Real-Time System Constraints: Where do They Come From and Where do They Go?

Cecilia Ekelin and Jan Jonsson

Department of Computer Engineering
Chalmers University of Technology
S–412 96 Göteborg, Sweden
{*cekelin,janjo*}@*ce.chalmers.se*

Abstract

Within the real-time community a great deal of research has been, and is being, conducted on scheduling. There exist a diversity of different scheduling algorithms as well as constraints that should be satisfied by a feasible schedule. What is not commonly discussed though is how the constraints relate to the requirements on the actual real-time system. In fact, many constraints are mere artifacts that cannot be traced to the system design. This often results in an over-constrained and infeasible scheduling problem. To avoid this we are conducting a survey on industrial real-time system requirements. The aim is to give a better understanding of how to derive adequate constraints. Furthermore, we believe that by considering constraints imposed by industrial requirements such as product cost or hardware limitations, the time complexity of schedule generation can be greatly reduced. In this paper we present some early results from our survey. This includes a description of real-time system requirements and how they relate to commonly used real-time constraints.

Keywords: *real-time constraints, real-time scheduling, system design, constraint programming, complexity reduction*

1 Introduction

Scheduling in distributed real-time systems involves assigning n tasks to m processors such that all constraints on the tasks are satisfied. A common instance of the problem is to find the optimal schedule according to some objective. In real-time literature, several scheduling algorithms that deal with specific real-time constraints (periods, release times etc.) have been proposed. However, in most cases there is no clear connection between the theoretical scheduling problem and the practical design of the real-time system being scheduled. This makes it difficult to verify the correctness of the constraints with respect to the requirements in the system design. Are the scheduling constraints due to limitations in hardware, cost or system behavior? Or are they more or less intelligent guesses? Previous research [1, 2] have argued that the latter often is the case. Adding constraints manually is an iterative process because the *ad hoc* assignment often leads to an over-constrained and infeasible system. Apparently, the reason for fabricating constraints is the lack of expressive power in current tools and the lack of scheduling tools that handle complex constraints. The facility to consider factors such as cost and hardware performance is also lacking in most proposed scheduling algorithms. However, for industrial real-time system developers, these factors are extremely important in order to provide cost-effective solutions. In particular since the trend in real-time system development is moving towards open systems [3], that is, the system designer is restricted to use the hardware components available on the market. Designing real-time systems with these prerequisites also affects the definition of an optimal schedule. The optimality of a schedule might not only concern the behavior of the tasks but also the cost of the resulting hardware configuration.

In an attempt to aid industry in this design process, we have begun work that attempts to identify requirements like cost and hardware limitations, as well as behavioral aspects of industrial real-time systems. This paper reports some results from this work and also demonstrates how these requirements can be translated into constraints. Apart from increasing the expressive power we also believe that, by introducing additional constraints, the scheduling search space will be reduced without making the system over-constrained. In particular, we are interested in investigating how this hypothesis can aid in reducing the search complexity for optimal schedule generation.

The rest of this paper is organized as follows. Section 2 describes some common real-time system requirements. Section 3 lists some real-time constraints typically found in literature. Section 4 presents our allocation and scheduling approach. In Section 5 related work is discussed and Section 6 outlines our future work.

2 Identification of requirements

When trying to identify the requirements on a real-time system from an industrial perspective it is necessary to work in close relation to industry. We are therefore performing our investigation in cooperation with Mecel AB in Göteborg, Sweden. Mecel AB develops tools and distributed real-time systems for the automotive industry. However, as an initial attempt we have limited the survey to requirements and constraint constructs found in literature. So far, we have found that the requirements can be divided into three groups which are further described below.

2.1 Behavioral requirements

A model of a real-time system includes tasks, that determine the behavior of the system, and resources, that are used by tasks. The behavior of the system imposes restrictions on its implementation which affects the scheduling constraints. Typical behavioral requirements could be task execution order or task allocation.

2.2 Temporal requirements

Most development tools for real-time systems focus on modeling the system behavior but lack the expressive power to handle temporal requirements [4]. This is a serious disadvantage since real-time systems have a temporal behavior as well as an operational. The temporal behavior depends on the environment that the system interacts with. That is why the requirements rarely are stated for each task individually but rather on chains of tasks constituting a specific function. For instance, a function like cruise control might have a required update rate which is set independent of the number of tasks that are part of the function. Most scheduling constraints stem from temporal requirements.

