Self Configuration of Dependent Tasks for Dynamically Reconfigurable Automotive Embedded Systems

Lei Feng, DeJiu Chen, Martin Törgren

Abstract—The configurations of current automotive embedded systems are normally fixed during the development process 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 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 configuration items, assessing the system-wide impacts of changes, and making dynamic decisions about new configurations. Our solution is verified by both mathematical analysis and simulation.

I. INTRODUCTION

A modern automobile often contains dozens of electronic control units (ECUs) [1], [2], which communicate and collaborate through networks. These networked ECUs support various functions such as comfort control, cruise control, and anti-lock braking. Automotive manufactures currently configure the functions of the ECUs statically at design time. AUTOSAR (AUTomotive Open System ARchitecture) [3], for instance, is an emerging automotive standard to address the software integration challenge. It strongly promotes 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. Among its use cases, we would like to point out the most compelling two. (1) 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. (2) A vehicle should allow various functionalities to be migrated among different ECUs in case of hardware failure or low quality of service (QoS).

The main research objectives of the DySCAS project include a middleware architecture that supports context-awareness and dynamic configuration management using policies [6]–[8], and a set of algorithms that determine new system configurations in the events of, e.g., attaching new devices [6], load balancing [9], and improving QoS [7]. 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 [8] service of the DySCAS middleware.

The self-configuration problem is as follows. Given a set of ECUs connected by local area networks (LANs), the limited memory and CPU resources of the ECUs, groups of tasks with known memory footprints and timing specifications to run at the ECUs, and the dependency relationship among the tasks, find an algorithm to determine the tasks that can successfully perform under timing requirements at every ECU, and the communication paths among the ECUs. Here a task is a concurrent thread of an application program running on top of the OS and middleware at an ECU. We assume that the task-to-ECU allocation [10], [11] has been decided, and do not consider in this paper task migration or replication [9], [12]. The only action on the tasks is to switch them on or off. This problem is motivated by the dynamics of the automotive embedded systems. Events such as hardware failures, addition of new computational resources, and wrong assumptions on application load, may lead to completely new situations, providing a need for task self-configuration.

While many research projects and publications on dynamic and reconfigurable software emerged recently, including dynamicTAO [13], 2K [14], ExORB middleware [15], and RUNES middleware [16], [17], most of them propose new software architectures and focus on implementation techniques. To the best of our knowledge, there is no report on the algorithmic solution of the self-configuration problem formalized above.

Ravindran [12] presented a systematic approach to system specification, run-time monitoring and failure detection, diagnosis, and dynamic reconfiguration and resource reallocation. The approach is indeed a key initial reference for this work, but does not consider the influence of the causal dependencies among tasks on system reconfiguration. Roman and Islam [15] developed a middleware infrastructure that supports dynamically programmable and reconfigurable software systems. They specify the execution order of micro building blocks (MBBs), the counterparts of our tasks, as a directed graph, through the direct manipulation of which the dynamic reconfiguration is performed. Such an explicit dependency graph, however, may not be available in run-time, because DySCAS employs the publish/subscribe pattern [8].
to realize task modularity and communication transparency. The task dependency is implicit through the input and output signals of the tasks.

Section II presents the model of task execution process. Section III introduces a simple recursive algorithm to solve the task self-configuration problem, and proves its convergence and correctness through mathematics. Section IV delineates the network communication activity during the configuration process. The algorithm is further verified through a Matlab simulation in Section V. Finally Section VI draws the conclusion and outlines future works.

II. TASK EXECUTION MODEL

In a networked embedded system, application tasks communicate and interact via input and output signals. A signal is often implemented using a variable, i.e., a designated segment of memory that may be used as a scalar, a matrix, a queue, or a data structure provided by an operating system (e.g., a mailbox) for network communication. Thanks to communication transparency support, 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 \( k \in \mathbb{m} := \{1, \ldots, m\} \) 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 \( IP_k \) \( (k \in \mathbb{m}) \) be the name set of the signals associated with the input ports at ECU \( k \) and \( OP_k \) the set associated to the output ports.

The state machine in Fig. 1 shows the execution procedure of one task. To avoid clutter, we ignore the indexes of the events. 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 prevent the task from successful deployment.

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

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

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 (e.g., encoded as a policy [6]): 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 off.

