Off-Line Scheduling of a Real-Time System

Klaus Schild
Daimler-Benz AG
Research & Technology
Alt-Moabit 96a, D-10559 Berlin, Germany
e-mail: schild@DBresearch-berlin.de

Jörg Würtz
Programming Systems Lab, DFKI
Stuhlsatzenhausweg 3, D-66123 Saarbrücken
Germany
e-mail: wuertz@dfki.de

Keywords
Constraint Programming, Time-triggered real-time systems, Scheduling.

ABSTRACT

This paper shows how a recently introduced class of applications can be solved by constraint programming. This new type of application is due to the emergence of special real-time systems, enjoying increasing popularity in such diverse areas as automotive electronics and aerospace industry. These real-time systems are time triggered in the sense that their overall behavior is globally controlled by a recurring clock tick. For this off-line scheduling problem a potentially indefinite, periodic processing has to be mapped onto a single time window of a fixed length. We make this new class of applications amenable to constraint programming. We describe which traditional scheduling and real-time computing techniques led to success and which failed when confronted with a large-scale application of this type. Global constraints were used to reduce memory consumption and to speed up computation. An elaborate heuristic, borrowed from Operations Research, was employed to solve the problem. Furthermore, we show that mere serialization is sufficient to find a valid schedule. The actual implementation was done in the concurrent constraint programming language Oz.

Introduction

There is a growing number of distributed real-time applications whose processing must obey a strictly regular pattern, and so must the communication involved. Exactly for this type of application, a particular class of architectures has been devised, the so-called *time-triggered architecture* [12]. The term *time triggered* refers to the fact that the overall behavior of the system is controlled by a recurring clock tick, the only event which may invoke any action. Time-triggered architectures fit especially the needs of safety-critical applications. This special type of architecture proved to be successful in such diverse areas as automotive electronics and aerospace industry.

Being able to compute an appropriate pre-runtime schedule automatically is the major challenge for a time-triggered architecture. What makes this specific off-line scheduling problem somewhat untypical is that a potentially indefinite, periodic processing has to be mapped onto a single time window. This time window determines the overall behavior of the system. Take a thermostat process T_{10}, transmitting at a frequency of 10Hz the particular temperature chosen to a controller C_{10}. Let this controller respond with exactly the same frequency, reporting potential malfunctioning. This induces a cyclic dependency which is explicitly allowed. If two application processes are allocated to different processors, then the only way for them to communicate with each other is a common data bus. All inter-processor messages must therefore be broadcasted through this data bus. A further restriction is that this bus is not able to transmit more than one message at a time.

Let us furthermore assume that there is a second controller, call it C_{20}, which gets at a frequency of 20Hz fresh data from a certain sensor S_{20}. In this case, however, the communication is one-way only. We assume that S_{20} and C_{20} are hosted by different processors. In particular, let S_{20} reside on the same processor as T_{10}, while C_{20} shares its host with C_{10}. The off-line scheduling problem then consists in mapping these quite different communication patterns onto a single time window, with no preemption being allowed. When repeated indefinitely, this time window should produce the intended overall behavior. Fig. 1 depicts a proper time window which does achieve this.

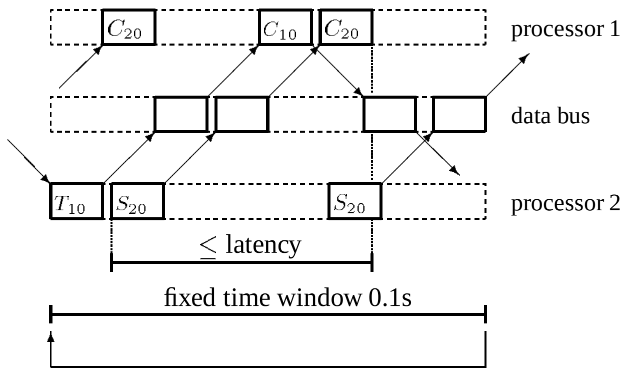

Figure 1: A sample repetition window

There may also be relative timing constraints between processes allocated to different resources. Constraints of this sort are called *latencies*. Fig. 1 includes such a latency between S_{20} and C_{20}. This latency is an admissible upper time bound on the time which may pass between the start of S_{20} and the termination of C_{20}, with the actual message passing in between. This is to guarantee that the relevant information processing is completed within the time bound specified.

These uncommon characteristics make this off-line scheduling problem a challenging new application domain. Only a few approaches exist to tackle similar problems. For an overview, especially on applications in the automotive industry, see [6]. The real-time applications tackled by constraint techniques are usually small on-line scheduling problems rather than off-line problems; see e.g. [19].

To the authors' knowledge, however, it is the first time that constraint techniques are applied to a time-triggered architecture. We modeled the relevant off-line scheduling problem with the help of the constraint programming language Oz, developed at the DFKI [18]. The actual problem that we solved was part of a large-scale industrial application. It involves a finite domain of 6 million different starting times, over 2,000 processes and messages, and more than one million constraints created during the solution search.

The contributions of this paper are as follows. We compare different techniques to solve traditional scheduling problems and real-time applications and show for the present case which techniques lead to success and which fail. We make use of global constraints in form of a single computational agent that constrain up to more than a thousand variables simultaneously. These global constraints significantly reduce the number of constraints that are active at a time. Search strategies coming from traditional real-time applications and Operations Research are tested. Only a recent strategy from Operations Research proves to be successful in solving the actual application. We show that it is sufficient to find a total ordering of the processes and messages to construct a valid schedule. We furthermore show that the usual criterion of utilization (or load) is insufficient for a bottleneck analysis for the application at hand. To solve problems of that size, it is important that redundant constraints are garbage collected as soon as possible.

