Implementing Efficient Propagation Control

Christian Schulte[1] and Guido Tack[2]

[1] KTH – Royal Institute of Technology, Sweden
[2] Katholieke Universiteit Leuven, Belgium

Abstract. In propagation-based constraint solvers, *propagators* implement constraints by removing inconsistent values from variable domains. To make propagation efficient, modern constraint solvers employ two mechanisms of propagation control, *event-based* and *prioritized* propagation. Events, such as a bounds change of a particular variable, control which propagators need to be scheduled for re-evaluation. Prioritization controls which of the scheduled propagators is executed next.
While it has been shown that the combination of event-based and priority-based scheduling is an efficient approach for propagation control, this is the first publication on the implementation details of such a system. This paper presents the design of efficient data structures for propagator priority queues and the event system. The paper introduces the notions of *modification events* and *propagation conditions*, which refine the event-based model of propagation and yield an efficient implementation. The presented architecture is the basis of Gecode.

1 Introduction

In propagation-based constraint solvers, each constraint is implemented by a *propagator*, whose task it is to prune variable domains, removing values that are inconsistent with the implemented constraint. The constraint solver executes the propagators in turn until none of them can prune any domain any longer, thus establishing a *fixpoint*.

The fixpoint computation is the core of the solving process, its efficiency in terms of runtime and memory is vital for a solver's performance. Apart from actually executing the propagators, the solver has to perform efficient **propagation control**, which has two aspects: (1) if the propagator has modified some variable domains, the solver decides which other propagators it has to reconsider, or *schedule*, for propagation due to the changes; and (2) the solver picks one of the scheduled propagators for execution.

The standard technique for (1) is to specify which *events* a propagator *depends on.* An event describes how a particular variable has been changed, and a propagator that depends on a certain event will only be scheduled if that event happens. For example, a propagator for the constraint $x \leq y$ only depends on the events that the lower bound of x or the upper bound of y changes. In any other case, it cannot prune any variable domain and is thus at a fixpoint.

For (2), it is essential to organize the scheduled propagators in a *queue* (as opposed to a stack, which can suffer from starvation). Furthermore, it can be beneficial to *prioritize* propagators by their estimated runtime cost of propagation, first picking cheap propagators, and only later executing more expensive ones that can then take advantage of the cheap pruning done before.

Schulte and Stuckey [16] have performed a thorough evaluation of event-based and prioritized propagation.

Contributions. This paper develops implementation techniques and concrete data structures for efficient event-based, prioritized propagation control. It establishes the notions of *modification events* and *propagation conditions*, which are used to describe sets of events in a compact way. The paper develops indexed dependency arrays as an efficient data structure for the dependencies, and priority bucket queues for prioritized propagator scheduling. The developed architecture has been fully implemented and is the basis of Gecode [5].

Plan. After setting up the preliminaries in Section 2, Section 3 introduces propagation conditions and modification events, and Section 4 develops indexed dependency arrays. Section 5 discusses prioritized propagation and presents the priority bucket queue. Section 6 discusses related work, and Section 7 concludes the paper.

2 Preliminaries

This section recapitulates propagator-centered propagation, presenting the basic notions such as variables, propagators, and events.

Variables, constraints, domains, propagators. Constraint problems are modeled in terms of *variables*, representing the objects of the problem, and *constraints*, representing the relations that the objects are engaged in. In propagation-based constraint solvers, each variable has a *variable domain*, a set of values that it can take.

A *domain* d is a function mapping variables to sets of values. The set of values in d for a particular variable x, $d(x)$, is called the *variable domain* of x. A domain d is *stronger* than a domain d', written $d \subseteq d'$, iff for all variables x, $d(x) \subseteq d'(x)$.

Each constraint is implemented by a *propagator*, whose task it is to *prune* the domains. A propagator is a function that takes a domain and returns a stronger domain, possibly removing inconsistent values from the variable domains.

The constraint solver executes the propagators in turn until none of them can contribute any more pruning. The system thus reaches a mutual fixpoint. For a discussion of the properties of propagators that guarantee fixpoints, see [17]. Constraint propagation is incomplete, so the solver interleaves propagation with tree search, which yields a sound and complete solution procedure.

A propagator is *subsumed* in a domain d if it is at a fixpoint for any stronger domain $d' \subseteq d$, i.e., $p(d') = d'$. As subsumed propagators do not prune in the remaining subtree of the search, they can be removed.

In this paper, we will use the terms variable, domain, and propagator more freely, sometimes referring to the mathematical objects introduced above, sometimes referring to objects as implemented in a concrete constraint solver.

