Findings from introducing state-of-the-art real-time techniques in vehicle industry

Christer Norström, Mikael Gustafsson*, Kristian Sandström, Jukka Mäki-Turja and Nils-Erik Bånkestad**
Mälardalen Real-Time Research Centre, Department of Computer Engineering
Mälardalen University, Västerås, Sweden
*TietoEnator ArosTech AB, Västerås, Sweden
** Volvo Construction Equipment Components AB, Eskilstuna, Sweden
cen@mdh.se

Abstract

The use of state-of-the-art real-time techniques in industry is still rare. The reason for this is three-folded: (1) the lack of commercially available tools, (2) the lack of methodologies that considers real-time throughout the complete development process, and (3) the lack of competence in real-time theory among industrial practitioners.

In this paper we present a case study of introducing state-of-the-art real-time techniques in industry. The case study was done as a collaboration between Mälardalen University and the industrial partners Volvo Construction Equipment AB (VCE) and TietoEnator ArosTech. VCE develops computer control systems for construction equipment vehicles, such as wheel loaders, graders, and articulated haulers. TietoEnator ArosTech is a firm of consultants with expertise competence in the area of embedded real-time systems.

We will present both the used methodology and the findings from introducing this methodology in an industrial project. The methodology emphasis is on introducing timing requirements early in the design of a system and it relies on the use of a well defined design language. We will present our findings categorized into methodological aspects, technology transfer, and technical aspects. The main result reported can be summarized as "*people, not paper, transfer technology*".

1 Introduction

Development of complex embedded systems is a growing area, i.e., we see more and more applications that are dependent on the use of embedded computers. Examples include highly complex systems, such as medical control equipment, mobile phones, and vehicle control systems.

Most of the embedded systems can also be characterised as real-time systems, which means that their correct function is dependent on both correct functional results and that the results are produced at the correct time.

The increased complexity of these systems leads to increasing demands on issues such as requirements engineering, high level design, early error detection, productivity, integration, verification, and maintenance. This calls for methods and models that enable a controlled and structured way of working during the complete life cycle of the system [Kal88].

There exist many design methods for real-time systems like, UML-RT and HRT-HOOD. However, these methods often concentrate on the logical and structural decomposition rather than focusing on the temporal behaviour. The temporal behaviour is often added on top. This is not so strange since these methods are based on general software development methods that are not focusing on embedded real-time systems. Furthermore, these methods have no, or limited, support for high level timing analysis and do not provide support for automatic mapping from the design to a resource structure. This often leads to a semantic gap between the design and the implementation, that is, the code and design description may not describe the same version of the system. Thus, classic problems during integration may occur, such as erroneous synchronisation and communication interfaces and that the system is hard to maintain.

Therefore, we have developed a model and method focused on the real-time properties of a system. The key property of the model and method is specification of a high level design that includes the specification of temporal constraints, communication and synchronisation. Furthermore, the model and method supports formal verification of these properties, early system integration, and efficient testing.

The aim of this paper is to briefly present this model and method, as well as our findings from introducing and using them in an industrial project. This project was performed as a cooperation between Mälardalen University, Volvo Construction Equipment AB (VCE) and TietoEnator ArosTech.

VCE has had onboard electronics since 1981 for specific functionality. Currently more and more functionality is provided by the computer control system. This has led to an increased number of people involved in the development of each product, and thus the need for better development methods and tools.

This was the motivation for the university to participate in the development of a new computer control system for the next generation wheel loaders. Since a complete new architecture was to be developed we were given the opportunity to introduce new technologies and methods.

Many functions are similar in different vehicles and therefore it would be a desired property to be able to reuse existing solutions. This was the starting point for defining a new architecture that could be used for all types of future construction equipment. Hence, the result of this project will act as a basis for extracting a product line architecture [Bos00]. However, the latter step is outside the scope of this paper.

Thus, this paper is focused on presenting our findings from introducing state of the art real-time technology in an industrial project. The validity of these findings is based on a single, but extensive, case study of one industrial project. Some of the findings are strengthened by similar results in other industrial projects that also have utilized state of the art real-time technology [Cas98, Mel98].

The outline of the paper is as follows: Section 2 presents briefly the characterization of the application. The design language used is described in Section 3. Section 4 presents briefly the tool that maps the design to a resource structure. Thereafter, in Section 5, the development methodology is presented. In Section 6 we present our findings categorized into findings related to methodological aspects, technology transfer, and technical aspects. Finally, in Section 7 some conclusions are given.

