Composing constraint solvers
Zoeteweij, P.

Citation for published version (APA):
Chapter 7

Applications of Nested Search

Nested search entails that a limited branch-and-propagate tree search is performed during constraint propagation. In this chapter we propose a generic reduction operator for nested search, and investigate the extent to which it can be used to express a number of well-known techniques, from different application domains, for improving the efficiency of constraint solving. Generalizing solving techniques has several advantages. From a modeling perspective, the technique extends easily to other application domains, and from a software engineering perspective, it avoids duplicate code with only small variations for different applications.

This is the first of three chapters that demonstrate the use of OpenSolver as a software component. In this case OpenSolver implements the generic reduction operator for nested search. Consequently, we gain rich facilities for expressing the nested search, at the cost of an overhead for using a general-purpose constraint solver for very specific search problems. We demonstrate that despite this overhead, our approach leads to a viable implementation of the techniques that we are interested in.

7.1 Introduction

Constraint propagation is usually implemented as the repeated application of reduction operators that enforce some form of local consistency, such as arc consistency, or an approximation thereof. In this context ‘local’ means that only individual constraints, applying to a small subset of the variables are considered when removing values from the domains of the variables. For arithmetic constraints, these individual constraints are usually the result of a decomposition of complex constraints into atomic constraints, for which the resulting form of consistency is weaker than for the original constraints.

Sometimes the efficiency of constraint solving can be improved by enforcing a stronger, less local form of consistency. Operators that enforce such a form
of consistency typically still update the domain of a single variable, but more than a single constraint is considered when removing values. This is achieved by actively trying different subdomains for the variable that we want to reduce, and then verifying by means of constraint propagation that this does not violate the combined constraints. The resulting trial-and-refutation mechanism can be seen as a limited form of branch-and-propagate search, where branching is on the domain of a single variable only.

We will look at two examples of such operators: enforcing box consistency for arithmetic constraints, and removing (shaving off) unfeasible activity starting times and completion times in scheduling problems. Box consistency can be explained as to consider all atomic constraints in the decomposition of a single user constraint, and is therefore a stronger notion of consistency than what is achieved for the decomposition alone. Shaving considers the full set of capacity and precedence constraints in a scheduling problem, when trying to refute possible values for the variable that it is applied to. We will show that both operators can be expressed as applications of a generic operator for nested search. A third application is optimization by means of a bisection search in the range for the outcome of an objective function.

Instead of implementing dedicated operators for each of these techniques, our approach entails that these operators are composed from a limited set of basic facilities, which includes the generic operator for nested search. This is an advantage in hybrid solvers, supporting multiple domain types, where we want to avoid that techniques like shaving are available only for a subset of the domain types. It is of even greater importance for open-ended solvers, where the set of domain types can be extended. A disadvantage is that a generic operator is likely not as efficient as a dedicated implementation. We provide the results of experiments that show that despite a general-purpose constraint solver is used for the nested search, we still obtain a workable implementation.

The rest of this chapter is structured as follows. The reduction operator for nested search is defined in Section 7.2. In Section 7.3 we show how it can be used to define optimization, box consistency, and shaving. Section 7.4 details the implementation of the operator, and Section 7.5 describes the experiments.

### 7.2 An Operator for Nested Search

In this section we propose a generic operator for nested search. It is presented as a domain reduction function that, like all other DRFs, is used in the context of a branch-and-propagate search for a solved form of an ECSP. The DRF is evaluated during the constraint propagation phase, and may yield a smaller domain for one of the variables of the ECSP. Since we are defining a generic operator, the DRF for this operator is parameterized with some extra information. This extra information instantiates the DRF to perform a particular reduction step.
7.2. An Operator for Nested Search

Inner and Outer ECSP

In this case, the information that parameterizes the DRF is another ECSP. So unlike the DRFs that we encountered so far, the DRF for nested search involves more than one ECSP:

1. The ECSP whose domains are reduced by the DRF. For regular DRFs, this is the only ECSP that we need to consider. Here we will call this ECSP the outer ECSP.

2. The ECSP that parameterizes the DRF. We will call this ECSP the parameter ECSP.

3. The parameter ECSP is combined with the domains that the DRF is evaluated for (detailed below). This results in a third ECSP that we will call the inner ECSP.

Evaluation of the DRF for nested search involves solving the inner ECSP by means of a branch-and-propagate search. The term nested search refers to this branch-and-propagate search on the inner ECSP, and emphasizes that it occurs as a single domain reduction step in the encompassing branch-and-propagate search on the outer ECSP.

The inner ECSP is a modified version of the parameter ECSP. It is obtained by replacing the domains of certain variables with the domains that the DRF is evaluated for. We introduce the following notation to describe this modification.

7.2.1. Definition. Let \( P \) be an ECSP with variables \( x_1, \ldots, x_m \) and corresponding domain types \( T_1, \ldots, T_m \). For \( D_1 \in T_1, \ldots, D_n \in T_n \), and \( n \leq m \), let \( (P, (D_1, \ldots, D_n)) \) denote the ECSP obtained by replacing in \( P \) the domains of the first \( n \) variables \( x_1, \ldots, x_n \) with \( D_1, \ldots, D_n \), and projecting the new sequence of domains on the constraints.