2.3 Cost requirements

Apart from mere system requirements, it is in the interest of industry that development of real-time systems is made cost-effective. That is why development using off-the-shelf hardware components has become an interesting alternative. Instead of making hardware for the software, the software has to be made for the hardware. This puts additional strain on the system design since the flexibility is decreased. Even though less flexibility is likely to reduce scheduling complexity, additional hardware constraints have been introduced.

3 Derivation of constraints

In the introduction it was stated that most real-time system constraints are artifacts. The reason for this is the inability of current tools to translate system requirements into system constraints in an adequate way. In the translation the requirements are usually divided into a number of simple constraints. Unfortunately, the *ad hoc* methods used to derive the constraints often leads to an over-constrained and infeasible system. In our approach we try to keep the constraints as close as possible to the original requirements. Below we discuss how a number of constraints, that frequently appear in real-time literature, relates to typical system requirements.

3.1 Task model

We use the following variables to denote task properties for a task i.

- S_i actual start time
- E_i worst-case execution time
- R_i release time, i.e., the earliest allowed start time
- D_i deadline, i.e., the latest allowed finish time
- P_i period, i.e., the rate that the task is executed with
- B_i blocking time, i.e., the time the task is spent preempted by other tasks
- N_i execution node, i.e., the processor the task is scheduled to execute on

Also, c is used to denote an arbitrary constant where applicable.

3.2 Preemption

Apart from the derivation of constraints, it has to be decided whether or not tasks are allowed to preempt each other. By allowing preemption it might be possible to find schedules for designs that are otherwise infeasible. Let n denote the number of tasks. Then the time a task i is blocked due to preemption is defined as:

$$B_i = \sum_{j=1}^{n} B_j :$$
$$\{i \neq j, S_i \leq S_j, S_j \leq S_i + B_i + E_i, N_i = N_j\}$$

Context switching or task invocation time is usually assumed to be small enough to be neglected or included in the task's worst-case execution time. However, there might be some real-world problems where this approximation is not desirable. For example, in preemptive scheduling where it is not known beforehand how many times a task will be preempted. Most likely, the context switch occurring at a preemption will be significant since the state of the preempted task must be stored away or retrieved. The context switch time c can easily be incorporated into our previous equation as:

$$B_i = \sum_{j=1}^{n} (B_j + c) :$$
$$\{i \neq j, S_i \leq S_j, S_j \leq S_i + B_i + E_i, N_i = N_j\}$$

3.3 Absolute timing constraints

Absolute timing constraints are direct restrictions on the temporal behavior of a task.

Execution times express the worst-case execution times of the tasks. They depend on the hardware the task is scheduled to execute on. This is of importance in a distributed system where the execution time for a task may differ between the nodes. Let $E_{i,N}$ be the execution time of task i on node N. Then this constraint is defined as:

$$E_i = E_{i,N} : \{N = N_i\}$$

Deadlines correspond to the responsiveness required by the system. Usually, there only exists an explicit deadline for a chain of tasks (end-to-end deadline) and not for each individual task. A common approach is to split the end-to-end deadline into shorter deadlines which are assigned to each task [5]. This operation may put an unnecessary strain on the task set. To avoid this, we assign the end-to-end deadline as the deadline for all tasks in the chain. Let D be the end-to-end deadline. Let T_D be all tasks restricted by D. Then the deadline constraint is defined as:

$$\forall i \in T_D | D_i = D$$

To ensure that no deadlines are missed, the following constraint is needed:

$$S_i + B_i + E_i \leq D_i$$

This assumes that the precedence relations have been modeled as described later in this paper.

Release times are mostly connected to periodic processes where they restrict the start time of task invocations. This is explained further below. Many scheduling algorithms impose release times on the tasks as a way to obtain mutual exclusion between tasks that access the same resource. The drawback of this technique is the same as for the deadline assignment mentioned previously, namely the risk of over-constraining the task set. To ensure that a task is not started before its release time, the following constraint is needed:

$$S_i \geq R_i$$

Periods express how often a task should be executed. This is linked to how accurate the system needs to be for interaction with its environment. Periods may restrict the deadlines since the deadline of a task often is less than its period ($D_i \leq P_i$). Even if that is not true ($D_i > P_i$), the k:th invocation of a task must be completed before invocation $k + 1$. A way to handle scheduling of periodic tasks is to find the least common multiple of all periods and then schedule each invocation as an individual task. The different scenarios impose the following constraints:

$$R_{i,k} = (k - 1) \cdot P_i : \{k > 0, D_i \leq P_i\}$$

$$R_{i,k} = (k - 1) \cdot D_i : \{k > 0, D_i > P_i\}$$

Separation constraints [1] express an interval of values that the period of a task should belong to. In contrast to deadlines, periods might be limited by a maximum and/or minimum period value which ensures that the required functionality is obtained. Let l and u be the minimum and maximum value respectively. Then this constraint is defined as:

$$l \leq P_i \leq u$$

3.4 Relative timing constraints

Relative timing constraints are also known as local constraints. That is, they express how two tasks relate to each other. Some common constructs are explained below. Other constructs are possible [6, 7] and can be expressed in a similar manner.