2 Application characteristics

The application is a vehicle control system with high demands on safety, reliability, and timeliness. The hardware in the system consists of two nodes that are connected via redundant buses. The application contains tasks, running at different period times, which collaborate to perform certain control functions. The system contains about 80 tasks with well-defined functionality. Each node is very I/O intensive. The complete system has about 150 I/O channels connected to it.

The execution times of the tasks in the application range from about 10 μs to 1 millisecond. The application is, due to the construction of the hardware, interrupt intensive. Since this application has many interrupts, the effect of these interrupts can not be neglected when scheduling the application tasks.

The worst case utilization of the processors for the critical part is around 80%, divided into 35% for interrupts and 45% for application tasks. The spare capacity left is used by soft real-time tasks. At run-time, the spare capacity will be more than the remaining 20% if the load is less than the worst case.

The reason for the extensive use of interrupts is mainly due to the hardware design. The hardware could not be modified since it was already designed and certified when the software development started.

3 Design language

The design language should be simple with a few, but powerful, constructs with clearly defined syntax and semantics. The reason for this is twofold: 1) parts of the implementation can be automatically generated by tools and, 2) the traceability from specification to implementation is improved since it is easier to overcome the semantic gap between design specification and implementation. The design is tightly coupled to the implementation; it is easier to fix a bug by correcting the design than to just make a modification in the code, when tools generate parts of the implementation directly from the design specification.

Another important principle is the separation of concerns. A specific example in the language is to separate communication and synchronization constructs from the C-code. This gives advantages in verifying the temporal behavior of the system (analyzing or estimating the execution time of the code is easier since it is independent from other components of the system). The integration phase also becomes easier when the interaction and synchronization is specified and analyzed early in the design.

The most important contribution, however, is that temporal constraints are defined early in the development process, which enables an early temporal verification.

The key elements of the language in increasing order of granularity are:

- Application – defines the top level of a complete software system.
- Modes and mode transitions – defines a high level state machine.
- Transactions – describes the functionality in a mode.
- Interaction graphs – describes the interactions between tasks that make up a transaction.
- Tasks – the computational elements of the design language.

3.1 Application model

A classical way of attacking problems is by ”divide and conquer”, i.e., by decomposing the problem into more manageable sub-problems. This is done here by

hierarchical decomposition, where an application is broken down into modes. A mode is an operational state of the application. Different modes contain different functionality. Each mode should only include the functionality that is needed for the desired behavior. A picture of this hierarchy is shown in Figure 1.

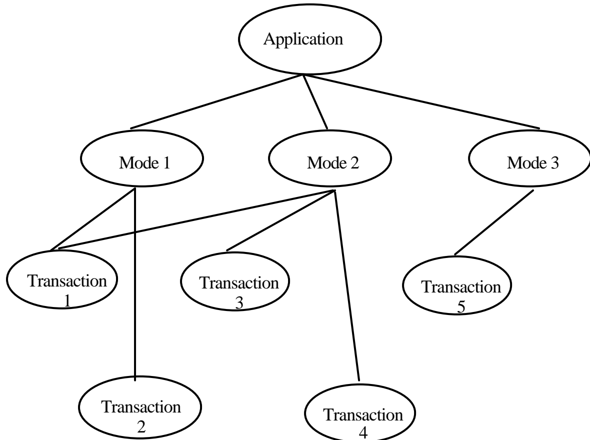

Figure 1: Application model

3.2 Modes and mode transitions

A mode describes specific functionality in a system state. If the functionality differs substantially from one state to another, one should separate them into two different modes. An example is the control system in a vehicle, which can have different functionality depending on the status of the vehicle. If the vehicle is fully functional, the control system is in full operating mode. If a severe error occurs the control system can take the vehicle into a reduced functionality, mode where only the most critical functions of the control system are provided, so that the vehicle can be taken for repair.

Modes in the system are described in a mode transition graph, comparable to a state transition graph, where all legal transitions between modes are depicted. An example mode transition graph for our vehicle is illustrated in Figure 2.

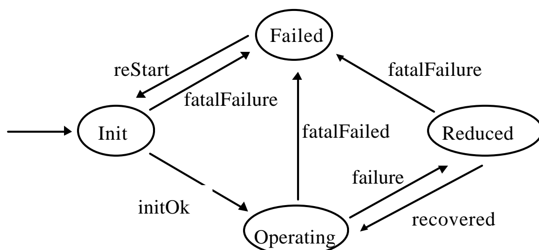

Figure 2: Mode transition graph for a control system for a vehicle

Why modes? In almost all application there is some kind of mode concept, even if implicit. For example when the system is starting up, initialisation functionality is provided which is no longer needed when the system is fully operating. Many systems also have a failure mode with reduced functionality. If there is no way of specifying these modes they have to be implemented ad hoc in the code which makes it hard to understand and maintain the system.