The paper is organized as follows. The following section gives the details of the specific off-line scheduling problem to be tackled. The third section then briefly introduces the basics of constraint programming. The fourth section shows how the scheduling problem can be captured by finite-domain constraints. An elaborate search heuristic is included as well. The fifth section demonstrates that this heuristic enables us to solve even large-scale industrial applications. The paper closes with a brief discussion of the results obtained.

The Scheduling Problem

This section gives a description of the scheduling problem that we solved. Due to space limitations we completely ignore the possibility of asynchronous communication. Asynchronousness refers to all those cases where the sender's frequency is different from that of the receiver. An in-depth treatment of this topic can be found in the full version of this paper, which will be published as a technical report.

The whole problem is about off-line scheduling of a special class of multi-processor systems with only a single data bus. For the present purpose, it suffices to treat a *multi-processor system* just as a finite number of processors—the data bus will be considered later on. Allocated to each processor there is a finite number of different *application processes* (or *processes* for short). Each process is allocated to exactly one processor, called the *host* of the process. A process can be executed on its host only. It may, of course, be executed more than once. A particular execution of a process, P, is denoted by P_i, for an index $i \geq 1$. Every such process execution has a non-negative *starting time*, $start(P_i)$. The duration of different executions of a particular process is always the same. Thus each particular process P has allocated to it a specific non-negative number $dur(P)$, its *execution time*. The execution times of two processes may, of course, differ. The *completion time* of an execution, P_i, is then uniquely determined by a simple calculation:

$$compl(P_i) \quad = \quad start(P_i) + dur(P). \qquad (1)$$

We thereby implicitly assume that no execution of a process may ever be preempted.

A specific process, P, can either be *periodic* or *aperiodic*. If it is periodic, it must be invoked with a certain frequency.

The reciprocal value of this frequency is called the *period* of P, or simply $period(P)$. A frequency of 10Hz, for instance, results in a period of exactly 0.1 seconds. The distance in time between the starting points of two consecutive executions of a particular process must always agree with the period of that process. Therefore, we have the following isochronousness condition:

$$start(P_i) = start(P_{i-1}) + period(P), \text{ for } i \geq 2. \qquad (2)$$

This condition, however, does make sense only when

$$period(P) \quad \geq \quad dur(P). \qquad (3)$$

In practice, the period of a process is not only expected to be always greater than its execution time, but it is even much more greater.

We assume that the overall control of a multi-processor system is organized *a priori* as a fixed time window. It is exactly this time window which, when invoked indefinitely, will deterministically control the behavior of the overall system. This time window is called the *repetition window*. The length of the repetition window is called the *cycle time* CT. This length should, of course, be chosen such that every periodic process fits into the repetition window without running out of phase. Not only the isochronousness condition (2) has to be met *within* the repetition window itself, but the unraveled version of the repetition window must obey a similar condition as well (see Fig. 2). There is a simple way to accomplish this: just identify the cycle time with the least common multiple of all periods used. This is not an unusual approach to off-line scheduling of periodic processes, see e.g. [5]. The approach works as follows. A periodic process, P, has to be executed exactly $n = \frac{CT}{period(P)}$ times within the repetition window. Let the relevant executions be $P_1, ..., P_n$. If all these executions meet the isochronousness condition (2), then the distance in time between $start(P_1)$ and $start(P_n)$ is exactly $(n-1) \cdot period(P)$. But then, the distance in time between $start(P_n)$ within the current repetition window and $start(P_1)$ within the subsequent cycle is $CT - (n-1) \cdot period(P)$. This yields exactly $period(P)$ if $n = \frac{CT}{period(P)}$. This proves that also the unraveled version of the repetition window satisfies the isochronousness condition if only, within the repetition window itself, (2) holds and there is a number of $\frac{CT}{period(P)}$ executions of P.

An aperiodic process, P, is treated as if it had a period of $period(P) = CT$. This means that during every invocation of the repetition window an aperiodic process gets the opportunity to work exactly once.

We require all executions of a process, whether periodic or aperiodic, to be scheduled within the repetition window. This is what we have implicitly assumed so far. We thus impose the following general upper time bound:

$$compl(P_i) \quad \leq \quad CT. \qquad (4)$$

If two processes share a common host, none of their executions may overlap in time. This means that two process executions may work in parallel only if they are hosted by different processors. A single processor, however, does allow only sequential executions, all executions running on it must therefore be serialized in the following way:

$$compl(P_i) \leq start(Q_j) \text{ or } compl(Q_j) \leq start(P_i) \qquad (5)$$
$$\text{whenever } P \text{ and } Q\ (P \neq Q) \text{ have the same host.}$$

Note that for a single process serialization is already guaranteed by the isochronousness condition (2). The disjunctive constraint above, therefore, does not include the case where P and Q are identical.

Perhaps it is worthwhile noting that this type of serialization constraint is not as harmless as one might think. This is because the number of constraints (5) increases quadratically in the number of application processes.

Inter-Processor Communication

If two processes have different hosts, the only way for them to communicate with each other is to broadcast a *message* through a common data bus. There is only one such data bus available and all processors are connected to it. Two processes sharing the same host are able to communicate directly with each other.