Events. An event describes how a variable domain was modified. Events are used to determine which propagators have to be re-executed when a variable domain is modified.

In the most basic event system, propagators only notify the solver of which variables they are *interested in.* For example, a propagator for the constraint $x \leq y$ only needs to be re-executed if either x or y has been modified since its last execution.

Most solvers use more complex events, which not only describe *which* domains have changed, but also *how* they have changed. A typical system for integer variables may consist of the events asn (the variable was assigned a value), lbc (the lower bound changed), ubc (the upper bound changed), and dmc (the domain changed). Events can overlap, for example, whenever the event lbc happens, also dmc happens. Looking again at the $x \leq y$ example, the propagator only needs re-execution when a lbc event on x or a ubc event on y happens.

An event e is characterized by a condition $e(d(x), d'(x))$ for two variable domains $d'(x) \subseteq d(x)$. For variable domains $d''(x) \subseteq d'(x) \subseteq d(x)$, it must satisfy $e(d(x), d''(x))$ if and only if $e(d(x), d'(x))$ or $e(d'(x), d''(x))$. An event always describes an actual domain modification: if $d(x) = d'(x)$, then $e(d(x), d'(x))$ must be false.

We define the set $\text{events}(d(x), d'(x)) := \{e \mid e(d(x), d'(x))\}$ for two variable domains $d'(x) \subseteq d(x)$. This construction ensures that events are *monotonic*, they are never discarded by further changes to a variable domain. For three variable domains $d''(x) \subseteq d'(x) \subseteq d(x)$, monotonicity implies that $\text{events}(d(x), d''(x)) = \text{events}(d(x), d'(x)) \cup \text{events}(d'(x), d''(x))$.

We say that an event e *implies* an event e' (written $e \rightarrow e'$) if and only if for all variable domains $d(x)$ and $d'(x)$, $e(d(x), d'(x))$ implies $e'(d(x), d'(x))$.

Our example event system features the following implications:

$$\begin{tikzcd}
 & \mathsf{dmc} & \\
\mathsf{lbc} \arrow[ur] & \mathsf{asn} \arrow[u] & \mathsf{ubc} \arrow[ul]
\end{tikzcd}$$

A simple propagation loop. The goal of the propagation loop is to compute a fixpoint of all propagators. Instead of re-executing all propagators in a loop, constraint solvers typically keep track of which propagators are known to be at a fixpoint; these are called *idle.* All other propagators are called *active* and are kept in a data structure called the *agenda.* The propagation loop picks a propagator p from the agenda, executes it, and determines which other propagators need to be put on the agenda again, based on what kind of events p caused. When the agenda is empty, propagation stops with a mutual fixpoint of all propagators.

The following pseudo-code is a high-level implementation of the propagation loop, assuming an object-oriented implementation language where q is an agenda of propagator objects, and each propagator object has a method `propagate` that (destructively) performs the propagation. This code leaves out the details of how the agenda q is implemented, or how the dependent propagators P are computed exactly. The rest of this paper fills in these details.

```
fixpoint()
1 while not $q$.empty()
2     $p = q$.head()
3     $q$.idle($p$)
4     $p$.propagate()
5     $P$ = propagators depending on events generated by $p$
6     $\forall p' \in P :$ $q$.enqueue($p'$)
```

Recomputation versus trailing. Constraint solvers *backtrack* during search: The solver builds one path in the tree, and if it hits a failed state, it returns to a previous search state, chooses a different alternative, and continues the search.

Backtracking can be implemented using two different techniques, *trailing* and *recomputation* [15]. The former relies on storing *undo* information, which can be used to revert all the changes made during propagation and search between the failed state and the state that the solver is backtracking to. The latter, on the other hand, stores information how to *redo* the steps between a copy of the state higher up in the tree and the backtracking target state.

This paper describes the architecture of the Gecode [5] propagation kernel, which is based on copying and recomputation. The data structures are therefore designed to be memory efficient, because in a copying system, memory efficiency immediately yields a runtime advantage, too.

3 Modification Events and Propagation Conditions

This section introduces the notions of *modification events* and *propagation conditions*, which both capture different sets of events that a solver needs for propagator scheduling.

In order to perform propagator scheduling, the solver needs to record which events happen on which variables, and then determine the propagators that depend on these events. We are thus dealing with two different types of sets of events. A *modification event* is the set of events that happen when a variable domain is modified. A *propagation condition* is the set of events on a particular variable that a certain propagator depends on.