7.2.2. Example. For

\[
P := \langle C_P ; x, y, z \in \{0, 1\} ; D_x, D_y, D_z \in \mathbb{Z} ; A_x, A_y, A_z \rangle.
\]

\( (P, (\{0, 1\}, \{1, 2\})) \) denotes the ECSP obtained by replacing the domain of \( x \) in \( P \) with \( \{0, 1\} \), and the domain of \( y \) in \( P \) with \( \{1, 2\} \):

\[
(P, (\{0, 1\}, \{1, 2\})) = \langle C'_P ; x \in \{0, 1\}, y \in \{1, 2\}, z \in \{0, 1, 2\}
\]

\[
; D_x, D_y, D_z \in \mathbb{Z}
\]

\[
; A_x, A_y, A_z
\]

where \( C'_P \) is \( C_P[\{0, 1\}, \{1, 2\}, \{0, 1, 2\}] \), the projection of the new domains on the constraints of \( P \). The domains of \( x \) and \( y \) in \( P \), and their sets of final domains are irrelevant for the purpose of this notation. The projection is needed only to maintain the property that constraints are subsets of Cartesian products of domains.
Definition of the Operator

The operator for nested search is now defined as follows.

7.2.3. Definition. Given

- an ECSP $P$ with at least $n$ variables, and
- a variable $x_j$ with $1 \leq j \leq n$,

let $T_1, \ldots, T_n$ be the domain types of the first $n$ variables in $P$. We define the function

$$f_{P,x_j} : T_1 \times \cdots \times T_n \rightarrow T_j$$

as follows:

$$f_{P,x_j}(D_1, \ldots, D_n) = \begin{cases} D'_j & \text{if } (P, \langle D_1, \ldots, D_n \rangle) \text{ is consistent} \\ \emptyset & \text{otherwise} \end{cases}$$

where if $P' := (P, \langle D_1, \ldots, D_n \rangle)$ is consistent, $D'_j$ is the domain of $x_j$ in a $\gamma$ solved form of $P'$, for some notion of local consistency $\gamma$.

To elucidate this definition, note that

- $P$ and $P'$ are the argument ECSP and inner ECSP, respectively. $f_{P,x_j}$ is a domain reduction function for an outer ECSP $Q$ that has variables $x_1, \ldots, x_n$ and the corresponding domain types $T_1, \ldots, T_n$ in common with $P$. For our applications, $P$ can always be defined such that $x_1, \ldots, x_n$ is a subsequence of the variables of $Q$.

- If $P'$ is consistent, there may exist more than a single $\gamma$ solved form of $P'$. In this case, the definition is not specific about which $\gamma$ solved form delivers the outcome of the function. We will comment on this after the example below.

Further, recall from Definition 2.2.14 on page 16 that a $\gamma$ solved form of an ECSP is a subproblem that is $\gamma$ consistent, and whose domains are elements of their respective sets of final domains. $\gamma$ refers to some notion of local consistency, e.g., $\gamma = \text{arc}$ for arc consistency.

Operationally, the evaluation of $f_{P,x_j}$ on a given sequence of argument domains $D_1, \ldots, D_n$ consists of the following three steps (see also Figure 7.1).

1. Construction of the inner ECSP $(P, \langle D_1, \ldots, D_n \rangle)$

2. Branch-and-propagate search on the inner ECSP. This is the actual nested search.
7.2. An Operator for Nested Search

\[ f_{P,x_j}(D_1, \ldots, D_n) = \]

1. construct \( P' := (P, (D_1, \ldots, D_n)) \)

2. solve \( P' \)

3. consistent \( \rightarrow \) \( \emptyset \)
   inconsistent \( \rightarrow D_j \)

\( \gamma \) solved form
\( \langle C_P ; \ldots, x_j \in D_j', \ldots ; \ldots \rangle \rightarrow D_j' \)

Figure 7.1: Evaluation of the DRF for nested search

3. If step 2 determines that the inner ECSP is inconsistent, \( f_{P,x_j}(D_1, \ldots, D_n) \) evaluates to \( \emptyset \). Otherwise, \( f_{P,x_j}(D_1, \ldots, D_n) \) evaluates to the domain of variable \( x_j \) in the \( \gamma \) solved form that is found in step 2.

In two of the three applications considered in the next section, the inner ECSP has exactly one decision variable, and in all applications, the inner ECSP has one or more auxiliary variables. As a result, the branch-and-propagate search of step 2 is limited in the sense that it is not an exhaustive exploration of all possible combinations of canonical domains. Search takes place on a subset of the variables only.

**An Example**

We will see three examples of specific uses of the DRF of Definition 7.2.3 in the next section. Just to illustrate the interaction between the inner ECSP, the DRF, and the outer ECSP, we present the following (contrived) example.

**7.2.4. Example.** Let \( Q \) be the ECSP

\[ Q := \langle C_Q ; w, x, y \in \{0, 1, 2\} ; D_w, D_z, D_y \in \mathbb{Z} ; A_w, A_{Q,x}, A_{Q,y} \rangle \]

Consider the instance \( f_{P,y} \) of the DRF for nested search that is parameterized by the ECSP

\[ P := \langle x < y, y \neq z ; x, y \in \mathbb{Z}, z \in \{0, 1, 2\} ; D_x, D_y, D_z \in \mathbb{Z} ; A_{P,x}, A_{P,y}, A_z \rangle \]

having sets of final domains \( A_{P,x} = [\mathbb{Z}], A_{P,y} = \mathbb{Z} - \{\emptyset\}, \) and \( A_z = [\mathbb{Z}], \) i.e., \( x \) and \( z \) are decision variables, and \( y \) is an auxiliary variable in \( P \).