Any message has exactly one *sender*, but may have multiple *receivers*. In each particular case, the sender is a process, just like all the receivers. If at least one of the receivers has another host than the sender, then the relevant message is an *inter-processor message*; otherwise it is an *intra-processor message*.

Associated with each message M, there is a non-negative number $dur(M)$, the *transfer time* of M. If M is an intra-processor message, then $dur(M)$ is always 0.

There is a special type of inter-processor message which has neither any sender nor a particular receiver. It is a broadcast message to adjust the local clocks of the processors, called a *resynchronization message*. For convenience, we assume a dummy sender and a dummy receiver for this special kind of message. Both the sender and the receiver can be hosted by any processor if only they are not hosted by the same processor. Their relevant execution times are set to 0. The transfer time of this special broadcast message is very short, while its frequency—that is, the frequency of its dummy sender—is very high, typically some 1.000Hz.

An actual transmission of a single message, M, is denoted by M_i, for an index $i \geq 1$. Each such transmission has a particular starting time and a completion time, $start(M_i)$ and $compl(M_i)$ for short. Similar as for process executions, the completion time of a message transmission depends on

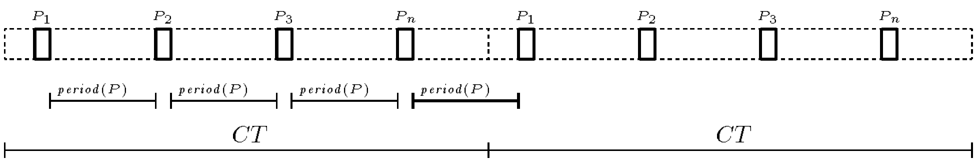

Figure 2: The wrapping over the repetition window

its starting time and the relevant transfer time:

$$compl(M_i) \quad = \quad start(M_i) + dur(M). \tag{6}$$

If S is the sender of M, then there must be a sequence of transmissions $M_1, ..., M_n$ with $n = \#S$, no matter whether it is an inter-processor or an intra-processor message. This, of course, applies to the synchronous case only. In this case, a message is always transferred with exactly the same frequency as its sender is executed. For asynchronous communications, this is not necessarily the case. In the synchronous case, however, the frequency of the message does always coincide with that of its sender:

$$start(M_i) = start(M_{i-1}) + period(S), \text{ for } i \geq 2. \tag{7}$$

Any message transmission, of course, must obey the general upper time bound, too:

$$compl(M_i) \quad \leq \quad CT. \tag{8}$$

It is not necessary for a message to be transferred *immediately* after the sender terminated; rather, a single message can be buffered as long as no fresh version of this message arives. However, what *is* necessary is that a message is never transferred before the relevant execution of the sender terminated:

$$compl(S_i) \quad \leq \quad start(M_i). \tag{9}$$

There is no similar precedence constraint between the message transmission and its actual receiver. That is to say, if R_j is one of the actual receivers of M_i, then no constraint of the form $compl(M_i) \leq start(R_j)$ is imposed. This is because R_j may be postponed until the subsequent invocation of the repetition window, in which case R_j would typically occur at the beginning of the repetition window. In a cyclic communication pattern this is even necessarily so. Take the particular cyclic communication pattern depicted in Fig. 3. Here, P sends a message to Q, which in turn sends a second message back to P. If the two processes have the same frequency, then the actual receiver of the second message *cannot* be scheduled after the message transmission, at least within the current cycle of the repetition window. In this case, P would occur twice within the repetition window and, therefore, would have twice the frequency of Q. This is why the second message will be received only within the subsequent cycle of the repetition window. As a matter of fact, there is no precedence relation between

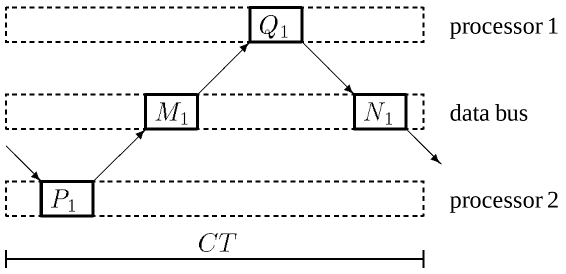

Figure 3: Postponing the receiver of a message

a message transmission and potential receivers. They may even overlap in time, in which case the *actual* receivers are again postponed until the subsequent repetition cycle.

What *is* required is that a transmission of a message must be scheduled before the next execution of its sender (if there is any):

$$compl(M_i) \quad \leq \quad start(S_{i+1}).$$

However, there is no need to impose this condition explicitly. This is because it already follows from (7)–(9), together with the fact that there are always as many transmissions of M as there are executions of S.

The data bus can always transfer only one message at a time. Thus all transmissions through the data bus have to be serialized:

$$compl(M_i) \leq start(N_j) \text{ or } compl(N_j) \leq start(M_i) \tag{10}$$
$$(M \text{ and } N \ (M \neq N) \text{ are inter-processor messages})$$

For intra-process messages, it is not necessary to explicitly rule out any mutual overlapping in time. This is because their transfer time is always 0.