3.3 Transactions and interaction graphs

For each mode a number of functions must be provided, we call these transactions. A transaction consists of a collection of tasks that together provide the desired functionality. The interaction and dependencies between tasks are described by communication and synchronisation constructs. Communication is specified as a directed relation from one task to another. Synchronisation can be described by precedence relationships between tasks or by mutual exclusion of task that share a common resource. The temporal behaviour of tasks in the transaction is specified by temporal attributes of single tasks. Besides tasks, interrupts can also be specified. Including interrupts in the specification makes it possible to include them in the analysis.

3.4 Task

A task is the smallest executable unit. A task is described with a set of functional and temporal parameters:

Functional:

- **Entry function**. The entry function specifies the function to perform on each invocation. This, together with the state and input, defines the functionality of the task.
- **State**. A task has some state variables, (comparable to instance variables of an object), which keep their values across activations of the task. The variables constitute the task state.
- **Ports**. Since communication primitives are not allowed in the code, communication is specified in the interaction graph. Each task is equipped with in- and out-ports. The in-ports acts as input to the entry function and the result of the entry function is placed on the out-ports of the task.

Temporal:

- **Period time**. The period time of the task.
- **WCET**. Worst Case Execution Time of the entry function. Note that this value is assessed and used as

an additional design parameter during the design and verified after implementation.

- **Release time**. Remember that every task is a member of a precedence graph and therefore has a period. The release time is the earliest time the task can be activated, relative to its period start.
- **Deadline**. The deadline is the latest time a task is allowed to terminate, relative to its period start.

The *execution semantics* of a task is at activation to read the in-ports, thereafter perform the function, and before termination write the result to its out-ports. This construction means that each task can be designed without knowing where the input data was produced and where the produced output data will be used.

4 Mapping of the design to a resource structure

The Configuration Compiler tool maps a textual based description of the design to a resource structure, as illustrated in Figure 3. The Configuration Compiler is a pre-run-time scheduler that generates dispatch tables and communication infrastructure for each mode. Besides the mapping of the model, the tool also supports specification of architecture specific attributes like performance, the time granularity of the run-time dispatcher, communication times, and number of nested pre-emptions allowed. The implementation of the Configuration Compiler is based on a heuristic tree search strategy, similar to the one presented in [Ram90]. The major difference is that this scheduler takes interrupts and architecture specific attributes into account. The current version of the tool is adapted to the real-time operating system Rubus[1].

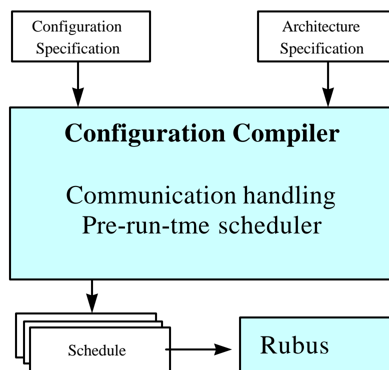

Figure 3: The Configuration Compiler

[1] Rubus and the Configuration Compiler are commercial products, see www.arcticus.se.

5 Development methodology

The development methodology defines the workflow when developing an application. The methodology employed in this project is iterative and quite traditional. The emphasis in the method is to derive a high level design that enables early schedulability analysis. To facilitate this it is required that synchronization, communication, and temporal attributes are defined early in the design process, which is of no problem except for execution times of the tasks. The execution times are normally derived from the code. However, in this approach we specify (estimate) an execution time budget for each task. The execution time budget is later in the implementation phase used as an implementation requirement. Estimating the execution time budgets is a delicate issue that requires highly skilled engineers with a lot of experience. However, if the estimate can not be fulfilled a negotiation strategy has to be employed. That is, execution time may be borrowed from another task, which does not utilize the allocated execution time. The development methodology is general and can be adapted to different design languages (modeling languages). The method is briefly described in Figure 4 and by the following text.

I. **Requirements engineering**. Here are the requirements formulated by the customer of the system.

II. **Requirements analysis**. In this stage the functions of the application are identified from the requirements specification. An important aspect here is also to determine temporal constraints for these functions.

III. **High-level system decomposition**. In this stage the application's different operational modes are identified together with valid transitions between them, by specifying the mode transition graph.