For example, when increasing the lower bound of an integer variable, the corresponding modification event may be $\{\mathsf{lbc}, \mathsf{dmc}\}$. A propagator that reacts to changes of either bound of a variable x has propagation condition $\{\mathsf{lbc}, \mathsf{ubc}\}$ on x. The solver schedules those propagators where for any variable, the intersection of the modification event and the propagation condition is not empty.

Propagator scheduling is one of the critical operations of a constraint solver kernel. The goal is therefore to make two operations as efficient as possible:

maintaining the set of events that has happened, and computing its intersection with the propagation conditions.

This can be achieved by representing each modification event and each propagation condition by an integer that can be used as an index into an appropriate data structure that represents the dependencies, as developed in the following section. The remainder of this section shows how to arrive at minimal definitions for propagation conditions and modification events, in order to keep the integer encoding as small as possible.

Propagation conditions. Some event sets are equivalent for the purpose of propagator scheduling. For example, the event sets $\{\mathsf{lbc}, \mathsf{dmc}\}$ and $\{\mathsf{dmc}\}$ are equivalent, as lbc implies dmc. More generally, if a propagator should be scheduled by an event e, then it will also be scheduled by any event e' such that $e' \to e$. We therefore restrict propagation conditions to the equivalence classes with respect to reverse implication:

A *propagation condition* π is a set of events that is closed under the converse of implication: for any two events e and e', if $e \in \pi$ and $e' \to e$, then $e' \in \pi$.

Our example event system yields the following propagation conditions:

$$\begin{array}{ccc}
 & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\
 & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}\} & \\
\{\mathsf{asn}\} & \{\mathsf{asn}, \mathsf{lbc}\} & \{\mathsf{asn}, \mathsf{ubc}\} \\
\{\mathsf{lbc}\} & \{\mathsf{ubc}\} & \{\mathsf{lbc}, \mathsf{ubc}\}
\end{array}$$

Modification events. Event-directed scheduling requires determining the set events$(d(x), d'(x))$ when the domain changes from d to d'. Instead of computing this set from two given domains, an implementation will maintain a set of events incrementally during propagation. Such a set of events represents the modifications between two variable domains, and we call it a *modification event.*

Similar to propagation conditions, not all sets of events actually occur as modification events in practice. For instance, the set $\{\mathsf{lbc}\}$ cannot occur, as an lbc event always implies that a dmc event has happened, too. In general, we therefore define that a *modification event me* is a set of events that is closed under implication.

The definition implies that modification events are closed under union. This makes it easy to maintain the set of events incrementally for any variable. Again, as there are only few events in typical event systems, we can enumerate all modification events for a particular event system. An implementation can therefore represent modification events as small integers. In our example event system, we can simplify even further, as the asn event always implies *either* lbc or ubc. Therefore, the modification event $\{\mathsf{asn}, \mathsf{dmc}\}$ does not have to be represented.

The following modification events are therefore derived from the example event system:

$$\begin{array}{ccc}
 & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\
\{\mathsf{asn}, \mathsf{lbc}, \mathsf{dmc}\} & \{\mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \{\mathsf{asn}, \mathsf{ubc}, \mathsf{dmc}\} \\
\{\mathsf{lbc}, \mathsf{dmc}\} & \{\mathsf{dmc}\} & \{\mathsf{ubc}, \mathsf{dmc}\}
\end{array}$$

The asn event. The event system presented above can be simplified further by treating the asn event as if it implies any other event:

$$\begin{tikzcd}
 & \mathsf{dmc} & \\
\mathsf{lbc} \arrow[ur] & & \mathsf{ubc} \arrow[ul] \\
 & \mathsf{asn} \arrow[uu] \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Consequently, the propagation conditions are now:

$$\begin{array}{ccc} & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\ & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}\} & \\ \{\mathsf{asn}\} & \{\mathsf{asn}, \mathsf{lbc}\} & \{\mathsf{asn}, \mathsf{ubc}\} \end{array}$$

And the modification events look as follows:

$$\begin{array}{ccc} & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\ & \{\mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\ \{\mathsf{lbc}, \mathsf{dmc}\} & \{\mathsf{dmc}\} & \{\mathsf{ubc}, \mathsf{dmc}\} \end{array}$$

In this simplified system, all propagators are executed at least once when all variables are assigned. In a solver that is based on copying, this additional strong invariant proves useful because it means that all propagators will eventually test for and report subsumption. Subsumed propagators are removed from the system and therefore do not have to be copied any longer, which saves memory and runtime. Gecode uses the simplified system for all its variable types[3].

