CROSS cyclic resource-constrained scheduling solver

Alessio Bonfietti *, Michele Lombardi, Luca Benini, Michela Milano

Università di Bologna, Viale Risorgimento, 2, 40136 Bologna, Italy

Abstract

Cyclic scheduling problems consist in ordering a set of activities executed indefinitely over time in a periodic fashion, subject to precedence and resource constraints. This class of problems has many applications in manufacturing, embedded systems and compiler design, production and chemical systems. This paper proposes a Constraint Programming approach for cyclic scheduling problems, based on modular arithmetic: in particular, we introduce a modular precedence constraint and a global cumulative constraint along with their filtering algorithms. We discuss two possible formulations. The first one (referred to as CROSS) models a pure cyclic scheduling problem and makes use of both our novel constraints. The second formulation (referred to as CROSS*) introduces a restrictive assumption to enable the use of classical resources constraints, but may incur a loss of solution quality. Many traditional approaches to cyclic scheduling operate by fixing the period value and then solving a linear problem in a generate-and-test fashion. Conversely, our technique is based on a non-linear model and tackles the problem as a whole: the period value is inferred from the scheduling decisions. Our approach has been tested on a number of non-trivial synthetic instances and on a set of realistic industrial instances. The method proved to effective in finding high quality solutions in a very short amount of time.

1. Introduction

Cyclic scheduling problems arise in a number of application areas, such as in hoist scheduling [10], mass production [19, 13], flow shop [30,31], cyclic job-shop [8,9], compiler design (optimizing loops on parallel architectures) [27,19], software pipelining [35], and data-flow computing [22]. Optimal cyclic schedulers are lately in great demand, as stream computing paradigms are gaining momentum across a wide spectrum of computing platforms, ranging from multi-media encoders and decoders in mobile and consumer devices, to advanced packet processing in network appliances, and high-quality rendering in game consoles. In stream computing, an application can be abstracted as a set of tasks that have to be performed on incoming items (called coding units, packets, pixels, depending on the context) of a data stream. A typical example is video decoding, where a compressed video stream has to be expanded and rendered. Since video compression exploits temporal correlation between successive frames, decoding is not a pure process-and-forward activity (because the computation on the current frame depends on the previously decoded frame). These dependencies must be taken into account in the scheduling model. In embedded computing contexts, resource constraints (computational units and buffer storage) imposed by the underlying hardware platforms are of great importance. In addition, the computational effort which can be spent to compute an optimal schedule is often limited by cost and time-to-market considerations.

From a combinatorial optimization standpoint, cyclic scheduling is the problem of assigning start times to a set of activities (indefinitely repeated over time) such that the average repetition time for the overall set is minimal, i.e. the execution frequency is maximal. From a practical point of view, one is typically interested in finding a periodic schedule, where the same pattern of start times is repeated every \(\lambda\) time units. The value \(\lambda\) is the schedule period.

* Corresponding author.

0004-3702/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.artint.2013.09.006
Traditional constraint-based scheduling techniques have achieved a good level of maturity in the last decade [3]. They can be applied to cyclic scheduling problems, but the adaptation techniques defined so far have failed to produce state-of-the-art results. In particular, the related literature is focused on two approaches:

- The blocked scheduling approach [5] builds a schedule for a single repetition of the set of activities. The schedule is then repeated after the last activity is over, so that consecutive repetitions do not overlap. As a side effect, however, blocked scheduling may fail to make optimal use of the available resources, with possibly large decrease of solution quality.

- The unfolding approach [33] schedules a number of consecutive repetitions of the application. The schedule is then repeated after the last activity is over, like in blocked scheduling. Unfolding allows to exploit inter-repetition overlaps, but at the cost of an increased instance size. Moreover, determining the right number of repetitions is a non-trivial problem.

On the other hand, a number of approaches investigate the so-called modulo scheduling. This technique builds a schedule for a single repetition, which is repeated every \( \lambda \) time units, where \( \lambda \) is called modulus and is in fact the schedule period. Hence, the modulo scheduling approach builds a periodic schedule, with no loss of optimality. Existing modulo scheduling techniques operate by fixing the modulus and searching for a feasible schedule. In this process, inter-repetition overlaps are taken into account, so that dealing with resource constraint becomes more complicated. The smallest modulus with a feasible schedule is the optimal solution.

In this paper we introduce a novel Constraint Programming approach based on modular arithmetic to compute resource-constrained cyclic schedules with minimum period. In our approach, the modulus is a decision variable, linked via global constraints to the other decision variables in the model (i.e. the activity start times). As a consequence, we gain efficiency with respect to other modulo scheduling approaches and solution quality with respect to the blocked and the unfolding approaches. To obtain these gains however, we cannot reuse constraint based techniques, but we have to define extensions to cope with modularity and periodic schedules.

In particular, we have developed novel constraints to model temporal and resource restrictions in a cyclic context. Namely, we have introduced the Modular Precedence Constraint (MPC) and the Global Cyclic Cumulative Constraint (GCCC), and we have devised efficient filtering algorithms for both of them. The new constraints are the foundation for the constraint model employed within CROSS (Cyclic Resource Constrained Scheduling Solver). The solver makes use of a novel random restart search strategy, where we set the upper bound of the modulus variable while the lower bound is inferred from the scheduling decisions. This is in contrast with classical modular approaches that fix the modulus and solve the corresponding scheduling sub-problem.

As another contribution, we have proposed a restricted approach (called CROSS\(^*\)), where we simplify the model by using traditional resource constraints instead of the GCCC. This comes at a cost of a limitation: namely, we make the assumption that the end times of all the activities should be assigned within the modulus. As a consequence, the approach may lose optimality.

We have performed an extensive experimental evaluation on a number of non-trivial synthetic instances and on a set of industrial instances. We have compared our approach with a state-of-the-art ILP (Integer Linear Programming) scheduler and the Swing Modulo Scheduling heuristic technique (SMS). SMS is a non-constraint-based (heuristic) modular approach adopted in the gcc compiler [18]. CROSS (and its restricted version CROSS\(^*\)) proved to be able to deal effectively with temporal and resource constraints and to compute very high quality solutions in a short time. The experiments also show that, for some instance classes, our technique outperforms both the blocked and the unfolding approaches in terms of search speed and solution quality.

The paper is structured as follows: in Section 2 we formally define the considered problem, while in Section 3 we describe all related works highlighting the differences w.r.t. our approach. Section 4 formally describes the modular formulation adopted in this work. In Sections 5 and 6 we respectively describe the model with the novel constraints and their filtering algorithms. Section 7 discusses two search strategies and the restricted version of the solver, and formally defines a dominance rule used to narrow the search space. Experimental results are reported in Section 8 and Section 9 concludes the paper.

2. The problem

The problem we consider is the Cyclic Resource-Constrained Scheduling Problem (CRCSP). As a main difference with traditional scheduling, the set of activities is repeated indefinitely (in practice, a very large number of times). In the following, we formally define the concepts needed to address the CRCSP and we provide some intuitions. The problem is defined over a Project Graph:

**Definition 1.** A Project Graph \( \mathcal{G} \) is a directed (possibly cyclic) graph consisting of a pair \( \langle \mathcal{V}, \mathcal{A} \rangle \), where:

- elements in \( \mathcal{V} (\| \mathcal{V} \|= n) \) are nodes that represent activities;
- elements in \( \mathcal{A} (\| \mathcal{A} \|= m) \) are arcs (formally described in Section 2.2). An arc \( (i, j) \) represents a temporal dependency between activities \( i \) and \( j \).
We assume that every activity \( i \in \mathcal{V} \) in the graph \( G \) has a fixed duration \( d_i \). We refer to \((i, \omega)\) as the \( \omega \)-th execution of activity \( i \in \mathcal{V} \), where \( \omega \in \mathbb{Z} \) is called execution number. A set of executions \((i, \omega)\) of all the activities in \( \mathcal{V} \) with the same \( \omega \) value is referred to as a repetition. A schedule is defined as an assignment of start times to all executions \((i, \omega)\). We refer to start\((i, \omega)\) as the starting time of activity \( i \) at execution \( \omega \). Without loss of generality, we also assume that start\((i, \omega) \geq \) start\((i, \omega')\) if \( \omega \geq \omega' \) and that start\((i, \omega) \geq 0\).

The most important performance metric of a cyclic schedule is the average inter-execution distance, which is strictly related to the concept of execution frequency (i.e. the throughput).

**Definition 2.** The average inter-execution distance \( \lambda(i) \) of an activity \( i \) is defined as the following limit:

\[
\lambda(i) = \lim_{\omega \to \infty} \frac{\sum_{\omega'=0}^{\omega'} (\text{start}(i, \omega + 1) - \text{start}(i, \omega))}{\omega'}
\]

(1)

Note that, since the activities are repeated indefinitely, the sum can start for \( \omega = 0 \) without loss of generality, even if \( \omega \) is in \( \mathbb{Z} \) as from the problem definition.

**Definition 3.** The average inter-execution distance \( \lambda \) of a set of activities \( \mathcal{V} \) is the worst case \( \lambda(i) \).

\[
\lambda = \max_{i \in \mathcal{V}} \lambda(i)
\]

(2)

The throughput of an activity \( i \) is defined as the average number of executions of \( i \) per time unit, and corresponds to the inverse of the average distance:

\[
\text{THP}(i) = \frac{1}{\lambda(i)}
\]

Analogously, the throughput of a set of activities is the inverse of \( \lambda \). Lower period (i.e. higher throughput) values are to be preferred since they correspond to more efficient schedules.

Note that a schedule has infinite size in principle, because each activity executes an infinite number of times. However, since building an infinite schedule is impossible in practice, we should find a way to compute a more compact problem solution. Typically, one wants to build a periodic schedule, i.e. a schedule where activities are executed regularly with a fixed period. Such an approach requires to specify only a start time for each activity, plus the period value. Periodic schedules will be formally defined in Section 2.3.

In practical cases, activities are subject to several restrictions. In particular, there may be resource constraints and temporal dependencies, described in the following sections.

**2.1. Resource constraints**

In this work, we consider activities requiring different amounts of a set \( \mathcal{R} \) of limited capacity resources. Each resource \( k \in \mathcal{R} \) has maximum capacity \( \text{CAP}_k \) and each activity \( i \in \mathcal{V} \) has a set of non-negative requirements \( r_{i,k} \), one per resource. A zero requirement denotes a non-required resource. A schedule is feasible iff, for each \( k \in \mathcal{R} \), at any point in time \( t \) the sum of the requirements \( r_{i,k} \leq \text{CAP}_k \) of the resource. Formally:

\[
\sum_{i \in \mathcal{V}, \omega \in \mathbb{Z}, \text{start}(i, \omega) \leq t} r_{i,k} \leq \text{CAP}_k \quad \forall t \in ]-\infty, \infty[, \forall k \in \mathcal{R}
\]

(3)

**2.2. Time constraints**

Temporal dependencies between activities corresponds to arcs in the project graph and may connect executions with different \( \omega \) number.

**Definition 4.** Each arc \((i, j) \in \mathcal{A}\) is a tuple \((i, j, \theta_{(i,j)}, \delta_{(i,j)})\), where:

- \( i \) is the source activity;
- \( j \) is the sink activity;
- \( \theta_{(i,j)} \in \mathbb{R} \) is called minimum time lag;
- \( \delta_{(i,j)} \in \mathbb{Z} \) is called the repetition distance;
The arc \((i, j, \theta_{(i,j)}, \delta_{(i,j)})\) enforces the relation:

\[
\text{start}(j, \omega) \geq \text{start}(i, \omega - \delta_{(i,j)}) + d_i + \theta_{(i,j)} \quad \forall \omega \in \mathbb{Z}
\]

Observe that, in a cyclic problem, a temporal dependency connects an infinite number of distinct pairs of executions of \(i\) and \(j\). The value \(\delta_{(i,j)}\) acts as a repetition offset: it declares the distance in terms of repetitions between the executions of the connected activities. If \(\delta_{(i,j)} = 0\), the edge \((i, j, \theta_{(i,j)}, 0)\) is called intra-repetition edge while if \(\delta_{(i,j)} \neq 0\), then \((i, j, \theta_{(i,j)}, \delta_{(i,j)})\) is an inter-repetition edge. The time lag \(\theta_{(i,j)}\) specifies the minimal temporal distance between the start of execution \((j, \omega)\) and the end of execution \((i, \omega - \delta_{(i,j)})\).

