Explaining Producer/Consumer Constraints

Andreas Schutt and Peter J. Stuckey

Decision Sciences, Data61, CSIRO, Australia, and Department of Computing and Information Systems, The University of Melbourne, Victoria 3010, Australia
{andreas.schutt,peter.stuckey}@data61.csiro.au

Abstract. Resource-constrained project scheduling problems are one of the most studied scheduling problem, and constraint programming with nogood learning provides the state-of-the-art solving technology for them, at least when the aim is minimizing makespan. In this paper we examine the closely related problem of scheduling producers and consumers of discrete resources and reservoirs. Producer/consumer constraints model consumable resources, such as raw materials (*e.g.*, water) and money, in which event times relate to a production or consumption event. In this paper, we investigate what is the most appropriate language of learning: should we learn about the event times for production and consumption, or should be instead learn about the temporal relationships between events? For this reason, we explore global constraint propagators with explanation for producer/consumer constraints and contrast this with simple decomposition approaches. Experiments on resource-constrained project scheduling problems involving producer/consumer constraints show that nogood learning solvers are highly effective at these problems.

1 Introduction

One of the most-studied and well-known scheduling problem is the resource-constrained project scheduling problem (RCPSP) that comprises non-preemptive activities, scarce renewable resources, and precedence relations between pairs of activities. A schedule determines the start and end time of activities such that no resource limit is exceeded over the planning horizon and all precedence relations are satisfied. In RCPSP, an activity *consumes* some units of the renewable resource and releases or *produces* them at its ends. Renewable resources are very common and model, *e.g.*, manpower and machines, whereas non-renewable resources model, *e.g.*, money and energy, and have a fixed initial capacity, *i.e.*, an activity only consumes them. Reservoirs (also called inventories [15]) are consumable resources, for which activities can produce and/or consume resource units at their start and/or end. Normally, they have a maximal capacity or level and a minimal level requirement, *e.g.*, safety stock. They generalize renewable and non-renewable resources. The constraints that the activities impose on the reservoir are called producer/consumer constraints. Typical examples for such a resource are fuel or water tanks, raw materials, and, even money in an investment project [15].

There is limited literature about producer/consumer constraints. Simonis et al. [22] brought it to the attention of the constraint programming (CP) community for the first time. They motivated different application scenarios and show how to model them with the global `cumulative` constraint. Barták [3] provided an overview of the intersection of industrial planning and scheduling problems which include producer/consumer constraints. Neumann et al. [15] studied the properties of the feasible regions, proposing a branch-and-bound algorithm and filtered beam search heuristic for solving producer/consumer constraints in combination with generalized precedence constraints. They also created the test set `ubo`. Laborie [12] investigated the intersection of planning and scheduling involving producer/consumer constraints with generalized precedence relations. He proposed a balance constraint that reasons about the temporal relations between two events and can detect new temporal relations. He then developed an exact solution method with several dedicated search heuristics, which were combined to closed the remaining open instances in the test set `ubo`. Beldiceanu et al. [5] presented the new global constraint `cumulatives`, which generalized the global `cumulative` constraint. This new constraint allowed negative heights in order to model producer/consumer constraints. Beck [4] investigated heuristics for scheduling problems involving inventories, and more recently Kinable [10] extended the balance constraints [12] to optional events.

We consider the scheduling of *production* and *consumption events* for discrete resources in reservoirs, as well as generalized precedence constraints (also called temporal constraints, minimal and maximal time lags, difference logic constraints) relating those events.

A *reservoir* is a finite pool for storing a resource, having a minimal $L_{\min}$ and maximal resource level $L_{\max}$. A *production event* adds an amount of resource to a reservoir at a certain time, the reservoir should not exceed its maximal level. A *consumption event* removes an amount of resource from a reservoir at a certain time, the reservoir should not go below its minimal level. An event x has a start or *event time* $t(x)$ and an effect or *height* $h(x)$ on the reservoir. If $h(x) < 0$ ($h(x) > 0$) then we have a consumption (production) event.

Generalized precedence constraints relate two events and are of the form: $t(x) - t(y) \leq d$ where d is integral. Note that this allows us to express, *e.g.*,

- *fixed separation of events*: $t(x) - t(y) = d$ as $t(x) - t(y) \leq d \wedge t(y) - t(x) \leq -d$,
- *x is before y* written $x \prec y$ as $t(x) - t(y) \leq -1$
- *x is no later than y* written as $x \preceq y$ as $t(x) - t(y) \leq 0$

Example 1. Consider a RCPSP with activities i and a resource of capacity R. Let s_i, e_i, d_i and r_i be the start time of i, the end time of i, the fixed duration of i and the fixed resource requirement of i on R. Then we can reformulate the problem as producer/consumer constraints with generalized precedence constraints. Each activity i is modeled with a consumption event x_i and production event y_i for which $t(x_i) = s_i$, $h(x_i) = -r_i$, $t(y_i) = e_i$, and $h(y_i) = r_i$. and precedence constraints enforcing $t(x_i) + d_i = t(y_i)$ and the renewable resource is modeled as a reservoir with $L_{\min} = 0$, $L_{\max} = R$, and initial resource level at R. The

initial resource level is modeled by a dummy production event z with $t(z) = 0$ and $h(z) = R$ where 0 is the start of the planning horizon. □

Given the success of nogood learning on RCPSP and related scheduling problems [6,9,20,21,19,11,17,23], it is interesting to explore what is the best learning approach to tackle the producer/consumer problems with generalized precedence constraints, a class of problems that generalizes RCPSP and many other scheduling problems.