4 Dependencies

This section develops the data structures that represent the propagator dependencies.

The propagation loop in Section 2 defined scheduling in a high-level way:

$P =$ propagators depending on events generated by p
$\forall p' \in P : \; q.\texttt{enqueue}(p')$

Instead of collecting the events and then computing the set of propagators to schedule, we embed the scheduling into the variable modification operations. Whenever a propagator updates a variable domain in its `propagate` method, the variable computes the corresponding modification event me and schedules all propagators with corresponding propagation conditions.

The dependency data structure must therefore provide three basic operations:

- $x.\texttt{subscribe}(p, \pi)$ adds the propagator p to the dependencies of x at propagation condition π.

[3] For integer variables, Gecode uses a single event bnd instead of two separate events lbc and ubc. This was shown to be sufficient and increase efficiency [16]. We chose the more complicated setup for this paper to be able to discuss the full system.

- x.`cancel`(p, π) removes the propagator p from the dependencies of x at propagation condition π.
- x.`schedule`(π_i, π_j) iterates over the propagators between propagation conditions π_i and π_j of x to schedule them.

The most important operation is iteration. It is performed whenever an event happens, so we will design the data structure to be as efficient as possible in this case. For subscription and canceling, efficiency is not quite as important, as they happen less frequently.

We enforce a strong contract between propagators and the dependencies. A propagator must not cancel subscriptions that it has not established before, and it must cancel all its subscriptions when it ceases to exist (e.g. because it detects subsumption). We will discuss below why the invariants enforced by this contract are important.

Furthermore, the data structure must be *backtrackable*, i.e., the solver must be able to revert it to a previous state. We will discuss how to achieve this using either copying or trailing.

4.1 Indexed dependency arrays

The most efficient data structure for fast iteration is an array. So, in principle, we could have one array of propagators per propagation condition. However, in practice a single modification event often triggers several propagation conditions.

The dependencies are therefore stored in a single *dependency array* `dep`, sorted by propagation condition. In addition, we maintain the *dependency index* `idx`, which partitions the dependency array by propagation condition. Figure 1 shows this architecture.

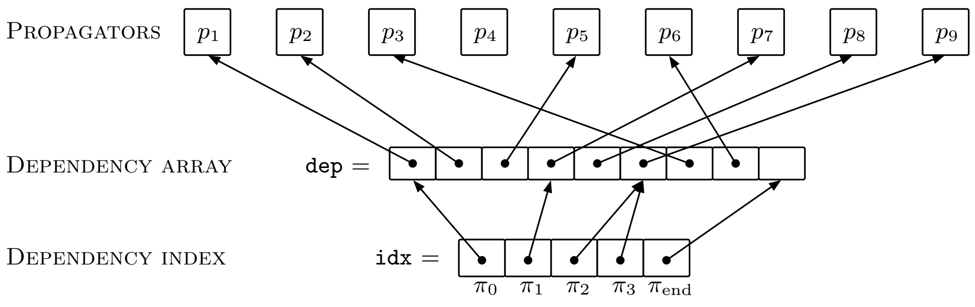

Fig. 1: Dependency data structures

For each propagation condition π_i, the dependency index points to the first propagator in the dependency array that is subscribed with π_i. For example, the first propagator subscribed with π_1 in Figure 1 is `dep`[`idx`$[\pi_1]] = p_7$. To iterate

```
subscribe($p, \pi_i$)
1  for $j = k$ downto $i$
2      dep[idx[$\pi_{j+1}$]] = dep[idx[$\pi_j$]]
3      idx[$\pi_{j+1}$] = idx[$\pi_{j+1}$] + 1
4  dep[idx[$\pi_i$]] = $p$
```

(a)

```
cancel($p, \pi_i$)
1  $j_p$ = idx[$\pi_i$]
2  while dep[$j_p$] $\neq p$ do $j_p = j_p + 1$
3  dep[$j_p$] = dep[idx[$\pi_{i+1}$] − 1]
4  for $j = i + 1$ to $k$
5      dep[idx[$j$] − 1] = dep[idx[$\pi_{j+1}$] − 1]
6      idx[$\pi_j$] = idx[$\pi_j$] − 1
7  idx[$\pi_{\text{end}}$] = idx[$\pi_{\text{end}}$] − 1
```

(b)

Fig. 2: Subscribing (a) and canceling (b)

over all propagators subscribed with a certain propagation condition π_i, we start at $\texttt{dep}[\texttt{idx}[\pi_i]]$ and finish at $\texttt{dep}[\texttt{idx}[\pi_{i+1}]-1]$. There is one additional propagation condition, π_{end}, so that π_{i+1} and $\texttt{idx}[\pi_{i+1}]$ are defined for all propagation conditions π_i.

