Towards Runtime Adaptation in Real-time, Networked Embedded Systems

Christian Prehofer, Marc Zeller
Fraunhofer Institute for Communication Systems ESK
Munich, Germany
{christian.prehofer, marc.zeller}@esk.fraunhofer.de

Abstract—In this work, we consider reliable runtime adaptation in networked, embedded systems with tight real-time constraints by adapting the placement of software components on a multitude of hardware components. We show the need for a hierarchical transaction concept in this context. In particular, we consider multiple adaptation steps under hard system constraints and also introduce a model with undesired configurations, which cannot be maintained for an extended time period. Furthermore, we discuss implementation issues for such an adaptation process, including the actual task migration implementation, for real-time, embedded systems.

I. INTRODUCTION

In many industrial application domains (e.g., railway, automotive, avionic, etc.), networked embedded systems realize safety-relevant applications which have high demands on dependability. There has been considerable work on self-adaptive systems, which can modify their software configuration at runtime [1], [2]. However, applying these techniques to networked, embedded systems poses several new problems due to the limitations and reliability requirements of embedded systems [3]. In particular, we focus on networked embedded systems (like automobiles), where the main system constraints are limited memory resources, task schedulability, timing dependencies between software components, heterogeneous hardware platforms, and different sub-networks connected by a gateway.

The problem is that the adaptation at runtime as such is also subject to the above constraints. This means that the timely operation of the system must be ensured during the adaptation. Hence, only the slack times between the tasks, which we assume to be periodic, can be used for this purpose. Earlier work has focused on modeling and solving above constraints efficiently [4], as well as formalizing the constraints for the actual adaptation process finding such adaptation steps efficiently [5]. As an application scenario, we consider an automotive embedded system consisting of multiple Electronic Control Units (ECUs). These are connected locally by different bus systems and then system-wide via gateways.

We consider the problem of adaptations in this application domain, consisting of several software migrations, under the mentioned system constraints. First, we consider the above timing constraints which have to be ensured. In addition, we have secondary criteria like energy consumption which are less time critical. For instance, a component may be migrated to an ECU which then consumes an amount of energy which is not sustainable for more than two seconds. Similarly, there may be reliability constraints which allow non-critical functions to be non-operational for very short time for an adaptation, but not for a longer period. Hence, we have system states which are not desirable or not sustainable. We call these undesired configurations. Such states can be used as intermediary configurations in a complex adaptation, but must be changed as soon as possible. This is in contrast to non-valid configurations, which violate hard system constraints and must be avoided at any time.

Here, we focus on the adaptation process and transaction concepts to address the efficiency and the urgency of the adaptations. The efficiency means to speed up the adaptation process, e.g., by parallelization and by merging steps. Urgency means that adaptation may reach intermediary configurations which are undesired, and must be adapted quickly. Such configurations are possible for a short period of time, but not sustainable.

Such adaptation problems with timing constraints have been noted in several other works, e.g., [6], [7], without modeling of the above system constraints. It also has been observed in [7] that in general transaction concepts are needed, if intermediate configurations are not viable. We will detail and extend this by showing the need for hierarchical transactions.

In this paper, we present a hierarchical transaction concept for adaptations depending on the above constraints and undesirable configurations. Secondly, we discuss implementation issues for such adaptations for real-time, embedded systems.

II. RELIABLE RUNTIME ADAPTATION IN EMBEDDED NETWORKS

We characterize networked embedded systems by a set of inputs \( I \) (sensors), a set of features \( F = \{ f_1, \ldots, f_k \} \) and a set of outputs \( O \) (actuators). The set of features is implemented by a set of software components \( SW = \{ s_1, \ldots, s_n \} \) [3]. Each feature \( f_i \) is implemented by a number of software components \( SW_{f_i} \) with \( SW_{f_i} \subseteq SW \). We assume that the software components are executed as periodic tasks on the hardware platforms and that their Worst Case Execution Times (WCETs), as well as memory and network consumption are known. The software components are executed on a set of hardware platforms \( P = \{ p_1, \ldots, p_m \} \). The system configuration \( c \) describes allocation of the respective software components to the available hardware platforms at time \( t \).
To guarantee correct system behavior, the design has to provide assertions about the system’s runtime behavior. For adaptive systems, the system must be able to check if system configurations are valid at runtime. Especially in distributed systems the adaptation and satisfaction of the requirements is a resource extensive task.

For such networked embedded systems there are several limitations and reliability requirements [3]. In particular, we focus on automotive embedded systems, where the typical system constraints are

1) Heterogeneous hardware platforms: Thus, a software component can only be executed on a subset of all available platforms (assignment constraints).
2) Restricted memory resources: For each hardware platform, the total resource usage by the assigned components must be checked.
3) Restricted network resources: For each network bus, the total resource usage by the assigned components must be checked.
4) Timing requirements can be represented by so-called end-to-end timing chains (TCs) of the function networks and ensure end-to-end timing dependencies of the system features.
5) Real-time task scheduling is needed to ensure the timing constraints. In our model, we consider the scheduling of real-time tasks with dynamic priorities.
6) Restricted gateway throughput, i.e. the network bandwidth of the network gateway has to be checked.