IV. **Function decomposition and structuring**. The functions, for each mode, are decomposed into transactions. Note that one transaction could belong to several modes. Transactions are decomposed into smaller units called tasks and their low-level functions are specified together with the data flow information between them. Some high level functions has parts that have a high demand of responsiveness or are very frequent (but small) so that implementing them as tasks would be infeasible. Therefore such low-level functions are implemented as interrupts. This is formally described in an interaction graph.

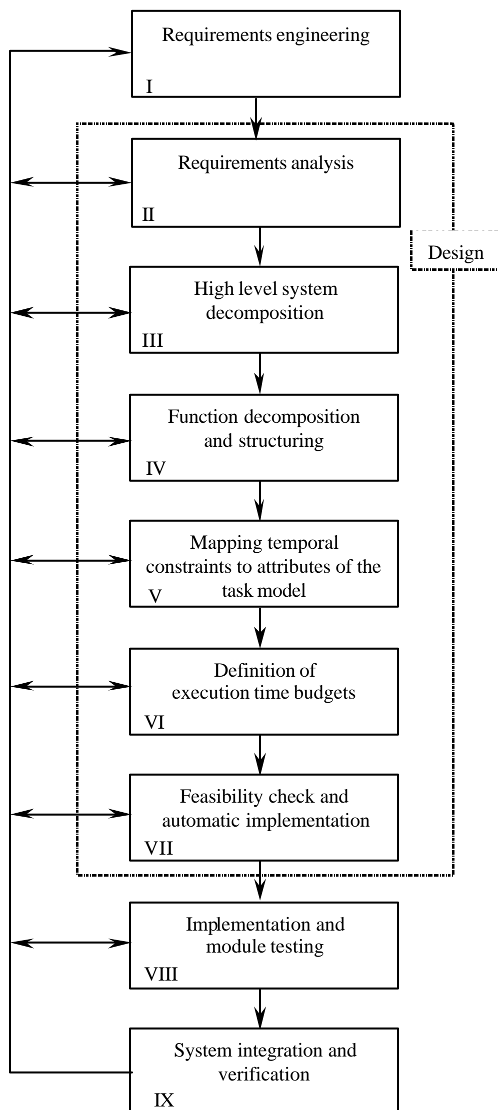

Figure 4: The design methodology

V. **Mapping temporal constraints to attributes of the task model**. In the previous stage the high level functions were decomposed into smaller units and structured according to the interaction between them. This step has to brake down the high level temporal requirements into temporal attributes for these smaller units. The expressiveness of the task model attributes are different, and lower level, than specified for the high level functions, so it is important that this transformation is done in a safe way, i.e., that the task model attributes does not violate any of the high level constraints. It is also important that this mapping does not overconstrain the system.

VI. **Defining Execution Time Budget**. Traditionally the assessment of WCET is done by either measurements or by statically analyzing the code produced for each task. In this approach, however, execution time budgets are defined, these budgets are later in step VIII used as implementation requirements. The reason for this is that a feasibility test for the system, and a possible re-engineering, can be done at an early stage, and thus provide early detection of design errors related to resource utilization, communication and synchronization.

VII. **Feasibility check and automatic implementation**. The formally described design can be checked for temporal correctness even if no actual (low-level) implementation has been done. This is done by a static scheduler, which tries to find a feasible schedule. Besides the schedule, the communication infrastructure is automatically generated.

VIII. **Implementation and module testing**. The implementation of tasks is simply done by traditional programming (coding). Besides the traditional functional specification, the programmer also has the execution time budget as an implementation requirement, i.e., the programmer has to implement the specified function in a way that it does not violate the budget. The module testing includes both verifying the functional behavior as well as that the time budgets are not violated. If the time budget can not be met a redesign has to be done.

IX. **System integration and verification**. The integration phase is usually done very quickly and without problems since the actual integration was done in the design with a strict semantics. The major work is to do the integration testing.

The above figure and listing defines the activities performed in each step, and the iteration when using this method.

6 Findings

In this section we will describe the findings acquired when introducing and using the design language and method earlier described in Section 2. The findings are categorized into those related to development methodology, technology transfer, and technical issues respectively. The development methodology covers the findings based on the use of the design language and method. The technology transfer part describes issues regarding the transferring and introduction of new technology and especially real-time technology into an organization. The technical issue part presents new or relevant technical challenges that have been discovered during this work.

6.1 Design methodology

Finding 1: The design language provides a good basis for the design description.