Again for the example in Figure 1, scheduling all propagators that are subscribed with propagation condition π_1 would amount to scheduling the propagators p_7 and p_8. No propagator is subscribed with π_2 (as $\texttt{idx}[\pi_2] = \texttt{idx}[\pi_3]$). Through this index data structure, iterating over all propagators subscribed with a particular propagation condition is as efficient as possible, taking constant time per propagator, and with low constants in practice.

We can now define the method $\texttt{schedule}(\pi_i, \pi_j)$, which schedules all propagators starting at propagation condition π_i and finishing at propagation condition π_j. For the above example, $\texttt{schedule}(\pi_0, \pi_2)$ would thus schedule p_1, p_2, p_5, p_7, and p_8. The following code implements `schedule`, assuming a method `enqueue` that puts a propagator into the right queue:

```
schedule($\pi_i, \pi_j$)
for $k$ = idx[$\pi_i$] to idx[$\pi_{j+1}$] − 1
    enqueue(dep[$k$])
```

4.2 Subscribing and cancelling

The remaining operations to be defined are subscribing and canceling. Subscribing a propagator p with propagation condition π_i means adding it at the appropriate position to the dependency array and modifying the index accordingly. Assuming that the dependency array is resized dynamically, subscription can be implemented to have amortized run-time $O(k-i)$ as shown in Figure 2(a). First, some space is cleared for the new subscription at $\texttt{dep}[\texttt{idx}[\pi_i]]$ (lines 1–3). Then the new subscription is entered (line 4).

A subscription can be canceled in $O(\texttt{idx}[\pi_{i+1}] - \texttt{idx}[\pi_i] + i)$ (Figure 2(b)). The **while** loop in line 2 finds the index of p in the dependency array (note that the loop is only correct if the propagator is actually subscribed to the variable). After finding the index j_p, the position `dep`$[j_p]$ is reused (lines 3–7).

Assigned variables. Variables that are assigned, i.e., whose domain is a singleton, cannot produce any events any more. Therefore, subscribing and canceling on these variables is useless, and the overhead can be avoided by a simple check.

Scheduling upon subscription. A propagator typically subscribes to its variables when it is created. A newly created propagator must be scheduled for execution (otherwise, no propagation would happen for the first fixpoint). There is one exception: If the propagator subscribes only with $\{\mathsf{asn}\}$ propagation conditions and none of the variables is assigned, it does not have to be scheduled.

Rewriting. A special case of canceling and subscribing is *propagator rewriting*. E.g., consider the case when all but two variables in a long linear equation are assigned. Then the propagator for the linear equation can be rewritten to a simpler, more efficient binary version. The old propagator cancels its subscriptions, and the new propagator subscribes. Note that this order is essential: it means that the dependency arrays will not need resizing, as the old propagator leaves enough space for the new one.

4.3 Scheduling

When a variable domain is modified resulting in a modification event me, the variable determines which ranges of propagation conditions intersect with me (this is implemented as a lookup table), and then schedules each of those ranges π_i, π_j using `schedule`(π_i, π_j).

Assume the following enumeration of the propagation conditions of our simplified event system:

$$\begin{array}{ccc} & \pi_4 = \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\ & \pi_3 = \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}\} & \\ \pi_0 = \{\mathsf{asn}\} \qquad & \pi_1 = \{\mathsf{asn}, \mathsf{lbc}\} & \qquad \pi_2 = \{\mathsf{asn}, \mathsf{ubc}\} \end{array}$$

The scheduling then looks as follows:

```
case $me$ of
    $\{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\}$: $x$.schedule($\pi_0, \pi_4$)
    $\{\mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\}$: $x$.schedule($\pi_1, \pi_4$)
    $\{\mathsf{lbc}, \mathsf{dmc}\}$: $x$.schedule($\pi_1, \pi_1$); $x$.schedule($\pi_3, \pi_4$)
    $\{\mathsf{ubc}, \mathsf{dmc}\}$: $x$.schedule($\pi_2, \pi_4$)
    $\{\mathsf{dmc}\}$: $x$.schedule($\pi_4, \pi_4$)
```

4.4 Copying and trailing

Dependencies can be modified dynamically. For example, a propagator may “lose interest” in some of its variables, if it can determine that no further change of their domains will cause any propagation. Or, propagators can replace themselves with simpler versions. Or, in the most dynamic case, the propagator only needs subscriptions to a dynamically changing subset of the variables, such as for the *watched literals* technique [12,7].