At state “active”, the task has secured the necessary inputs and resources, and operates under the scheduling of a real time operating system (RTOS). Event deactivate switches off the task in the CPU but does not affect the memory. Event off removes the task from the ECU. While the RTOS may frequently put the task into a waiting queue for context switching, these switches are generated by the RTOS and the scheduling states (running, ready, waiting) can be seen as substates of the “active” state. In a normal system, all allocated tasks should be active. Hence we choose state “active” as the only marker state of the state machine, denoted by an output arrow in Fig. 1.

At ECU \( k \in \mathbb{m} \), 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

\[
OA_k := \bigcup \{O_{ik} | i \in \mathbb{n}_k, task_{ik} \in TA_k \} \tag{2}
\]

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

\[
IA_k := [I_k \cap \bigcup_{j=1}^{m} O_{Aj}] \cup IP_k \tag{3}
\]

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

\[
TA_k \subseteq \{task_{ik} | I_{ik} \subseteq IA_k \} \tag{4}
\]

and must be schedulable, namely all members in the set meet deadlines. The necessary condition to activate task\(_{ik}\) 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 set \( TA_k \) \( (k \in \mathbb{m}) \) to satisfy the three equations. The startup and reconfiguration processes of the middleware are essentially iterative procedures to find a solution of this problem. A trivial one is to set \( TA_k = \emptyset \) for all \( k \in \mathbb{m} \). It is, however, a tough job to find a nontrivial solution. For an arbitrary distributed system, the nontrivial solution may not exist or be unique. We shall study this dynamic process in the next section.
III. DYNAMIC CONFIGURATION PROCESS

When dynamic configuration is wanted, e.g., in the cases of system startup or new device attachment, the middleware instances 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. Without Hard Time Deadline

Starting from a simple question, we first suppose that the CPUs at all ECUs are powerful enough to execute all tasks in time or the tasks do not have hard real time requirements. With this assumption, (4) is simplified to

\[ T A_k := \{ \text{task} | i \in n_k, I_k \subseteq I A_k \} \quad (5) \]

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. Since we disregard the timing requirements of tasks, we just need to verify 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 \( k \in m \). 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 \( k \in m \) we have

\[
T A_k(0) := T_k \quad O A_k(0) := \bigcup \{ O_{i_k} | \text{task} \in T A_k(0) \} \\
I A_k(0) := I_k \cap \left( \bigcup_{j=1}^{m} A_{j}(0) \right) \cup I P_k
\]

Because \( T A_k(0) := T_k \), we have \( O A_k(0) = O_k \). Let integer \( r = 0, 1, \ldots \). From \( T A_k(r), O A_k(r), I A_k(r) \), we can derive

\[
T A_k(r+1) := \{ \text{task} | I_k \subseteq I A_k(r) \} \\
O A_k(r+1) := \bigcup \{ O_{i_k} | \text{task} \in T A_k(r+1) \} \\
I A_k(r+1) := I_k \cap \left( \bigcup_{j=1}^{m} A_{j}(r+1) \right) \cup I P_k
\]

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

**Proposition 1:** By ignoring the schedulability at all ECUs, sequences \( \{ T A_k(r) \} \) for all \( k \in m \) as defined above converge.

**Proof:** Using mathematical induction, we prove that for all \( k \in m \) the three sequences \( \{ T A_k(r) \} \), \( \{ O A_k(r) \} \), \( \{ I A_k(r) \} \) are all nonincreasing.

When \( r = 0 \), we immediately have \( T A_k(1) \subseteq T A_k(0) \) and \( O A_k(1) \subseteq O A_k(0) \), as \( T A_k(0) = T_k \) and \( O A_k(0) = O_k \). Consequently,

\[
\bigcup_{k=1}^{m} O A_k(1) \subseteq \bigcup_{k=1}^{m} O A_k(0)
\]

and \( I A_k(1) \subseteq I A_k(0) \).