Motivation:

Using the design language described in Section 3 gives three major benefits when designing a system:

1. *It gives a skeleton of the application, which can be analyzed without having a single line of code.*
2. *The analysis leads to early error detection of communication, synchronization, and timing errors.*
3. *Simplified system integration.*

Currently we can analyze communication, synchronization, and timing requirements. Communication is analyzed in three different aspects. Firstly, the types of connected ports are checked, which ensures that the proper data types are passed to the tasks. Secondly, the analysis will reject a design where the amount and rate of data passed through the system makes it infeasible to fulfill the timing requirements. Thirdly, data consistency is checked. Again, if there is no possible way of fulfilling all timing requirements and at the same time guarantee data consistency, the design is rejected.

The analysis of the synchronization makes sure that all precedence and mutual exclusion relationships between tasks can be guaranteed in conjunction with guaranteeing the timing requirements.

Finally the analysis of the timing requirements reveals if it is possible to find a schedule for the given design and execution time budgets that fulfills these timing requirements. If it is impossible to fulfill the timing requirements the design will be rejected.

The analysis presented above leads to early detection of errors, in the design, of the properties that are analyzed. Such errors are otherwise often found in the integration phase of the project and thereby cost a lot of time and effort to correct.

System integration is also simplified by the early analysis. If the implementation of the code of each task comply with the interface given by the design, i.e., retrieving data only from the in-ports, performing the desired function within the given execution time budget, and producing data only to the out-ports, then the integrated system will fulfil the design and thus satisfy the requirements. Thus, a step of the development process, that often tends to be quite troublesome and leading to costly delays in the project, are simplified.

Note that the only thing that has to be added to implement the design is the task code, everything else is automatically generated, i.e., communication, synchronization and an execution scenario (schedule).

Finding 2: The use of a precise design language

a) Enables parallel implementation and testing of the tasks.

b) Facilitates efficient integration of new personnel into the project.

Motivation:

a) The task model stipulates tasks, which have no synchronization or communication within the code. Recall from section 2, that each task uses a computational model based on input - calculation - output. That leads to that each task can be implemented and tested in parallel since each task is only dependent on its own state and the values of its in-ports to make a calculation. The module testing is, thus, very simple to make, just feeding values to the in-ports and monitoring the output. This also allows regression testing of modules.

b) One small group of people, who have good knowledge about the system and a good feeling of future demands on the systems, develops the design. The design they come up with must be stable, that is, not too many major changes are allowed to occur after the implementation phase start. If that is accomplished, it is easy to introduce new personnel into the implementation phase since each new employee or consultant only has to understand the design language and obey the given interface to be able to start to implement and test. The design language has decreased the introduction time for new employees substantially.

Finding 3: The methodology increases the time spent in the design phase but shortens the implementation time.

Motivation:

We feel that the time to complete the design phase has increased compared to similar projects, which have used traditional informal techniques (such as structured analysis and design). This is not surprising since a precise design with analysis is harder to come up with, compared to a design that just is based on written documents. However, we feel also that the precise design has lead to shorter time spent on implementation, test, and integration due to reasons described earlier in this section. We also believe that it will be much easier to maintain a system based on a precise design compared to a traditional system. This is mainly due to two reasons:

1. Normally the implementation and the design tend to diverge which makes it hard to foresee the impact of changes and added functionality. This can be avoided by the fact that the tools are useful and actually produces verified functionality. It is for example quite natural and widely accepted to use the compiler

instead of adding object code here and there. Another restraining factor can be the fear of disturbing the order laid out by the tools, again compare with the compiler example.

2. Even if there is a good match between the design documents and the implementation it is not easy to foresee the impact of changes and added functionality. In our case several properties of the altered design can be analyzed, as discussed earlier, already in the design phase. So changes or add-ons that does not comply with the implemented functionality will be detected.

Finding 4: Execution time budgets for tasks turned out to be good as a design tool and implementation requirement.

Motivation:

To be able to make an early capacity analysis of the resources in the system, like processors and buses, each task has to have an execution time budget. This budget states how much of the processor capacity the task is allowed to utilize. The difficulty in specifying this budget is to relate the execution time budget to the functional requirements of the task, e.g., for a controller it should be possible to fulfil the desired control performance within the specified time budget. If it is not possible this time budget is erroneous. The execution time budgets are then used as implementation requirements.

In this project we were really surprised that these budget estimations where so good. However, the engineers that specified these budgets had many years of experience in control system design and good knowledge about hardware close programming.