In most of these cases, the dependencies must be *backtrackable*. We can achieve this either by *copying* them, or by *trailing* any changes [15].

Copying. When copying the dependency arrays, it is advantageous to allocate enough memory for all dependency arrays of all variables in one block. That way, the copy will be compact and the overhead for allocation is low.

Trailing. Dynamic dependencies change infrequently (typically much less than variable domains). A simple trailing scheme that stores a function pointer together with the data which dependency needs to be changed works well [15].

5 Propagator Priority Queue

This section develops a priority queue data structure that provides efficient operations for priority-based propagator scheduling. We first recapitulate priority-based propagator scheduling, and then present the priority bucket queue.

5.1 Propagator priorities

Schulte and Stuckey showed [16] that propagator priorities are an important technique for efficient propagator scheduling[4].

Obviously, prioritized propagation requires some measure to determine a propagator’s priority. For this paper, we assume that priorities model the estimated *runtime cost* of propagation. The execution of propagators with high estimated runtime cost is postponed, so that they can take advantage of the pruning of the cheaper propagators that are run first.

A straightforward way to estimate the cost is to classify the propagators according to their algorithmic complexity. We will use the following system of costs and priorities: `unary` = 7, `binary` = 6, `ternary` = 5, `linear` = 4, `quadratic` = 3, `cubic` = 2, `veryslow` = 1. The names suggest the arity of the corresponding propagator (for the highest three priorities), or the asymptotic run-time for n-ary propagation algorithms. The cost of propagation often changes dynamically. For instance, a typical algorithm for propagating linear equations has an asymptotic run-time linear in the number of *unassigned* variables. Accordingly, when all but three variables are assigned, the cost should be reported as `ternary` instead of `linear`.

[4] Priorities are particularly useful for implementing *staged propagation* [16], which is out of the scope of this paper, but can be implemented easily on top of the presented prioritized, event-based system.

5.2 Priority bucket queue

For our purposes, the priority queue must provide four operations:

- `enqueue`(p) adds propagator p at the priority determined by its `cost` method. If p is already in the queue, it is re-prioritized according to its current cost.
- `empty`() tests whether the queue is empty.
- `head`() returns the oldest propagator at the highest priority.
- `idle`(p) removes propagator p from its current queue and marks it as idle.

All four operations are performed extremely often during the fixed point computation, and are hence crucial for the solver's performance.

Common algorithms for priority queues are based on variations of the *heap* data structure (see for example [11,3]). Heaps support an arbitrary number of priority levels. For most types of heaps, the run-time complexity of the enqueue and dequeue operations depends on the number of elements in the queue. For example, using binary heaps, both operations require time in $O(\log n)$ if n is the number of elements in the queue.

In order to make `enqueue` and `dequeue` as efficient as possible, we restrict the number of priority levels to a small, fixed set of integers (such as the cost values introduced above). Then, a priority queue based on *buckets* can be used (see [11]), providing constant-time `enqueue` and `dequeue` operations. We will now see how a bucket-based priority queue of propagators can be implemented.

The bucket queue. A bucket-based priority queue consists of an array of doubly-linked lists of propagators. The list at array index i represents the queue of propagators at priority i. Furthermore, a propagator can only be in one queue at a time. We can hence embed the links for the doubly-linked lists into the propagator objects. In addition to the lists for each priority, the solver maintains the list of idle propagators, the so-called *idle queue* (in our case modeled as priority 0). The invariant is then that a propagator is always in exactly one queue.

Each list of propagators is cyclic and terminated by a *sentinel element*. The sentinels are kept in an array that represents the priority queue. Figure 3 depicts an example of this architecture. An empty queue is depicted as a sentinel with a simple cycle (as at priority $k-1$).

This implementation of a bucket queue yields efficient access. Inserting and removing a propagator can be done in constant time—unlink it from its current queue, and link it at the position before the sentinel element of the target queue. Finding the next propagator to schedule costs at most k tests. Queues are managed as follows:

- The solver can access the queue with priority i as $Q[i]$.
- p.`next`() returns the propagator following p in the linked list.
- p.`unlink`() removes propagator p from its current queue.
- $Q[i]$.`tail`(p) adds p as the last propagator to the queue with priority i.