Usually in CP there is fairly well understood tradeoff: a propagator that infers more is worthwhile as long as the cost of the inference does not outweigh the benefits of the extra inference. With nogood learning, this becomes more complicated. A propagator with weaker inference may be better if it makes more local inferences, that are more reusable, in particular a decomposition may be advantageous since it introduces new local variables which may be worthwhile learning about. In the case of producer/consumer we have three distinct possibilities:

- a decomposition may be best since the local variables it introduces are valuable for learning
- a time bounds approach may be best since it uses native literals to the problem, and bounds can succinctly capture lots of schedules independent of what happened to earlier events,
- an approach based on ordering may be best, since ordering of events captures the essence of the constraints on reservoirs

The aim of this paper is to answer the question of which approach is best.

2 Preliminaries

2.1 Lazy Clause Generation

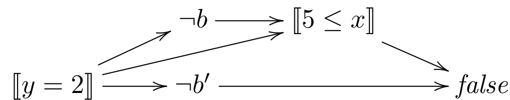
CP solves constraint satisfaction problems by interleaving propagation, which remove impossible values of variables from the domain, with search, which guesses values. All propagators are repeatedly executed until no change in domain is possible, then a new search decision is made. If propagation determines there is no solution then search undoes the last decision and replaces it with the opposite choice. If all variables are fixed then the system has found a solution to the problem. For more details see, *e.g.*, [18].

We assume we are solving a constraint satisfaction problem over set of variables $x \in \mathcal{V}$, each of which takes values from a given initial finite set of values or *domain* $D_{init}(x)$. The domain D keeps track of the current set of possible values $D(x)$ for a variable x. Define $D \sqsubseteq D'$ iff $D(x) \subseteq D'(x), \forall x \in \mathcal{V}$. The constraints of the problem are represented by propagators f which are functions from domains to domains which are monotonically decreasing $f(D) \sqsubseteq f(D')$ whenever $D \sqsubseteq D'$, and contracting $f(D) \sqsubseteq D$. If all values are removed from one domain of a variable x, *i.e.*, $D(x) = \emptyset$ then the constraints cannot be satisfied with the search decisions made and a failure is triggered.

We make use of CP with learning using the lazy clause generation (LCG) [16] approach. Learning keeps track of what caused changes in domain to occur, and on failure computes a *nogood* which records the reason for failure. The nogood prevents search making the same incorrect set of decisions again.

In an LCG solver integer domains are also represented using Boolean variables. Each variable x with initial domain $D_{init}(x) = [l..u]$ is represented by two sets of Boolean variables $[\![x = d]\!], l \leq d \leq u$ and $[\![x \leq d]\!], l \leq d < u$ which define which values are in $D(x)$. We use $[\![x \neq d]\!]$ as shorthand for $\neg[\![x = d]\!]$, and $[\![d \leq x]\!]$ as shorthand for $\neg[\![x \leq d - 1]\!]$. An LCG solver keeps the two representations of the domain in sync. For example if variable x has initial domain $[0..5]$ and at some later stage $D(x) = \{1, 3\}$ then the literals $[\![x \leq 3]\!], [\![x \leq 4]\!], \neg[\![x \leq 0]\!], \neg[\![x = 0]\!]$, $\neg[\![x = 2]\!], \neg[\![x = 4]\!], \neg[\![x = 5]\!]$ will hold. Explanations are defined by clauses over this Boolean representation of the variables.

Example 2. Consider a simple constraint satisfaction problem with constraints $b \rightarrow x + 3 \leq y$, $\neg b \rightarrow y + 3 \leq x$, $b' \rightarrow y \leq 3$, $\neg b' \rightarrow x \leq 3$, with initial domains $D_{init}(b) = D_{init}(b') = \{0, 1\}$, and $D_{init}(x) = D_{init}(y) = \{0, 1, 2, 3, 4, 5, 6\}$. There is no initial propagation. Setting $[\![y = 2]\!]$ makes the first constraint propagate $D(b) = \{0\}$ with explanation $[\![y = 2]\!] \rightarrow \neg b$, then the second constraint propagates $D(x) = \{5, 6\}$ with explanation $\neg b \wedge [\![y = 2]\!] \rightarrow [\![5 \leq x]\!]$. The third constraint propagates $D(b') = \{0\}$ with explanation $[\![y = 2]\!] \rightarrow \neg b'$ and the last constraint sets $D(x) = \emptyset$, with explanation $[\![5 \leq x]\!] \wedge \neg b' \rightarrow \mathit{false}$. The graph of the implications is



Any cut separating the decision $[\![y = 2]\!]$ from *false* gives a nogood. The simplest one is $[\![y = 2]\!] \rightarrow \mathit{false}$. □

2.2 Global Difference Logic Propagator

Constraints of difference, *i.e.*, of the form $x - y \leq d$ where d is a fixed value are one of the simplest form of constraints. Efficient algorithms based on shortest paths are known for computing satisfaction and implications, and propagation for this class of constraints. In this work we use them inside an explaining solver, hence the propagator needs to explain its reasoning, making it more akin to a difference logic theory propagator [7] in SAT modulo theories (SMT).

The global difference logic propagator [8] reasons with literals of the form $[\![x - y \leq d]\!]$, and for example can make transitive deductions like $[\![x - y \leq 1]\!] \wedge [\![y - z \leq 4]\!] \rightarrow [\![x - z \leq 5]\!]$ which are beyond the scope of usual finite domain propagation. The propagator will detect if a literal $[\![x - y \leq d]\!]$ becomes true or false given the current set of difference constraints (and bounds) which are asserted, and also detect unsatisfiability of the set of asserted constraints. This will be important for the producer/consumer constraint which can both make

use of difference logic information, and produce new difference logic information. Note that because bounds literals $[\![x \leq d]\!]$ and $[\![-x \leq d]\!]$ are so much more important in CP they are treated specially unlike in SMT, for details see [8].

3 The Producer/Consumer Constraint

Laborie [12] provides the framework of the producer/consumer (or balance) constraint for reasoning about reservoirs. In this section, we revisit his framework in order to extend it for nogood learning solvers in the next section.