To verify that the implementation fulfils the requirements the execution time for the tasks was measured and sometime calculated.

6.2 Technology transfer

Finding 5: To be able to transfer real-time technology to industry; tools, education (courses, tutorials), carriers, and adapters are required.

Motivation:

Tools:

When transferring theories to the industry it is necessary that the theory is encapsulated in a tool, which shows the practical use of the theory [Sch96], unless the theory is very simple [Bat99]. A good example of a tool that encapsulates advanced technology well is a traditional compiler. In this case the tool was in the first version an application written in a high level language that was easy to adapt to up-coming requirements from the industry. To handle these up-coming requirements in an efficient way is important to succeed in the transfer, a part where the carrier described below play a significant role. The tool was later ported to a low-level language to get an efficient implementation.

Courses:

We have found out that an engineer requires at least two days of training to understand the basic real-time theory and the added methodology to be able to work with design of new systems. So in reality for an experienced engineer it will take about one week including the training course to be productive, from the model and methodology point.

Carrier:

The success of this transfer is mainly because one person, that worked in the research group where the ideas where developed, started to work as a consultant for TietoEnator ArosTech at VCE. Regardless how many good reports we write we need people that carries the results [Dal94]. A related example is the development of the control system for Volvo S80 where Ken Tindell and others carried the response time analysis for the CAN bus into a tool and implanted that tool into Volvo Car Cooperation organization [Cas98, Mel98].

Citation: "Tech transfer is a contact sport. People not paper transfer technology" [Fol96].

Adapters:

Even if we have carriers we need early adapters at the company that take the technology into the company and its organization. These people need to be authoritative to be able to sell the new technology in the organization. There is always a healthy conservatism in all organization. Therefore one must find people that are ready to invest enough time and energy to find out if the technology is applicable and gives an added value to the development of their products or not [Ben96].

Finding 6: The major problems when introducing real-time technology in an organization is to change the requirements caption process to include timing requirements.

Motivation:

Several independent sources have given the same statement (Volvo Car and Volvo Construction Equipment). Especially since all engineering disciplines within a company has to change their way of specifying requirements on the electronics. The main problem is that once a timing requirement for a high-level function has been derived, it is very hard to reconsider it later on. It seems that a timing requirement becomes more and more truthful the older the timing constraint becomes. This really comes to the surface when a new function is added

and the schedulabilty test is negative depending on that the utilization of the system is too high. To add this function anyway you need to find either execution time budgets that are too generous or timing requirements that are too strict. Assuming the overestimation of execution times is neglectable, the timing requirements have to be reconsidered. To find out which timing requirements that have to be relaxed there must exist a notion of confidence of the timing requirements. As an example, the time from pressing a particular lamp switch until the light is turned on should it take 200 ms or 300 ms, if the requirements say that the confidence in specifying 200 ms is low this timing requirement could be considered to be relaxed. Thus the results from the requirements caption process must be clearly expressed and well motivated since it will be used during the complete life cycle of the system.

6.3 Technical issues

Finding 7: The task model used (described in Section 2) is in some cases too restricted when handling control jitter for simple controllers and especially for multirate controllers.

Motivation

The *limited expressiveness* in the used task model is related to the jitter problem and multirate communication problem. Specifying release times and deadlines of the tasks involved in the computation can be used to fulfil for example jitter requirements. However, this is a problem since the engineer has to distribute the release times and deadlines at the timeline to not overload a specific window of the timeline. This means that the engineer has to act as a pre scheduler to the scheduler, which is not efficient. Instead, a desired property of the task model would be to have the possibility to specify relative timing constraints. For example, a sampling task is required to run with a certain period time and have a tolerance of a specific amount (*Period time ± tolerance*). Relative timing constraints could also be used for specifying latency constraints, e.g., the time between sampling and actuation. Furthermore, when a controller consists of several entities that run with different period times, i.e., multirate control, one would also like to have the possibility to specify latency constraints. If the used task model supported this it would be much simpler to specify a system. Extending the task model is an easy task but to come up with useful tools to schedule a system based on such a task model is not an easy task.

Finding 8: Task model and scheduling techniques reported in literature has to be extended to take real-world requirements into consideration.

Motivation:

When the scheduling tool for this task model was developed we had to take several important aspects into account to be able to get a tool that utilized the resources of the target system efficient. The two aspects we will cover here are schedule representation and taking interrupt overhead into account when constructing the same schedule.