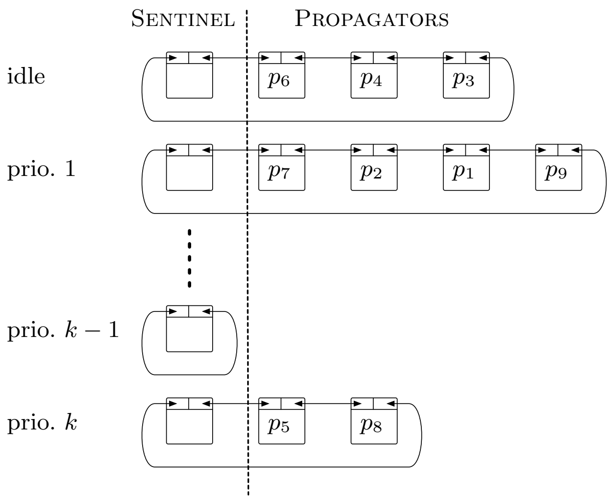

Fig. 3: Propagators in prioritized queues

A propagator is added to the queue that corresponds to its cost, which it reports using the `cost` method. The following code implements the `enqueue`, `head`, and `idle` methods, as well as a method `empty` that reports whether all queues except the idle queue are empty, indicating that propagation has reached a fixpoint.

```
enqueue($p$)
1  $p$.unlink()              // remove $p$ from current queue
2  $Q[p$.cost()$]$.tail($p$)    // put $p$ into new queue
```

```
head()
1  for $i = k$ downto 1
2      if $Q[i]$.next() $\neq Q[i]$ then return $Q[i]$.next()
```

```
idle($p$)
1  $p$.unlink()              // remove $p$ from current queue
2  $Q[0]$.tail($p$)             // put $p$ into idle queue
```

```
empty()
1  for $i = k$ downto 1
2      if $Q[i]$.next() $\neq Q[i]$ then return FALSE
3  return TRUE
```

Avoiding re-scheduling of propagators. The code above schedules a propagator anytime its propagation condition matches a modification event. This involves unlinking the propagator, re-evaluating its cost, and putting it into the queue again. If the propagator already was in the queue due to a previous variable modification, the overhead of re-scheduling it should be avoided.

As an indication of whether the propagator is already in the correct queue, we will use the set of events since its last invocation, called the *modification event delta* Δme, and store it in every propagator. Before executing a propagator, it is set to the empty set. The `schedule` method then takes the modification event that caused the scheduling as an additional argument:

```
schedule($\pi_i, \pi_j, me$)
1  for $k$ = idx[$\pi_i$] to idx[$\pi_{j+1}$] $- 1$
2      if $me \not\subseteq$ dep[$k$].$\Delta me$ then
3          dep[$k$].$\Delta me$ = dep[$k$].$\Delta me \cup me$
4          enqueue(dep[$k$])
```

Line 2 makes sure that a propagator is only added to the queue if the new modification event *me* is not already contained in the propagator's modification event delta. This is correct because if $me \subseteq$ `dep`$[k].\Delta me$, then the propagator has already been put into the queue before through line 3.

Memory and run-time efficiency. The bucket queue is as efficient as possible, both in terms of memory requirements and run-time. The asymptotic run-time for all operations is a small constant if we restrict the priorities to a small, fixed set. Priority-based scheduling with a fixed number of priorities is the standard in all propagation-based solvers (see Section 6), and has proven effective in practice. In terms of memory, this architecture requires two pointers per propagator for the doubly-linked list, which, theoretically, is an overhead of one pointer per propagator compared to an array-based implementation. However, using arrays for the queues would require dynamic resizing, which again costs memory and/or runtime. In practice, embedding the double links in the propagator objects is therefore without overhead.

6 Related Work

Most constraint solvers are based on propagator-centered, event-directed, prioritized propagation as presented in this paper.

▶ **SICStus Prolog** [1] employs a priority queue of propagators, using two priority levels.
▶ **Mozart OZ** [13] maintains a two-level priority queue of propagators, and scheduling is based on events. Mozart offers additional priorities for non-monotonic propagators (as described in [14]): each non-monotonic propagator gets its own priority level, effectively fixing the order in which non-monotonic propagators are run and hence maintaining the guarantee to compute a unique fixed point.
▶ **Eclipse Prolog** [18] has a feature called *suspension*, which attaches a Prolog goal to finite domain variables. When the variable domain changes, the goal, which may implement a propagator, is scheduled. The Eclipse system features twelve priority levels, but like SICStus and Mozart, its finite domain solver only makes use of two levels.