Latencies

For a real-time system it is important that critical information is guaranteed to be processed within certain time limits. If a process sends a message to a second process, then it may be important to guarantee that the overall information processing never exceeds a certain period of time. In particular, it is convenient to specify an upper bound for the admissible period of time which may pass between the generation of the information by the sender and its full processing by the receiver, with the actual exchange of the information in

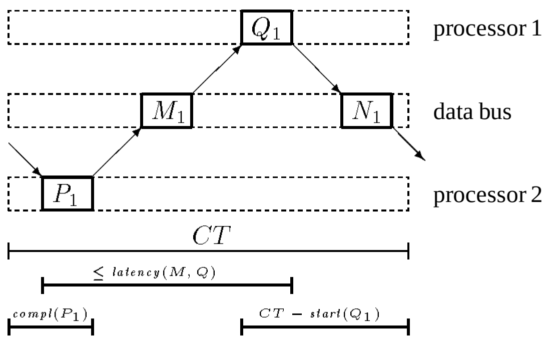

Figure 4: Latencies

between. We call an upper bound on exactly this period of time a *latency*. The specific value of a latency depends not only on the message itself, but also on the particular receiver of that message. This is because the receivers might differ in their criticality as well. The particular value of such a latency is henceforth denoted by $latency(M, R)$, where R is one of the receivers of M. If S is the sender of that message, we have the following relative timing constraint:

$$compl(R_i) - start(S_i) \quad \leq \quad latency(M, R).$$

Fig. 4 shows a sample latency, constraining the admissible distance in time between $start(P_1)$ and $compl(Q_1)$.

Notice that this type of latency condition presupposes that the ith transmission of M is actually received by the ith execution of R. This can only be guaranteed for a synchronous communication, for which the frequency of the sender does not differ from that of the receiver.

The latency condition above, however, does not cover all relevant situations, even with only the synchronous case taken into consideration. To see this, consider again Fig. 4. Not only does P send the message M to Q, but Q sends also N back to P. The actual receiver of M_1 is Q_1. Here, the latency constraint form above works perfectly well. The actual receiver of N_1 is P_1. P_1 does not occur after N_1 because the actual receiver of N_1 is postponed until the subsequent repetition cycle, a case which is not covered by the latency condition above. In this case, a proper latency condition should be that $CT - start(Q_1) + compl(P_1)$ must not exceed $latency(N, Q)$ (see also Fig. 4). In the general case, a proper latency condition has to take into account both cases. This looks as follows. Let S be the sender of a message M. For every receiver, R, of this message, we have:

$$\begin{array}{ll} \text{If} & start(R_i) \geq compl(M_i) \\ \text{then} & compl(R_i) - start(S_i) \leq latency(M, R) \\ \text{else} & CT - start(S_i) + compl(R_1) \leq latency(M, R). \end{array} \tag{11}$$

Such a latency constraint is, of course, only imposed if the transferred information is critical.

These are all the constraints we are considering. Now, suppose we are given a particular multi-processor system. The *scheduling problem* then consists in finding for each single process execution and message transmission a particular starting time such that the constraints introduced above are simultaneously satisfied. In a time-triggered architecture, the time scale is a discrete one, dictated by the recurring global clock tick. The frequency of the global clock coincides with that of the common data bus.

Constraint Programming

The problem under investigation was solved with the help of finite-domain constraints. The specific constraint language we used is Oz [18].

A *finite-domain constraint* is a first-order formula over a number of variables, each of which may take a specific value out of a finite set of non-negative integers, called a *finite domain*. In Oz, a distinction is made between constraints which are *basic* and those which are not. Only basic constraints may reside in the global *constraint store*. This is to guarantee that the satisfiability of the entire constraint store can always be decided efficiently, just like the question whether an arbitrary basic constraint is entailed by the current store or not [16]. The simplest form of a basic constraint is $x \in D$, where D can be an arbitrary finite domain. A basic constraint of this type is also referred to as a *domain constraint*. In this paper, we consider only two other types of basic constraints. These are of the form $x = y$ and $x = n$, where n can be an arbitrary non-negative integer.

As is well-known, deciding the satisfiability of nonbasic constraints such as $x + y = z$ is computationally intractable (because graph-coloring problems can be encoded, for example). This is why in Oz such nonbasic constraints do not reside in the global constraint store. Instead, a nonbasic constraint is imposed by what is called a *propagator*. This propagator implements the effects of the relevant constraint. It is a computational agent that tries to narrow down the domains of variables by adding appropriate basic constraints to the constraint store $\mathcal{S}$. Take a propagator for a nonbasic constraint, α. This propagator may impose a basic constraint not yet entailed by $\mathcal{S}$, say β, if β is entailed by $\mathcal{S} \cup \{\alpha\}$. In addition, of course, $\mathcal{S} \cup \{\beta\}$ has to be satisfiable. If, for example, $\mathcal{S}$ consists of the basic constraints $x \in \{1, ..., 10\}$, $y \in \{1, ..., 10\}$ and $z \in \{1, ..., 10\}$, then a propagator for $x + y = z$ might add $x \in \{1, ..., 9\}$, $y \in \{1, ..., 9\}$, as well as $z \in \{2, ..., 10\}$ to the current constraint store. The term *constraint propagation* refers to advancing the constraint store in this manner. A propagator for a constraint, α, signals inconsistency as soon as it is realized that α is inconsistent with the constraint store.

A *finite-domain problem* is a finite set of finite-domain constraints such that for each variable occurring in a constraint there exists at least one domain constraint. A *solution* to a finite-domain problem is a specific mapping which assigns a single non-negative integer to each variable such that all constraints are satisfied simultaneously.

