{t_r\}, & \text{if schedulable} \\ X_{r-1}, & \text{otherwise} \end{cases} \tag{8}$$

Finally, $S_k := X_{n_k}$.

Proposition 3: Set S_k obtained above is the most important schedulable task subset of task set T_k.

Proof: Because $\emptyset$ is schedulable, (8) ensures that all X_r are schedulable. Therefore, $S_k = X_{n_k}$ must be schedulable. For further derivation, we also claim that the sequence $\{X_r\}$ is monotonic in the way that

$$(\forall r, r' \in \mathbf{n_k})\ r \le r' \Rightarrow X_r \subseteq X_{r'} \tag{9}$$

For any task $t_i \in \{t_1, \cdots, t_{n_k}\}$, we define

$$HP(t_i) := \{s \in S_k | \mathbf{prio}_k(s) \le \mathbf{prio}_k(t_i)\}$$

to represent the tasks in S_k whose priorities are higher than or equal to t_i. Owing to the priority assignments of the tasks, we can prove that

$$HP(t_i) = X_i,\ i = 1, \cdots, n_k \tag{10}$$

by mathematical induction. The details are omitted because of the space limit.

Finally, we prove that S_k is the most important schedulable subset by the definition. Let $t_j \in T_k - S_k$. Then

$$\{s \in S_k | \mathbf{prio}_k(s) \le \mathbf{prio}_k(t_j)\} \cup \{t_j\}$$
$$= HP(t_{j-1}) \cup \{t_j\} = X_{j-1} \cup \{t_j\}$$

If this set is schedulable, then $X_j = X_{j-1} \cup \{t_j\}$. Eq. (9) implies $t_j \in S_k$, which conflicts with the original assumption on t_j. The above set, therefore, cannot be schedulable. ■

IV. SIMULATION

We implement the proposed configuration approach in Matlab and have it verified via simulation. To realize the real-time scheduling strategies of ECUs and the network communication, we use the Matlab/Simulink-based simulator TrueTime [18].

We demonstrate the proposed configuration methods and the Matlab implementation through the simulations on the simple imaginary distributed system depicted in Fig. 4. The system consists of two ECUs connected by a Controller Area Network (CAN). The two big squares represent two ECUs, with the left one numbered 1 and the right one 2. The small rounded squares represent application tasks.

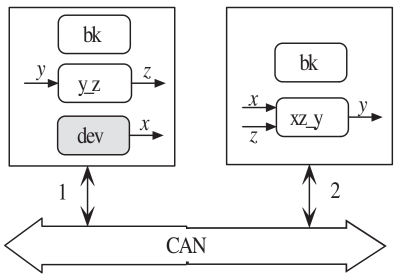

Fig. 4. A Simple Networked Embedded System

All tasks, except the highlighted task "dev", are permanently allocated to the target ECUs; thus the four tasks participate in the configuration process following the system startup, i.e., the beginning of the simulation. Task "dev" serves for a new device that might be attached to ECU1 in the future; hence it is normally absent in the system. When the device is connected to ECU1, it will trigger an external interrupt, which loads the application task associated to the device in the ECU and calls upon a system configuration using CAN communication.

All tasks in Fig. 4 are periodic. Task "bk" stands for a background task running at both ECUs. Task "y_z" reads input y and outputs z. Task "xz_y" has two inputs x, z and one output y. Task "dev" only outputs x. The three tasks form a simple control loop, with "y_z" acting as a plant,"xz_y" as a controller, and "dev" issuing reference input.

The parameters of the tasks are listed in Table I. In column 6, acronym "WET" means worst-case execution time. Without losing generality, the memory footprints of the tasks are assumed to be simple nominal numbers instead of exact byte sizes.

TABLE I
TASK PARAMETERS

Name	ECU	RAM	Prio	Perid	WET	In	Out
bk	1, 2	1	9	0.15	0.03	$\emptyset$	$\emptyset$
y_z	1	0.5	8	0.1	0.02	y	z
dev	1	1	7	0.1	0.03	$\emptyset$	x
xz_y	2	1	6	0.1	0.04	x, z	y

We assume that both ECUs have 3 slots of memory resources and adopt the fixed priority scheduling strategy. Based on the foregoing system setup, we first derive the expected simulation results using mathematics, and then examine if the prospect matches the TrueTime simulation.

The simulation starts without the new device; hence the full task sets at the two ECUs are $T_1 = \{bk,\ y\_z\}$ and $T_2 = \{bk,\ xz\_y\}$, respectively. The I/O ports of the two ECUs are both $\emptyset$.

When the configuration process begins, the ECUs first load the tasks into the ECU RAMs. According to the system setup, there are sufficient RAM to load all the tasks. The remaining memory at ECU 1 is 1.5 and that at ECU 2 is 1. The input and output name sets of the ECUs are $I_1 = \{y\}, O_1 = \{z\}$, and $I_2 = \{x, z\}, O_2 = \{y\}$.