Given a set of events Ev on the reservoir with $t(x)$ being the time of the event x and $h(x)$ the effect on the reservoir. Let Pr and Co be the set of producer and consumer events respectively, $Pr = \{x \in Ev \mid h(x) > 0\}$, $Co = \{x \in Ev \mid h(x) < 0\}$. The producer/consumer constraint enforces that the reservoir stays within the minimal (resource) level $L_{\min}$ and maximal (resource) level $L_{\max}$. Hence for every point in the planning horizon τ it enforces

$$L_{\min} \leq \sum_{x \in Ev : t(x) \leq \tau} h(x) \leq L_{\max} \quad .$$

The event time $t(x)$ and height $h(x)$ can be variable. We use the notation $t_{\min}(x)$ ($t_{\max}(x)$) to refer to the current minimum (resp. maximum) possible time for event x, and similarly $h_{\min}(x)$ ($h_{\max}(x)$) for minimum (maximum) height.

In order to propagate the constraint, the set of events are partitioned into the following six sets with respect to an event x.

before	$B(x) = \{y \in Ev \mid t(y) < t(x)\}$
before or with	$BS(x) = \{y \in Ev \mid t(y) \leq t(x) \wedge y \notin B(x) \cup S(x)\}$
with	$S(x) = \{y \in Ev \mid t(y) = t(x) \vee (t(y) \leq t(x) \wedge t(y) \geq t(x))\}$
after or with	$AS(x) = \{y \in Ev \mid t(y) \geq t(x) \wedge y \notin A(x) \cup S(x)\}$
after	$A(x) = \{y \in Ev \mid t(y) > t(x)\}$
unknown	$U(x) = \{y \in Ev \mid y \notin B(x) \cup BS(x) \cup S(x) \cup A(x) \cup AS(x)\}$

Since normally not all relationships between pairs of events are known in advanced, the partition of events dynamically changes during the search. However, only events in one of these sets $BS(x)$, $AS(x)$, or $U(x)$ can move to another set and that at most twice. At the end, all events are partitioned by $B(x) \cup S(x) \cup A(x)$. An event in $U(x)$ can move to all other sets. An event in $BS(x)$ can move to either $B(x)$ or $S(x)$. An event in $AS(x)$ can move to either $S(x)$ or $A(x)$.

Determining to which set an event y belongs to during search can be performed by checking the earliest and latest event time or using ordering constraints. If we use event time information then the sets are determined by:

$$\begin{aligned}
B(x) &= \{y \in Ev \setminus \{x\} \mid t_{\max}(y) < t_{\min}(x)\} \\
BS(x) &= \{y \in Ev \setminus \{x\} \mid t_{\max}(y) \leq t_{\min}(x) \wedge (t_{\min}(y) < t_{\max}(y) \vee t_{\min}(x) < t_{\max}(x))\} \\
S(x) &= \{y \in Ev \setminus \{x\} \mid t_{\max}(y) = t_{\min}(x) \wedge t_{\min}(y) = t_{\max}(y) \wedge t_{\min}(x) = t_{\max}(x)\} \cup \{x\}
\end{aligned}$$

$$
\begin{aligned}
AS(x) &= \{y \in Ev \setminus \{x\} \mid t_{\min}(y) \geq t_{\max}(x) \wedge (t_{\min}(y) < t_{\max}(y) \vee t_{\min}(x) < t_{\max}(x))\} \\
A(x) &= \{y \in Ev \setminus \{x\} \mid t_{\min}(y) > t_{\max}(x)\} \\
U(x) &= \{y \in Ev \setminus \{x\} \mid t_{\min}(y) < t_{\max}(x) \wedge t_{\min}(x) \leq t_{\max}(y)\}
\end{aligned}
$$

To make use of ordering constraints, we first introduce the ordering Booleans B_{xy} for all event pairs $\{x, y\} \subseteq Ev$, and the constraints

$$
b_{xy} \leftrightarrow t(x) < t(y) \qquad b_{yx} \leftrightarrow t(y) < t(x) \qquad \neg b_{xy} \vee \neg b_{yx}
$$

unless we use the global difference logic propagator where instead we simply define the ordering literals b_{xy} as

$$
b_{xy} \equiv [\![t(x) - t(y) \leq -1]\!] \qquad b_{yx} \equiv [\![t(y) - t(x) \leq -1]\!]
$$

Using the ordering literals the sets are determined by:.

$$
\begin{aligned}
B(x) &= \{y \in Ev \setminus \{x\} \mid b_{xy} = \mathit{true}\} \\
BS(x) &= \{y \in Ev \setminus \{x\} \mid D(b_{xy}) = \{\mathit{true}, \mathit{false}\} \wedge b_{yx} = \mathit{false}\} \\
S(x) &= \{y \in Ev \setminus \{x\} \mid b_{xy} = \mathit{false} \wedge b_{yx} = \mathit{false}\} \cup \{x\} \\
AS(x) &= \{y \in Ev \setminus \{x\} \mid b_{xy} = \mathit{false} \wedge D(b_{yx}) = \{\mathit{true}, \mathit{false}\}\} \\
A(x) &= \{y \in Ev \setminus \{x\} \mid b_{yx} = \mathit{true}\} \\
U(x) &= \{y \in Ev \setminus \{x\} \mid D(b_{xy}) = \{\mathit{true}, \mathit{false}\} \wedge D(b_{yx}) = \{\mathit{true}, \mathit{false}\}\}
\end{aligned}
$$

Note that the relationships determined by the ordering constraints will be strictly more informative than those determined from event time information (independent of whether we use difference logic or not). The conditions for $B(x)$, $S(x)$ and $A(x)$ for event time, will enforce the conditions for ordering literals. Thus, the ordering literals can potentially decide earlier to which set an event y belongs to and exploit that information for propagation.