Constraint propagation alone is often not sufficient for determining a solution. A solution for a finite-domain problem, Θ, can be obtained, anyway, if a list of (not necessarily basic) constraints $\alpha_1, \ldots, \alpha_n$ is chosen. Instead of solving the original problem, it is then tried to solve at least one of the stronger problems $\Theta \cup \{\alpha_1\}, \ldots, \Theta \cup \{\alpha_n\}$. In this case, we say that *we branch with* α_i in Θ. Examples for α_i might be $x = 1$ and $x > 1$. Such a modification of the original problem is perfectly admissible, in that a solution for $\Theta \cup \{\alpha_i\}$ is always a solution for Θ as well. The sequence of problems $\Theta \cup \{\alpha_i\}$ can simply be examined one after the other, so that $\Theta \cup \{\alpha_i\}$ is touched only if all preceding alternatives lead to an inconsistency. In general, the choice of the constraints to branch with, including the specific ordering in which they are examined, is called a *branching strategy*. To maximize the available information for any strategy, branching does not take place before constraint propagation has reached a fixed point. Solving a constraint problem thus consists in a sequence of interleaving propagation and branching steps.

In Oz, a propagator for a constraint α ceases to exist as soon as it is realized that α is entailed by the constraint store. The propagator is then automatically garbage collected by the system itself. This will turn out to be an important feature for being able to cope with the present problem.

A Solution to the Problem

This section explains how constraint programming can be used to solve the scheduling problem at hand. Such a solution always has two parts to it. The starting point is to recast the relevant problem in terms of formal constraints. This is not too difficult in the present case, but we have to consider efficiency and memory consumption. The next step is to choose a particular branching strategy, which *is* difficult in the present case. We shall not only describe the branching strategy which ultimately led to success, but also mention some of those that failed. It is difficult to say whether these results are specific to the actual constraint language that we used. Anyway, we hope that the results will give other programmers some helpful hints how to tackle similar problems.

The Constraints

The starting time of a process execution, P_i, can well be captured by a finite-domain variable and so can the starting time of a message transmission, M_j. For convenience, these variables are also denoted by $start(P_i)$ and $start(M_j)$ respectively. Because the cycle time CT is typically only a fraction of a second, the initial domains should be chosen carefully. The smallest relevant time unit is the reciprocal of the frequency ρ of the global clock. This is exactly the general time unit that we use. The starting times are then initialized as follows:

$$\begin{aligned} start(P_i) &\in \{0, \ldots, CT - dur(P)\}, \\ start(M_j) &\in \{0, \ldots, CT - dur(M)\}. \end{aligned}$$

The completion times are not introduced explicitly; rather, terms of the form $start(P_i) + dur(P)$ or $start(M_j) + dur(M)$ are used to replace them. This is to reduce the overall number of finite-domain variables.

The specific domains that we associated with $start(P_i)$ and $start(M_j)$ clearly guarantee that neither $compl(P_i)$ nor $compl(M_j)$ may ever exceed CT, thus satisfying constraints (4) and (8). The remaining constraints of the second section then form ordinary finite-domain constraints. The only exceptions are latency and serialization constraints.

Serialization Constraints

The constraint (5), pairwise serializing all process executions on a specific host, is a *disjunctive constraint*. It involves two possible alternatives. Of exactly the same nature is (10), serializing the communication through the data bus. Constraints of this sort can be handled, anyway, if the constraint language allows to reflect the truth value of a constraint into a 0/1 variable, see e.g. [15, 11, 10]. In Oz this kind of reflection is called *reification*. Take (5). A proper reification is as follows. First define two 0/1 variables, never taking the value 1 at the same time: $B_1 \in \{0, 1\}$, $B_2 \in \{0, 1\}$, and $B_1 + B_2 = 1$. Then state the following nested constraint.

$$\begin{aligned} (B_1 = 1) &\leftrightarrow \big(start(P_i) + dur(P) \leq start(Q_j)\big), \\ (B_2 = 1) &\leftrightarrow \big(start(Q_j) + dur(Q) \leq start(P_i)\big). \end{aligned}$$

This reification captures exactly the meaning of (5). To see this, assume that one of the alternatives, say, $start(P_i) + dur(P) \leq start(Q_j)$ becomes inconsistent with the current constraint store. B_1 is then set to 0 and, therefore, B_2 to 1. But then, the second alternative is immediately invoked.

In Oz this reification can also be phrased somewhat more succinctly:

$$\begin{aligned} \big(start(P_i) + dur(P) \leq start(Q_j)\big) &\quad + \\ \big(start(Q_j) + dur(Q) \leq start(P_i)\big) &\quad = \quad 1 \end{aligned}$$

But even with this succinct form used, we need altogether $\frac{n \cdot (n-1)}{2}$ reified constraints to serialize n process executions, and similar for message transmissions. Such a naive approach works well as long as n is comparatively small. With a number of about 1,800 message transmissions, a number we were faced with, this is no longer feasible. It is true that, in the course of the computation, many of these reified constraints can eventually be garbage collected; a large number of them will nevertheless exist for quite a long time.

Such an excessive memory consumption can be avoided by a special type of constraint, reasoning on a large set of variables simultaneously. A constraint of this kind is called *global*. In some cases, global constraints do not only save memory, but improve also the overall run-time. This might even happen when a global constraint does nothing more than mimicking the propagation of a host of non-global constraints like the reified ones from above. Instead of maintaining $O(n^2)$ propagators only a single (global) propagator needs to be maintained by the underlying runtime system. We dealt with the constraints (5) and (10) in exactly this way. Given an arbitrary number of items, the relevant global propagator mimics the pairwise serialization of these items, without the full computational burden of the original pairwise serialization. The global propagator essentially considers successively each pair of items and checks whether any clause of the corresponding disjunction is disentailed. In this case it imposes the right basic constraints implied by the remaining clause. The propagator was created with the help of a special interface, the so-called constraint propagator interface of Oz [14] and is available as `FD.schedule.serializedDisj`. This interface makes it possible to create new propagators efficiently.