The next step is to determine the active tasks according to the schedulability and data dependency. As the task priorities are assigned by the rate monotonic approach, we use the simple criterion that if overall CPU utilization U at an ECU meets the relation [16], [17] $U \leq n(2^{1/n} - 1)$, where n is the number of periodic tasks, then the task set is schedulable. When $n = 2$, the scheduling threshold is around 0.828. The utilizations for sets T_1 and T_2 are 0.4 and 0.6, respectively. Hence the two task sets are both schedulable, i.e., $S_1 = T_1$ and $S_2 = T_2$ in the terminology of Section III-B.

We derive the sequences formalized in Section III as follows.

$$TA_1(0) = \{bk, y\_z\}, OA_1(0) = \{z\} = IA_2(0)$$

$$TA_2(0) = \{bk, xz\_y\}, OA_2(0) = \{y\} = IA_1(0)$$

Since $IA_2(0)$ is smaller than the input set of task "xz_y", the task must be inactive and $TA_2(1) = \{bk\}$. Correspondingly output name y will not be available any longer, i.e., $OA_2(1) = \emptyset$ and $\Delta OA_2(1) = \{y\}$. Meanwhile, because $IA_1(0)$ is equivalent to the input set of task "y_z", $TA_1(1) = TA_1(0) = \{bk, y\_z\}$ and $\Delta OA_1(1) = \emptyset$.

$$IA_1(1) = IA_1(0) - \Delta OA_2(1) = \emptyset$$

$$IA_2(1) = IA_2(0) - \Delta OA_2(1) = \{z\}$$

On the next iteration, we further know $TA_1(2) = TA_2(2) = \{bk\}$, $\Delta OA_1(2) = \{z\}$, and all others are $\emptyset$. As a result, $TA_1(3) = TA_1(2)$ and $TA_2(3) = TA_2(2)$. The sequences converge at the second iteration. After the system configuration, only "bk" is active at every ECU. So the actual CPU utilizations at both ECUs are 0.2.

The configuration result is reasonable. Task "xz_y" requires input x, which can only be provided by the new device. Thus it cannot be active. Consequently, its output name y is absent in the system. Task "y_z" must be deactivated too.

The new device is attached to ECU1 in the simulation. The corresponding interrupt calls on a configuration process. In this process, owing to the new device, the full task set at ECU1 becomes $T_1 = \{bk,\ y\_z,\ dev\}$. The full task set at ECU2 is still the same $T_2 = \{bk,\ xz\_y\}$.

As stated before, the remaining memory at ECU1 is 1.5 and the memory footprint of "dev" is 1. So the new task can be loaded in ECU1. The loaded tasks at ECU1 is then the new set T_1. The remaining memory at ECU1 changes to 0.5. There is no memory change at ECU2. Because the new task "dev" provides output name x, the corresponding input and output become $I_1 = \{y\}, O_1 = \{x, z\}$, and $I_2 = \{x, z\}, O_2 = \{y\}$.

Since T_1 contains 3 tasks, the scheduling threshold for it is 0.78. The actual utilization for task set T_1 is 0.7. So T_1 is schedulable and we still have $S_1 = T_1$. The new configuration process to decide active tasks after introducing the new device is as follows.

$$TA_1(0) = \{bk, y\_z, dev\}, OA_1(0) = \{x, z\} = IA_2(0),$$

$$TA_2(0) = \{bk, xz\_y\}, OA_2(0) = \{y\} = IA_1(0)$$

Because $IA_1(0) = I_1$ and $IA_2(0) = I_2$, $TA_1(1) = TA_1(0)$ and $TA_2(1) = TA_2(0)$. The sequences converge at step 0. All tasks in Fig. 4 are active after the reconfiguration. The new CPU utilizations at the two ECUs are 0.7 and 0.6. If the device is detached later, a configuration process identical to the startup process will occur.

Finally we present the TrueTime simulation result. The plots in Fig. 5 show the execution schedules of all tasks at the two ECUs. The absolute values of the plots are meaningless. In one plot, *high* value means that the task is running, thus occupying the CPU in that time period, *low* value means that the task is idle, and *medium* value implies that the task is preempted by a higher priority task. The simulation time is 10s, and the new device is attached and detached at 3s and 7s, respectively. When the system starts up at 0s, the configuration process is delayed for 500ms to establish the network communication. Evidently the simulation agrees with the theoretical prediction.