The presence of inter-repetition dependencies may create feasible cycles in the project graph. Consider the following simple example about building a skyscraper, depicted in Fig. 1. The project graph contains the two activities floor and pillars (having respectively duration 5 and 2) and two arcs \((\text{floor, pillars}, 2, 0)\) and \((\text{pillars, floor}, 1, 1)\). Both arcs have a positive time lag. Activity floor represents the act of building the floor frame and activity pillars represents the edification of the pillars for the next floor. The first temporal dependency ensures that the pillars are built after the floor completion, while the second one conveys that without the pillars of the previous floor a new floor frame cannot be built. Note that \((\text{pillars, floor}, 1, 1)\) is an inter-repetition dependency.

Based on Eq. (4), we can formally define both the dependencies:

\[
\begin{align*}
\langle \text{floor, pillars}, 2, 0 \rangle & \rightarrow \text{start}(\text{pillars}, \omega) \geq \text{start}(\text{floor}, \omega) + 5 + 2 & \forall \omega \in \mathbb{Z} \\
\langle \text{pillars, floor}, 1, 1 \rangle & \rightarrow \text{start}(\text{floor}, \omega) \geq \text{start}(\text{pillars}, \omega - 1) + 2 + 1 & \forall \omega \in \mathbb{Z}
\end{align*}
\]

We now proceed by providing a numeric example to clarify the mechanics of the temporal dependencies. Since a cyclic schedule is infinite, we can choose for convenience a reference repetition and start time. Fixing the start time of one activity does not compromise completeness since \(\omega \in \mathbb{Z}\). Specifically, let us assume that \(\text{start}(\text{floor}, 0) = 0\) (i.e. the workers immediately start to build the ground floor). The following equations and Fig. 2 show how the start times of different executions are (lower) bounded by temporal dependencies:

**Step 1.** \(\omega = 0\):

\[
\langle \text{floor, pillars}, 2, 0 \rangle \rightarrow \text{start}(j, 0) \geq \text{start}(\text{floor}, 0) + 7
\]

\[
\text{start}(\text{pillars}, 0) \geq 7
\]

**Step 2.** \(\omega = 1\):

\[
\langle \text{pillars, floor}, 1, 1 \rangle \rightarrow \text{start}(\text{floor}, 1) \geq \text{start}(\text{pillars}, 0) + 3
\]

\[
\text{start}(\text{floor}, 1) \geq 7 + 3
\]

\[
\langle \text{floor, pillars}, 2, 0 \rangle \rightarrow \text{start}(\text{pillars}, 1) \geq \text{start}(\text{floor}, 1) + 7
\]

\[
\text{start}(\text{pillars}, 1) \geq 10 + 7
\]

**Step 3.** \(\omega = 2\):

\[
\langle \text{pillars, floor}, 1, 1 \rangle \rightarrow \text{start}(\text{floor}, 2) \geq \text{start}(\text{pillars}, 1) + 3
\]

\[
\text{start}(i, 2) \geq 17 + 3
\]

\[
\langle \text{floor, pillars}, 2, 0 \rangle \rightarrow \text{start}(\text{pillars}, 2) \geq \text{start}(\text{floor}, 2) + 7
\]

\[
\text{start}(\text{pillars}, 2) \geq 20 + 7
\]

\[
\vdots
\]

2.3. Periodic schedule

As stated in Section 2, a way to compute a compact solution to the problem is to consider schedules where the activities are executed regularly with a fixed period (i.e. periodic schedules). Note that in this case the fixed period (called iteration period) is the same as the maximum average inter-execution distance \(\lambda\) from Definition 3. Formally:
Definition 5. A periodic schedule is pair $\langle \mathbb{L}, \lambda \rangle$ where $\lambda$ is the iteration period and $\mathbb{L}$ is a vector containing the start times for the execution 0 of all the activities, i.e.:

$$\mathbb{L}[i] = \text{start}(i, 0) \quad \forall i \in V$$

A periodic schedule obeys the following restriction:

$$\text{start}(i, \omega) = \text{start}(i, 0) + \omega \cdot \lambda \quad \forall \omega \in \mathbb{Z} \forall i \in V \quad (7)$$

The iteration period of the schedule is defined as the distance between the start times of two consecutive executions of the same activity. We use the notation $\lambda$, since for a periodic schedule the iteration period is the same as the maximum average inter-execution distance. This can be checked by combining Eqs. (1) and (2) from Section 2 with Eq. (7). Finally, the makespan for a periodic schedule is defined as the distance between the start of the first activity and the end of the last activity in every repetition. For this reason, it will be referred in the following as schedule length. To avoid confusion, we will use the same terminology also for other, non-strictly periodic, approaches.

2.3.1. Iteration bound

Let $C$ be the set of all cycles of a project graph $G$, where a cycle is defined in the usual fashion as a sequence of precedence connected activities whose last element is also connected to the first one. Let $V(c)$ and $A(c)$ respectively be the set of activities and arcs that belong to a cycle $c \in C$.

Definition 6. The cycle bound $\text{CB}(c)$ of a cycle $c$ is the minimum time required in a periodic schedule to execute all the activities in $c$. This is equal to:

$$\text{CB}(c) = \frac{T(c)}{D(c)}$$

where:

- $T(c) = \sum_{i \in V(c)} d_i$ is the sum of the durations of the activities in $V(c)$.
- $D(c) = \sum_{(i,j) \in A(c)} \delta_{(i,j)}$ is the sum of the repetition distances of arcs in $A(c)$.

Note that the sum of the repetition distances must be strictly positive $D(c) > 0$, otherwise no feasible schedule exists. This can be intuitively checked since $\text{CB}(c) \to \infty$ as $D(c) \to 0$: more details and formal proofs can be found in [17]. As a consequence, at least an arc in the cycle must have a positive $\delta$. A graph with $D(c) > 0 \forall c \in C$ is called deadlock-free.

Definition 7. The Iteration Bound $\text{IB}$ of a graph is the maximum of the cycle bounds:

$$\text{IB} = \max_{c \in C}(\text{CB}(c))$$

The iteration bound is related to the concept of critical path in traditional scheduling and can be computed in polynomial time with the algorithm presented in [21]. The iteration bound $\text{IB}$ is an intrinsic lower bound on the iteration period: we can never achieve a $\lambda$ less than $\text{IB}$, even if unlimited resources are available. Periodic schedules are said to be periodically optimal if the iteration period $\lambda$ is the same as the iteration bound $\text{IB}$. If there are no resource restrictions, then a periodically optimal schedule is guaranteed to exist. This is not true once we add resource constraints to the problem. Further details can be found in [12,16].

2.4. Problem definition

We can now define the CRCSP as follows. Given:

- a Project Graph $G = (V, A)$ with:
  - a set of activities $i \in V$;
  - a set of temporal dependencies $(i, j, \theta_{(i,j)}, \delta_{(i,j)}) \in A$.
\[ A = 1, \quad R = 1 \]

\[ \begin{array}{ccc}
A & 1 & 6 \\
B & 1 & 4 \\
C & 1 & 12 \\
D & 1 & 3 \\
E & 1 & 5 \\
\end{array} \]

Total Capacity \( R = 3 \)

**Fig. 3.** Annotated representation of a cyclic graph.

- A set \( R \) of limited capacity resources, where each resource \( k \in R \) has capacity \( \text{CAP}_k \).
- A fixed duration \( d_i \) for each activity.
- A resource requirement \( r_{i,k} \geq 0 \) for each activity \( i \) and resource \( k \).

The CRCSP consists in finding a periodic schedule: i.e. an iteration period \( \lambda \) and a feasible assignment of \( \text{start}(i, 0) \forall i \in V \) such that all dependencies are consistent, no resource capacity is exceeded and the iteration period \( \lambda \) is minimized.

### 2.5. Overlapped schedules

We end this section with a couple of useful definitions. Cyclic schedules can be either non-overlapped or overlapped.

- A schedule is said to be non-overlapped if repetition \( \omega + 1 \) starts only once repetition \( \omega \) is over. Formally:

\[
\text{start}(i, \omega + 1) \geq \text{start}(j, \omega) + d_j \quad \forall i, j \in V
\]  

- A schedule is overlapped if Eq. (8) does not hold, i.e.:

\[
\exists i, j \in V \mid \text{start}(i, \omega + 1) < \text{start}(j, \omega) + d_j
\]

### 3. Related works

The cyclic scheduling literature mainly arises in industrial and computing contexts. While there is a considerable body of work on cyclic scheduling in the OR literature [1], the problem has not received much attention from the AI community ([13] is one of the few related papers).

Three main approaches have been proposed for the CRCSP. The first belongs to the non-overlapped class, while the other two exploit inter-repetition overlaps:

- The so-called blocked scheduling approach (see Section 3.1) builds a schedule for a single repetition and assumes that the period is equal to the schedule length. As a consequence, consecutive repetitions are not allowed to overlap, with an obvious loss in terms of solution quality.
- The unfolding approach (see Section 3.2) schedules a number of consecutive repetitions and then repeats the block similarly to the previous technique. Unfolding often leads to higher quality schedules, but it also requires to solve problem instances with artificially increased size.
- The modulo scheduling approach (see Section 3.3) schedules a single repetition, which is however repeated every \( \lambda \) time units. The value \( \lambda \) is called the modulus and it is the same as the iteration period. By exploiting repetition overlaps, the modulus can be made much smaller than the schedule length, obtaining considerable speed-ups.

**Fig. 3** depicts a simple example of a CRCSP. All minimum time lags \( \theta_{i,j} \) are assumed to be 0 and the repetition distance \( \delta_{i,j} \) is 0 whenever not explicitly mentioned. In the following subsections we use this instance to show how the three approaches described above work.

### 3.1. Blocked scheduling

Traditional static resource-constrained scheduling techniques for cyclic problems are non-overlapped [24,7]. These methods optimize the performance of a single repetition of the project graph and then repeat the schedule periodically.

**Fig. 4** shows the optimal blocked schedule for the simple problem described in **Fig. 3**. The output of the solution approach is the schedule (or block) identified by the black arrows, which is then repeated every iteration period \( \lambda = 27 \). Hence, the throughput is one over the length of the schedule \( T_{thp} = \frac{1}{27} = 0.037 \). The horizontal dotted line represents the resource capacity: note that this schedule leaves most of the resource idle. With blocked schedules, this may happen quite frequently.
3.2. Unfolded scheduling

Overlapped schedules exploit the repetitive nature of periodic schedules to achieve higher throughput. On this purpose, they take into account inter-repetition dependencies in addition to intra-repetition ones. The unfolding technique, presented in [32] and [33], consists in scheduling $u$ consecutive repetitions of the graph, where $u$ is referred to as unfolding factor (or as blocking factor). Then, the resulting schedule (say with total length $L$) is treated as a normal blocked schedule and repeated every $L$ time units. Note that the schedule length acts as a sort of period, but since $u$ repetitions are considered, the actual average inter-repetition distance is $\lambda = \frac{L}{u}$. In fact, $u$ executions are completed in $L$ time units.

Figs. 5 and 6 show the optimal unfolded schedule for the simple problem depicted in Fig. 3 with unfolding factor $u = 2$ and $u = 3$ respectively (with $u = 1$ we obtain the blocked schedule from Fig. 4). The schedule is restarted after $L$ time units: for $u = 2$ the optimal $L$ is 35 and $\lambda = 17.5$, while for $u = 3$, $L = 47$ and $\lambda = 15.6$. The throughput is higher than in the blocked schedule, but the problem size (and consequently the search space) is bigger because it consists of $u$ different repetitions. Because the problem is NP-hard [25], multiplying the number of activities by the unfolding factor leads to an exponential increase in the solution time.

Since the group of repetitions in the schedule is repeated in a blocked fashion, the unfolding approach may be incapable to optimally exploit inter-repetition overlaps to make the best use of the available resources. Moreover, despite increasing $u$ from 2 to 3 led to a lower $\lambda$ value in our example, it is not true in general that increasing the unfolding factor leads to better schedules. The optimal unfolding factor cannot be trivially computed.

On the other hand, it is known [19] that periodic schedules are dominated by K-periodic schedules (i.e. periodic schedules for sequences of K iterations) in the presence of finite capacity resources. Since the unfolding technique takes into account multiple iterations, in some cases an unfolded schedule may be better than a (single-iteration) periodic schedule. Hence, no strict dominance exists between the two approaches. An in-depth survey of traditional and unfolding cyclic scheduling techniques can be found in [5].