3.1 Immediate Maximal Resource Level Before Event

In this sub-section following [12], we describe the consistency check and the filtering on the event times and heights. The maximal resource level shortly before an event $x \in Ev$ can be approximated by

$$
L^{<}_{\max}(x) = \sum_{y \in B(x)} h_{\max}(y) + \sum_{y \in Pr \cap (BS(x) \cup U(x))} h_{\max}(y) \tag{1}
$$

where the last sum is an approximation because it neglects precedence relations and consumer events.

Consistency If the maximal resource level is too low then there does not exist any producer event that can be pushed for execution before the event x. Thus, the system is inconsistent, i.e.,

$$
L^{<}_{\max}(x) < L_{\min} \to \mathit{false} \quad .
$$

Time Bounds Filtering If the first sum in (1) is less than $L_{\min}$ then the lower bound on $t(x)$ can be improved.

$$\sum_{y \in B(x)} h_{\max}(y) < L_{\min} \rightarrow \exists \Omega \subseteq Pr \cap (BS(x) \cup U(x)) \text{ with}$$
$$\sum_{y \in B(x) \cup \Omega} h_{\max}(y) \geq L_{\min} : \forall y \in \Omega : t(y') < t(x)$$

Searching for an arbitrary set Ω is expensive, but a "practical" set can be computed as follows. Let the events $y_1, y_2, \ldots \in Pr \cap (BS(x) \cup U(x))$ be in chronological order according to their earliest event time, *i.e.*, $t_{\min}(y_1) \leq t_{\min}(y_2) \leq \ldots$. Then the lower bound on $t(x)$ can be updated as follows.

$$\exists k : \sum_{i=1}^{k-1} h_{\max}(y_i) < L_{\min} - \sum_{y \in B(x)} h_{\max}(y) \leq \sum_{i=1}^{k} h_{\max}(y_i) \rightarrow t_{\min}(y_k) < t(x)$$

Consumption and Production Level Filtering The consumption or production level of certain events y can be filtered with the respect to x. The following rule holds for all consumers $y \in Co \cap B(x)$ and all producers $y \in Pr \cap (B(x) \cup BS(x) \cup U(x))$.

$$L^{<}_{\max}(x) + h_{\min}(y) - h_{\max}(y) < L_{\min} \rightarrow L_{\min} - L^{<}_{\max}(x) + h_{\max}(y) \leq h(y)$$

This rules avoids a resource level that is below the safety level $L_{\min}$.

3.2 Other Resource Levels Regarding an Event

Similar reasoning to the immediate maximal resource level before an event can be performed for the immediate minimal resource level before the event (2) and the immediate maximal (3) and minimal resource level after the event (4). They lead to similar propagations to those described for $L^{<}_{\max}(x)$ in the previous subsection.

$$L^{<}_{\min}(x) = \sum\nolimits_{y \in B(x)} h_{\min}(y) + \sum\nolimits_{y \in Co \cap (BS(x) \cup U(x))} h_{\min}(y) \quad (2)$$

$$L^{>}_{\max}(x) = \sum\nolimits_{y \in B(x) \cup BS(x) \cup S(x)} h_{\max}(y) + \sum\nolimits_{y \in Pr \cap (AS(x) \cup U(x))} h_{\max}(y) \quad (3)$$

$$L^{>}_{\min}(x) = \sum\nolimits_{y \in B(x) \cup BS(x) \cup S(x)} h_{\min}(y) + \sum\nolimits_{y \in Co \cap (AS(x) \cup U(x))} h_{\min}(y) \quad (4)$$

4 Explanations

In this section, we describe how to explain propagations for $L^{<}_{\max}$, the propagations for other resource levels are explained similarly. Since we investigate different propagation algorithms, we introduce general explanations at first before refining them to each of the propagators.

4.1 Explanation of the Resource Level and Inconsistency

Explanation for inconsistencies or filtering regarding to an event x must express the reason for the current bound on the resource level $L^{<}_{\max}(x)$. For simplicity, we assume fixed consumption/production of the events at first. In this case, the bound on the level is caused by the following reasons.

- consumer events are executed before event x
- producer events are executed simultaneously to or after event x

All those events cause a lower level on the reservoir shortly before event x. Thus, the corresponding explanation would be

$$expl(L^{<}_{\max}(x)) = \bigwedge_{y \in Co \cap B(x)} expl(y \prec x) \wedge \bigwedge_{y \in Pr \cap Z(x): y \neq x} expl(y \succeq x) \quad (5)$$

where $Z(x) = S(x) \cup AS(x) \cup A(x)$ and the functions $expl(y \prec x)$ and $expl(y \succeq x)$ are defined later. Note that the relative position of events that are not considered are irrelevant for the current bound on the level. Thus, they can be left out of the explanation.

In the case, events can have flexible consumption/production then the bound on the resource level can be caused by these additional reasons.

- consumer events run before event x consume too many resource units
- production events that are not simultaneously to or after event x produce too few resource units

Thus, the explanation (5) must be extended by

$$\bigwedge_{y \in Co \cap B(x)} [\![h(y) \leq h_{\max}(y)]\!] \wedge \bigwedge_{y \in Pr \cap (B(x) \cup BS(x) \cup U(x))} [\![h(y) \leq h_{\max}(y)]\!]$$

Explanation for Inconsistency If we have resource underflow $L^{<}_{\max}(x) < L_{\min}$ immediately before an event x then the explanation is simply

$$expl(L^{<}_{\max}(x)) \rightarrow false \quad .$$

We generalize this explanation by removal of consumer and producer events considered in (5) in input order using the slack $L_{\min} - L^{<}_{\max}(x) - 1$.