\( Q \) and \( P \) share variables \( x \) and \( y \), so as a DRF on \( Q, f_{P,y} \) has the signature

\[ \mathcal{T}_x \times \mathcal{T}_y \rightarrow \mathcal{T}_y \]
with $T_x = Z$ and $T_y = Z$.

If we evaluate this function for the domains of $x$ and $y$ in $Q$, i.e., we compute $f_{P_y}(\{0, 1, 2\}, \{0, 1, 2\})$, we first construct the inner ECSP

$$P' = (P, \langle \{0, 1, 2\}, \{0, 1, 2\} \rangle).$$

This is the first step of Figure 7.1, and in this case we have

$$P' = \langle x < y, y \neq z ; x, y, z \in \{0, 1, 2\} ; D_x, D_y, D_z \in Z ; A_{P, x}, A_{P, y}, A_z \rangle$$

This ECSP is consistent, so suppose that the branch-and-propagate search of step 2 finds the arc solved form

$$\langle x < y, y \neq z ; x = 0, y \in \{1, 2\} ; D_x, D_y, D_z \in Z ; A_{P, x}, A_{P, y}, A_z \rangle,$$

which we also encountered in Example 2.2.15 on page 16. Now as step 3 we select the domain of $y$ in this arc solved form as the outcome of the function evaluation, and we have

$$f_{P_y}(\{0, 1, 2\}, \{0, 1, 2\}) = \{1, 2\}$$

In a branch-and-propagate search on $Q$, this result is then used as a new domain for $y$, and $Q$ is transformed into the following ECSP.

$$\langle C'_Q ; w, x \in \{0, 1, 2\}, y \in \{1, 2\} ; D_w, D_x, D_y \in Z ; A_w, A_{Q, x}, A_{Q, y} \rangle,$$

with $C'_Q := C_Q[\{0, 1, 2\}, \{0, 1, 2\}, \{1, 2\}]$. \hfill $\square$

One way in which this example is contrived is that the relation between $P$ and $Q$ is not clear. In our applications, $P$ is constructed so that applying $f_{P, x_j}$ will remove values for which the constraints in $C_Q$ cannot be satisfied.

As we mentioned after Definition 7.2.3, the functionality of the operator depends on the solved form that is found by the branch-and-propagate search in step 2 of Figure 7.1. If in Example 7.2.4, we had found another arc solved form for example one having $D_x = \{0\}$, $D_y = \{1\}$, and $D_z = \{2\}$, then $f_{P_y}(\{0, 1, 2\}, \{0, 1, 2\})$ would have evaluated to $\{1\}$ instead of $\{1, 2\}$.

In two of the three applications discussed in the next section, we require a specific kind of branch-and-propagate search in step 2 of Figure 7.1, namely the one based on a depth-first, leftmost-first traversal strategy. This traversal strategy will lead to a specific solved form, which then fully determines the functionality of the DRF. We will use a a superscript $L$ to indicate this requirement: $f_{P, x_j}^L$.

Because the output variable $x_j$ is also in the input scheme of $f_{P, x_j}$, and because $f_{P, x_j}(D_1, \ldots, D_n)$ evaluates to the domain of $x_j$ in a solved form of the inner ECSP, by Definition 2.2.14 on page 16 we have $f_{P, x_j}(D_1, \ldots, D_n) \subseteq D_j$. Returning to the discussion of DRF properties on page 21, this leads to inflationary updates of ECSPs, which ensures that instantiations of the operator for nested search will not cause non-termination of generic iteration algorithms. However, as we shall see in the next section, the operators are not necessarily monotonic, or, as was already demonstrated by Example 7.2.4, equivalence preserving.
7.3 Applications

In our applications, we will not use the operator for nested search to find a particular value for the output variable $x_j$, but only to update one of its bounds. However, without special care, the domain of $x_j$ may be reduced to a singleton set during search. This happens if $x_j$ is a decision variable of the inner ECSP, but also if its value is uniquely determined by the decision variables of the inner ECSP. Therefore we will use a copy of the output variable during the search. This copy is a regular CSP variable that is added to the parameter ECSP, and serves to update one of the bounds of the output variable. For this we use the regular inequality constraints.

The initial domain of the copy variable is set to that of the output variable. We cannot use an equality constraint here, because that would still reduce the domain of the output variable to a singleton set, once a solved form is found. Therefore we introduce the following constraint.

7.3.1. Definition. Given two variables $x$ and $y$ with respective domains $D_x$ and $D_y$ of the same domain type, let the constraint $x := y$ denote the subset $(D_x \cap D_y) \times D_y$ of $D_x \times D_y$.

This constraint can be thought of as an assignment operator: modifications of the domain of $y$ are propagated to the domain of $x$, but not the other way around. Our first application below will demonstrate its use.

Of the three example applications discussed in this section, optimization by means of a bisection search involves inner ECSPs that have the most similarity with the ECSPs that we have encountered so far: they represent purely combinatorial problems to which two auxiliary variables are added for optimization. Therefore we discuss this application first.