3.3. Modulo scheduling

The modulo scheduling method consists in finding a schedule for a single repetition, plus a modulus value $\lambda$ (usually lower than the whole schedule length). The schedule is repeated every $\lambda$ time units and the activities of consecutive repetitions, like in a pipelined system, overlap over time. This technique is the best in exploiting the available resources. Fig. 7 shows the optimal modular schedule for the graph of Fig. 3. The figure shows that, after an initial transient phase, the execution reaches a periodic phase where a repetition of the graph is completed every $\lambda = 10$ (corresponding to a throughput $\frac{1}{\lambda} = 0.1$).
Note that a time window of length $\lambda$ (evidenced with the black arrows) contains the start time of exactly one execution of each activity. The activities may however appear with different execution numbers (i.e. $\omega$ values), following a well defined pattern. Such a collection of activity executions is called an *iteration* and is a fundamental concept for our method.

Fig. 8a depicts the schedule subpart corresponding to a specific iteration starting at a multiple of $\lambda$ (in the periodic phase). In the picture, with $x_\omega$ we refer to the $\omega$-th execution of activity $x$, i.e. to $(x, \omega)$. Note that an iteration contains a single execution of each activity and that not all the activities appear with the same $\omega$ values. For example, $(C, \omega+1)$ and $(A, \omega+2)$ are executing concurrently with $(D, \omega)$, i.e. the first execution of activity $D$ runs together with the second execution of $C$ and the third execution of $A$.

One of the key ideas of this paper is to focus on scheduling a single iteration (Fig. 8a) instead of a single repetition with a large horizon (Fig. 8b). In this context, it is convenient to restrict to iterations starting at a multiple of the period $\lambda$, so that we can refer as the $\omega$-th iteration to the iteration starting at $\omega \cdot \lambda$. Further details will be given in Section 5.

### 3.3.1. Specific modulo scheduling techniques

**OR approaches** An advanced ILP formulation for the modulo scheduling approach has been proposed in [14] by Dupont de Dinechin and is based on a time-indexed model. A recurring issue for this kind of approach is that, in periodic problems, the schedule length (which determines the number of problem variables) may be fairly big compared to the period. In an attempt to circumvent this difficulty, a second formulation was proposed in [15] by Eichenberger and Davidson (by exploiting a decomposition of start times) at the cost of a reduction in the quality of the LP bound. An excellent overview of state-of-the-art ILP formulations is given in [2], where the authors present also a new model based on the Danzig-Wolfe Decomposition. Another good overview of complete methods can be found in [19] while McCormick et al., in [31], study the complexity in cyclic scheduling. All the mentioned approaches are based on iteratively solving resource constrained subproblems obtained by fixing the period value. To the best of our knowledge, this is a common trait of all the state-of-the-art approaches in the OR field. The main reason is that fixing $\lambda$ allows to solve the Resource Constrained Cyclic Scheduling Problem via an integer linear program, while modeling $\lambda$ as an explicit decision variable yields non-linear models.

**A constraint programming approach** In [13] the authors present a formulation for solving cyclic job shop scheduling as a Constraint Satisfaction Problem. They also describe how their formulation could be generalized to face problems with cumulative resources. The approach they propose is again based on fixing the period and solving the derived subproblems. The obvious drawback is that a resource constrained scheduling problem needs to be repeatedly solved for different $\lambda$ values to obtain the feasible/optimal solution. Our method is also based on Constraint Programming, but it does not require to fix a $\lambda$ value, thanks to the use of a global constraint to model resource restrictions.

**Incomplete approaches** Several heuristic approaches have been proposed to find the smallest possible $\lambda$. These are usually instruction scheduling techniques that are used by several current compilers.\(^1\) The *Iterative Modulo Scheduling* algorithm is presented in [35]: the main feature of the algorithm is its iterative nature, in the sense that each activity can be scheduled and unscheduled (therefore considering backtrack operations) several times before a suitable slot is found. Another heuristic approach (called SCAN) is presented in [6]. SCAN is built on some of the main ideas behind Iterative Modulo Scheduling, but

\(^1\) Heuristic-based Modulo Scheduling is a family of Software Pipelining techniques that produce effective schedules with a relatively small compilation time. These Modulo Scheduling techniques take as input the application to be scheduled represented by its data dependence graph and a description of the architecture and produce a schedule for the application.
it is based on an ILP model. In [20] Huff presented the slack modulo scheduling algorithm. Key of this method is the use of a bidirectional scheduling approach (top-down and bottom-up) and the use of slacks. The slack of an unbound activity is a measure of the freedom that the activity would have if it was scheduled in the partial solution. Another interesting approach is described in [8], where Brucker and Kampmeyer apply tabu-search to cyclic job-shop scheduling problems. The state of the art for incomplete methods is probably Swing Modulo Scheduling, described in [26, 27]. The approach is implemented in the optimization chain of the gcc compiler [18]. The method produces effective schedules with a low computational cost (see [11] for a well structured comparison between most of these modulo scheduling approaches). Key for the efficiency is a valid variable selection function considering both the period bound imposed by the current iteration and the criticality of the path to which the activity to be scheduled belongs to.

Those heuristic approaches compute a schedule for a single repetition of the application. The schedule is characterized by its length and by an initiation interval, which is the same as the iteration bound and determines the throughput. However, the schedule length can be extremely large, with implications on the size of the model in case of ILP based approaches (e.g. SCAN). Our model is considerably more compact, since we schedule a single iteration as opposed to a repetition, so that we restrict to a time window with length $\lambda$.

4. Modular representation for cyclic schedules

In this section, we recall some modular arithmetic notions that are the foundations of the cyclic scheduling solver we propose. The main underlying idea is to focus on a $\lambda$-width time window in the periodic phase (see Fig. 7). First, we present a start/end time decomposition similar to that in [15]. The start time of execution 0 of activity $i$ (i.e. $\text{start}(i, 0)$) can be expressed as:

$$\text{start}(i, 0) = s_i + \beta_i \cdot \lambda$$

where $s_i$ is a value in the half-open interval $[0, \lambda]$ and $\beta_i$ is an integer number. In practice, $\beta_i$ identifies the iteration when activity $i$ is first scheduled (see Section 3.3) and $s_i$ is its relative start time within the corresponding $\lambda$-width time window, i.e. its modular start time. Analogously, the end time $\text{end}(i, 0)$ can be decomposed into a modular end time $e_i$ and an iteration number $\eta_i$.

$$\text{end}(i, 0) = e_i + \eta_i \cdot \lambda$$

Fig. 9 shows the optimal modulo scheduling related to the graph from Fig. 3 described in Section 2. Note that activity $C$ is longer than the period $\lambda$, hence the activity begins in a period and ends in the following one. For instance, the third execution of $C$ (i.e. $C'''$) starts at instant $\text{start}(i, 3) = 30$ and terminates at 42. Using the modular representation the activity has $s_i = 0$, $\beta_i = 3$ and $e_i = 2$, $\eta_i = 4$. Hence it starts at $\text{start}(i, 3) = 0 + 3 \cdot \lambda$ and ends at $2 + 4 \cdot \lambda$. Since $\lambda = 10$ we have $s_i + \beta_i \cdot \lambda = 30$ and $e_i + \eta_i \cdot \lambda = 42$.

Note that there is a strong correlation between iteration numbers and schedule length. In particular, a larger difference between the highest and smallest $\beta_i$ in an iteration corresponds to a larger schedule length. Moreover, start and end times are constrained by the relation $\text{end}(i, 0) = \text{start}(i, 0) + d_i$, hence we have $e_i + \eta_i \cdot \lambda = s_i + \beta_i \cdot \lambda + d_i$ and hence:

$$d_i = e_i - s_i + (\eta_i - \beta_i) \cdot \lambda$$

Moreover, since $e_i - s_i$ is strictly less than $\lambda$, we have $\eta_i - \beta_i = \lfloor \frac{d_i}{\lambda} \rfloor$, which means that $\eta_i$ is unambiguously determined once $\beta_i$ and $\lambda$ are known. We can also rewrite the temporal dependencies using the modular formulation. In particular, the relation:

$$\text{start}(j, \omega) \geq \text{start}(i, \omega - \delta_{(i,j)}) + d_i + \theta_{(i,j)}$$

is rewritten as:

$$\text{start}(j, 0) + \omega \cdot \lambda \geq \text{start}(i, 0) + (\omega - \delta_{(i,j)}) \cdot \lambda + d_i + \theta_{(i,j)}$$

$$s_j + \beta_j \cdot \lambda + \omega \cdot \lambda \geq s_i + \beta_i \cdot \lambda + (\omega - \delta_{(i,j)}) \cdot \lambda + d_i + \theta_{(i,j)}$$

$^2$ The method can take advantage of the use of bidirectional scheduling method as the period $\lambda$ is fixed.
performing the usual eliminations we have the following inequality that no longer depends on $\omega$:

$$s_j + \beta_j \cdot \lambda \geq s_i + (\beta_i - \delta_{i,j}) \cdot \lambda + d_i + \theta_{i,j}$$  \hspace{1cm} (12)$$

This is a very important result and it is equivalent to say that in a periodic schedule, if the (modular) precedence constraints are satisfied for iteration 0, then they are satisfied for every other iteration.

4.1. Resource modeling

Before introducing our constraint model it is necessary to get a better insight into the resource usage profile of a periodic schedule. The most unusual peculiarity of this case is that the value of $\lambda$ may modify the resource usage over time, as depicted in Fig. 10. The picture presents the profiles corresponding to schedules with the same start times, but with 4 different period values:

- case a) represents a classic non-overlapped schedule.
- In b) the modular start time $s$ is equal to the modular end time $e$, since the period is equal to the length of the activity.
- In case c) the period is shorter than the activity, hence the schedule is overlapped and the resource profile exhibits a “pulse” in the middle.
- In d) the period is slightly larger than the half of the activity duration. In this profile the modular end time $e$ precedes the modular start time $s$, which is a bit counterintuitive.

Note that changes in the modulus affect not only the resource usage but also the modular end time $e_i$. In general, in [28] we showed that the amount of resource $k$ required by activity $i$ in a time window having width $\lambda$ is given by the following expression:

$$r_{q_{i,k}}(s_i, d_i, t, \lambda) = \begin{cases} 
   r_{i,k} \cdot (\lfloor \frac{d_i}{\lambda} \rfloor + 1) & \text{if } s_i \leq t < e_i \text{ or } e_i < s_i \leq t \text{ or } t < e_i < s_i \\
   r_{i,k} \cdot \lfloor \frac{d_i}{\lambda} \rfloor & \text{otherwise}
\end{cases}$$  \hspace{1cm} (13)$$

In other words, the resource usage is given by a constant factor $r_{i,k} \cdot \lfloor \frac{d_i}{\lambda} \rfloor$, plus an additional $r_{i,k}$ in case the considered time point lies “between” the modular start and the modular end (the quotes are used since we may have $e_i < s_i$). In Fig. 10, the constant usage factor is always 1 except for a), where it is 0. As a particular case, if $\beta_i = \eta_i$ the constant usage factor is zero and $r_{q_{i,k}}(s_i, d_i, t, \lambda)$ becomes a classical resource usage function. This suggests that forcing the end times to be within the modulus allows the use of classical resource constraints (see Section 5).

5. Constraint-based model

In this section we propose a complete constraint-based approach for the cyclic scheduling problem, based on modular arithmetic. Our CP model features three classes of variables, representing the (modular) start times, the corresponding iteration numbers and the modulus. The modular start time variables have domain $[0, \text{MAX\_TIME}]$, the iteration numbers are in $[-\|V\|, +\|V\|]$ and the modulus is in $[\text{IB}, \text{MAX\_TIME}]$. We recall that $\|V\|$ is the number of nodes in the graph. The value $\text{MAX\_TIME}$ instead is given by the sum of the durations of the activities and the sum of the time lags of the edges.

For sake of brevity, in the rest of the paper we will use the notation $s_i, e_i, \beta_i, \eta_i$ and $\lambda$ to refer to variables rather than values. Moreover, we will speak of “start/end times”, implicitly meaning their modular counterparts: non-modular start/end times are never used in the model. In the rest of the paper we also assume that:

- $\text{EST}_i$ is the earliest start time of $i$: the minimum in the domain of $s_i$;
- $\text{LST}_i$ is the latest start time of $i$: the maximum in the domain of $s_i$;
- $\text{EET}_i$ is the earliest end time of $i$: the minimum in the domain of $e_i$;
- $\text{LET}_i$ is the latest end time of $i$: the maximum in the domain of $e_i$.