Suppose that for any integer \( r \), it is true that

\[
T A_k(r+1) \subseteq T A_k(r) \\
O A_k(r+1) \subseteq O A_k(r) \\
I A_k(r+1) \subseteq I A_k(r)
\]

By the definition of the sequences and the induction assumption, we can show

\[
T A_k(r+2) = \{ \text{task} | I_k \subseteq I A_k(r+1) \} \\
\subseteq \{ \text{task} | I_k \subseteq I A_k(r) \} = T A_k(r+1) \\
O A_k(r+2) = \bigcup \{ O_{i_k} | \text{task} \in T A_k(r+2) \} \\
\subseteq \bigcup \{ O_{i_k} | \text{task} \in T A_k(r+1) \} = O A_k(r+1) \\
I A_k(r+2) = [I_k \cap \left( \bigcup_{j=1}^{m} A_{j}(r+1) \right) \cup I P_k] \\
\subseteq [I_k \cap \left( \bigcup_{j=1}^{m} A_{j}(r+1) \right) \cup I P_k] = I A_k(r+1)
\]

We now proclaim that sequences \( \{ T A_k(r) \} \), \( \{ O A_k(r) \} \), \( \{ I A_k(r) \} \) are all nonincreasing.

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

For concise representation of the limits of the sequences, we suppose that all the sequences converge at the index \( r^* \). Accordingly the limits of sequences \( \{ T A_k(r) \}, \{ O A_k(r) \}, \{ I A_k(r) \} \) are respectively \( T A_k(r^*), O A_k(r^*), I A_k(r^*) \).

This configuration process begins with the full task sets \( T_k \) and gradually reduces those whose inputs are not all available. 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., \( T A_1(0) = T A_2(0) = \emptyset \), then \( O A_1(0) = O A_2(0) = \emptyset \), \( I A_1(0) = \{ \text{inPort} \} \), and \( I A_2(0) = \emptyset \). Consequently, \( T A_1(1) = T A_2(1) = \emptyset \) and the two sequences both converge to \( \emptyset \). Evidently, this is not a correct answer to this system. The sequences proved in Proposition 1, on the other hand, can achieve the right answer.

![Fig. 2. Tasks Forming a Loop](image-url)
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, say $x$, we simply rename the two output names as, for example, $x_1, x_2$. Define the following sequences:

$$T_A_k(0) := T_k$$
$$OA_k(0) := \bigcup \{O_{ik} | task_{ik} \in T_A_k(0)\}$$
$$IAO_k(0) := I_k \cap (\bigcup_{j=1}^m OA_j(0))$$
$$IA_k(0) := IAO_k(0) \cup IP_k$$

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

$$T_A_k(r+1) := \{task_{ik} | I_k \subseteq IAO_k(r)\}$$
$$OA_k(r+1) := \bigcup \{O_{ik} | task_{ik} \in T_A_k(r+1)\}$$
$$\Delta OA_k(r+1) := OA_k(r) - OA_k(r+1)$$
$$IAO_k(r+1) := IAO_k(r) - \bigcup_{j=1}^m \Delta OA_j(r+1)$$
$$IA_k(r+1) := IAO_k(r+1) \cup IP_k$$

We introduce two new sequences 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 $k \in m$ and all possible integers $r$, $T_A_k(r) = T_A_k'(r)$, $OA_k(r) = OA_k'(r)$, and $IA_k(r) = IA_k'(r)$.

**B. With Hard Time Deadline**

Next we consider the timing constraint in the configuration process. The major difference is that the active task set $T_A_k$ at ECU $k$ is determined by (4) and $T_A k$ must be schedulable. Following the same procedure developed in Section III-A, we define the following sequences for all $k \in m$,

$$T_A k(0) := T_k$$
$$OA_k(0) := O_k$$
$$IA_k(0) := I_k \cap (\bigcup_{j=1}^m OA_j(0)) \cup IP_k$$

and for integer $r = 0, 1, \cdots$, $T_A k(r+1)$ is a schedulable subset of $\{task_{ik} | I_k \subseteq I_k(r)\}$,

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

There exist two problems for this straightforward approach. First, the result of $T_A k(r+1)$ is not unique. Second, the more important problem is that the sequences $\{T_A k(r)\}$ 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 $\{T_A k(r)\}$, therefore, may not converge. This situation is illustrated by the example in Fig. 3.