7.3.1 Optimization

As an alternative to branch-and-bound, in optimization we can perform a bisection search in the range of an objective function that is defined on the variables of a combinatorial problem. The following search procedure is adapted from [BLPN01], where it is applied to job-shop scheduling problems. Suppose we want to minimize an integer objective function that evaluates in the range $[l..h]$. Assuming that a solution to the combinatorial problem exists (this is often the case for constrained optimization problems, and certainly for job-shop scheduling), we can determine the minimum as follows:

1. Split the domain $[l..h]$ for the outcome of the objective function into two halves $[l..m]$ and $[m+1..h]$, with $m := \lceil \frac{1}{2}(l + h) \rceil$, and first try to solve the combinatorial problem with the domain for the outcome of the objective function set to $[l..m]$. 

2. If the combinatorial problem has a solution for which the objective function evaluates to a value \( v \in [l..m] \), then if \( v \) equals \( l \) this is the minimum, and we are done. If \( v > l \), restart the search at step 1, now using \([l..v]\) as the range for the objective function.

3. If no solution exists for which the objective function evaluates in \([l..m]\), restart the search at step 1, now using \([m + 1..h]\) as the range for the objective function.

We now describe this optimization scheme as an application of our operator for nested search. Let

\[ R := \langle C \mid x_1 \in D_1, \ldots, x_n \in D_n ; T_1, \ldots, T_n ; A_1, \ldots, A_n \rangle \]

be the ECSP for a combinatorial problem. In what follows we consider that we want to minimize the outcome of an integer objective function \( g \) on variables \( x_1, \ldots, x_n \).

### The Parameter ECSP

We will construct a parameter ECSP \( P \) such that evaluating \( f_{P,c}(l..h) \) verifies that a solved form of \( R \) exists, for which \( g(x_1, \ldots, x_n) \) falls within the range \([l..h]\). If it exists, \( f_{P,c}(l..h) \) evaluates to \([l..v]\) where \( v \) is the outcome of \( g \) for the particular solved form found during the evaluation of \( f_{P,c} \). If it does not exist, \( f_{P,c}(l..h) \) evaluates to \( \emptyset \). \( R \) is the basis of the parameter ECSP \( P \), but two variables and three constraints are added.

First we add a variable \( c' \) for the outcome of the objective function, and constrain it accordingly:

\[ c' = g(x_1, \ldots, x_n). \]

Assuming that \( R \) contains no auxiliary variables, the value of \( c' \) is fixed when \( g \) is evaluated for a solved form of \( R \). Since in general, a solved form for \( R \) will not yield the minimal outcome of \( g \), we can only use this value as an upper bound, and we add a second variable \( c \), and constrain it to be less than, or equal to \( c' \).

\[ c \leq c' \]

Now \( c \) will be the output variable of the DRF for nested search, and \( c' \) is its copy for performing the search, as we described just before Section 7.3.1. All that is needed now is a third constraint that gives \( c' \) its initial domain:

\[ c' := c \]

Assuming that \( c \) and \( c' \) do not occur in \( R \), the parameter ECSP now becomes

\[ P := \langle C_P \mid c \in \mathbb{Z}, c' \in \mathbb{Z}, x_1 \in D_1, \ldots ; T_c, T_{c'}, T_1, \ldots ; A_c, A_{c'}, A_1, \ldots \rangle \]

where the underlined elements are the additions to the combinatorial problem \( R \), and where
7.3. Applications

\[ C_P := C \cup \{c' = g(x_1, \ldots, x_n), c \leq c', c' := c\}, \]

- \( T_c \) and \( T_{c'} \) both equal \( I \) (\( g \) is an integer function), and
- \( A_c \) and \( A_{c'} \) equal \( I - \{\emptyset\} \).

The sets of final domains \( A_c \) and \( A_{c'} \) render \( c \) and \( c' \) auxiliary variables of \( P \): search is on the variables of \( R \) only.

The Domain Reduction Function \( f_{P,c} \)

The ECSP \( P \) gives rise to the following instance of the generic operator for nested search:

\[ f_{P,c} : I \rightarrow I \]

When \( f_{P,c} \) is evaluated for an integer interval domain \([l..h]\), as step 1 of Figure 7.1, the inner ECSP is constructed by substituting the domain of \( c \) in \( P \) with \([l..h]\). Then, as step 2 of Figure 7.1, branch-and-propagate search is performed on \( P' \). Propagation of the constraint \( c' := c \) will set the domain of \( c' \) equal to \([l..h]\) initially, but during the search, the domain of \( c' \) will be modified further. According to Definition 7.3.1 these modifications will not propagate back to the domain of \( c \), but through the constraint \( c \leq c' \) the upper bound for \( c \) is modified yet. However, these modifications do not affect the outcome of \( f_{P,c}([l..h]) \) until a solved form is found. In such a solved form, the domain of \( c' \) has likely been reduced to a singleton set \( \{v\} \) because \( c' \) is tied to the decision variables by the constraint \( c' = g(x_1, \ldots, x_n) \). Through the constraint \( c \leq c' \), the domain of \( c \) in this solved form equals \([l..v]\), which is the outcome of \( f_{P,c}([l..h]) \). When no solved form of \( P' \) exists, \( f_{P,c}([l..h]) \) evaluates to the empty set.