The allocation of the software components to the available hardware platforms in a system configuration c is called valid at time t, if all these system constraints are satisfied, as defined in full detail in [4]. By this formalism, it is possible to describe self-adaptation in networked embedded systems (in the automotive domain) in form of linear equations. It has been shown in [4] that these linear constraints can be solved efficiently, even though the number of equations grows enormously. For instance, with 2000 software components on 100 ECUs, this results in about 4.5 million variables and 170,000 equations.

The adaptation process A changes the configuration c of the networked embedded system: c \xrightarrow{A} c_{new}. Thereby, the adaptation process A can be decomposed into a sequence of atomic steps mig1,...,mig_i with 1 \leq i \leq n. Each of these steps mig_i \in A represents the migration of one specific software component s_mig_i \in SW from its current hardware platform p_{cur}, where the component is currently executed, to another hardware platform p_{new}. Hence, the adaptation process of a networked embedded system is defined as follows:

\[
\begin{align*}
    c & \xrightarrow{mig_1} c^1 \\
    c^1 & \xrightarrow{mig_2} c^2 \\
    \vdots & \\
    c^{i-1} & \xrightarrow{mig_{j}} c^i \\
    \end{align*}
\]

The configurations c^1, \ldots, c^{i-1}, which are reached after the successful execution of a migration mig_1, \ldots, mig_{i-1}, describe the allocation of the software components during the adaptation process. All these intermediate configurations must fulfill the same predefined requirements as the "normal" system configurations. We assume here that the goal of the adaptation is to reach some final configuration c_{new}. Such a sequence of migrations has to be performed as a transaction, which means that all steps are performed successfully or, in case of a failure, there is a rollback to the old configuration (cf. [7]).

While it is generally important to perform adaptations efficiently, our context poses strict limitations for the adaptation process. As our tasks are critical and are executed periodically, the adaptation has to take place within the inactive time slot of a component. As these time slots may be quite tight, it may only be possible to transfer a small number of components from one hardware platform to another one.

To be able to execute a certain migration mig_j \in A, it must be possible to carry out the migration of the software component while the whole system remains schedulable and the overall system behavior is not influenced (see [5]). Therefore, the following constraint must be satisfied in order to guarantee that the adaptation process does not affect the correct behavior of other features within the networked embedded system:

\[
    WCMT_{s_mig_j} < T_{s_mig_j} - WCET_{(s_mig_j,p_{cur})} \quad (1)
\]

where WCMT_{s_mig_j} is the Worst Case Migration Time and T_{s_mig_j} is time between two executions of a software component, which is typically the period of the software component s_mig_j \in SW. WCET_{(s_mig_j,p_{cur})} represents the WCET of s_mig_j active on the hardware platform p_{cur}.

In addition, we define undesired configurations as a certain subset of valid configurations, which are undesired. There are several reasons why some configuration may be undesirable. For instance, inefficient resource utilization, physical reliability considerations (e.g. accidents) or energy efficiency. As we aim for an extensible and open system, we keep the set of undesirable configurations abstract. We assume however that checking membership of this set is efficiently computable. A sample criteria for such undesirable configurations is for example the inefficient resource utilization on a certain hardware platform.

For instance, if a hardware platform is little used, it is better to turn it off and move the components to some other hardware platforms. This means a certain minimal utilization of a hardware platform is desirable. Similarly, we may have specific placement constraints for some components as some hardware platforms may have higher reliability than others. Such constraints describe undesirable system configurations which are acceptable, but not desired. We view these undesired configurations as temporary, intermediate configurations.

### III. Hierarchical Transactions for Runtime Adaptation

In this section, we discuss the problem of finding suitable and efficient transactions for the adaptation process. Each adaptation A which is executed during runtime to adapt the networked, embedded system can be decomposed into atomic steps A = \{mig_1, \ldots, mig_i\} 1 \leq i \leq n. Such a sequence of
migration has to be performed as a transaction, which means all steps are performed successfully or, in case of a failure, there is a rollback to the old configuration.

In the simplest case, all intermediate configurations of the adaptation are valid and all migration steps follow the transition constraints introduced in Section II. In this case, we consider the sequence as a single global transaction.

\[ c_{\text{new}} \leftarrow c_{i-1} \leftarrow c_{i-2} \leftarrow \cdots \leftarrow c_2 \leftarrow c_1 \rightarrow c \]

Thereby, \( i - 1 \) intermediate configurations are reached during the adaptation process. All these intermediate configurations \( c_1, \ldots, c_{i-1} \) fulfill the predefined system constraints.

In general, we will however need a hierarchical transaction concept. Hierarchical means that, in addition to the above global transaction, we form local transactions. Similar concepts have been used for data base transactions, called nested transactions [8] or multilevel transactions [9], focusing more on efficient and reliable execution in a distributed database context. Here, we use local transactions for a different reason. Local transactions express a higher sense of urgency in case of a failure and must be rolled-back instantly.