Hence the domain of a start time variable $s_i$ is actually $[\text{EST}_i, \text{LST}_i]$, with $\text{LST}_i < \lambda$. Similarly, the domain of $e_i$ is $[\text{EET}_i, \text{LET}_i]$. Furthermore, we use the notation $\underline{x}$ and $\overline{x}$ to refer to the highest and the lowest values (i.e. the upper and the lower bounds) of the domain of a generic variable $x$. Obviously the lower/upper bound of a time variable is the same as its earliest/latest time, e.g. $\underline{s_i} = \text{EST}_i$ and $\overline{s_i} = \text{LST}_i$. Finally, as stated in Section 4, the $\beta_i$ and $\eta_i$ values are equal to $s_i + (d_i \mod \lambda)$ and $\beta_i + \lfloor \frac{d_i}{\lambda} \rfloor$ respectively. Therefore the model is actually based only on the $s_i$ and $\beta_i$ variables.

These variables are subject to temporal and resource constraints. In order to model a temporal dependency $(i, j, \theta_{i,j}, \delta_{i,j})$ we have implemented a modular precedence constraint (ModPC), with the following signature:

\footnote{As stated in Section 2.3.1 the iteration bound $\text{IB}$ is a valid lower bound for the modulus.}
Fig. 10. Resource profile modifications.

\[ \text{ModPC}(e_i, s_j, \eta_i, \beta_j, \lambda, \theta_{i,j}, \delta_{i,j}) \]

where \( e_i \), \( s_j \), \( \eta_i \), \( \beta_j \), \( \lambda \) are the variables representing the end time of activity \( i \), the start time of activity \( j \), their respective iteration numbers and the period. The parameters \( \theta_{i,j} \) and \( \delta_{i,j} \) are assumed to have fixed values. The filtering algorithm for the constraint is described in Section 6.1.

In the CRCSP all activities are subject to resource constraints. Unlike in traditional scheduling, the resource profile depends on the period value (see Section 4.1). For this reason we devised a global cumulative constraint based on a modular time table, that ensures a consistent resource usage (the GCCC, discussed in Section 6.3). The signature is as follows:

\[ \text{GCCC}([s_i], [d_i], [r_i], \text{CAP}, \lambda) \] (14)

where \([s_i]\) is a vector of start time variables, \([d_i]\) is the vector of corresponding durations and \([r_i]\) are the requirements for the modeled resource. The \( \text{CAP} \) value is the resource capacity and \( \lambda \) is the period variable.

As an alternative, it is possible to use traditional resource constraints by making the assumption that \( \beta_i = \eta_i \), i.e. that the end and the start time of an execution \((i, \omega)\) must be within to the same period. Formally:

\[ s_i \leq e_i \leq \lambda \quad \forall i \in \mathbb{V} \]

Since no activity is scheduled across different periods, the classical cumulative constraint can be used. As a main drawback, we may incur a quality loss due to the presence of an unnecessary restriction, which can be substantial for large resource capacities and large activity durations. In such situations, if we make no special assumption and we allow \( \eta_i \geq \beta_i \) and \( e_i < s_i \), we may obtain much better schedules. We refer to the solution approach making use of both the modular precedence constraint and the GCCC as CROSS, with no restricting assumption. Conversely, in the CROSS\(^{\ast}\) approach we force the end
times to be within the modulus and replace the GCCC with traditional cumulative constraints. The two approaches differ also for the adopted search strategy, discussed in Section 7.

5.1. Buffer constraints

In a real world context, a precedence constraint often implies an exchange of intermediate products between activities that should be stored in buffers. For example, in the context of embedded systems two activities may exchange data packets in the buffer is consumed. It is common to have on each buffer a size limit, which can be modeled through the following constraint:

$$\beta_j - \beta_i + (\epsilon_i \leq s_j) \leq B_{(i,j)} - \delta_{(i,j)} \quad (15)$$

where $B_{(i,j)}$ is the size limit and the reified constraint $(\epsilon_i \leq s_j)$ evaluates to one if the condition is satisfied. Inequality (15) limits the number of executions of activity $i$ (the producer) before the first execution of $j$ (the consumer): this ensures that the buffer capacity is always respected. Obviously $B_{(i,j)} \geq \delta_{(i,j)}$, otherwise the problem is infeasible. In fact, the value $\delta_{(i,j)}$ can be thought as the number of products already accumulated in the buffer $(i, j, \theta_{(i,j)}, \delta_{(i,j)})$ when the project starts for the first time.

6. Filtering algorithms

This section presents the filtering algorithms of the Modular Precedence Constraint, the Buffer Constraint and the Global Cyclic Cumulative Constraint, used respectively to model temporal dependencies, buffer and resource constraints.

6.1. Filtering of the Modular Precedence Constraint ModPC

We recall that the Modular Precedence Constraint (ModPC) constraint has the following signature:

$$\text{ModPC}(\epsilon_i, s_j, \eta_i, \beta_j, \lambda, \theta_{(i,j)}, \delta_{(i,j)})$$

where $\epsilon_i, s_j, \eta_i, \beta_j, \lambda$ are variables representing respectively the end time of activity $i$, the start time of activity $j$, their respective iteration numbers and the modulus, and $\theta_{(i,j)}, \delta_{(i,j)}$ are constant values representing the minimum time lag and the iteration distance associated to the arc. For sake of simplicity, in the following we will omit the subscript when referring to $\theta$ and $\delta$. We recall that the value $\theta_{(i,j)}$ must be non-negative, i.e. $\theta \geq 0$.

The filtering algorithm of the Modular Precedence Constraint has three fundamental components:

- The filtering rules for the iteration variables, which updates the bounds of the $\eta_i$ and $\beta_j$ variables so that a proper distance (in terms of number of iterations) exists between activity $i$ and $j$.
- The filtering rules for the start time variables, which modify the start times of the involved activities to avoid infeasible overlaps.
- The filtering rules for the modulus variable, which compute a lower bound on the modulus.

The algorithm is executed whenever the domain bounds of any involve variable change. Filtering a single precedence relation achieves bound consistency and takes constant time.

6.1.1. Filtering the iteration variables

With reference to the temporal model proposed in Section 5 we can rewrite Inequality (12) as:

$$s_j + \beta_j \cdot \lambda \geq \epsilon_i + \theta + (\eta_i - \delta) \cdot \lambda. \quad (16)$$

Starting from the equation above, we have

$$\eta_i - \beta_j - \delta \leq \frac{s_j - \epsilon_i - \theta}{\lambda} \quad (17)$$

with $-\lambda < s_j - \epsilon_i \leq \lambda$ and $\theta \geq 0$. Eq. (17) can be used to obtain bounds over the $\beta_j$ (and $\eta_i$) variables, in particular:

$$\eta_i \leq \beta_j + \delta + \left[\frac{s_j - \epsilon_i - \theta}{\lambda}\right] \quad \beta_j \geq \eta_i - \delta - \left[\frac{s_j - \epsilon_i - \theta}{\lambda}\right] \quad (18)$$

As an example, suppose that during search two activities $i$ and $j$ connected by a temporal dependency $\langle i, j, 0, 0 \rangle$ are overlapping and have $s_j = 0, \epsilon_i = 3$: then Inequality (18) appears as follows:
\[
\beta_j \geq \eta_i - \left\lfloor \frac{-3}{\lambda} \right\rfloor
\]  
(19)

which implies that \( \beta_j > \eta_i \). In fact, two connected activities can overlap iff they have different iteration values: in particular \( \beta_{\text{sinkNode}} > \eta_{\text{sourceNode}} \).

### 6.1. Filtering the start time variables

Let \( \Delta_\omega \) be \( \eta_i - \beta_j - \delta \) and hence \( \Delta_\omega \) be \( \eta_i - \beta_j - \delta \). Constraint (16) can now be written as:

\[
s_j - e_i + \theta \geq \Delta_\omega \cdot \lambda.
\]  
(20)

Note that we must have \( \Delta_\omega \leq 0 \): in fact \( \Delta_\omega > 0 \) would imply that \( \beta_j < \eta_i - \delta \) which is, by definition, impossible. From \( \Delta_\omega \leq 0 \), we can deduce the two inequalities, that lead to bounds on the start/end variables:

\[
s_j \geq e_i + \theta + \Delta_\omega \cdot \lambda \geq e_i + \theta + \Delta_\omega \cdot \lambda
\]  
(21)

\[
e_i \leq s_j - \theta - \Delta_\omega \cdot \lambda \leq s_j - \theta - \Delta_\omega \cdot \lambda
\]  
(22)

Note that if \( \Delta_\omega = 0 \) the modular constraint boils down to a classical precedence constraint, i.e. \( s_j \geq e_i + \theta \). In fact, two connected activities with the same iteration value cannot overlap and Inequality (21) “pushes” the destination activity \( j \) after the end time of \( i \) plus the time lag \( \theta \).

### 6.1.3. Filtering the modulus variable

The domain of the modulus variable can be pruned only when \( \Delta_\omega \) is strictly negative, i.e. \( \Delta_\omega < 0 \). In this situation, we can derive from Eq. (20) the following inequality, resulting in a lower bound on the modulus variable:

\[
\lambda \geq \frac{e_i - s_j + \theta}{-\Delta_\omega} = \left\lceil \frac{e_i - s_j + \theta}{\beta_j - \eta_i + \delta} \right\rceil
\]  
(23)

### 6.2. Filtering for the buffer constraints

From Inequality (15) we can derive two expressions to compute bounds over the \( \beta \) variables:

\[
\beta_j \leq \beta_i - (e_i \leq s_j) + (B_{i,j} - \delta_{i,j})
\]  
(24)

\[
\beta_i \geq -\beta_j + (e_i \leq s_j) - (B_{i,j} - \delta_{i,j})
\]  
(25)

where \( B_{i,j} \) is the size limit of the buffer and the reified constraint \( (e_i \leq s_j) \) equals one if the condition is satisfied. In other words Inequality (15) limits the iteration distance between the activities involved.

### 6.3. Filtering of the Global Cyclic Cumulative Constraint GCCC

The Global Cyclic Cumulative Constraint for resource \( k \) ensures consistency in the use of a single resource. The notion of consistency is formally given by Eq. (3). In our approach we make use of an equivalent formulation, which can be obtained in two steps. First, observe that in a periodic schedule, the resource constraints are satisfied if and only if they are satisfied in any \( \lambda \)-long interval. Therefore, without loss of generality we can restrict to \([0, \lambda]\). Then, the resource usage due to a single activity in such an interval is given by the \( r_{q_i(s_i, d_i, t, \lambda)} \) function from Eq. (13) and we can rewrite Eq. (3) as:

\[
\sum_{i \in V} r_{q_i(s_i, d_i, t, \lambda)} \leq \text{CAP}_k \quad \forall t \in [0, \lambda]
\]

Since the GCCC refers to a single resource, for the sake of readability we remove the \( k \) index from the requirement functions. Hence \( r_{q_i} \) becomes \( r_i \) and \( \text{CAP}_k \) becomes \( \text{CAP} \).

The filtering is inspired by the timetable filtering for the cumulative constraint [3]. Similarly to timetable filtering, our algorithm works with the compulsory parts of activities.

Activity \( i \) has a compulsory part if and only if there exists a time span where the activity is necessarily executing. This happens if \( \text{EST}_i + d_i > \text{LST}_i \). To take into account the resource usage corresponding to compulsory parts, we introduce a generalized version of Eq. (13) from Section 4.1:

\[
r_{q_i(s_i, d_i, t, \lambda)} = \begin{cases} 
\frac{r_i \cdot \left\lfloor \frac{d_i}{\lambda} \right\rfloor + 1}{\text{if}} & \text{LST}_i \leq t < \text{EET}_i \text{ or } \text{EET}_i < \text{LST}_i \leq t \text{ or } t < \text{EET}_i < \text{LST}_i \\
\frac{r_i \cdot \left\lfloor \frac{d_i}{\lambda} \right\rfloor}{\text{otherwise}} 
\end{cases}
\]  
(26)
Where \( s_i \) is a start time variable, rather than a value as in the non-generalized form of the function. Basically, the generalized \( \hat{r}_q_i (s_i, d_i, t, \lambda) \) represents the resource usage of an unbound activity, corresponding in practice to that of its compulsory part. If the \( s_i \) variable is bound, then \( \hat{r}_q_i (s_i, d_i, t, \lambda) = r_q_i (s_i, d_i, t, \lambda) \). The shape of the function is the same as before, namely a constant factor plus a "pulse". Note that if the activity duration is longer than the period (i.e. \( d_i \geq \lambda \)), then there exists a compulsory part at least as wide as the period itself. The GCCC constraint guarantees that:

1. the start time of each activity is not lower than the minimum instant where enough resources are available.
2. The modulus is not lower than the minimum value such that the cumulative usage due to the compulsory parts does not exceed the resource capacity.