Optimization as Branch-and-Propagate Search

With \( f_{P,c} \) the optimization scheme can be described as a branch-and-propagate search on an outer ECSP

\[ Q := \langle C_Q \mid c \in [l..h]; D_c \in \mathcal{Z}; A_c = |\mathcal{Z}| \rangle \]

This outer ECSP contains a single decision variable \( c \), with initial domain \([l..h]\), where \( l \) and \( h \) are trivial lower bounds for \( g \) that follow from the domains of \( x_1, \ldots, x_n \) in \( R \). \( C_Q \) can be thought of as the constraint that there exists a solved form of \( R \) for which \( g \) evaluates to \( c \). Nodes of the search tree are created by bisection of the domain of \( c \), and in the constraint propagation phase \( f_{P,c} \) is applied once to the domain of \( c \). If we perform a depth-first, leftmost-first traversal, the first solution is guaranteed to contain the minimum value of the outcome of \( g \) for any solved form of \( R \).
7.3.2. Example. Let \( R \) be an ECSP with variables \( x_1, \ldots, x_n \), and let \( g \) be an integer function on \( x_1, \ldots, x_n \). Suppose that for all possible combinations of values allowed by the domains of these variables in \( R \), \( g \) evaluates in the range \( 0..100 \). Suppose further that \( R \) is consistent, and that the minimum value of \( g \) for any solved form of \( R \) is 23. The search for this minimum proceeds as follows.

Initially, in the outer ECSP \( Q \) we have \( c \in [0..100] \). As the initial constraint propagation phase we evaluate \( f_{P,c}([0..100]) \). As a part of this evaluation, in step 1 of Figure 7.1, we transform

\[
P := \{ C_P : c \in \mathbb{Z}, c' \in \mathbb{Z}, \ldots ; \ldots ; \ldots \}
\]

into

\[
P' := \{ C_P : c \in [0..100], c' \in \mathbb{Z}, \ldots ; \ldots ; \ldots \}.
\]

Now in step 2 of Figure 7.1, we search for a solved form of \( P' \). Through propagation of the constraint \( c' := c \) in \( C_P \), the domain of \( c' \) is immediately changed from \( \mathbb{Z} \) to \([0..100]\). As the search progresses, the domain of \( c' \) undergoes further changes, and the upper bound for \( c \) is updated accordingly. Suppose that the nested search in step 2 of Figure 7.1 finds a solved form for which \( g \) evaluates to 36, i.e., this yields a suboptimal value for the objective function \( g \). This solved form looks like this:

\[
\{ C_P : c \in [0..36], c' \in \{36\}, \ldots ; \ldots ; \ldots \},
\]

the domain of \( c' \) is fixed, but \( c \) only has its upper bound modified, and \( f_{P,c}([0..100]) \) evaluates to \([0..36]\).

In the branch-and-propagate search on the outer ECSP \( Q \), the value of \( f_{P,c}([0..100]) \) is used as the new domain for \( c \). This is not yet a singleton set, and we proceed by branching on \( c \), yielding subdomains \([0..18]\) and \([19..36]\). Because we do a depth-first leftmost-first search on \( Q \), we continue the search in the \( c \in [0..18] \) branch. It will turn out that no solution to \( P \) exists for which \( g(X) \) lies in this range, i.e., \( f_{P,c}([0..18]) = \emptyset \), which voids the domain of \( c \) in this branch. Then search proceeds in the \( c \in [18..36] \) branch, where \( f_{P,c}([18..36]) \) yields a tighter upper bound for \( c \), and so on, until finally the domain of \( c \) has been narrowed to \{23\}. Because the search is depth-first leftmost-first, this is guaranteed to be the minimum.

\[\square\]

Discussion

It is important that during the constraint propagation phase, the domain reduction function \( f_{P,c} \) is applied only once, to verify that a solution for the current domain of the criterion variable exists, and to update the upper bound of this domain for the actual solution that is found. If unless we deduce a failure, we keep iterating the function until it makes no more modifications to the domain of the criterion variable, we will apply it at least once more. This second application
may find the same solved form that caused the initial reduction, in which case we just duplicate the work, but because we start the nested search with a different range for the objective function, we may well find a solved form that implies a tighter bound. In the latter case, we achieve a further reduction, and the operator is applied again. This way we embark on a limited branch-and-bound search in the constraint propagation phase, that could lead all the way to the minimum. Because the bisection search is proposed as an alternative to branch-and-bound, this behavior is undesirable.

Note that $f_{P,c}$ is not a monotonic function. In Example 7.3.2, $f_{P,c}([0..100])$ evaluates to $[0..36]$ because the objective function $g$ evaluates to 36 for the particular solved form found $c \in [0..100]$. However, it is possible that for a narrower domain for $c$, say $[0..99]$, the search on the inner ECSP leads to a solved form with a higher outcome of the objective function, say 37. While $[0..99] \subseteq [0..100]$, we then have $f_{P,c}([0..99]) \not\subseteq f_{P,c}([0..100])$, which entails that the transformation is non-monotonic with respect to the subproblem relation. Returning to the discussion of DRF properties on page 21, this implies that generic iteration is no longer guaranteed to terminate in the least common fixed point of the DRFs involved. For this application, this is not a problem, because the operator is applied only once.

Further, note that our proposed optimization scheme does not lead to a fully accurate implementation of the procedure described at the beginning of this section. As we described it there, the combinatorial problem is solved in left branches only. Right branches are guaranteed to contain a solution if the left branch fails, and these are split again immediately. In Section 7.5.1 we see how the same effect can be achieved with nested search in a branch-and-propagate setting.