Precedence constraints deal with the order the tasks should be executed in. Such constraints are often derived from the system design because the behavior of a system is modeled as sequences of operations. If task i should precede task j the constraint is defined as:

$$S_i + B_i + E_i \leq S_j$$

Distance constraints are a stronger variant of precedence constraints. In addition to constraining their order of execution, they also express the minimum distance in time between two tasks i and j. A reason for this could be limitations in the processing speed of the environment that the tasks interact with. It could also be derived from delays in the communication hardware between two communicating tasks. If the distance is c, the constraint is defined as:

$$S_i + B_i + E_i + c \leq S_j$$

Freshness constraints [1] are the opposite of distance constraints. They express the maximum distance in time between two (consecutive) tasks i and j. This is used to express that a task uses some result produced by another task. If the tasks are too far apart in time, the result have become inaccurate when it is about to be used. This constraint is typically found in database applications. If the freshness is c, the constraint is defined as:

$$S_i + B_i + E_i \leq S_j \leq S_i + B_i + E_i + c$$

Correlation constraints [1] are related to freshness constraints in that they describe the maximum difference between the finish times of two (concurrent) tasks i and j. This appears when a third task uses the results from the two tasks and a too large a time-skew between the parameters makes the task's computation faulty. If the correlation is c, the constraint is defined as:

$$|(S_i + B_i + E_i) - (S_j + B_j + E_j)| \leq c$$

Harmonicity constraints [1] relate the periods of two communicating tasks. It is desired that the period of the consumer task i is exactly divisible by the period of the producer task j. If this is the case it is much easier for the consumer to keep track of the received messages since they always will arrive with the same interval. This constraint is defined as:

$$P_i = c \cdot P_j : \{c > 0\}$$

3.5 Resources

To be able to perform their operations, the tasks may require to use specific hardware components. The performance or functionality of the available components is limited by how expensive the hardware is allowed to be.

Locality constraints concern the allocation of tasks to different nodes. A task might require a specific unit for its execution that is not present on all nodes. We express this by simply removing impossible nodes from the task's node set. This is a *hard-coded* constraint. Let N be an impossible node. This is then defined as:

$$N_i \neq N$$

It could also be the case that communicating tasks (i and j) should be located on the same node. A reason for this could be that the operating system requires that all tasks within a process are located on the same node. Another reason could be to lower the cost for the communication network. This is known as a *clustering* constraint. The constraint is defined as:

$$N_i = N_j$$

Locality constrains are often implementation recommendations made by the designer, and less frequently given in the specification.

Communication between tasks requires a communication media. The time to send a message depends on performance offered by the communication channel. Messages sent between tasks must themselves be scheduled on the communication network which affects the scheduling of the tasks. Message scheduling is non-preemptive. There exist predefined constraint constructs for this kind of scheduling, e.g., the *serialized* constraint [8].

Devices are resources other than processors used by the tasks during execution. It might be the case that only a limited number of tasks can access a device at the same time, e.g., shared memory or bus communication. This means ensuring that the amount of the resource used, by concurrent tasks, never at any point in time is exceeding a given limit. This can be expressed using the *cumulative* constraint [8].

3.6 ORing

ORing constraints [6] are not that common but can be used to express alternative operations. For instance, to explicitly state that a task i must not preempt another task j (an EXCLUDES relation) means than task i must execute before or after task j. Any types of constraints can be ORed. For c constraints this is simply defined as:

$$constraint_1 \vee constraint_2 \vee ... \vee constraint_c$$

4 Allocation and scheduling

Most published real-time scheduling algorithms are "hard-coded" with respect to the constraints, task properties and resources they can handle. Even-though a specific method in most cases outperforms a generic, the former soon becomes quite complex and it becomes difficult to include new features. That is why we have chosen the constraint programming approach.