Schedule representation. A common representation of a static schedule is a vector, where one position in the vector represents a discrete point in time at which the execution of a task can start. The granularity of time has to be matched with the frequency of the periodic clock that drives the dispatcher, which will execute the tasks according to the schedule. If the execution time of a task is less than this granularity, or if it exceeds a multiple of the granularity with a small fraction, then the utilization of the CPU resource will decrease. This because there will be time intervals that can not be used to execute tasks. An apparent solution to this is to increase the granularity (frequency) of the periodic clock. However, with a higher frequency of the clock the dispatcher will instead use more of the CPU resources, since it will execute more often.

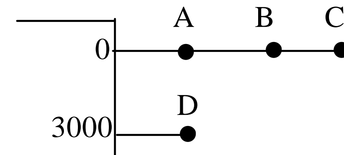

Figure 5: The representation of a schedule.

Another way of representing a schedule is as a list of rows, see Figure 5, where each row represents a point in time at which the dispatcher is to start the execution of a sequence of one or more tasks. The first task in this sequence, or *chain*, is started at the given point in time. All other tasks in the chain are started as soon as the preceding task in the sequence has completed its execution, without need for the clock to trigger the dispatcher. This representation will allow several tasks to be executed during an interval less than the period time of the dispatcher clock. Hence, the dispatcher overhead can be kept low at the same time, as the utilization of the CPU resource is high.

Interrupt overhead. Typically, pre-run-time scheduling does not account for interrupts, assuming their execution can be ignored or incorporated into task execution times. In many applications, the interrupts are, however, non-negligible and inclusion in task execution is too pessimistic and inefficient. Furthermore, as inter-arrival and execution times of interrupts are smaller than the granularity of the online dispatcher and the arrival times are unknown, interrupt-handling routines cannot be

modeled as pre scheduled tasks. The application of server algorithms, e.g., sporadic server [Spr89] total bandwidth server [Spr95], and slack stealing [Leh92] are not feasible due to the short response times that are required.

The key issue for static scheduling accounting for interrupts is the consideration of the overhead. If interrupts occur at run-time, interrupt-handling routines are executed. The delay this poses on task execution must be accounted for when the system is scheduled. Evidently, an inherent, minimum amount - the worst case penalty - to handle a worst case scenario has to be reserved, according to minimum inter-arrival times and execution times. Any amount exceeding this, however, is overhead imposed by the used method. It is this overhead that has to be kept small for efficient utilization of the processor. During this project we had to develop a method that handled interrupts in an efficient manner. This method combines a tree search algorithm with response time analysis, see the paper by Sandström et al [San98].

Finding 9: To make a pre-run-time scheduler tool really useful, feedback has to be provided to the user when the system is not schedulable.

Motivation

When applying scheduling in industrial projects, engineers are faced with a problem that only to a very limited degree has been attacked by the real-time research community, namely how to provide constructive feedback to the user in cases when a feasible schedule can not be found.

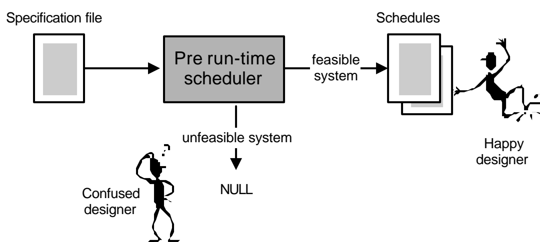

Figure 6: Pre run-time scheduling: present situation

This *limited feedback problem* leads to confused designers, as illustrated in Figure 6, which more or less at random have to optimize and modify the specification. However, to help the designer to come up with a specification for which the pre-run-time scheduler can find a feasible schedule, there is a need for heuristics that analyze the specification for semantic problems and give constructive feedback to the user. That is, the user should be provided hints to how the problem can be resolved, i.e., how the specification can be modified to allow the generation of a feasible schedule.

We have developed a method to provide feedback to the user by calculating a load function for the system. By identifying bottlenecks in the system specification we can guide the designer in modifying the input to the pre-run-time scheduler. The underlying hypothesis is that there is a correlation between the points in time when the load function has a high value, and the locality of the bottlenecks in the specification that leads to an infeasible schedule, [All96].

Finding 10: To minimize the verification effort when only small updates have been done to the application an incremental scheduling is needed.

Motivation

When an application has been tested and used in a vehicle for some time without any problems, the application is accepted and released. If then later some new functionality is added one wants to keep as much as possible of the execution order in the application to avoid major re-verification efforts.

This is not possible today, that is, when adding new functionality to the application a completely new schedule has to be generated. The major drawback of this approach is that the application verification and validation has to be completely redone to guarantee the functionality.