Finally, we did not specify the level of local consistency that the nested search is to be based on. Apart from the requirement that assignments violating the constraints $C$ in $R$ should be filtered out, we only require that the constraint $c' := c$ is bounds consistent, and that the domain of $c'$ is voided if it does not contain the outcome of $g$, for a solved form of $R$. All further propagation helps to speed up the solving process. If the constraint $c' = g(x_1, \ldots, x_n)$ propagates back to the domains of $x_1, \ldots, x_n$, the nested search accelerates. As is demonstrated in Example 7.3.2, the search on the outer ECSP also benefits from bounds consistency of $c \leq c'$.

### 7.3.2 Box Consistency

In Section 4.5 we introduced hull consistency, which is an approximation of arc consistency used for arithmetic constraints and floating-point interval domains. Box consistency [BMVH94] is another such approximation. It was introduced to avoid decomposing constraints, and as such it partly avoids the dependency problem that we also discussed in Section 4.5. Before we can define box consistency, and demonstrate how it is enforced using our generic operator for nested search,
we first need to recall the notion of an interval extension of a constraint.

An **interval extension** of a constraint \( C \subseteq \mathbb{R}^n \) is a relation \( C \subseteq \mathcal{F}^n \) such that for all \( D := (D_1, \ldots, D_n) \in \mathcal{F}^n \), \( D \in C \) if there exists a tuple \( (d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n \) for which \( (d_1, \ldots, d_n) \in C \).

**7.3.3. Example.** An interval extension \( C_{eq} \subseteq \mathcal{F} \times \mathcal{F} \) of the equality constraint is

\[
(D_1, D_2) \in C_{eq} \text{ iff } D_1 \cap D_2 \neq \emptyset.
\]

As an example of an interval extension of a particular class of constraints, we can define interval extensions for the other relational symbols \( \leq \) and \( \geq \) as well, and modify the definition of the natural interval extension of Section 4.5 to include not only arithmetic expressions, but also arithmetic constraints. □

Interval extensions of constraints are called **interval constraints**.

Now a constraint \( C \subseteq \mathbb{R}^n \) on variables \( x_1, \ldots, x_n \) with associated domains \( D_1, \ldots, D_n \in \mathcal{F} \) is said to be **box consistent** if for all \( 1 \leq j \leq n \)

\[
D_j = \text{hull}(D_j \cap \{ r \in \mathbb{R} \mid (D_1, \ldots, D_{j-1}, \text{hull}\{\{r\}), D_{j+1}, \ldots, D_n) \in C})
\]

where \( C \) is an interval extension of \( C \).

In [BGGP99] a notion of box consistency is defined that supports using different interval extensions for different occurrences of variables, and that also captures a number of other, alternative definitions. In what follows, we always use the natural interval extension for \( C \). We will further limit ourselves to polynomial **equalities**, for which we use the interval extension proposed in Example 7.3.3.

**7.3.4. Example.**

- The constraint \( x^3 + x = 0 \) on \( x \in \text{hull}([-1, 1]) \) is not box consistent: the domain of \( x \) properly contains \( \text{hull}\{\{-1\}\} \), which is not in the interval extension of the constraint \( x^3 + x = 0 \).

- The constraint \( x^3 + x = 0 \) on \( x \in \text{hull}\{\{0\}\} \) is box consistent. □

Given a compound constraint, enforcing box consistency for this constraint may yield narrower domains than enforcing hull consistency for the decomposition of the constraint [CDR99]. This can be seen by comparing Example 7.3.4 and Example 4.5.1. However, the accuracy of the condition

\[
(D_1, \ldots, D_{j-1}, \text{hull}\{\{r\}), D_{j+1}, \ldots, D_n) \in C
\]

is still subject to the dependency problem. Therefore it will not achieve hull consistency for the compound constraint in general. In other words, box consistency is weaker than hull consistency, but stronger than hull consistency for the decomposed constraint.
Enforcing Box Consistency

The idea behind enforcing box consistency is to fix, for every variable \( x_j \) that a constraint applies to, the domains of the other variables to interval constants. The interval extension of the constraint with all but one variable replaced by an interval constant is then a *unary interval constraint*, and we can remove those subintervals from \( D_j \) that do not satisfy it. Because we will take the intersection of the remaining domain with \( D_j \), and compute the hull of this intersection to be able to represent it as an \( \mathcal{F} \) interval again, we only need to know the leftmost and the rightmost canonical interval that are a subset of \( D_j \), and for which the unary interval constraint holds. We can then intersect \( D_j \) with the hull of the union of these canonical intervals. This is illustrated in Figure 7.2 for a constraint \( f(x_1, \ldots, x_n) = 0 \). The marks on the \( x_j \) axis are the floating-point numbers in \( \mathbb{F} \), which delimit the canonical intervals. The boxes drawn along the curve represent the ranges for the outcome of \( f \) for a particular canonical interval in the domain of \( x_j \), and in presence of the current domains of the other variables.