4.2 Explanation for Time Bounds Filtering

The lower bound on the event time of x can be updated if all events that are before x are not enough to achieve the minimal resource level. In that case, some producer events that are currently in the before-or-simultaneously ($BS(x)$) or unknown relationship ($U(x)$) to x, must happen before x in any solution. In addition to the reasons considered for the current bound on the maximal resource level, these reasons need to be considered for a bound update

– producer events in $BS(x) \cup U(x)$ that start too late

These reasons result in the following explanation.

$$expl(L^{<}_{\max}(x)) \wedge \bigwedge_{i=k}^{\infty} [\![t_{\min}(y_k) \leq t(y_i)]\!] \rightarrow t_{\min}(y_k) < t(x)$$

We generalize this explanation in the simliar way as in the case of inconsistency, but we use the slack $\sum_{i=1}^{k} h_{\max}(y_i) - L_{\min}$. At first, we remove events in $B(x)$ in input order and then the remaining events in chronological order of their minimal event time.

4.3 Explanation for Consumption and Production Level Filtering

The lower bound on the height of an event y can be updated if it would cause a resource underflow shortly before event x. The potential underflow is only related the current bound on the maximal level and an explanation is simply

$$expl(L^{<}_{\max}(x)) \rightarrow L_{\min} - L^{<}_{\max}(x) + h_{\max}(y) \leq h(y) \ .$$

5 Global Reservoir Propagators

In this section, we describe three different reservoir propagators and specialize the general explanations described in the previous section.

5.1 Bounds Propagator

The bounds propagator `bounds` uses the current bounds on the event times in order to relate pairs of events, *i.e.*, partitioned all events into the six sets described in the previous section with respect to an event x. For each event, the propagator can easily determine the event partition in linear time by scanning over all events. Thus, it can also determine all four reservoir levels and time for the time bounds filtering in linear time, for the last one we assume that the events are sorted with respect to $t_{\min}(.)$. The sorting can be done at the beginning of the propagator's execution. Therefore, the worst case complexity of the bounds propagator is $\mathcal{O}(|Ev|^2)$.

When the propagator `bounds` detects a inconsistency or performs filtering it additional generates the explanation described in the previous section and specializes them by using following bounds on the event times.

$$expl(y \prec x) = [\![t(y) < t_{\min}(x)]\!] \wedge [\![t_{\min}(x) \leq t(x)]\!]$$
$$expl(y \succeq x) = [\![t_{\max}(x) \leq t(y)]\!] \wedge [\![t(x) \leq t_{\max}(x)]\!]$$

Note the literals $[\![t_{\min}(x) \leq t(x)]\!]$ and $[\![t(x) \leq t_{\max}(x)]\!]$ will only appear once in an explanation. Since most of the explanation, e.g. $expl(L^{<}_{\max}(x))$, can be pre-computed in linear time for an event x. It does not add to the worst case complexity of the propagator.

5.2 Order Propagator

The order propagator `order` works in the same way as `bounds`, but it uses ordering variables between pairs of events for propagation. Hence, it has the same runtime complexity $\mathcal{O}(|Ev|^2)$. It uses ordering literals in the explanations.

$$expl(y \prec x) = b_{yx} \qquad\qquad expl(y \succeq x) = \neg b_{yx}$$

Note that Laborie [12] uses the non-learning version of the propagator `order`.

5.3 Timetable Propagator

The timetable propagator `tt` does not consider the time relations between pair of events, but relates events to time points in the planning horizon. Instead of performing propagation at each time point in the horizon, it only has to consider time points at which the minimal or maximal level may changes. These time points are the bounds on the event times, *i.e.*, $t_{\min}(x)$ and $t_{\max}(x)$. Hence, it considers a number of time points which is linear in the number of events.

The propagation and explanation work similar to the `bounds` propagator. Consider the time point $\tau = t_{\max}(x)$ of an event x for propagation and let z be an *artificial event* with $t(z) = t_{\min}(z) = t_{\max}(z) = \tau$ and $h(z) = 0$ then the same propagation can be done as the `bounds` propagator with respect to this fixed artificial event. Consequently, the explanation are the same and reduce to one of the following literals due to the fixed event z.

$$expl(y \prec z) = [\![t(y) < \tau]\!] \qquad\qquad expl(y \succeq z) = [\![\tau \leq t(y)]\!]$$

It is obvious that the `tt` propagator can easily be implemented with a worst case complexity $\mathcal{O}(|Ev|^2)$. Note that this propagator is a specialized form of the timetable propagator of the global `cumulative` constraint. Thus, it will prune the same as the timetable propagator in `cumulative` when the reservoir constraints are modeled as cumulative propagators as described in [22].

6 Models

We use the solver-independent modeling language MiniZinc [13] for describing our models. The input parameter of an instance are as follows. Their meaning is giving in the comment after the declaration.

```
set of int: R;  % Set of reservoirs
set of int: E;  % Set of events
set of int: P;  % Set of generalized precedence relations
array [R] of int: Lmin;  % Minimum level of the reservoirs
array [R] of int: Lmax;  % Maximum level of the reservoirs
array [R, E] of int: rr;     % Amount of resource production/consumption of
     the events
array [P, 1..3] of int: prec;    % Generalized precedence relations between
     two events: start(prec[i,1]) + prec[i,2] <= start(prec[i,3])
array [R] of set of int: resE = [ {i | i in E where rr[r,i] != 0} | r in R ];
        % Set of "non-zero" events for each reservoir
```

Then the start (event) time variables `s` and the objective `objective` are declared as follows where `0..UB` is the planning horizon.

```
set of int: Times = 0..UB;      % Planning horizon
array [E] of var Times: s;      % Event time variables
var Times: objective = s[sink]; % Objective variable
```

The initial upper bound on the objective `UB` is initialized by `sum(i in E)(max([0] ++ [prec[p,2] | p in P where prec[p,1] = i]))` unless otherwise stated. Note that the set of events `E` contains one dummy source (`source`) and sink (`sink`) event. The source start time is constrained by `s[source] = 0`. Generalized precedence constraints are expressed by one binary linear inequality constraints as usual.

```
constraint forall(p in P)( s[prec[p, 1]] + prec[p, 2] <= s[prec[p, 3]] );
```

Reservoir Decompositions The first decomposition is the time-indexed formulation (`dTT`), which allows learning about the event times. This model creates two linear constraints for each point in time in the planning horizon. Thus the model size is time-dependent.

```
array[E, Times] of var bool: bit =
    array2d(E, Times, [s[i] <= t | i in E, t in Times]);
constraint forall(r in R, t in Times)(
    sum(i in res_events[r])(rr[r,i] * bit[i,t]) <= Lmax[r]
/\  sum(i in res_events[r])(rr[r,i] * bit[i,t]) >= Lmin[r] );
```

Note that the additional Boolean variables `bit` are literals of the Boolean representation of `s` in LCG solver, *i.e.*, those already exists for LCG solvers. In total $\mathcal{O}(|\mathtt{R}| \cdot \mathtt{UB})$ linear constraints of size $\mathcal{O}(|\mathtt{E}|)$ are created.