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. We derive the sequences for this example subject to hard deadlines. Evidently $T_A(0) = \{Task1, Task2\}$, $OA(0) = \{x\}$, and $IA(0) = \{inPort, x\}$. Then either $T_A(1) = \{Task1\}$ or $T_A(1) = \{Task2\}$, because the two tasks cannot both meet deadlines at ECU1. Since Task2 has higher priority, it seems more appropriate to set $T_A(1) = \{Task2\}$. Correspondingly, $OA(1) = \emptyset$ and $IA(1) = \{inPort\}$. Now that $x$ is not included in $IA(1)$, Task2 should be deactivated and Task1 can now become active, i.e., $T_A(2) = \{Task1\}$, $OA(2) = \{x\}$, $IA(2) = \{inPort, x\}$. Notice that $IA(2) = IA(0)$. The same results for the sequences recur and this iterative process goes forever. Even if we chose $T_A(1) = \{Task1\}$ at step 1, we shall still encounter this endless loop.

A simple solution to the convergence problem is to always guarantee sequences $\{T_A k(r)\}$ ($k \in m$) monotonic. Based upon this rule, the sequences $\{T_A k(r)\}$ 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 constraint of schedulability, with the only modification of setting $T_A(0) := S_k$. Since we have already shown that
sequences \( T_A(k) \) are monotonically nonincreasing, the limits \( T_A(k) \) must be schedulable.

At ECU \( k \in \mathcal{M} \) there are generally many candidates of the schedulable set \( S_k \). If the ECU adopts the fixed priority scheduling strategy, 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.

\[
prio_k : T_k \rightarrow \mathbb{Z}^+
\]

To guarantee that every task has a unique priority, we suppose function \( \text{prio}_k \) 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

\[
\{ s \in S_k | \text{prio}_k(s) \leq \text{prio}_k(t) \} \cup \{ t \}
\]

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

Given the 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,

\[
T_k = \{ t_1, \ldots, t_{n_k} \}
\]

and

\[
(\forall i, j \in n_k) i \leq j \Rightarrow \text{prio}_k(t_i) \leq \text{prio}_k(t_j)
\]

Let \( X_0 = \emptyset \). For all \( r = 1, \ldots, n_k \),

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

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

\[
(\forall r, r' \in n_k) r \leq r' \Rightarrow X_r \subseteq X_{r'}
\]

For any task \( t_i \in \{ t_1, \ldots, t_{n_k} \} \), we define

\[
HP(t_i) := \{ s \in S_k | \text{prio}_k(s) \leq \text{prio}_k(t_i) \}
\]

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 easily see

\[
HP(t_i) = S_k \cap \{ t_1, \ldots, t_i \}
\]

We shall further prove that

\[
HP(t_i) = X_i, \ i = 1, \ldots, n_k
\]

by mathematical induction.

\[
HP(t_1) = \{ t_1 \} \text{ if set } \{ t_1 \} \text{ is schedulable; otherwise } HP(t_1) = \emptyset
\]

This result matches the definition of \( X_1 \) in (8). If the identity holds for some \( i < n_k \), we prove that it must be true for \( i + 1 \) as follows.

\[
HP(t_{i+1}) = S_k \cap \{ t_1, \ldots, t_i, t_{i+1} \} = (S_k \cap \{ t_1, \ldots, t_i \}) \cup (S_k \cap \{ t_{i+1} \}) = HP(t_i) \cup (S_k \cap \{ t_{i+1} \}) = X_i \cup (S_k \cap \{ t_{i+1} \})
\]

If task set \( X_i \cup \{ t_{i+1} \} \) is schedulable, then \( X_{i+1} = X_i \cup \{ t_{i+1} \} \) by (8). Owing to (9), we know \( X_{i+1} \subseteq X_{n_k} = S_k \) and hence \( t_{i+1} \in S_k \). Consequently \( HP(t_{i+1}) = X_i \cup \{ t_{i+1} \} \).

On the other hand, if the task set \( X_i \cup \{ t_{i+1} \} \) is not schedulable, then \( X_{i+1} = X_i \) and then \( t_{i+1} \notin S_k \). Again we have \( HP(t_{i+1}) = X_i = X_{i+1} \). We finish the proof of (10).

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

\[
\{ s \in S_k | \text{prio}_k(s) \leq \text{prio}_k(t_j) \} \cup \{ t_j \} = HP(t_{j-1}) \cup \{ t_j \} = X_{j-1} \cup \{ t_j \}
\]

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

**IV. COMMUNICATION BETWEEN ECUS**

This section describes the communication activity for the middleware instance at ECU \( k \in \mathcal{M} \). At the beginning of the configuration process, the middleware broadcasts the output list \( OA_k(0) \) and collects the same messages from other ECUs in the network. When the middleware receives a message about \( OA_k(0) \) (\( k' \in \mathcal{M} \) and \( k' \neq k \)), it updates the set of available inputs \( IAO_k(0) \). After a period long enough to receive the broadcast messages from all ECUs, the configuration process starts the converging process and generates the remaining elements of the sequences.