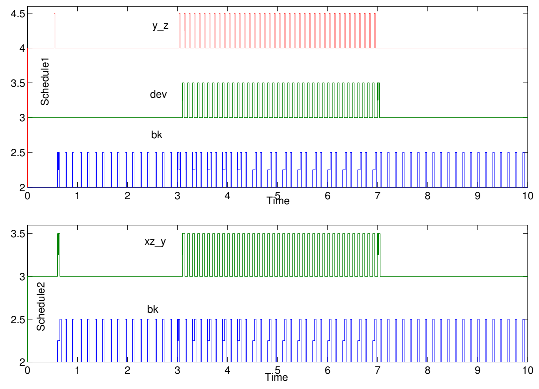

Fig. 5. Task Schedules of the Configuration Process

V. CONCLUSION AND FUTURE WORKS

To support the embedded reasoning and decision services that enable dynamically self-configuring automotive software systems, we formalize and propose a preliminary solution to the task self-configuration problem, which seeks to decide which tasks should be active on ECUs subject to resource constraints and task dependency. Our result has been verified by both mathematical proof and simulation.

We are planing to refine this study with the following approaches: (1) More accurate enforcement of real-time requirements. The schedulability criterion based on CPU utilization is very conservative. To be more realistic, we shall measure task latencies online and detect deadline overrun. (2) Task migration and load balancing [10]. These are unique features that DySCAS project investigates for automotive software systems. (3) Dynamic collaboration between this configuration decision and adaptive resource management service [7].

REFERENCES

[1] J.A. Cook, I.V. Kolmanovsky, D. McNamara, E.C. Nelson, and K. Venkatesh. Control, Computing and Communications: Technologies for the Twenty-First Century Model T, *Proc. IEEE*, 95(2):334-355, 2007.

[2] K. Grimm. "Software Technology in an Automotive Company - Major Challenges", *Proc. 25th Intl. Conf. Software Eng.*, Portland, OR, USA, pp.498-503, 2003.

[3] AUTOSAR - Automotive Open System Architecture, http://www.autosar.org, Aug. 2008.

[4] A. Sangiovanni-Vincentelli and M.D. Natale. Embedded System Design for Automotive Applications, *Computer*, 40(10):42-51, 2007.

[5] DySCAS - Dynamically Self-Configuring Automotive Systems, http://www.dyscas.org, Aug. 2008.

[6] R. Anthony, A. Rettberg, O. Redell, T. Quereshi, M. Törngren, C. Ekelin, and G. deBoer. "Dynamically Reconfigurable Automotive Control Systems", *Proc. Advanced Automotive Electronics*, Gaydon, UK, Jan. 31, 2007

[7] D.J. Chen, R. Anthony, M. Persson, D. Scholle, V. Friesen, G. deBoer, A.Rettberg, and C. Ekelin. "An Architectural Approach to Autonomics and Self-management of Automotive Embedded Electronic Systems", *Proc. 4th European Congress ERTS*, Toulouse, France, Jan. 29-Feb. 1, 2008.

[8] S.H. Bokhari. On the Mapping Problem, *IEEE Trans. Comput.*, C-30:207-214, 1981.

[9] M. Kafil and I. Ahmad. Optimal Task Assignment in Heterogeneous Distributed Computing Systems, *IEEE Concurr.*, 6(3):42-51, 1998.

[10] I. Jahnich, I. Podolski, and A. Rettberg. "Integrating Dynamic Load Balancing Strategies into the Car-Network", *Proc. IEEE Intl. Symp. Electronic Design, Test and Applications*, Hong Kong, China, Jan. 23-25, 2008.

[11] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell. "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB", *Proc. Middleware 2000 Conf.*, ACM/IFIP, New York, USA, Apr. 3-7, 2000.

[12] F. Kon, J.R. Marques, T. Yamane, R.H. Campbell, and M.D. Mickunas. Design, Implementation, and Performance of an Automatic Configuration Service for Distributed Component Systems, *Softw. Pract. Exper.*, 35(7):667-703, 2005.

[13] M. Roman and N. Islam. "Dynamically Programmable and Reconfigurable Middleware Services", *Middleware 2004, LNCS 3231*, H.-A. Jacobsen (Ed.), pp.372-396, 2004.

[14] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G.P. Picco, and S. Zachariadis. Reconfigurable Component-based Middleware for Networked Embedded Systems. *Int. J. Wireless Information Networks*, 14(2):149-162, Jun. 2007.

[15] J.S. Kinnebrew, A. Gupta, N. Shankaran, G. Biswas, and D.C. Schmidt. "A Decision-Theoretic Planner with Dynamic Component Reconfiguration for Distributed Real-Time Applications", *Proc. 8th Intl. Symp. Autonomous Decentralized Systems*, Sedona, AZ, USA, Mar. 2007.

[16] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd ed., Springer Science + Business Media, Inc., 2004.

[17] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *J. ACM*, 20(1): 46-61, 1973.

[18] M. Ohlin, D. Henriksson, and A. Cervin. *TrueTime 1.5 - Reference Manual*, Department of Automatic Control, Lund University, Sweden, http:// www.control.lth.se/truetime, Mar. 2008.