The filtering algorithm exploits incremental computation and consists of three procedures:

- **Trigger**: this procedure is executed whenever any variable bound changes. The aim of this algorithm is to update the time tabling data structure.
- **Core**: this algorithm is executed at the end of all trigger procedures and it is structured in two independent phases:
  1. Start Time Propagation, that propagates the lower bound of the start time variables.
  2. Modulus Propagation,\(^4\) that computes the minimum \( \lambda \) needed to guarantee feasibility.
- **Coherence**: the procedure is executed whenever the modulus upper bound changes. The procedure modifies the data structure to guarantee the coherence with the new \( \lambda \) bound.\(^5\)

In the rest of this section we focus on the two phases of the **Core** procedure.

6.3.1. **Core phase 1**: start time filtering algorithm

The filtering algorithm guarantees that the start time of each activity is not lower than the minimum instant where enough resources are available, i.e.:

\[
\begin{align*}
\forall i \in [0, \lambda]: \sum_{j \in V \setminus \{i\}} r_q_j (s_j, d_j, t', \lambda) & \leq \text{CAP} - r_q_i (s_i, d_i, t', \lambda) \quad \forall t' \in [t, t + d_i^*] \\
\end{align*}
\]

where \( d_i^* = d_i \mod \lambda \) is the length of the pulse and is referred to as modular duration.\(^6\) Similarly to the timetable approach, we adopt a data structure to store the minimum resource usage, given the current scheduling decisions. For every \( t \in [0, \lambda) \), this is given by the following expression:

\[
\sum_{i \in V} \hat{r}_q_i (s_i, d_i, t, \lambda)
\]

Note that changes in the expression value occur at the \( \text{LST}_i \) and \( \text{EET}_i \) of all the activities: this is a direct consequence of the \( \hat{r}_q_i \) definition in Eq. (26).

Intuitively the algorithm proceeds as follows: for each unbound activity \( i \) the algorithm scans the resource profile, starting from \( \text{EST}_i \), to search for a schedulability window. A schedulability window is a time slice large enough and with enough resources to allow the activity execution. The process stops when a window is found or the search goes beyond the Latest Start Time (\( \text{LST}_i \)). Since the solver is based on modular arithmetic, the procedure follows a modular time wheel and the \( \text{LST}_i \) and \( \text{EET}_i \) values (i.e. the time points when a profile change may occur) are stored in a circular queue. Recall that, since \( \text{LST}_i \) and \( \text{EET}_i \) are modular values, they are guaranteed to be lower than \( \lambda \). The asymptotic complexity of the algorithm is discussed at the end of the section.

**Data structure** As stated in Section 5, each activity \( i \in V \) has four relevant time points: two of them are related to the start time, namely \( \text{EST}_i \) (or \( \underline{s}_i \)) and \( \text{LST}_i \) (or \( \overline{s}_i \)), and two related to the end time, i.e. \( \text{EET}_i \), \( \text{LET}_i \).

The constraint relies on an a circular queue \( \Omega [0, (\|V\| \cdot 2)] \) where each activity \( i \in V \) is represented via two queue items, respectively corresponding to the \( \text{LST}_i \) and its \( \text{EET}_i \). Each item \( \Omega [\text{idx}] \) stores three values:

- \( \Omega [\text{idx}] . \text{activity} \): the activity corresponding to item \( \Omega [\text{idx}] \).
- \( \Omega [\text{idx}] . \text{time} \): the time value, either the \( \text{LST}_i \) or the \( \text{EET}_i \).
- \( \Omega [\text{idx}] . \text{res} \): the total resource usage at instant \( \Omega [\text{idx}] . \text{time} \); formally

\[
\Omega [\text{idx}] . \text{res} = \sum_{i \in V} \hat{r}_q_i (s_i, d_i, \Omega [\text{idx}] . \text{time}, \lambda)
\]

The items are sorted by increasing \( \Omega [\text{idx}] . \text{time} \).

---

\(^4\) For ensuring the termination of the algorithm the period \( \lambda \) is considered as integer value.

\(^5\) The Trigger and the Coherence algorithms have an asymptotic complexity of \( O(n \cdot \log n) \), where \( n \) is the number of the activities.

\(^6\) In case \( d_i \) is not integer, then \( d_i^* = d_i - \lambda \cdot \left\lfloor \frac{d_i}{\lambda} \right\rfloor \).

---

Algorithm 1: Core 1: start times filtering algorithm.

Data: Let $S$ be the set of activities not already scheduled

\[
\begin{align*}
1 & \text{begin} \\
2 & \quad \text{for all the unbound activities } i \in S \text{ do} \\
3 & \quad \quad \text{feasible} = \text{false} \\
4 & \quad \quad \text{canStart} = \text{EST}_i \\
5 & \quad \quad t_0 = \text{EST}_i, \ idx_0 = \text{largest } idx \text{ such that } \Omega[idx].time \leq \text{EST}_i; \\
6 & \quad \quad \text{startIdx} = idx_0; \\
7 & \quad \quad \text{stop} = \text{false}; \\
8 & \quad \quad \text{while } \text{stop} \text{ do} \\
9 & \quad \quad \quad \text{// Resource availability check} \\
10 & \quad \quad \quad \quad \text{avRes} = \Omega[idx_0].\text{res} - r_q(t_0, d_i, t_0, \lambda); \\
11 & \quad \quad \quad \quad \text{if } \text{avRes} + r_q(\text{canStart}, d_i, t_0, \lambda) \leq \text{CAP} \text{ then} \\
12 & \quad \quad \quad \quad \quad \text{if } \neg \text{feasible} \text{ then} \\
13 & \quad \quad \quad \quad \quad \quad \text{feasible} = \text{true}; \\
14 & \quad \quad \quad \quad \quad \quad \text{canStart} = t_0; \\
15 & \quad \quad \quad \quad \text{else } \text{feasible} = \text{false}; \\
16 & \quad \quad \quad \text{// Offset determination} \\
17 & \quad \quad \quad \quad \text{if } \text{idx}_0 < \text{startIdx} \text{ then } \text{offset} = \lambda; \\
18 & \quad \quad \quad \quad \text{else } \text{offset} = 0; \\
19 & \quad \quad \quad \text{// Pruning} \\
20 & \quad \quad \quad \quad \text{if } \text{feasible} \land \text{canStart} + d_i \leq \text{offset} + t_0 \text{ then} \\
21 & \quad \quad \quad \quad \quad \text{EST}_i \leftarrow \text{canStart}; \\
22 & \quad \quad \quad \quad \quad \text{stop} = \text{true}; \\
23 & \quad \quad \quad \quad \text{else if } \neg \text{feasible} \land \text{offset} + t_0 > \text{LST}_i \text{ then} \\
24 & \quad \quad \quad \quad \quad \text{fail();} \\
25 & \quad \quad \quad \text{// Move forward} \\
26 & \quad \quad \quad \text{idx}_0 = \text{idx}_0 + 1, t_0 = \Omega[\text{idx}_0].\text{time}
\end{align*}
\]

The algorithm The pseudo-code is reported in Algorithm 1 where $S$ is the set of unscheduled activities. Lines 3–7 contain the algorithm initialization; the variable canStart represents the candidate start time of the schedulability window and initially assumes the value of the Earliest Start Time of the selected activity $i$. The feasible flag is used to remember if enough resource is available at the current canStart value. The $t_0$ and $idx_0$ values are respectively the time instant and the $\Omega$ index currently being processed. Initially, they are respectively equal to EST, and to the largest index in $\Omega$ corresponding to an item with $\Omega[idx].time \leq t_0$. Note that since profile changes always correspond to queue items, the resource usage at $t_0$ is the same as at $\Omega[idx_0].time$. The startIdx variable stores the first index in $\Omega$ examined by the algorithm, i.e. the first idx0. The stop flag is used for loop termination.

The algorithm searches for a schedulability window by scanning the queue $\Omega$, item by item. The window is assumed to start at the current canStart value and to end at $t_0$. Note that, with the exception of the first while-loop iteration (where $\Omega[idx_0].time$ may be strictly less than $t_0$), the current time point under examination (i.e. $t_0$) always matches the time value of the queue index under examination (i.e. $\Omega[idx].time$).

From line 9 to 14 the algorithms checks the resource availability at $t_0$. The value avRes computed at line 9 is the resource usage, adjusted by subtracting the contribution of the compulsory part of activity $i$. Line 10 checks if scheduling $i$ at time canStart would cause an over-usage at time $t_0$. Note that $r_q(canStart, d_i, t_0, \lambda)$ corresponds to the resource consumption of activity $i$ at $t_0$, assuming it is scheduled at time canStart. Depending on the result of the check, the feasible flag is updated. A transition between feasible = false to feasible = true means that a new candidate time for the schedulability window has been found and that the canStart value must be updated.

Since $\Omega$ is circular, at some point the process may reach the end of the queue and re-start from the first element: the filtering algorithm makes use of an offset value to distinguish between items encountered before and after crossing the end of the queue. In particular, for the latter ones it holds $idx_0 < startIdx$ and the offset is set to $\lambda$ (lines 10–11). We recall that $\lambda$ (i.e. the maximum value in the domain of $\lambda$) is the modulus value corresponding to the minimum resource usage profile (see Fig. 11).

At lines 17–19 the algorithm checks if the current schedulability window, from canStart to $t_0$: 1) has enough available resource and 2) it is long enough to contain activity $i$. The time window length is adjusted by adding the value offset. If both the conditions hold, then a valid schedulability window has been found and the start time is filtered (line 18). If this is not the case, then the algorithm checks if the current $t_0$ (adjusted with the offset value) has become larger than LST, (line 20–21); in this case there is no way to schedule activity $i$ and the algorithm fails. If neither of the two situations occurs, at line 22 the process moves to the next item in $\Omega$ by updating $t_0$ and $idx_0$.

Note that the filtering procedure guarantees that an activity can be scheduled at its earliest start time (i.e. has sufficient resources), as formalized in Expression (27).
40
A. Bonfietti et al. / Artificial Intelligence 206 (2014) 25–52

Fig. 11. Resource Profile of a partial allocation with minimum and maximum modulus.

Traditional cumulative constraint With the assumption that the activities are scheduled within the period (i.e. \(e_i < \lambda\)), as in the CROSS\(^*\) approach, the algorithm proposed achieves the same filtering of a traditional cumulative constraint, i.e. obtained through the timetabling or the sweep algorithm (respectively in [3] and [4]). In case this hypothesis is relaxed, the traditional cumulative constraint cannot be applied.

Time complexity The filtering algorithm has two nested cycles, respectively over the set of non-scheduled activities \(S\) and over the items of the circular queue \(\Omega\). The corresponding maximum numbers of iterations are \(n\) and \(2 \cdot n\). Hence the asymptotic complexity is \(O(n^2)\).

6.3.2. Core 2: modulus filtering algorithm

In cyclic scheduling, it is possible to reduce the cumulative usage at time \(t\) by increasing the modulus. As a consequence, unlike in classical scheduling, the compulsory parts in the current schedule may enforce a non-trivial lower bound on the feasible \(\lambda\). The goal of lambda filtering is to find the minimum period where sufficient resources are available for the current schedule. Formally:

\[
\lambda \geq \min_{\lambda' \in [0, \lambda]} \sum_{i \in V} r_q_i(s_i, d_i, t, \lambda') \leq \text{CAP} \quad \forall t \in [0, \lambda']
\]

(28)

The algorithm makes again use of a circular queue \(\Omega\). However, the value \(\Omega[idx].time\) of the items corresponding to the Earliest End Time is computed assuming that \(\lambda = \lambda_{\min}\). In other words, they are the Earliest End Time for the scenario where the period takes its minimum value, corresponding to the most constrained resource profile.