This serialization propagator does not employ any domain-specific information other than that of a linear time scale. Global constraints, however, may also employ more elaborate propagation techniques for domain-specific purposes. Applegate and Cook [1] proposed such a technique, called *edge finding*, see also [3]. Variants of this technique enjoy a run-time which essentially grows quadratically in the number of items to be serialized. In computational terms, even these more efficient variants are yet more expensive than the global constraint described above. For certain problems, however, edge finding may lead to a drastic reduction of the search space, outweighing the additional computational costs. No such effect can be observed for the present problem. This is due to the fact that the domains of the starting time variables are rather huge in comparison to the item's duration. But one premise of edge finding to enable strong reasoning are narrow domains. While the data bus is the real bottleneck of the current scheduling problem (see also below), this is not mirrored in the domains of the variables immediately. Compared with the global constraints that we employed, edge finding does not shorten the search for a solution at all. Edge finding even increased the overall run-time by a factor of five.

Latency Constraints

The latency condition (11) distinguishes between communication patterns which are wrapped over the repetition window and those which are not. These two possibilities might be captured with the help of a reified constraint, too; however, this would lead to weak constraint propagation. An experiment showed that such an approach does not allow to solve the present problem in one day of computation time. Propagation, however, can be improved by a specific preprocessing phase. If, for example, P_i sends a message to Q_j, it can be determined before whether Q_j is to be postponed until the subsequent repetition window or not. This works as follows. If the relevant communication is not involved in a cycle, then it is assumed that Q_j is not postponed. In this case, $start(Q_j) \geq compl(P_i)$ is imposed. As a consequence, the following version of a latency condition is sufficient.

$$start(Q_j) + dur(Q) - start(P_i) \quad \leq \quad latency(M, Q).$$

If, on the other hand, the communication between P_i and Q_j is part of a cycle, then an ordering between P_i and Q_j can be chosen arbitrarily. If $start(Q_j) \geq compl(P_i)$ is chosen, then again the latency condition just given is sufficient. If, otherwise, $start(P_i) \geq compl(Q_j)$ is chosen, then the following condition must be met:

$$CT - start(P_i) + start(Q_j) + dur(Q) \leq latency(M, Q).$$

In this way, we can dispense with the disjunction of (11), at least for all those processes which are not involved in a cyclic communication pattern. This guarantees adequate propagation. In principle, it might happen that this preprocessing phase cuts off the only admissible solution to the problem. This, however, seems to be very unlikely. At least it never happened in any of our experiments.

A Branching Strategy from Real-Time Applications

Branching strategies off the shelf (like first-fail) turned out to be too simple for the present problem. Branching strategies can be divided into two categories. The first class of strategies branches with basic constraints such as $x = c$. The second type of strategies branches with non-basic constraints such as $x + c \leq y$. In this case, new propagators are created during run time. We experimented with strategies from either category.

The first branching strategy that we investigated was motivated by Liu and Layland's [13] work on scheduling algorithms for hard real-time applications. This strategy works as follows. The finite domain which models the starting time of a process execution or a message transmission denotes the set of possible integer values for the starting time. Let $lct(P_i)$ and $lct(M_i)$ be the latest possible completion time of P_i and M_i, respectively. The value $lct(P_i)$, for example, is thus the maximal value of the domain of P_i plus the execution time of P_i. Exactly this pair (P_i, M_i) is then selected for which $(lct(P_i), lct(M_i))$ is minimal with respect to lexicographic order. In addition, a pair may be selected only if either $start(P_i)$ or $start(M_i)$ is not already fixed to a single value. If $start(P_i)$ stays unfixed, it is branched with basic constraints of the form $start(P_i) = c$, beginning the enumeration of possible candidates for c with the smallest possible starting time of P_i. As soon as a consistent value for $start(P_i)$ is found, the starting time of M_i is fixed by a similar procedure. As a result, the process with minimal latest possible completion time (usually a process with a high frequency) determines which starting time is fixed first. While this strategy works well if no latency constraints are considered, no solution is obtained in one day of computation time with latencies included.

Branching Strategies from Operations Research

As an alternative, we investigated several strategies of the second category. These strategies were motivated by similar strategies from Operations Research, which have recently been used also in the field of constraint programming [2, 4]. Let us first explain what the key idea of this strategy is. Consider a set, S, of items. The task is to schedule all elements of S in a mutually non-overlapping manner. The branching strategy is divided into two phases, the determination of an appropriate serialization and the assignment of concrete starting times. *Serialization* in this case means that for all $s, s' \in S$ such that $s \neq s'$, it has to be determined whether s is scheduled before s' or vice versa. An appropriate total ordering of S can be obtained by branching with one of the following propagators at run-time:

$$start(s) + dur(s) \leq start(s') \\ \text{or} \\ start(s') + dur(s') \leq start(s).$$

In the best case, a number of $\frac{|S| \cdot |S-1|}{2}$ branching steps is needed to serialize S completely. This number may be reduced if some ordering decisions can be decided deterministically. For example, the strategy proposed in [4] avoids some branching steps by detecting orderings which must necessarily hold. The assignment of concrete starting times could then be done with a branching strategy of the first category. But a number of branching steps which can grow quadratically in S might be, of course, not feasible for a large-scale problem. Nevertheless, we have tried the strategies suggested in [4] and [17]. These strategies employ one pairwise ordering decision at each branching step and have been shown to yield good results in the area of job-shop scheduling [7]. Unfortunately, both strategies failed even after several hours of computation time for a small test problem. This means that the strategies using local pairwise ordering decisions are too weak for the considered problem. Both strategies [4, 17] consider the size of the domains of starting time variables to make branching decisions. But because these domains are rather wide compared to the item's duration (especially in the beginning of the search), this heuristic might guide the search in a wrong direction. Due to the size of the considered problems these wrong decisions cannot be recovered by simple backtracking.