On the basis of \( IAO_k(r) \ (r = 0, 1, \ldots) \), the middleware determines a new set of active tasks \( T_A(k+1) \), computes a new available output list \( OA_k(r+1) \), and broadcasts the outputs that should be removed from the available output list, i.e., those in \( \Delta OA_k(r+1) = OA_k(r) - OA_k(r+1) \). It also listens to the network communication. As soon as receiving a message from other ECUs about the removed outputs in \( \Delta OA_k(r+1) \), it calculates another new \( IAO_k(r+1) \) and repeats the converging process until the sequences reach the limits or the configuration time hits a predetermined deadline.

During the configuration process, the middleware establishes the communication paths among the ECUs. If an
output name of ECU \( k \) is used by others, the middleware manages a list of all these ECUs for the output name. On the other hand, if the value of an input name comes from another ECU, the middleware also records its source ECU.

We represent the communication flow among the system ECUs with a digraph \((N, V, A)\), where

- \( N \) represents the set of ECUs of the network. Recall that there are \( m \) ECUs in the network. Hence the cardinality of \( N \) is \( m \), i.e., \( N := \{ n_i \mid i \in m \} \).
- \( V \) is the set of all available names in the system, namely, \( V := \bigcup_{k=1}^{m} OA_k(r^*). \) Elements in \( V \) label the arrows of the digraph.
- \( A \subseteq N \times V \times N \) is the set of labeled arrows \((n_i, x, n_j)\), where \( n_i \neq n_j \) and \( x \in OA_i(r^*) \cap IA_j(r^*) \). The intuition is that ECU \( n_i \) provides the value of \( x \) to ECU \( n_j \).

The complete model of the digraph, however, does not exist in the system. Its information is distributed on the ECUs, which store adjacent arrows only.

V. 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 ECU 1 and the right one ECU 2. The small rounded squares represent application tasks.

![Fig. 4. A Simple Networked Embedded System](image)

All tasks, except the highlighted task “devTsk”, 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 “devTsk” 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. In the simulation, the device attachment is fulfilled by a square wave signal connected to an external trigger of the TrueTime kernel.

All tasks in Fig. 4 are periodic. Task “backTsk” stands for a background task running at both ECUs. Task “y\_Tsk” reads input \( y \) and outputs \( z \). Task “xz\_yTsk” has two inputs \( x, z \) and one output \( y \). Task “devTsk” only outputs \( x \). The three tasks form a simple control loop, with “y\_Tsk” playing as a plant,”xz\_yTsk” as a controller, and “devTsk” 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>
<thead>
<tr>
<th>Name</th>
<th>ECU</th>
<th>RAM</th>
<th>Prio</th>
<th>Period</th>
<th>WET</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>backTsk</td>
<td>1, 2</td>
<td>1</td>
<td>9</td>
<td>0.15</td>
<td>0.03</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>y_Tsk</td>
<td>1</td>
<td>0.5</td>
<td>8</td>
<td>0.1</td>
<td>0.02</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>devTsk</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>0.1</td>
<td>0.03</td>
<td>( \emptyset )</td>
<td>x</td>
</tr>
<tr>
<td>xz_yTsk</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>0.1</td>
<td>0.04</td>
<td>x, z</td>
<td>y</td>
</tr>
</tbody>
</table>

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 \( T_1 = \{ \text{backTsk}, \ y\_Tsk \} \) and \( T_2 = \{ \text{backTsk}, \ xz\_yTsk \} \), respectively. The I/O ports of the two ECUs are both \( \emptyset \).