Fig. 11 shows two different resource usage profiles for the same activity start times, but with different period value. The former corresponds to the maximum \(\lambda\) value (i.e. \(\lambda_{\max}\)) and the latter corresponds to the minimum \(\lambda\) value (i.e. \(\lambda_{\min}\)). Note that with \(\lambda_{\min}\) the activities \(A\) and \(B\) now cross the modulus, increasing the resource consumption at time 0. This causes a resource over-usage, represented by the shaded area. The effect of the period \(\lambda\) on the usage profile is a direct consequence of how the resource usage function \(r_q(s, d, t, \lambda)\) is defined (see Section 4.1). The goal of the modulus filtering algorithm is to get rid of the over-usage by increasing the lower bound \(\lambda_{\min}\). This is done in an iterative fashion, by repeatedly computing the over-usage integral and pushing the \(\lambda\) lower bound ahead.

The filtering algorithm The pseudo-code for the filtering procedure is reported in Algorithm 2. It is an iterative process, repeated until the resource over-usage becomes 0.

The cumulative resource over-usage at each iteration is referred to as \(E_n\). At the beginning of each iteration (lines 3–5), the algorithm updates and reorders the data structure \(\Omega\): this is necessary since \(\lambda_{\min}\) is changed at each iteration, causing a modification of all the modular end times. Moreover, the procedure resets the values \(\text{lastres}\) and \(\text{lasttime}\), respectively referring to the last processed resource consumption and to the corresponding time point.

Then the items in the queue are processed one by one. At lines 9–12 the resource consumption is increased or decreased depending on whether the item corresponds to a start or an end time. At line 13, the procedure checks if the resource consumption of the last processed time point (i.e. \(\text{lastres}\)) exceeds the resource. In this case, the cumulative resource over-usage on the time window \([\text{lasttime}, \Omega(idx).time]\) is summed to the current \(E_n\) quantity. At line 15 the algorithm updates the \(\text{lastres}\) and \(\text{lasttime}\) values, before starting to process the next item. The period lower bound is updated at the end of each iteration (line 16), by dividing the cumulative over-usage by the capacity of the resource and summing the quantity to the current \(\lambda_{\min}\).

In the following we provide the correctness and termination proofs. Three important reminders for the proofs are:

- the resource consumption of a compulsory part is a decreasing function of the period (see Eq. (26)).
- The algorithm works with partial solutions as not all activities have been scheduled.
The resource profile is defined as the cumulative usage due to the compulsory parts of all the activities. We recall that the profile depends on the value of $\lambda$ and on the start time variable domains (see Eq. (28)).

**Correctness proof** Let us assume that $\lambda$ is a real number (i.e. $\lambda \in \mathbb{R}$). At each algorithm step a new bound is computed:

$$\lambda \leftarrow \lambda + \frac{E}{\text{cap}}.$$

The algorithm is correct if the last computed bound is valid. A bound $\lambda$ is valid if for any lower $\lambda$ value the resource profile exceeds the resource capacity (i.e. $\lambda < \lambda \Rightarrow$ infeasible profile).

The algorithm is therefore correct if at each step the computed offset (i.e. $\frac{E}{\text{cap}}$) is the minimum bound shifting that might get rid of the resource over-usage.

By shifting the bound $\lambda$ by an offset $\Lambda$, a new resource area $E$ is introduced in the schedule: formally $E = \Lambda \cdot \text{cap}$. This shifting might reduce the over-usage energy $E$ and obviously the maximal reduction is $E$. Note that if $E \leq E$, then $E = \Lambda \cdot \text{cap}$. We can therefore compute the minimum offset which corresponds to $\lambda = \frac{E}{\text{cap}}$ (from $E = \Lambda \cdot \text{cap}$).

Finally, as $\lambda$ is actually an integer variable, the ceiling function $\lceil \frac{E}{\text{cap}} \rceil$ guarantees to find the minimum integer valid bound. The algorithm is therefore correct.

**Termination proof** First, observe that the offset $\Lambda$ computed at step $k$ is never greater that the offset at step $k+1$.

Now, the Algorithm terminates if (1) it finds a valid bound or (2) it proves an infeasibility. Note that an infeasibility means that no valid bound exists in the domain of $\lambda$, i.e. that even the value $\lambda$ corresponds to an infeasible resource profile. However, this condition is already checked by Algorithm 1, meaning that, when Algorithm 2 executes, the profile for $\lambda = \lambda$ does not exceed the resource usage. We can therefore safely disregard termination condition 2. Moreover, we know that the computation will converge to some bound value $\lambda^* \leq \lambda$.

However, assuming a real $\lambda$ variable, it may happen that the computation takes an infinite number of steps. This may occur in case the computed offset $\Lambda$ tends to zero but never becomes null. Now, it is sufficient to assume a finite precision for the computation to converge. Since in practice we model $\lambda$ with an integer variable the termination proof holds.

**Time complexity** The algorithm inner loop has two main steps:

- Updating and sorting the modular end time for items in $\Omega$, with asymptotic complexity $O(n \cdot \log n)$.
- Computing the cumulative over-usage. This takes up to $2 \cdot n$ steps, and thus has asymptotic complexity $O(n)$.

It is difficult to obtain a tight bound on the number of iterations of the main loop (lines 2–16). As $\lambda$ is an integer, then $\lambda \leftarrow \lambda + \frac{E}{\text{cap}}$ provides a trivial bound and the overall asymptotic complexity is $O((\lambda - \lambda) \cdot n \cdot \log n)$. In practice, however, the number of iterations of the main loop is very small.
7. Search

In this section, we introduce two search strategies tailored for two models of the problem. The first (called Path-based method) works for the restricted case where the end time is assumed to be within the modulus, $\beta_i = \eta_i \quad \forall i \in V$. The second (called random restart method) is used in the generic version and it is based on restarts, although in this case they are employed to ensure correctness rather than to speed up search. As discussed in Section 5, by constraining the end times to be within the modulus, we can model the resource constraints with traditional cumulative constraints.

In this section we present both the search strategies, together with their variable and value selection heuristics. Finally we also present a dominance rule on $\beta$ variables used to narrow the search space.

7.1. Path-based method

This is the search strategy employed in CROSS*. Since our approach is designed to build periodic schedules, the start time value of each activity can be decided with respect to another, arbitrarily chosen, reference activity. The $s_i$ and $\beta_i$ variable of this reference activity (called The reference node) can be fixed to zero. Such a restriction does not compromise the method completeness, as long as the $\beta_i$ variable domains are large enough. We always choose as a reference activity a node with no in-going arc having $\delta(i,j) = 0$. Formally, for a node src to be a candidate, it must hold $\exists β(i, src), δ(i, src) \in \mathcal{A}$ such that $\delta(i, src) = 0$. Note that there always exists at least one node with this property, otherwise the graph is in deadlock and the problem is infeasible. A formal proof can be found in [5].

The reference activity (let its index be src) is always immediately scheduled, by posting $s_{src} = 0, \beta_{src} = 0$, with no backtrack possible. Then, at any search node the next activity $j$ to be used for branching is chosen among those connected by an arc $(i, j)$ to one of the activities $i$ already scheduled. Those candidates are ranked according to the number of outgoing arcs (the more, the better). This selection criterion seems to lead to better filtering of the Modular Precedence Constraint and for the dominance rule described in Section 7.3. In case of ties, the algorithm prefers activities with longer duration, and finally with smaller index.

Then we assign a value to the start time and to the iteration number of the selected activity, i.e. to variables $s_j$ and $\beta_j$. This is done in two successive search steps, rather than by posting the conjunction of the two assignments as a single constraint.

In particular, the algorithm attempts to schedule the activity at its earliest start time, i.e. it assigns $s_j$ to its minimum value. On backtrack, the activity is postponed, i.e. marked as non-selectable until its earliest start time is modified by propagation. This is analogous to what it done in the classical schedule-or-postpone strategy for the RCPSP [23] and can be done since in CROSS* we enforce resource restrictions by means of traditional cumulative constraints.

The $\beta_j$ variables are assigned via labeling, with value order given by increasing $|v|$ (where $v$ is the value to be assigned). In case of ties, the positive $v$ is chosen. This criterion is based on the existing correlation between iteration numbers and schedule length (see Section 4) and is designed to produce schedules as short as possible. A deeper discussion on the values that the iteration variables could assume can be found in Section 7.3. On backtrack, the previously assigned value is simply removed from the domain of $\beta_j$.

Unlike other modulo scheduling approaches, our solution method does not require a-priori fix the period. When all the activities have been scheduled, we simply assign to the $\lambda$ variable the minimum in its domain. This value is guaranteed to be feasible by the propagation of precedence and resource constraints. Optimization is performed in a traditional CP fashion by posting a permanent upper bound on $\lambda$ whenever a feasible solution is found.

7.2. Random Restart method

If the activities are allowed to cross different iterations, like in our generic CROSS approach, the modular starting times depend non-monotonically on $\lambda$. As a major consequence, updating the upper bound of $\lambda$ may cause some values that were previously inconsistent to become consistent (or vice versa).

Consider the following example (see Fig. 12): we have two activities A and B whose duration is 9 and 3 respectively. There is a precedence constraint between A and B with $\delta = 0$ and a precedence constraint between B and A with $\delta = 1$. A is scheduled with $s_A = 0$ and $\beta_A = 0$. Suppose also that $\bar{\lambda} = 6$, which implies that the modular end time of $A$ is $3$. Then the feasible values for $s_B$ are 3, 4, 5. Now suppose we update $\bar{\lambda} = 4$. The modular end time of $A$ becomes 1 and therefore, we have that values 1 and 2, that were previously infeasible for the modular start time of $B$, are now feasible.

For overcoming this issue, we have decided to restart the search from scratch any time the upper bound of $\lambda$ is updated.

The structure of the search strategy is the same as before: an activity $j$ is selected, then a value is assigned to its start time and iteration variables (i.e. $s_j$ and $\beta_j$) in two successive search nodes. Once all the activities have been scheduled, the $\lambda$ variable is fixed to $\bar{\lambda}$. The iteration numbers are assigned as in CROSS*, giving priority to values $v$ in the domain having the minimum absolute value $|v|$ and posting $\beta_j \neq v$ on backtrack.

As a first difference, we do not fix a reference activity. At every search node, the activity $j$ to be scheduled is chosen at random among those connected to at least one already scheduled activity $i$. At the root node (where no scheduling decision has been taken) the choice is simply made at random.
Second, the selected activity is not scheduled at EST_i. Instead, the algorithm keeps a list of candidate start times, corresponding to the values s_i + (d_i mod λ) for all the already scheduled activities.\(^7\) The value 0 is always added to the list in case it is not present. Then the selected activity j is scheduled at the smallest candidate start time t falling in the interval [EST_j, LST_j]. If there is no such t, the algorithm backtracks without even opening a choice point. Otherwise, the activity j is scheduled at time t and on backtrack the constraint s_j > t is posted.

7.3. Dominance rules

One important observation is that assigning different iteration values to activities connected by an arc (i, j) allows to place activity j before activity i in the λ-long window, apparently violating the precedence constraint. In particular, this requires to β_j to be greater than β_i by a minimum amount: any further increase obtains the same effect. This observation serves as the basis to devise a dominance rule that filters redundant values for the β variables.

Suppose we want to schedule two activities i and j, connected by a temporal dependency (i, j, θ_{i,j}, δ_{i,j}). From (12) we derive:

$$\beta_j \geq \beta_i - \delta_{i,j} + \left[\frac{s_i + d_i - s_j + \theta_{i,j}}{\lambda}\right]$$

and let $\Delta_\beta = \frac{s_i + d_i - s_j + \theta_{i,j}}{\lambda}$. Let’s focus on the successor j. We analyze the dependency in Eq. (29) in two different cases:

1. Activity j follows activity i: in this case $s_j \geq s_i + d_i + \theta_{i,j}$. Since $s_j < \lambda$, $\Delta_\beta$ is in the interval $]-1,0]$. Hence, $\lceil \Delta_\beta \rceil = 0$ and then $\beta_j \geq \beta_i - \delta_{i,j}$.

2. The two activities overlap in the λ-long window, or activity j precedes activity i: in this case $s_j < s_i + d_i + \theta_{i,j}$ and $\Delta_\beta$ is strictly positive. Therefore, the lowest possible value that $\lceil \Delta_\beta \rceil$ can assume is 1, which implies $\beta_j > \beta_i - \delta_{i,j}$.