In [HMD97] a very general algorithm for enforcing box consistency is given. In this algorithm, procedures LeftNarrow and RightNarrow search in the domain \( D_j \) for the leftmost and rightmost canonical intervals \( \text{hull}(\{r_l\}) \) and \( \text{hull}(\{r_r\}) \) that satisfy the unary interval constraint described above, and update \( D_j \) accordingly. Both procedures can be described as instances of the generic operator for nested search. Left narrowing for variable \( x_j \) of a constraint \( C \subset \mathbb{R}^n \) on variables \( x_1, \ldots, x_n \) is realized by

\[
f^L_{P,x_j} : \mathcal{F}^n \to \mathcal{F}
\]
where $P$ is the following ECSP.

$$\begin{align*}
\langle \{x'_j/x_j\}C, & \quad x'_j := x_j, x_j \geq x'_j \\
: & \quad x_1, \ldots, x_n \in \mathbb{R}, x'_j \in \mathbb{R} \\
: & \quad D_1, \ldots, D_n \in \mathcal{F}, D_{x'_j} \in \mathcal{F} \\
: & \quad \mathcal{A}_1, \ldots, \mathcal{A}_n = \mathcal{F} - \{\emptyset\}, \mathcal{A}_{x'_j} = [\mathcal{F}]
\end{align*}$$

In this parameter ECSP, $\{x'_j/x_j\}C$ is the constraint that we want to enforce box consistency for, with all occurrences of $x_j$ replaced by $x'_j$, the copy of $x_j$ introduced for performing the search. Through its set of final domains $\mathcal{A}_{x'_j} = [\mathcal{F}] = \{\text{hull}\{r\} \mid r \in \mathbb{R}\}$, $x'_j$ is the only decision variable in $P$. All other variables, including $x_j$ have $\mathcal{F} - \{\emptyset\}$ as their set of final domains. They are auxiliary variables in the inner ECSP, and no branching needs to be performed on their domains to reach a solved form.

When $f^L_{x_j}$ is evaluated for a sequence of intervals $D_1, \ldots, D_n$, in step 1 of Figure 7.1 the domains of $x_1, \ldots, x_n$ in the parameter ECSP $P$ are replaced by $D_1, \ldots, D_n$. This yields the inner ECSP $P'$. During step 2, propagation of the constraint $x'_j := x_j$ ensures that the domain of $x'_j$ is equal to that of $x_j$ initially. Because no branching takes place on the domains of $x_1, \ldots, x_n$, $\{x'_j/x_j\}C$ now effectively has become a unary interval constraint on the domain of $x'_j$. If $P'$ is consistent, and we perform a depth-first, leftmost-first search for a solved form, as specified by the superscript $L$, then the domain of $x'_j$ in this solved form is the leftmost canonical interval in $D_j$ that satisfies the unary interval constraint obtained by replacing in $C$ the domains of $x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n$ with $D_1, \ldots, D_{j-1}, D_{j+1}, \ldots, D_n$, and by taking the natural interval extension. Propagation of the constraint $x_j \geq x'_j$ updates the lower bound for $x_j$ accordingly, and in step 3 of Figure 7.1, $f^L_{x_j}$ evaluates to $D_j$ with the lower bound set to that of the leftmost canonical interval that satisfies the unary interval constraint.

Coupled with analogous operators for right narrowing, and for the other variables that participate in the constraint, $f^L_{x_j}$ enforces box consistency for constraint $C$. Intuitively, narrowing domains of variables can only move the leftmost and rightmost canonical intervals in the domain of these, and other variables inwards. This can be used to demonstrate that operators for enforcing box consistency are in fact monotonic functions, so the order in which they are applied by an iteration algorithm is irrelevant for the outcome of their combined computation.

For evaluating the unary interval constraint we can use the facilities described in Section 4.5. This way the operators for enforcing box consistency are composed from the generic reduction operator for nested search, and the facilities for enforcing hull consistency for a decomposition of an arithmetic constraint into atomic constraints.
7.3.3 Shaving

Shaving is a constraint propagation technique used for solving scheduling problems. We refer to the description of this technique in [MS96], and use job-shop scheduling as an example.

Recall from Chapter 6 that a job-shop scheduling problem (JSSP) instance consists of a set of activities and a number of machines. An activity is characterized by the machine that it must be processed on, and by a processing time. Activities are grouped in jobs, and all activities of a job have to be executed in a specified order. The problem is to find for each activity an interval in which it can be executed on the specified machine, such that no two activities require the same machine simultaneously, and such that the precedence constraints inside the jobs are respected. An optimal schedule minimizes the makespan of the schedule, being the completion time of the activities that finish last.

A possible CSP formulation of the JSSP contains an integer interval variable for the starting time of each activity. The lower bound for the starting time of an activity is called the release date, and the upper bound plus the processing time of the activity is called the deadline. Here we will consider the procedure for updating release dates. The procedure for deadlines is analogous.

The shaving technique entails that starting with the release date, we see what happens if we fix the activity to start at that time. After experimentally fixing the starting time of an activity we apply constraint propagation. If propagation of the fixed starting time leads to a failure, we can safely remove this candidate starting time from the domain of the variable, and we proceed by trying the next possible starting time, and so on, until we encounter a starting time that does not lead to a failure. This is then the new release date for the activity. Shaving can be explained as a leftmost-first search in the domain of a single variable, and as such it can be expressed as an application of the generic operator for nested search.