Instead of ordering only a single pair of items at a time, many pairs can be ordered in a single branching step. The number of branching steps can thus be significantly reduced by considering a whole number of propagators at a time. This proceeds as follows. First, select an item $s \in S$. Then it is branched with a number of constraints, altogether stating that s precedes each item in S, except for s itself:

$$start(s) + dur(s) \leq start(s'),$$

for all $s' \in S$ such that $s' \neq s$. If this leads to an inconsistency, then the entire procedure is retried with another $s \in S$; otherwise s is deleted from S. As soon as the subsequent constraint propagation has reached a fixed point, the overall procedure is invoked again, a process which continues until all items are serialized. In the best case, no more than $|S| - 1$ branching steps are needed to serialize S completely. Of course, we still have a quadratic number of pairwise ordering decisions, but the depth of the search tree may be reduced.

The question remains which of the potential candidates for $s \in S$ should be considered first. For this purpose, information on the possible starting times can be exploited. In particular, it is possible to extract those items among S which may precede all others, see e.g. [3, 2, 20]. This subset of S can be computed in time $O(|S|)$. Let F be this set of items which may precede all others in S. If F is empty, there clearly exists no serialization at all. If $|F| = 1$, no branching step is necessary because F's single element is the only candidate for s. If otherwise $|F| > 1$, those elements of F are tried first whose earliest possible starting time is minimal. This guarantees that for the remaining items there is as much free space on the time scale left as possible. This strategy is available in Oz as `FD.schedule.firstsDist`.

Order of Branchings

Now we have to decide which resource should be serialized first. It is common practice to schedule the resource first which can be seen as the bottleneck of the problem. As an obvious criterion we have chosen first the utilization (load) values. Because the utilization of the processors is up to 92%, whereas the utilization of the data bus is only about 11%, we have tried to serialize the processes first. Surprisingly, we could only solve a few problems (only two out of seven). Analyzing the problem more carefully, we found out that the primary source of complexity are the different periods at which processes may be executed. What makes the scheduling of the data bus so hard is the fact that it is the only resource commonly used by all inter-processor communications. Through the common data bus processes on different processors with different periods interact in a non-trivial way. Thus, the kind of branching described before is first applied to the messages sent through the data bus. The serialization of the messages automatically narrows down the possible starting times of the sending processes. The process executions are then serialized by the same strategy. Currently it is not necessary to impose a certain order which processor to serialize first.

Assignment of Concrete Starting Times

The second phase is the assignment of concrete starting times. It turns out that a valid schedule can be obtained just by simply assigning the starting times to the *minimal* value of those still remaining. This means that no further search is needed. This observation can be gathered from a similar theorem of Van Hentenryck [8]. This theorem essentially states that mere constraint propagation is indeed sufficient for detecting any inconsistency if there are only constraints of the form $x + c \leq y$ and $x + c = y$ such that c is an arbitrary integer. The theorem moreover states that a solution can be obtained in a way very similar to the one described in this section. Van Hentenryck's theorem, however, does not include disjunctive constraints of the shape $x + c_1 \leq y \vee y + c_2 \leq x$. On the other hand, the kind of serialization that we described above leaves only one of the two disjuncts of each disjunctive constraint, so that Van Hentenryck's theorem still applies.

Empirical Results

The constraint program described has been applied to several large-scale problems of the Daimler-Benz holding. Currently, we have the data of seven such problems available. A typical problem of this particular class looks as follows. The frequency of the global clock leads to possible starting times ranging from 0 to 6 million. There are 20 processors, hosting altogether about 170 processes. Taking the executions of periodic processes into account, there are about 350 process executions to be scheduled. The process executions transfer a number of about 1,200 inter-processor messages. With additional 600 resynchronization messages, there are about 1,800 messages to be transferred through the common data bus. This is also the largest number of arguments with which the global serialization constraint is invoked. Among these 1,800 message transmissions there are 1,200 that are transferred with a frequency higher than that of the repetition window. The overall utilization of the data bus is about 11%, the maximal utilization of a processor is about 92%.

An appropriate program is implemented in Oz [14]. Running the program on a Pentium Pro 200 MHz will bring about a first solution within 10 minutes with 7.5 MB active data on the heap. In the course of the computation, more than 1.6 million propagators are created, thereof about 23,000 before the first branching step takes place. The number of backtracking steps ranges from 0 to 40. The overall number of branching steps is about 2,200 in the best case. The number of propagators created is conformable to the reported memory consumption. Many of those propagators which are imposed during the branching are entailed very early and, thus, are garbage collected. It is important that the entailment detection is not delayed until the constrained variables are determined. This would be insufficient because of the branching strategy used (only after all process executions and the messages are serialized the starting times are fixed).

Recomputation was used to keep the memory consumption as low as possible. This means that whenever an inconsistency occurs, previous computation states are reconstructed by recomputation. Copying (trailing in other systems) of earlier computation states is thereby avoided [9].