Note that $\beta_j$ must be at least equal to $\beta_i - \delta_{i,j}$, and must be strictly greater if we want activity j to overlap with or precede activity i in the time window. In particular, the smallest value that allows the apparent violation of the precedence constraint is $\beta_i - \delta_{i,j} + \lceil \Delta_\beta \rceil$: any higher value provides no benefit from a schedule building perspective. This information can be used to devise a dominance constraint that removes dominated (but feasible) values from the domain of β variables. In detail, we can post:

$$\beta_j \leq \max_{i \in \text{PRECE}} \left(\beta_i - \delta_{i,j} + \left[\frac{s_i + d_i - s_j + \theta_{i,j}}{\lambda}\right]\right)$$

where $\text{PRECE} = \{i : \{i, j, \theta_{i,j}, \delta_{i,j}\} \in A\}$ is the set of the predecessors of j. Note that, as a consequence of the combination of the modular precedence constraints (i.e. Eq. (29)) and the dominance rule, $\beta_j$ is strongly constrained by the $\beta_i$ of the predecessor activities: this is the reason why scheduling activities according to their topological order may result in a much better propagation (see Section 7.1).

We can obtain a second dominance rule by reasoning in a similar fashion for the successors of an activity i:

$$\beta_i \geq \min_{j \in \text{NEXT}_i} \left(\beta_j + \delta_{i,j} - \left[\frac{s_i + d_i - s_j + \theta_{i,j}}{\lambda}\right]\right)$$

where $\text{NEXT}_i = \{j : \{i, j, \theta_{i,j}, \delta_{i,j}\} \in A\}$ is the set of the successors of i.

\(^7\) Note that the list corresponds to the set of all feasible start times coinciding with the modular end times of the scheduled activities. These instants correspond to the start times of all the feasible schedulability windows of the considered activity (see Section 6.3.1 for further details).
To make the rules clearer, we make a simple example. Assume we have a graph with three activities $A, B, C$ connected with two arcs: $(A, B, 0, 0)$ and $(B, C, 0, 0)$. Assume that each variable is initially unbound, with $s_i \in [0, \lambda]$ and $\beta_i \in [-3..+3]$ for every activity. If we schedule activity $A$ by posting $s_A = 0$ and $\beta_A = 0$, then the $\beta_i$ domain for all other activities becomes $(0..+3)$. The application of the dominance rule (30) results in a further reduction of the domains, in particular $\beta_B \in [0..+1]$ and $\beta_C \in [0..+2]$. Note that the values $\beta_B = 2$ and $\beta_B = 3$ are feasible but redundant. At this point, if the activity $B$ is scheduled so that it overlaps with $A$ (i.e. $s_B < s_A + d_A$), its iteration variable is be forced to $\beta_B = 1$.

From an implementation perspective, the direct formulation of the rules may be difficult to model and heavy to propagate. It is therefore convenient to modify the right hand side expression in Inequality (30) with an upper bound, obtaining the following, updated rule:

\[
\beta_j \leq \max_{i \in \text{PRE}} \left( \beta_i - \delta(i, j) + \left\lceil \frac{\lambda + d_i + \theta(i, j)}{\lambda} \right\rceil \right)
\]

i.e.

\[
\beta_j \leq \max_{i \in \text{PRE}} \left( \beta_i - \delta(i, j) + 1 + \left\lceil \frac{d_i + \theta(i, j)}{\lambda} \right\rceil \right)
\]

For a fixed $\lambda$ value, the expression $\lceil \frac{d_i + \theta(i, j)}{\lambda} \rceil$ can be replaced with a fixed value to obtain a (much simpler to compute) upper bound on $\beta_j$. This technique is employed in the context of the CROSS search strategy, which works by restarting the search whenever the $\lambda$ upper bound is modified.

8. Experimental results

We have evaluated the effectiveness and scalability of both the approaches discussed in Section 5 (CROSS and CROSS*) on various sets of instances. Moreover, we have compared them against a state-of-the-art ILP method and a state-of-the-art heuristic.

In the first part of this section we will focus on the CROSS* approach: we show that the solver is able to compute good quality solutions in terms of seconds in the context of a real world problem. In the second part of the section we present a comparison between CROSS and CROSS*, plus a blocked and an unfolded approach. More in detail, this section contains:

1. An experimentation on industrial instances: the main purpose of this experimental evaluation is to show the our CROSS* approach to cyclic scheduling is viable in a practical setting. In particular, the experimentation will focus on instruction scheduling benchmarks, namely an industrial set of 36 instances for the ST200 processor by STmicroelectronics (also employed in [2]). We compare CROSS* with two state-of-the-art approaches: the complete ILP method presented in [2] and Swing Modulo Scheduling (SMS), a heuristic used by the GCC compiler [18].

2. An evaluation of the solution quality obtained by the CROSS* approach: the second group of tests is performed on a set of synthetic instances and compares the best solution obtained by CROSS* (within 300 s) with a lower bound on the instance period (namely, the iteration bound from Section 2.3.1).

3. A comparison between CROSS/CROSS* and blocked/unfolded scheduling: this experimentation is performed on a set of synthetic instances, for which we compare the solution quality of both the approaches with respect to classic blocked and unfolded scheduling.

4. A comparison between CROSS and CROSS*, w.r.t. the Throughput / Resource trade-off: the fourth group of results compares our two approaches in terms of Throughput / Resource-usage trade-off.

All our solvers have been implemented in IBM ILOG Solver and Scheduler 6.7. All the experiments are performed on an Intel Core 2 Duo 3.3 GHz with 8 GB of RAM. A 300 s time limit was set on each run. The synthetic instances were built by means of an internally developed task-graph generator, designed to produce graphs with realistic structure and parameters.

8.1. Evaluation of CROSS* on industrial instances

The first set of 36 instances is obtained from a compiler for a VLIW (Very-Long Instruction Word) architecture. In this setting, cyclic scheduling problems arise when optimizing inner loops at instruction level: in this context the activities represent instructions, the precedence constraints model data and control dependencies and the resources are the hardware units required to execute the instructions. In the considered instances, all the resources have unary capacity and all the instructions have unary duration. With the objective to evaluate the approaches on a more diversified setting, in [2] the authors have obtained an additional set of modified instances, by replacing the original resource capacities and consumptions with randomly generated integer values.

The smallest instance features 10 nodes and 42 arcs, while the largest one has 214 nodes and 1063 arcs. In [2] the authors present two ILP formulations for the resource-constrained modulo scheduling problem (RCMSP), which is equivalent to the CRCSP discussed in this paper, with the (easy to enforce) exception that the period must be integer. As described in [2], both the ILP approaches adopt a dual process by iteratively increasing an infeasible lower bound; as a consequence, the method does not provide any feasible solution before the optimum is reached. Given a large time limit (604800 s)
their solvers found the optimal solution for almost all the instances: our experiments compare the optimal value with the solution found by our method within a 300 s time limit.8 We also compare our approach with an incomplete method, namely a state of the art heuristic approach called Swing Modulo Scheduling (SMS), presented in [26] and used by the gcc compiler [18].

Note that the ILP and the SMS approaches were developed to tackle instruction scheduling problems; hence both work with activities with unary duration.

As we mentioned, our benchmarks consists of two subsets of instances: the original problems extracted by the compiler and the modified ones (which tend to be more challenging). The results of the experiments performed with both the sets are reported respectively in Table 1 and Table 2. In the tables the first three columns describe the instance (name, number of nodes and arcs), the third shows the run-time of the ILP approach in [2], the fourth and the fifth respectively report the solution time and the quality of our solutions, evaluated by means of the gap w.r.t. the optimal solution.9 The last two columns present the solution time and the quality gap for the SMS heuristic approach. For some instances (see the missing time information) the ILP approach was not able to find a solution: in such cases we report the best solution found by our method within a 300 s time limit.8 We also compare our approach with an incomplete method, namely a state of the art heuristic approach called Swing Modulo Scheduling (SMS), presented in [26] and used by the gcc compiler [18].

Note that the ILP and the SMS approaches were developed to tackle instruction scheduling problems; hence both work with activities with unary duration.

As we mentioned, our benchmarks consists of two subsets of instances: the original problems extracted by the compiler and the modified ones (which tend to be more challenging). The results of the experiments performed with both the sets are reported respectively in Table 1 and Table 2. In the tables the first three columns describe the instance (name, number of nodes and arcs), the third shows the run-time of the ILP approach in [2], the fourth and the fifth respectively report the solution time and the quality of our solutions, evaluated by means of the gap w.r.t. the optimal solution.9 The last two columns present the solution time and the quality gap for the SMS heuristic approach. For some instances (see the missing time information) the ILP approach was not able to find a solution: in such cases we report the best solution found by CROSS*, which is used as a reference for computing the SMS gap.

On the industrial set, CROSS* is able to find the optimal solution within one second for all but one instance (adpcm-st231.2, whose optimality gap is 2.44%). We also found a solution for the gsm-st231.18 instance that was previously unsolved. The SMS heuristic method obtains an average optimality gap of 14.58%. Its average run time is less than few seconds, with a maximum of 58.36 s, corresponding to instance gsm-st231.18. For the previously unsolved instance (i.e. gsm-st231.18), the gap between SMS and CROSS* is 3.8%.

8 In a preliminary experimentation the solver was found to typically provide the best solution in less than a second. Hence a 300 s time limit should be widely sufficient for the method to converge to best solution it can find.

9 For both CROSS* and SMS, the quality gap is computed with the following formulation: 100 * (Sol – ILP_Opt)/ILP_Opt, where ILP_Opt ≤ Sol.
the highest computation time corresponds to instance gsm-st231.18, but it is considerably larger compared to that of the industrial set (i.e. 429.36 s).

Finally, note that for the instances gsm-st231.25 and gsm-st231.33 no optimal solution is available, since in [2] the authors were only able to find a suboptimal schedule within the time limit of 604,800 s (no details are given on how the dual process they propose converges to sub-optimal solutions). On the instance gsm-st231.25 both the modular and the SMS approaches find the same solution as [2]. On gsm-st231.33 the best period value found in [2] has value 52 and the SMS solver finds the same solution; our method however finds in one second a solution with value 47 and within 56 s a solution of value 46.

Discussion on the optimality proof The results of our experimentation show that CROSS* is able to find very good solutions in a very short time (a few seconds). The method appears however much less effective in proving optimality: despite the optimal solution was found on most instances, an optimality proof was achieved only for 12.5% of them within the time limit.

This is mainly due to the fact that cyclic instances are frequently characterized by a large number of precedences having high repetition distance $\delta$ values. Unfortunately this makes the precedence constraints propagation less effective (see Section 6.1).

As formulated in Section 2.3.1 a cycle $c$ with a high $D(c)$ (we recall that $D(c) = \sum_{(i, j) \in A(c)} \delta(i, j)$) has less impact on the iteration bound than a cycle with a lower $D(c)$.

A simple example is considering two activities $i, j$ connected in a cycle with: $D(c) = 1$, or $D(c) > 1$ or without precedences. Let also the activity $i$ be scheduled. In the first case (where $D(c) = 1$) the precedence constraint propagation has a strong impact forcing the activity $j$ to be scheduled after the end of $i$. In the other two cases (where $D(c) > 1$ or without precedences) the precedence constraint cannot filter on the start time domain of activity $j$. In fact the two activities might overlap their executions with different iteration values (i.e. $\beta$).
Table 3
Solution quality.

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>avg (%)</th>
<th>best (%)</th>
<th>worst (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.706%</td>
<td>2.28%</td>
<td>5.18%</td>
</tr>
<tr>
<td>2</td>
<td>3.68%</td>
<td>2.105%</td>
<td>5.04%</td>
</tr>
<tr>
<td>5</td>
<td>3.51%</td>
<td>1.81%</td>
<td>5.015%</td>
</tr>
<tr>
<td>10</td>
<td>3.37%</td>
<td>1.538%</td>
<td>4.98%</td>
</tr>
<tr>
<td>60</td>
<td>3.14%</td>
<td>1.102%</td>
<td>4.83%</td>
</tr>
<tr>
<td>300</td>
<td>2.9%</td>
<td>0.518%</td>
<td>4.73%</td>
</tr>
</tbody>
</table>

Fig. 13. Graphical representation of the optimality gap improvement over time.

To conclude, high $D(c)$ values mean that most of activities of $c$ can overlap their executions. There is therefore a strong analogy with those instances and acyclic scheduling problems with few precedence constraints (or even no precedences). It is well known that CP approaches based on start time variables and cumulative constraints have difficulties in proving optimality for this class of problem instances.