After a startup waiting period, the configuration process takes place. 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

\[
I_1 = \{ y \}, O_1 = \{ z \}, \quad \text{and} \quad I_2 = \{ x, z \}, O_2 = \{ y \}
\]

The next, and the most important step, is to determine the active tasks according to the schedulability and data dependency. As the ECUs both apply fixed priority strategy, we use the simple criterion that if overall CPU utilization \( U \) at an ECU meets the relation [19], [20]

\[
U \leq n(2^{1/n} - 1)
\]

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

\[
U_1 = 0.03/0.15 + 0.02/0.1 = 0.4 < 0.828
\]

\[
U_2 = 0.03/0.15 + 0.04/0.1 = 0.6 < 0.828
\]
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.

$$T_A(0) = \{ \text{backTsk}, y_{zTsk} \}$$
$$T_A(1) = T_A(0) = \{ \text{backTsk}, y_{zTsk} \}$$
$$O_A(0) = \{ z \}$$
$$O_A(1) = O_A(0)$$
$$IA(0) = I_1 \cap (O_A(0) \cup O_A(0)) = \{ y \}$$
$$IA(1) = \emptyset$$
$$\Delta A(0) = \emptyset$$
$$\Delta A(1) = \emptyset$$

Since $IA(2)$ is smaller than the input set of task “xzTsk”, the task must be inactive and

$$T_A(2) = \{ \text{backTsk} \}$$

Correspondingly output name $y$ will not be available any longer, i.e., $O_A(1) = \emptyset$ and $\Delta A(2) = \{ y \}$. Meanwhile, because $IA(1)$ is equivalent to the input set of task “y_{zTsk}”,

$$T_A(1) = T_A(0)$$
$$O_A(1) = O_A(0)$$
$$IA(1) = IA(0) - \Delta A(2) = \emptyset$$
$$IA(2) = IA(0) - \Delta A(0) = \{ z \}$$

On the next iteration, we further know

$$T_A(2) = \{ \text{backTsk} \}$$
$$O_A(2) = \emptyset$$
$$\Delta A(1) = \emptyset$$
$$\Delta A(2) = \{ z \}$$

As a result, $T_A(3) = T_A(2)$ and $T_A(3) = T_A(2)$. The sequences converge at the second iteration. After the system configuration, there is only one active task “backTsk” at every ECU. So the actual CPU utilizations at ECUs 1 and 2 are both 0.2.

The configuration result is reasonable. Task “xz_{zTsk}” 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_{zTsk}” must be deactivated too.

The new device is attached to ECU 1 in the simulation. The corresponding interrupt calls on a configuration process. In this process, owing to the new device, the full task set at ECU 1 becomes $T_1 = \{ \text{backTsk}, y_{zTsk}, devTsk \}$. The full task set at ECU 2 is still the same $T_2 = \{ \text{backTsk}, x_{zTsk} \}$.

As stated before, the remaining memory at ECU 1 is 1.5 and the memory footprint of “devTsk” is 1. So the new task can be loaded in ECU 1. The loaded tasks at ECU 1 is then the new set $T_1$. The remaining memory at ECU 1 changes to 0.5. There is no memory change at ECU 2. Because the new task “devTsk” provides output name $x$, the corresponding input and output become

$$I_1 = \{ y \}, O_1 = \{ x, z \}$$
$$I_2 = \{ x, z \}, O_2 = \{ y \}$$

Since $T_1$ contains 3 tasks, the scheduling threshold for it is 0.78. The actual utilization for task set $T_1$ is

$$U_1 = 0.03/0.15 + 0.02/0.1 + 0.03/0.1 = 0.7 < 0.78$$

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.

$$T_A(0) = \{ \text{backTsk}, y_{zTsk}, devTsk \}$$
$$T_A(1) = T_A(0) = \{ \text{backTsk}, y_{zTsk}, devTsk \}$$
$$O_A(1) = \{ x, z \}$$
$$O_A(2) = \{ y \}$$
$$IA(0) = \{ y \}$$
$$IA(2) = \{ x, z \}$$

Because $IA(1) = I_1$ and $IA(2) = I_2$, $T_A(1) = T_A(2)$ and $T_A(1) = T_A(2)$. The sequences converge at step 0. All tasks in Fig. 4 are active after the middleware configuration. Consequently, the task numbers at ECUs 1 and 2 are 3 and 2, respectively; 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.

VI. 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 planning 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. Moreover, for different application types and timing constraints, we shall investigate different scheduling strategies. (2) Task replication and migration. These are unique features that DySCAS project investigates for automotive software systems. (3) Other types of constraints and task dependency, such as reliability and safety constraints.
Fig. 5. Task Schedules of the Configuration Process

REFERENCES