A local transaction can be formalized as the subsequence \( c^1 \) to \( c^3 \) in the sequence below, where \( c^2 \) is an undesired configuration.

\[ c_{\text{new}} \leftarrow c_{i-1} \leftarrow c_{i-2} \leftarrow \cdots \leftarrow c_2 \leftarrow c_1 \rightarrow c \]

In summary, the execution of a sequence of migrations \( A \) consists of a global transaction and possibly several local transition transactions for subsequences. In case a local transaction fails, it can be locally repaired to fall-back to a valid configuration. Then, it can be re-tried or the global transaction has to fail as well.

We use such local transactions for several reasons: First, we may have non-valid intermediate configurations. In this case, we have to find a valid transition sequence which reaches a valid, workable configuration. Secondly, an adaptation process can be performed faster if several steps are merged into a valid, local transition sequence, whenever possible. These subsequences are executed in a local transaction which means that the overall transaction has fewer steps and can often be executed faster, as it uses fewer slack times.

If the system must be adapted newly while the global transaction is still in progress, the global transaction may be aborted, but the local transaction has to be completed to reach a valid configuration. Then a new adaptation process can be started.

IV. IMPLEMENTATION ISSUES AND DISCUSSION

In the following, we discuss several issues and options for actually implementing the presented adaptation concept.

The first issue is about implementing the actual migration of a software component. There are generally two options. The first one is moving only the required data and system states to a different ECU. The second option is to transfer the program code and the current execution state as well.

For performing full task migration in real-time systems the Total Copy [10] mechanisms can be used. It freezes the execution of the task that is going to be migrated on the source node, transferring its entire data (context and/or binary code) and resuming its execution on the destination node. However, this technique leads to a so-called freeze time or blackout time - a time interval during which a task cannot be executed due to the ongoing migration. Since we use the slack of the task to perform the adaptation, the migration is performed while the task to be migrated is not in operation. Thus, the system is still in operation and the predefined system behavior is not influenced by the adaptation process. This strategy is restricted to software components which are periodically executed and the adaptation process does not last longer than their distinct period.

A disadvantage of the full task migration is that the full program code and dynamic data has to be moved, which can be considerable. In the case of simple, periodic tasks, we often have the case that the task is always started from a well-defined position. In this case, we only have to transfer the needed data, i.e. we do not have to transfer the program counters, registers or data on the heap/stack area. Hence, the full task migration may not be needed.

For the first option of simple state transfer, the program code of the software has to be made available at the target ECU. For this, there can be several options. The simplest one is to provision the program code for the task in advance, which however requires local storage. The alternative is to use one or more servers to provide the needed program code. In this case, we have to consider the network load when transferring the data. It may turn out that both options have to be used in specific cases. For small ECUs with slow or fully reserved networks, full task migration may not be realistic. On the other hand, in segments where faster networks are available, loading the software from a server is preferable.

An important enabler for task migration in networked embedded systems is virtualization. As such embedded systems often have heterogeneous hardware and software platforms, migration is limited to compatibility of these platforms. To address this problem, virtualization can provide virtual computation environments for adding new tasks on an ECU. While this is promising, virtualization makes it more difficult to calculate timing. Also, start-up times for virtual environments and emulations have to be considered. An overview of the different virtualization options is presented in [11], including both virtual machines like Java and hypervisors as in operating system virtualization.

Another implementation issue is the control and synchronization of the adaptation process. In general, such control can be realized in a central or a distributed manner. Note that the control is a very sensitive challenge here. In case the control itself has failures, e.g. lost data packets, inconsistent system states may be reached. This leads to new challenges on reliable
and distributed control. A possible solution is to use reliable, distributed transaction mechanisms [12], as discussed in [13]. Compared to typical data-base transaction we have real-time constraints, but on the other hand we may not have a large number of simultaneous transactions.

Moreover, failure handling and rollback issues must be considered when implementing the hierarchical transaction concept for runtime adaptation in real-time systems. The problem is that failure handling is very time critical. For instance, in case a task migration is unsuccessful (e.g. because software is not delivered from a server to the target ECU), the old instance has to be used. For a full task chain, all tasks have to be informed in a timely manner about an unsuccessful transaction. However, this requires control and coordination overhead. A networked embedded system must be able to perform a rollback during the adaption process quickly in order to avoid new failures resulting from the unsuccessful transaction.

V. CONCLUSION AND FUTURE WORK

In this work, we have presented a transaction concept for reliable runtime adaptation in networked, embedded systems with tight real-time constraints. Building on earlier work on system and transition constraints, we have shown the need for a hierarchical transaction concept. In particular, we have considered multiple adaption steps under hard system constraints and also assuming a model with undesired states, which cannot be maintained for an extended time period. Our motivation here was that in a real system many other constraints exist - such as energy efficiency - which require additional constraints. In other words, formal systems for adaptation must be open and expendable also for other constraints. Secondly, we have discussed implementation issues for such an adaptation process, including the actual task migration and virtualization to solve the problem of heterogeneous hardware and software platforms.

REFERENCES