We are currently investigating how to improve the efficiency of the proof of optimality through no-good learning. Note however that the optimality gap is so small that the method is very appealing even if used as a heuristic.

8.2. Evaluating CROSS* solution quality on synthetic benchmarks

The second set of experiments targets a task scheduling problem over a multi-processor platform. The benchmark in this case contains 1200 synthetic instances with 20 to 100 activities, cycles in the graph and high concurrency (i.e. the precedence constraints allow many activities to run in parallel). In a preliminary experimentation, this kind of graph structure appears to be the toughest to solve for our approach. The set of processors on the platform is represented in the generated instances as a cumulative resource with capacity 6 (representing the number of parallel threads). All activities have unary resource consumption.

Table 3 shows the average, best and worst gap over time between the CROSS* solution and a lower bound, computed as:

$$lb = \left\lceil \max \left( IB, \frac{\sum_{i \in V} d_i}{\text{cap}} \right) \right\rceil$$

that is the maximum between the iteration bound $IB$ (see Section 2.3.1) and the ratio between the sum of the execution times and the total capacity.

As one can see in the table, CROSS* finds a solution which is about 3.7% distant from the lower bound value within one second. After 300 s the gap has decreased to 2.9%. Hence, even on this new instance set the solver is able to find a high quality solution very quickly. After that, improvements are obtained much more slowly. Fig. 13 provides a more detailed view of the progress of the gap value. Once again, we can conclude that our approach converges very quickly to period values close to the lower bound. The actual optimum lies somewhere in-between the two values and therefore is even closer.

8.3. CROSS and CROSS* vs blocked and unfolding scheduling

Our third experimentation contains in first place a comparison between our approach, producing overlapped schedules, and the blocked approach that takes into account a single iteration. Since the problem is periodic and the schedule is iterated indefinitely over time, the latter method pays a penalty in the quality of the schedule obtained. A technique often
employed to address this limitation and to allow inter-iteration parallelism is unfolding (see Section 3). Unfolding often leads to improved blocked schedules, at the cost of an increase of the instance size. In detail, we report the results for two separate sets of experiments, respectively evaluating the CROSS\(\ast\) and CROSS method.

First set of experiments: The first experimentation is performed on a set of 220 instances, divided into three classes: small instances (14 to 24 activities), medium-size instances (25 to 44 activities) and big instances (45 to 65 activities). We compared our approach with the blocked one and with unfolded scheduling, using seven different unfolding factors (referred to as Unfold\(X\), where \(X\) is the number of unrolled iterations).

Table 4 shows the average gap between the above mentioned configurations and CROSS\(\ast\). The last column presents the average optimality gap over the whole experimental set. As expected, the worst gap is relative to the blocked schedule. Less obviously, the Unfold\(X\) configurations tend to have an oscillatory behavior. Fig. 14 depicts the relation between the gap (Y axis) and the blocking factor (X axis) for a selected instance with 30 activities. With an unfolding factor \(u = 11\) the solver finds a solution with the same period as the overlapped one. Increasing the unfolded factor to 12 deteriorates, rather than improving, the solution quality. In general (see Section 3.2), it is false that increasing \(u\) necessarily results in better solutions, making the determination of the optimal unfolding factor a non-trivial problem.

Furthermore, there are cases where finding a schedule with the same quality of a periodic approach is not possible at all: such an example is provided in [33]. On the other hand, we recall that periodic schedules are dominated by K-periodic schedules, i.e. periodic schedules for a sequence of consecutive iterations. K-periodic schedules are known to dominate 1-periodic schedule in the presence of finite capacity resources [19]. Therefore, since the unfolding technique produces a restricted class of K-periodic schedules, there may be cases where an unfolded schedule is strictly better than any possible 1-periodic schedule. In summary: no strict dominance exists between the two approaches. Our experimentation shows however that, on the considered benchmarks, periodic schedules tend to be much better in practice.

Second set of experiments: The second experimentation is performed on a group of 200 synthetically generated project graphs with 5 to 25 activities. The resource capacity is fixed to 5 times the average consumption. The durations are randomly generated so that around 10% of the activities in each graph are much longer than the remaining 90%. We compare the CROSS approach with blocked and the unfolded scheduling, for unfolding factors up to 5 (this value is sufficient to see the overall trend). We use the classical (and very effective) schedule or postpone search strategy for the blocked and the unfolded approach. For the CROSS one, we couple the Random & Restart method from Section 7.2 with a binary search on \(\lambda\) (using a time limit of 2 s on each iteration).
Table 5 shows the result of the comparison, grouped by the number of activities in the graph. In particular, the whole set of 200 instances has been partitioned into two subsets, respectively with 5–14 and 15–25 activities. The Time column shows the average time taken by the CROSS approach to reach the best solution found within 300 s. The table then reports, for all the considered approaches, the (average) quality gap w.r.t. the CROSS solution (i.e. $\frac{\lambda - \lambda_{CROSS}}{\lambda_{CROSS}}$) and an (average) $\Delta$ time value, representing the difference between the time taken by the considered approach and by CROSS to reach the best solution found within 300 s.

The results show the existence of a relevant trade-off between solution time and quality. The blocked scheduling approach represents one of the extremes, providing its best solution in a fraction of second, but with a quality gap as large as 216%. Increasing the unfolding factor leads to better $\lambda$ values, at the cost of increased solution time. The Unfold5 configuration eventually manages to produce better solutions compared to CROSS (1.3% improvement), but the process is up to 6 times longer. Our approach seems to provide a very good compromise, providing high quality solutions in a short amount of time.

8.4. Throughput/resource trade-off investigation

Cyclic scheduling allows to improve the resource efficiency by partially overlapping different schedule iterations. In particular, it is possible to exploit the available resources to reduce the period, even when the makespan cannot be further minimized (e.g. due to precedence constraints). Loops in the project graph limit the degree of such an improvement (see Section 2.3.1): if the graph is acyclic, however, the throughput can be arbitrarily increased by augmenting the resource capacity. A large number of practical problems (e.g. in VLIW compilation or stream computing) are described by project graphs with a few cycles or no cycle at all. In such a case, identifying the optimal throughput/resource-usage trade-off is the primary optimization problem.

We performed an experimentation on two groups of 20 synthetically generated instances, respectively consisting of cyclic and acyclic graphs. Durations are again unevenly distributed, as described in Section 8.3. In order to investigate the throughput/resource trade-off, we solved a set of period minimization problems with different resource availability levels. In detail, we considered a single resource and activities in each graph were labeled with random resource requirements, following a normal distribution. The resource capacity (referred to as CAP) ranges between 4 times and 14 times the average consumption level: the minimum value is chosen so as to guarantee the problem feasibility, while the maximum one is chosen to assess the solution quality in case of abundant resources.

Fig. 15 shows the average period after 300 s (the time limit was hit in all cases) obtained for different resource capacity values. The vertical bars report the corresponding standard deviation. The solid line corresponds to the CROSS approach and the dashed one to CROSS$. Note that CROSS obtains considerably better results for higher capacity values, i.e. the scenario
when we expected the highest benefit from allowing activities to cross iterations. The gap is larger for acyclic graphs, where the lack of loops enables to fully exploit the available resources.

The iteration period difference corresponds to a much larger gap in terms of “wasted” resource capacity, which can be assessed by measuring the value of the expression:

$$\text{slack} = \text{CAP} \cdot \lambda - \sum_{i \in \mathbb{V}} r_i \cdot d_i$$

The average slack values for this experimentation are reported in Fig. 16, where the overall resource waste is shown to grow according to a roughly quadratic law for the CROSS* approach. The growth is much slower for CROSS (in fact, it is approximately constant for acyclic graphs). The amount of wasted resource capacity is an important measure of how efficiently the resources are used and in a practical setting directly translates to platform/machine costs.

Interestingly, the two approaches have comparable performance for small capacity values. This suggest that the time limit is not severely limiting the search effectiveness. This is a relevant remark, since we expected CROSS to be considerably slower in finding good solutions. More details are reported in the histograms from Fig. 17, that show the instance count, grouped by the time (in seconds) employed by each method to get 1% close to the final best solution. As one can see, CROSS* is indeed faster on average, but both methods manage to provide high quality schedules in seconds.

9. Conclusions

We have proposed a constraint approach to solve cyclic scheduling problems, based on modular arithmetic. In particular, we have devised global constraints to model temporal dependencies (the Modular Precedence Constraint, ModPC) and resource restrictions (The Global Cyclic Cumulative Constraint, GCCC). For both of them, we devised original filtering
algorithms. We have also described a restricted version of our solver where we rely on a specific assumption to model the resource restrictions via traditional cumulative constraints, rather than via the GCC. The general instantiation of our approach is referred to as CROSS, while its restricted version as CROSS∗.

The approach has been tested extensively on industrial as well as synthetically generated instances. Our method was able to return solutions very close to the known optimum or to a lower bound in a matter of seconds (or a fraction of second), on problems of practical size. Depending on the considered benchmarks, our approach either outperformed existing heuristic or unfolding-based techniques, or provided a very good compromise between solution quality and time. All this makes our method one of the best available solvers for cyclic scheduling problems with cumulative resources.

Future research directions include improving the effectiveness of the proof of optimality, which is currently a weak point of the method. A second, very interesting, research topic concerns the design of a more effective search strategy for the unrestricted CROSS approach. Since the start times assigned by our Random & Restart strategy depend on the period upper bound λ, they tend to lead to solutions with period quite close to λ itself, thus making the optimization process slower. This could be addressed by exploiting ideas from the Precedence Constrain Posting technique [29,34], developed for non-cyclic scheduling problems.

Acknowledgements

We thank Maria Ayala and Christian Artigues for the useful discussions and the industrial benchmarks. The work described in this publication was supported by the SMECY Project, JTI ARTEMIS, Grant agreement No.: 100230.

References

A. Bonfietti et al. / Artificial Intelligence 206 (2014) 25–52


Alessio Bonfietti is a Post-Doctoral Fellow at DISI, University of Bologna, Italy. He received his Bachelor’s Degree in 2005 and his Master’s Degree in 2007, both in Computer Science from the University of Bologna. Since the graduation, his research interest is in the area of Mapping and Scheduling of periodic applications on multi-core architectures and in the area of Constraint Programming. He has published papers and performed reviews for international conferences and journals.

Michele Lombardi is a Post-Doctoral Fellow at DISI, University of Bologna; his research activity is related to Constraint Programming and its integration with Integer Programming and Artificial Intelligence techniques; in particular, his focus is on resource allocation and scheduling problems. Michele Lombardi got a PhD in Computer Engineering at University of Bologna, performing internships at EPFL Lausanne (CH) and Cornell University (NY). He has published papers and performed reviews for international conferences and journals; he is in the program committee of IJCAI 2011 and CPAIOR 2011.

Luca Benini is Full Professor at the Department of Electrical Engineering and Computer Science (DEI) of the University of Bologna. He also holds a visiting faculty position at the Ecole Polytechnique Federale de Lausanne (EPFL) and he is currently serving as Chief Architect for the Platform 2012 project in STmicroelectronics, Grenoble. He received a Ph.D. degree in electrical engineering from Stanford University in 1997. Dr. Benini’s research interests are in energy-efficient system design and Multi-Core SoC design. He is also active in the area of energy-efficient smart sensors and sensor networks for biomedical and ambient intelligence applications.

He has published more than 500 papers in peer-reviewed international journals and conferences, four books and several book chapters. He is a Fellow of the IEEE, a member of the Academia Europea, and a member of the steering board of the ARTEMISIA European Association on Advanced Research & Technology for Embedded Intelligence and Systems.

Prof. Michela Milano, PhD in Computer Science, is associate professor at the University of Bologna since 2001. Her research interest is in the area of Constraint Programming and its integration with Integer Linear Programming. In these field Michela Milano has achieved international visibility. She is program chair of CPAIOR 2005 and CPAIOR 2010. She has published more than 100 papers on peer reviewed international journals and conferences. She edited two books on Hybrid Optimization. She is Area Editor of INFORMS Journal on Computing, Constraint Programming Letters and member of the Editorial Board of the Constraint International Journal. She is project coordinator of the EU-FP7 project e-POLICY: Engineering the Policy Making like cycle, that fosters the use of optimization and decision support techniques for aiding political decision making and impact assessment.