▶ **B-Prolog** [19] queues *action rules*, which correspond to propagator invocations. A particularity of B-Prolog is that the same propagator can appear several times in the queue, once for each variable that triggered its scheduling.
▶ **Choco** [9] provides a sophisticated priority system with seven levels and both FIFO and LIFO scheduling, but is not propagator-centered, as explained below.

Variable-centered propagation. In our setup, the agenda holds the propagators that are not necessarily at a fixed point. Some solvers, notably ILOG Solver [8], Choco [2], and Minion [6], use an alternative approach: an *agenda of modified variables* instead of an agenda of propagators. A solver that bases scheduling on an agenda of variables performs *variable-centered propagation.*

ILOG Solver, Choco, and Minion actually implement a hybrid approach. When a modified variable is taken from the queue, its dependent propagators can either be run immediately, or put into a queue of propagators.

The advantage of variable-centered over propagator-centered propagation is that whenever a propagator is invoked, the information which variable exactly triggered the propagation is directly available. The propagator can take this information into account in order to compute the new domain *incrementally*, without recomputing from scratch. Lagerkvist and Schulte [10] show how *advisors* can be used to implement incremental propagation in a propagator-centered system. Their implementation is a straightforward extension of the data structures presented in this paper.

Propagator queues. It is folklore knowledge that propagators should be scheduled in a FIFO fashion. Similarly, using events to prevent gratuitous scheduling of propagators has been used in constraint solvers for a long time—one can argue that it was already present in the early DPLL algorithm [4]. Schulte and Stuckey [16] perform detailed experiments with different agenda strategies as well as priority queues, substantiating this folklore knowledge with empirical evidence. They also provide a comprehensive study of events, including a detailed experimental evaluation of different event schemes, fixed point reasoning, and staged propagation.

7 Conclusions

This paper developed an architecture and concrete data structures for event-based, prioritized propagator scheduling. It introduced the notions of *modification events* and *propagation conditions*, which capture exactly the sets of events that occur during propagator scheduling. Based on these notions, the paper developed *indexed dependency arrays*, an efficient data structure for storing and accessing the dependency information.

Furthermore, the paper presents the design of *priority bucket queues*, which are used to implement propagator scheduling prioritized by estimated cost of propagation.

The presented data structures are the core of the Gecode constraint solver, one of the most efficient solvers available today.

Acknowledgements. The authors would like to thank Mikael Z. Lagerkvist for many discussions about details of the Gecode architecture, and the anonymous reviewers for helpful comments that improved this version of the paper.

References

1. Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *PLILP'97*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.
2. CHOCO, 2010. http://choco-solver.net.
3. Thomas M. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd ed. edition, 2001.
4. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
5. Gecode, generic constraint development environment, 2010. http://www.gecode.org.
6. Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI 2006*, pages 98–102. IOS Press, 2006.
7. Ian P. Gent, Christopher Jefferson, and Ian Miguel. Watched literals for constraint propagation in Minion. In Frédéric Benhamou, editor, *CP 2006*, volume 4204 of *LNCS*, pages 182–197. Springer, 2006.
8. ILOG Solver, part of ILOG CP, 2009. http://www.ilog.com/products/cp.
9. F. Laburthe. Choco: Implementing a CP kernel. In *TRICS*, pages 71–85, September 2000.
10. Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In Christian Bessière, editor, *CP 2007*, volume 4741 of *LNCS*, pages 409–422. Springer, 2007.
11. Kurt Mehlhorn and Stefan Näher. *LEDA - A platform for combinatorial and geometric computing*. Cambridge University Press, 1999.
12. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *DAC '01*, pages 530–535, New York, NY, USA, 2001. ACM Press.
13. The Mozart programming system, 2009. http://www.mozart-oz.org.
14. Tobias Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.
15. Raphael M. Reischuk, Christian Schulte, Peter J. Stuckey, and Guido Tack. Maintaining state in propagation solvers. In Ian Gent, editor, *CP 2009*, volume 5732 of *LNCS*, pages 692–706. Springer, 2009.
16. Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, dec 2008.
17. Christian Schulte and Guido Tack. Weakly monotonic propagators. In Ian Gent, editor, *Proceedings of the 15th international conference on principles and practice of constraint programming*, volume 5732 of *LNCS*, pages 723–730. Springer, 2009.
18. Mark Wallace, Stefano Novello, and Joachim Schimpf. Eclipse: A platform for constraint logic programming. Technical report, IC Parc, Imperial College, London, 1997.
19. Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming*, 6:483–507, 2006.