Conclusions

We presented a new potential application domain for constraint programming with industrial strength. The very heart of this class of problems was described in full detail. We did so in order to make this new class of applications amenable to constraint programming.

With the help of the concurrent constraint language Oz, we were able to solve several large-scale problems from this class. The specific problems that we solved involved finite domains ranging from 0 to 6 million and more than 1,800 message transmissions to be scheduled in a mutually non-overlapping manner. Up to 3.9 million propagators were created in the course of computing a valid schedule.

We also described which techniques ultimately led to success and which failed. Global constraints and an elaborate search heuristic, borrowed from Operations Research, turned out to be a pre-requisite for any successful solution. More sophisticated global constraints like edge-finding were not able to improve the scheduling. This might be due to a quite small load of the data bus, leaving the relevant finite domains rather unconstrained.

The data bus yet turned out to be the bottleneck of the problem. This is certainly *not* what one would expect of a typically 11% load of the data bus and a processor load of up to 92%. What makes the scheduling of the data bus so hard is the fact that it is the only resource commonly used by all inter-processor communications. None of the common branching strategies was capable of cracking this bottleneck. We experimented with one well-known strategy from real-time applications and several others from constraint-based scheduling. One strategy was the serialization of a number of items by considering one pair after the other. What ultimately led to success was a new strategy, minimizing the overall number of branching steps. This special branching strategy was implemented with the help of the constraint propagator interface of Oz, exactly as the global constraint for serializing an arbitrary number of items.

Another important observation was that after a complete serialization, no further search is needed in order to assign concrete starting times. It actually suffices to take in each case just the smallest value among those still remaining. This is due to a special characteristic of the problem under consideration. Another observation was that excessive memory consumption can be avoided by garbage collecting redundant propagators as early as possible, a characteristic that Oz enjoys.

Currently we are validating our approach by tackling even more complicated problems with up to 3,000 message transmissions and about 500 process executions. These problems can also be solved in the presented framework.

Acknowledgements

The authors would like to thank Denys Duchier and the anonymous referees for their invaluable comments on a draft version of this paper. The second author was partly supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FKZ ITW 9601), and the Esprit Working Group CCL-II (EP 22457).

References

[1] APPLEGATE, D., AND COOK, W. A computational study of the job-shop scheduling problem. *Operations Research Society of America, Journal on Computing 3*, 2 (1991), 149–156.

[2] BAPTISTE, P., PAPE, C. L., AND NUIJTEN, W. Constraint-based optimization and approximation for job-shop scheduling. In *Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems* (1995).

[3] CARLIER, J., AND PINSON, E. An algorithm for solving the job-shop problem. *Management Science 35*, 2 (1989), 164–176.

[4] CASEAU, Y., AND LABURTHE, F. Disjunctive scheduling with task intervals. LIENS Technical Report 95–25, Laboratoire d'Informatique de l'Ecole Normale Superieure, Paris, France, 1995.

[5] CHENG, S., AND AGRAWALA, A. Allocation and scheduling of real-time periodic tasks with realative timing constraints. In *Proceedings of the Second International Workshop on Real-Time Computing Systems and Applications* (1995).

[6] ERIKSSON, C. *A framework for the design of distributed real-time systems*. ISRN KTH/MMK/R–97/2-SE, Royal Institute of Technology, KTH, Stockholm, Sweden, 1997.

[7] GAREY, M., AND JOHNSON, D. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[8] HENTENRYCK, P. V., AND DEVILLE, Y. Operational semantics of constraint logic programming over finite domains. In *Proceedings of the AAAI Spring Symposium Series* (1991), pp. 128–146.

[9] HENZ, M., MÜLLER, M., SCHULTE, C., AND WÜRTZ, J. The Oz standard modules. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1997.

[10] HENZ, M., AND WÜRTZ, J. Using Oz for college timetabling. In *Proceedings of the First International Conference on Practice and Theory of Automated Timetabling* (1996), vol. 1153 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 162–178.

[11] ILOG. ILOG SOLVER *3.2, User Manual*. URL: `http://www.ilog.com`, 1996.

[12] KOPETZ, H. Event-triggered versus time-triggered real-time systems. In *Proceedings of International Workshop on Operating Systems of the 90s and Beyond*, vol. 563 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1991, pp. 87–101.

[13] LIU, C., AND LAYLAND, J. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (1973), 46–61.

[14] MÜLLER, T., AND WÜRTZ, J. The constraint propagator interface of DFKI Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1997.

[15] OLDER, W., AND BENHAMOU, F. Programming in CLP(BNR). In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming* (1993), pp. 239–249.

[16] SCHULTE, C., SMOLKA, G., AND WÜRTZ, J. Encapsulated search and constraint programming in Oz. In *Proceedings of Principles and Practice of Constraint Programming* (1994), vol. 874 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 134–150.

[17] SMITH, S., AND CHENG, C.-C. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the 11th National Conference of the American Association for Artificial Intelligence* (1993), pp. 139–144.

[18] SMOLKA, G. The Oz programming model. In *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, 1995, pp. 324–343.

[19] WALLACE, M. Practical applications of constraint programming. *Constraints 1*, 1&2 (1996), 139–168.

[20] WÜRTZ, J. Oz Scheduler: A workbench for scheduling problems. In *Proceedings of the Eighth International Conference on Tools with Artificial Intelligence* (1996), pp. 149–156.