A desired feature of a scheduler would be to have the possibility to incrementally add new tasks to the application without affecting the already verified and unchanged part. A scheduler that takes both the updated design specification and the old verified schedule as input could solve this problem. The scheduler could try to find space in the old schedule for the new tasks or if not minimize the number of changes.

We believe that it is more important to keep the order than keeping the exact start times of the tasks, as long as the timing requirements are fulfilled. We believe this because there often are margins in the execution windows for the tasks while a change of order could have severe impact for example on multirate transactions, which often are sensitive on data age.

7 Conclusion and Future research

We believe the presented project has been successful in transferring real-time technology from a university to an industrial partner. As a result, the industrial partner has adopted a more systematic and formalized design process, which have shortened the overall development

cycle compared to similar previous projects. It also seems that the quality of the products has met the requirements.

However this transfer goes both ways, industry has also provided new relevant challenges for academia. An example of this is the limited expressiveness of the task model for real world constraints including specification of jitter constraints and specifying relative timing constraints (suited for multi rate control systems). It would be quite easy to extend the task model with such attributes, but the mapping of these to an implementation and the feasibility check, including schedule construction, is not a trivial task. Another example is the limited feedback problem of the static scheduler when it is unable to find a feasible schedule. There is a lot to gain if information can be given to the designer where to find the bottlenecks in the design and specification. The need of a incremental scheduler is also pointed out, which would be very useful when maintaining the application.

Remember: tech transfer is a contact sport, people not paper transfers technology!

Acknowledgements: We would like to thank Jack Stankovic, Hans Hansson, Sasikumar Punnekat, and Ivica Crnkovic for valuable discussions and for reviewing earlier versions of this paper. We would also like to thank Krithi Ramamritham for encouraging us to write this paper.

Mälardalen Real-Time research Centre (MRTC; www.mrtc.mdh.se) is a research centre in Västerås, Sweden, supported by Swedish industry, the Swedish Foundation for Knowledge and Competence Development (KK-stiftelsen) and Mälardalen University.

References

[Ben96] J. L. Bennett. Building Relationships for Technology Transfer. Communications of the ACM, Volume 39 Number 9. Sep. 1996.

[Spr94] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. The Journal of Real-Time Systems 1, 27-60 (1989)

[Leh92] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft aperiodic tasks in fixed priority preemptive systems. In Proc. IEEE Real-Time Systems Symposium. Dec. 1992.

[All96]B. Allwin, K. Sandström, and C. Eriksson. Constructive Feedback Turn Failure into Sucess for Pre-Run_time Schduled Systems. 11th Euromicro Workshop on real-time systems.

[Spu94] M. Spuri and G. C. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In Proc. IEEE Real-Time Systems Symposium. Dec. 1994.

[San98] K Sandström, C. Eriksson, and G. Fohler. Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System. In Proceedings of the fifth International Conferance on Real-Time Computing Systems and Applications, pp. 158-165, October 1998. ISBN 0-8186-9209-X.

[Cas98] Jim Foley. Technology Transfer from University to Industry. Communications of the ACM, Volume 39 Number 9. Sep. 1996.

[Cas98] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg Volcano a revolution in on-board communications. Volvo Technology Report. 98-12-10.

[Mel98] K. Melin. Volvo S80: Electrical system of the future Volvo Technology Report. 98-12-11.

[Bos00] J Bosch. Design and Use of Software Architectures, Adopting and Evolving a Product-Line Approach Addison Wesley. ISBN 0-201-67494-7, June 2000 (forthcoming)

[Kal88] D. Kalinsky and J. Ready. Distinctions between requirements specification and design of real-time systems. Conference proceedings on TRI-Ada '88 , 1988, Pages 426 – 432.

[Dal94] M Dalziel. Effective university-industry technology transfer. Canadian Conference on Electrical and Computer Engineering, 1994 , Conference Proceedings, Page(s): 743-746 vol.2

[Bat99] I Bate and A. Burns. An Approach to Task Attribute Assignment for Uniprocessor Systems. Proceedings of the 11th Euromicro Conference on Resal-Time Systems, York, England, UK, June, 1999

[Sch96] J. Scholtz. Technology Transfer through Prototypes. Communications of the ACM, Volume 39 Number 9. Sep. 1996.

[Ram90] K. Ramamritham. Allocation and Scheduling of Complex Periodic Tasks. In 10th Int. Conf. on Distributed Computing Systems, pages 108-115, 1990.