The tool we have selected for our experiments is SICStus Prolog [8] and its associated constraint solver for finite domains [9]. The identified real-time system requirements can be expressed using primitive (low-level) constraints which are combined to express more complex (high-level) constraints. The advantage of this generality is that it is easy to introduce new high-level constraints as long as they can be composed by a combination of low-level constraints. The translation of the constraints into code is pretty straightforward. An interesting feature is the possibility to guide the search for feasible schedules by varying the search parameters. By using "correct" parameters the search time can be significantly reduced. We hope that it will be possible to automatically detect which heuristic that is best suited for a specific problem instance.

5 Related work

A great motivation for this paper was the analysis of the origins of timing constraints presented by Ramamritham in [2]. The gap between the system design process and the implementation has been discussed in [10]. We continue this work by proposing guidelines for the derivation of constraints from system requirements. In [1], a technique to automatically generate periods and deadlines for each task from end-to-end deadlines, is proposed. A similar technique to generate intervals instead of single numbers for the start times has been presented in [7]. Both techniques are based on constraint solving using Fourier-Motzkin variable elimination. The ideas and techniques for constraint solving are already contained in the concept of constraint programming systems.

The use of constraint programming for scheduling has been investigated in the areas of operations research and artificial intelligence. However, the scheduling aspects in these areas are somewhat different from the ones concerning real-time systems. For instance, the optimization objective is different as well as the constraints to be considered. So far, we have only found one paper that addresses the use of constraint programming for scheduling of real-time systems [11]. However, this work only addresses the problem of finding a feasible schedule, while we are more interested in finding the optimal and most cost-effective schedule.

6 Future work

As previously mentioned, we are currently working together with the industry on a survey on real-time system requirements. A remaining issue of importance is the definition of an optimal schedule. To this end, we plan to implement a scheduling framework in SICStus Prolog which will incorporate the described real-time constraints, including a new notion of optimality. At a later stage we will investigate how the constraints and search heuristics can be used to reduce the complexity of the scheduling process. Previous work have shown that this is a viable approach [12, 13].

References

[1] R. Gerber, S. Hong, and M. Saksena, "Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes," *IEEE Trans. on Software Engineering*, vol. 21, no. 7, pp. 579–592, July 1995.

[2] K. Ramamritham, "Where do Time Constraints Come From and Where do They Go?," *International Journal of Database Management*, vol. 7, no. 2, pp. 4–10, 1996.

[3] J. A. Stankovic, "Real-Time and Embedded Systems," *http://www-ccs.cs.umass.edu/sdcr/*.

[4] G. Bucci, M. Campanai, and P. Nesi, "Tools for Specifying Real-Time Systems," *Real-Time Systems*, vol. 8, no. 2/3, pp. 117–172, Mar. 1995.

[5] M. Di Natale and J. A. Stankovic, "Dynamic End-to-End Guarantees in Distributed Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, Dec. 7–9, 1994, pp. 216–227.

[6] T. M. Chung and H. G. Dietz, "Language Constructs and Transformation for Hard Real-Time Systems," *Proc. of ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, Canada, June 1995, pp. 4–10.

[7] R. Gerber, W. Pugh, and M. Saksena, "Parametric Dispatching of Hard Real-Time Tasks," *IEEE Trans. on Computers*, vol. 44, no. 3, pp. 471–479, Mar. 1995.

[8] Intelligent Systems Laboratory, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, 1995, http://www.sics.se/sicstus/.

[9] M. Carlsson, G. Ottosson, and B. Carlson, "An Open-Ended Finite Domain Constraint Solver," *Proc. of the Int'l Symposium on Programming Languages: Implementations, Logics, and Programs*, H. Glaser et al., Eds., Southampton, UK, Sept. 3–5, 1997, vol. 1292 of *Lecture Notes in Computer Science*, pp. 191–206, Springer Verlag.

[10] M. Saksena, "Real-Time System Design: A Temporal Perspective," *Proc. of IEEE Canadian Conference on Electrical and Computer Engineering*, Waterloo, May 1998.

[11] M. Schild and J. Würtz, "Off-Line Scheduling of a Real-Time System," *Proc. of CP97 Workshop on Industrial Constraint-Directed Scheduling*, Schloss Hagenberg, Austria, Oct. 1997.

[12] J. Jonsson, "Effective Complexity Reduction for Optimal Scheduling of Distributed Real-Time Applications," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, Austin, Texas, May 31 –June 5, 1999, pp. 360–369.

[13] I. Ahmad and Y.-K. Kwok, "Optimal and Near-Optimal Allocation of Precedence-Constrained Tasks to Parallel Processors: Defying the High Complexity Using Effective Search Techniques," *Proc. of the Int'l Conf. on Parallel Processing*, Minneapolis, Minnesota, Aug. 10–14, 1998, pp. 424–431.