The second decomposition is an event-based formulation (`dAfter`), which ensures that the resource levels are within the limits at the event time for each event. For modeling it, we required two additional sets of Boolean variables `s_eq_0` and `s_leq_s`, where the first set describes whether an event starts at time point 0 and the second set are order variables between pairs of events.

```
array[E] of var bool: s_eq_0   = [s[i] <= 0 | i in E];
array[E,E] of var bool: s_leq_s = array2d(E, E, [s[i] <= s[j] | i, j in E]);
% Constraints for correct reservoir levels at event time
constraint forall(r in R, i in resE[r])(
        sum(j in resE[r])( rr[r,j] * s_leq_s[j,i] ) <= Lmax[r]
    /\  sum(j in resE[r])( rr[r,j] * s_leq_s[j,i] ) >= Lmin[r] );
% Constraints for correct reservoir level at time point 0
constraint forall(r in R)(
    if Lmin[r] > 0
    then sum(i in res_events[r])(rr[r,i] * s_eq_0[i]) >= Lmin[r]
    else if Lmax[r] < 0
         then sum(i in res_events[r])(rr[r,i] * s_eq_0[i]) <= Lmax[r]
    else true endif endif);
% Constraints for s_leq_s
constraint forall(i, j in E where i < j)(s_leq_s[i, j] \/ s_leq_s[j, i]);
```

The first set of constraints enforces the reservoir level at each event time, whereas the second enforces it for time point 0, and the last set of constraints explicitly models that one of the pair of order variables must be true. Due to the order variables `s_leq_s`, a learning solver is able to learn about the temporal relations between pairs of events. The reified constraints `s_leq_s[i,j] <-> s[i] <= s[j]`

are created in the definition of `s_leq_s`. The decomposition (after) creates $\mathcal{O}(|\mathtt{R}| \cdot |\mathtt{E}|)$ linear constraints of size $\mathcal{O}(|\mathtt{E}|)$ and $\mathcal{O}(|\mathtt{E}|^2)$ Boolean variables and reified binary linear constraints.

The third decomposition (`dBefore`) is an extension of the second one (`dAfter`). It adds redundant linear constraints for the reservoir level shortly before the event time for each event. It re-uses the Boolean variables `s_eq_0` and `s_leq_s`.

```
constraint forall(i in resE[r])(
   sum(j in resE[r])(rr[r,j]*not(s_leq_s[i,j])) <= Lmax[r]-Lmin[r]*s_eq_0[i]
/\ sum(j in resE[r])(rr[r,j]*not(s_leq_s[i,j]))-Lmin[r]*not(s_eq_0[i]) >= 0);
```

In addition to the constraints and variables created by the decomposition (`dAfter`), it creates $\mathcal{O}(|\mathtt{R}| \cdot |\mathtt{E}|)$ linear constraints of size $\mathcal{O}(|\mathtt{E}|)$.

Global Reservoir Constraints Rather than using decompositions to model the reservoir constraints we can make use of the global propagators. The `tt` propagators has the same propagation strength as the decomposition `dTT`, but drastically reduces the model size and makes it time independent, but note it does not have the same *language of learning*. The `order` propagator is equivalent to the `dBefore` decomposition, but again drastically smaller in size. Again it does not have many intermediate variables which might be useful for learning. The `bounds` propagator is weaker than the `dBefore` and `dAfter` decompositions, since it does not reason about order, but it is, of course, concise, and avoids the need for any order variables.

7 Experiments

The experiments were run on a machine running Ubuntu 14.04 with an Intel(R) Core(TM) i7 CPU running at 2.8GHz with 8GB memory. We implemented the described propagators in the lazy clause generation solver Chuffed (`chuffed`). We ran all model with a smallest first search alternating on each restart with activity based search. We used the benchmarks `ubo10`, `ubo20`, `ubo50`, and `ubo100` created by [15] consisting of 90 instances with 10, 20, 50, and 100 events, respectively. Since all instances are closed, we constructed the new benchmark `ubo200`[1] consisting of 90 instances with 200 events in the same way as in [15].

While the main purpose of our experiments is to determine the best form of explanation for this class of problem, in order to calibrate the learning methods against other approaches we also ran the method of Neumann and Schwindt [15] (`neu&sch`) on a machine running Windows 10 and having a Intel(R) Core(TM) i5 CPU with 3.2GHz and 4GB memory (since we only have a Windows executable). We were unable to obtain an executable of Laborie's method [12] to compare with. All runs were limited to 10 minutes.

We compare the following variations of decompositions and global propagators: (`dTT`) `dTT` decomposition, (`dAfter`) `dAfter` decomposition, (`dBefore`) `dBefore` decomposition, (`order`) `order` propagator, (`bounds`) `bounds` propagator, (`tt`) `tt` propagator, (`order+diff`) `order` and `diff` propagator, (`bounds+diff`) `bounds` and `diff` propagator, and (`tt+diff`) `tt` and `diff` propagator.

[1] Available at `http://people.eng.unimelb.edu.au/pstuckey/rcpsp/`

Table 1. Results on `ubo50`

solver	#inst	#opt	#sat	#unk	#inf	m.rt	m.it	m.#confl	m.#nodes
`neu&sch`	90	47	1	0	42	16.23s	0.0s		
`order+diff`	90	48	0	0	42	0.07s	0.01s	48	757
`order`	90	48	0	0	42	0.33s	0.01s	305	1823
`bounds+diff`	90	48	0	0	42	0.50s	0.01s	2143	6517
`bounds`	90	48	0	0	42	0.56s	0.01s	2384	6741
`tt+diff`	90	48	0	0	42	0.94s	0.01s	314	2131
`tt`	90	48	0	0	42	0.86s	0.01s	282	2108
`dTT`	90	48	0	0	42	1.21s	0.71s	259	735
`dAfter`	90	48	0	0	42	0.35s	0.08s	330	2115
`dBefore`	90	48	0	0	42	0.53s	0.14s	346	2101

Table 2. Results on `ubo100`

solver	#inst	#opt	#sat	#unk	#inf	m.rt	m.it	m.#confl	m.#nodes
`neu&sch`	90	50	7	2	31	60.11s	0.0s		
`order+diff`	90	57	0	0	33	0.89s	0.03s	229	2818
`order`	90	55	2	0	33	18.16s	0.03s	3996	17758
`bounds+diff`	90	55	2	0	33	20.55s	0.02s	38048	103067
`bounds`	90	55	2	0	33	22.02s	0.02s	44001	122138
`tt+diff`	90	55	2	1	32	31.96s	0.02s	6718	32048
`tt`	90	55	2	1	32	33.61s	0.02s	6655	33665
`dTT`	90	57	0	1	32	17.23s	3.45s	2045	12001
`dAfter`	90	56	1	0	33	11.41s	0.30s	1800	12509
`dBefore`	90	56	1	0	33	18.22s	0.57s	2347	14017

7.1 Results

The results are shown in Tab. 1–3. Each table shows: (#inst) the number of benchmark instances, (#opt) the number of instances proved optimal in the time limit, (#sat) the number where a solution was found, but was not proven optimal in the time limit, (#unk) the number where no solution was found, (#inf) the number proved unsatisfiable, (m.rt) the mean runtime in seconds, (m.it) the mean initialization time in seconds, (m.#confl) the mean number of conflicts, and (m.#nodes) the mean number of nodes. If the solver timed out for an instance then the number of conflicts (nodes) at that time are counted in m.#confl (m.#nodes).

Note that learning was crucial for solving these problems, even on smaller benchmarks (`ubo20`) the best method `order+diff` was unable to solve all problems within the 10 minute time limit without learning, whereas with learning all our methods could easily solve all these problems, and the larger ones (`ubo50`).

The method of Neumann and Schwindt is not competitive, for the smallest sizes it can solve most problems, but significantly slower, and it is very poor at detecting infeasible problems. The order based explanations are clearly dominant in terms of number of conflicts and number of nodes. The difference logic propagator, which clearly supports the order based explanations is significantly

Table 3. Results on `ubo200`

solver	#inst	#opt	#sat	#unk	#inf	m.rt	m.it	m.#confl	m.#nodes
`neu&sch`	90	54	10	11	15	148.92s	0.0s		
`order+diff`	90	66	0	0	24	24.67s	0.09s	2056	22454
`order`	90	53	12	4	21	154.49s	0.09s	6676	29181
`bounds+diff`	90	60	5	4	21	82.70s	0.05s	75348	245707
`bounds`	90	61	4	4	21	83.86s	0.05s	75297	242386
`tt+diff`	90	52	13	4	21	160.49s	0.05s	4498	21846
`tt`	90	51	14	5	20	166.96s	0.05s	4644	23048
`dTT`	90	0	0	90	0	—	∞	—	—
`dAfter`	90	52	14	3	21	155.67s	1.61s	4636	24930
`dBefore`	90	51	15	2	22	172.75s	3.03s	4634	21900

advantageous to the point where in Tab. 3 even though order based search without the difference logic propagator is massively smaller than linear explanation search, it cannot prove optimality of as many instances.

What is surprising is how effective the decompositions are, at least in terms of search. Clearly the intermediate literals introduced by the decompositions are improving the learning. The decompositions are actually better than all methods other than the order-based globals until size 200 where the size of the decompositions become disadvantageous. The time decomposition initialization time grows quickly, to the point that for `ubo200` it prevents the method running.

As the size grows the tradeoff of weaker propagation but better runtime complexity the `bounds` global propagator (`bounds`,`bounds+diff`) over the `tt` global propagator (`tt`,`tt+diff`) becomes evident. It still cannot compete with the `order` propagator, when used in conjunction with globals difference propagation. This may well change with incremental versions of these propagators.

We also examined the 12 hard instances of the benchmark set generated by Neumann *et al* [14] and closed by Laborie [12] using ILOG Scheduler. Table 4 compares with the published results of Neumann *et al* [15] and Laborie [12] on a HP-UX 9000/785 workstation. Note that Laborie only presents the best results from his nine different heuristics for each instance.

Clearly `order+diff` is comparable to the best of Laborie's 9 method (although his CPU is somewhat slower). His approaches includes the inference of new ordering relationships, and specialized search routines that make use of the information about precedences inferred by the search. ILOG scheduler includes a component equivalent to the global difference logic propagator, without learning. As the combinatorics of these problems grow substantially as the size increases we are confident that our learning solver would outperform his approach on larger problems, although perhaps we would need to invest in incremental propagators and inference of precedences to match the well engineered commercial solver.

Table 4. Comparison with Neumann et al. [15] and Laborie [12].

instance	optimal	neu&sch	best of laborie	order+diff
50_10	92	time out	0.28s	0.03s
50_27	96	346.483s	2.43s	0.1s
50_82	unsat	509.161s	0.05s	0.03s
100_6	211	time out	0.97s	2.3s
100_12	197	time out	0.72s	0.88s
100_20	199	time out	0.46s	13.6s
100_30	204	time out	2.11s	0.85s
100_41	337	time out	0.62s	1.62s
100_43	unsat	time out	7.65s	0.54s
100_54	344	time out	0.46s	6.95s
100_58	317	time out	0.49s	0.37s
100_69	unsat	time out	1.96s	0.41s

8 Conclusion

The producer/consumer constraints is a powerful tool for specifying complex scheduling problems with renewable and non-renewable resources. In this paper we have explored what is right approach to solving these problems once we are using nogood learning. The key language of learning is the ordering literals b_{xy} used by the `order` propagator, *but with the proviso* that we need to use a global difference logic propagator to see the interaction of the inference between ordering literals. Surprisingly without the use of the global difference logic propagator, decompositions based on ordering are competitive, since they generates many intermediate literals which prove to be useful for learning, even if the models they create are far larger.

There is considerable scope for improving the producer/consumer propagators with learning. Making the propagators incremental, and extending the order propagator to create new order inferences are likely to be highly beneficial. Given the effectiveness of the decompositions, at least in terms of search, it might be worth investigating a global propagator that supports lazy decomposition [2,1] where intermediates are made available during search in parts of the global where many explanations are generated.

Acknowledgments We thank to Christoph Schwindt for providing us the instance generator and an executable of the method used in [15]. This work was partially supported by Asian Office of Aerospace Research and Development grant 15-4016.

References

1. Abio, I., Nieuwenhuis, R., Oliveras, A., Rodriguez-Carbonell, E., Stuckey, P.J.: To encode or propagate: The best choice for each constraint in SAT. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming – CP 2013. LNCS, vol. 8124, pp. 97–106. Springer (2013)

2. Abio, I., Stuckey, P.J.: Conflict directed lazy decomposition. In: Milano, M. (ed.) Principles and Practice of Constraint Programming – CP 2012. LNCS, vol. 7514, pp. 70–85. Springer (2012)
3. Barták, R.: Conceptual models for combined planning and scheduling. Electronic Notes in Discrete Mathematics 4, 1 (2000)
4. Beck, J.C.: Heuristics for scheduling with inventory: dynamic focus via constraint criticality. Journal of Scheduling 5(1), 43–69 (2002)
5. Beldiceanu, N., Carlsson, M.: A new multi-resource cumulatives constraint with negative heights. In: Van Hentenryck, P. (ed.) Principles and Practice of Constraint Programming – CP 2002. LNCS, vol. 2470, pp. 63–79. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
6. Berthold, T., Heinz, S., Lübbecke, M., Möhring, R., Schulz, J.: A constraint integer programming approach for resource-constrained project scheduling. In: Lodi, A., Milano, M., Toth, P. (eds.) Integration of AI and OR Techniques in Constraint Programming. LNCS, vol. 6140, pp. 313–317. Springer Berlin / Heidelberg (2010)
7. Cotton, S., Maler., O.: Fast and flexible difference constraint propagation for DPLL(T). In: In Theory and Applications of Satisfiability Testing - SAT 2006. LNCS, vol. 4121, pp. 170–183. Springer (2006)
8. Feydy, T., Schutt, A., Stuckey, P.J.: Global difference constraint propagation for finite domain solvers. In: Antoy, S. (ed.) Proceedings of 10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming. pp. 226–235. ACM Press (2008)
9. Horbach, A.: A boolean satisfiability approach totheresource-constrained project scheduling problem. Annals of Operations Research 181(1), 89–107 (2010)
10. Kinable, J.: A reservoir balancing constraint with applications to bike-sharing. In: Quimper, C.G. (ed.) Integration of AI and OR Techniques in Constraint Programming. LNCS, vol. 9676, pp. 216–228. Springer International Publishing, Cham (2016)
11. Kreter, S., Schutt, A., Stuckey, P.J.: Modeling and solving project scheduling with calendars. In: Pesant, G. (ed.) Principles and Practice of Constraint Programming – CP 2015. LNCS, vol. 9255, pp. 262–278. Springer International Publishing (2015)
12. Laborie, P.: Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. Artificial Intelligence 143(2), 151–188 (2003)
13. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming – CP 2007. LNCS, vol. 4741, pp. 529–543. Springer-Verlag (2007)
14. Neumann, K., Zimmermann, J.: Project Scheduling: Recent Models, Algorithms and Applications, chap. Methods for resource-constrained project scheduling with regular and nonregular objective functions and schedule-dependent time windows, pp. 261–287. Kluwer (1999)
15. Neumann, K., Schwindt, C.: Project scheduling with inventory constraints. Mathematical Methods of Operations Research 56(3), 513–533 (2002)
16. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
17. Schnell, A., Hartl, R.F.: On the efficient modeling and solution of the multi-mode resource-constrained project scheduling problem with generalized precedence relations. OR Spectrum 38(2), 283–303 (2015)
18. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. ACM Transactions on Programming Languages and Systems 31(1), Article No. 2 (2008)

19. Schutt, A., Chu, G., Stuckey, P.J., Wallace, M.G.: Maximising the net present value for resource-constrained project scheduling. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) Integration of AI and OR Techniques in Constraint Programming. LNCS, vol. 7298, pp. 362–378. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
20. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. Constraints 16(3), 250–282 (2011)
21. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Solving RCPSP/max by lazy clause generation. Journal of Scheduling 16(3), 273–289 (2013)
22. Simonis, H., Cornelissens, T.: Modelling producer/consumer constraints. In: Montanari, U., Rossi, F. (eds.) Principles and Practice of Constraint Programming – CP '95. LNCS, vol. 976, pp. 449–462. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
23. Szeredi, R., Schutt, A.: Modelling and solving multi-mode resource-constrained project scheduling. In: Rueher, M. (ed.) Principles and Practice of Constraint Programming – CP 2016. LNCS, vol. to appear, p. to appear. Springer International Publishing (2016)