Let
\[ Q := \langle C_Q ; x_1 \in D_1, \ldots, x_n \in D_n ; T_1, \ldots, T_n : A_1, \ldots, A_n \rangle \]
be an ECSP for a job-shop scheduling problem, and assume that \( x_j \), with \( 1 \leq j \leq n \) is the variable for the starting time of an activity \( A \). Shaving the starting time of \( A \) can be expressed as the domain update
\[ D_j := f_P(x_j)(D_1, \ldots, D_n) \]
where \( P \) is the ECSP
\[ \langle \{ x'_j/x_j \}C_Q , x'_j := x_j , \ x_j \geq x'_j ; \ x_1 \in T'_1, \ldots, x_n \in T'_n, \ x'_j \in T'_j \]
\[ ; T_1, \ldots, T_n, \ T'_j = T_j \]
\[ : A_1 = T_1 - \{ \emptyset \} , \ldots, A_n = T_n - \{ \emptyset \}, \ \mathcal{A}_{x_j} = |T'_j| \rangle. \]
In this case, the parameter ECSP $P$ is a full copy of the outer ECSP $Q$, with a single variable $x'_j$ added. This variable is a copy of the variable whose starting time we want to update. The set of constraints in the parameter ECSP, $\{x'_j/x_j\}C_Q$, is the set of constraints of the outer ECSP, with every occurrence of $x_j$ replaced by $x'_j$. All variables of the outer ECSP are auxiliary variables in the inner ECSP, and search takes place only on the “copy” variable $x'_j$. It is coupled to the original $x_j$ in the usual way, through constraints $x'_j := x_j$ and $x_j \geq x'_j$. The domain of the variables $x_1, \ldots, x_n$ in the inner ECSP are irrelevant, and we set them here to the largest elements of the corresponding domain types $T_i^+, \ldots, T_n^+$. These domains are replaced by their counterparts in the outer ECSP when the DRF is evaluated.

### 7.3.5. Example

The example JSSP of Figure 6.1, on page 132 consists of three jobs, each having three activities that require three different machines. To implement shaving for this problem, we need 18 operators: one for each of the 9 activity starting times, and one for each of the completion times. Consider the operator for one of the starting times. The nested search finds the smallest value for this starting time that has the property that if the activity is actually scheduled to start at that time, regular constraint propagation on the full problem, involving all 17 other starting times and completion times, does not lead to a failure. In the global CSP, all earlier starting times are removed from the domain of the variable.

The extent to which infeasible starting times are removed depends on the level of consistency that we enforce during the nested search. In [BLPN01] it is suggested that we use the level of consistency enforced by the edge finding algorithm. The precedence constraints inside the jobs propagate modifications to the other machines, during the nested search. Also the constraint $x_j \geq x'_j$ must be enforced in order that the infeasible starting times are actually removed.

A depth-first leftmost-first search combined with bisection or enumeration branching on the domain of $x'_j$ will correctly update the lower bound of $x_j$, but in [MS96] a different branching scheme is described for step 2 of Figure 7.1. This alternative scheme entails that we first try to shave off a single value, and double the size of interval to shave off until an interval is found that allows for a feasible schedule. Then we search for the lower bound in this interval by regular bisection.

Several different notions of shaving exists. In [VHPP00] a different form is implemented to demonstrate nested search in OPL. Here the nested search does not directly modify release dates and deadlines, but only the ranking of the activities. Constraint propagation verifies whether individual activities can be ranked first among the set of unranked activities on a machine. If this fails, constraints are added to ensure that at least one of the other unranked activities is ranked before that particular activity.

The locality of our shaving operation is in between that of the other two applications. Box consistency performs nested search on a single variable, and
propagates only a single constraint. For optimization, the nested search is on all variables, propagating all constraints. Like box consistency, nested search for shaving branches only on one variable, but consistency checking involves all constraints. Using the same reasoning as for box consistency, we can demonstrate that DRFs for shaving are monotonic functions.

7.4 Implementation

The plug-in that implements the operator for nested search is an almost\(^1\) autonomous OpenSolver instance, acting as a reduction operator. A special coordination layer plug-in forms the interface between the solver that uses the nested search, and the solver that performs the nested search. This is illustrated in Figure 7.3. A benefit of this implementation is that all facilities of the OpenSolver framework are immediately available for nested search.

Programs 7.1 and 7.2 show examples of how the operator is used. The plug-in name is NestedSearch, and its specifier string consists of the following:

- The name of a Boolean variable, whose only purpose is that its domain will be voided if we find that the inner ECSP is inconsistent.

- The names of the variables that the operator applies to (the input variables). The output variable, whose domain is updated by the reduction operator, is identified by a prefix \&.

- Between curly brackets, an OpenSolver configuration for the parameter ECSP, in the language of Figure 3.2 on page 37. All input variables must appear in this configuration, but their domains are irrelevant. These will be provided by the coordination layer plug-in each time the operator is applied.

The Boolean variable was introduced because the operator cannot make any assumptions about the domains of the other variables, and hence has no uniform way of voiding their domains. It is a deviation from the specification in

\(^1\)There is a single thread of control throughout.
Section 7.2, but this is no fundamental difference. Also, more than a single variable can be marked as an output variable. This is a fundamental difference, but we have not found a use for this facility yet, and taking it into account would complicate the formal description of the operator further.

When it is activated during constraint propagation, the NestedSearch operator will pass the domains of the input variables to the interface coordination layer, and run the local OpenSolver instance. The solver configuration in the specifier string is parsed the first time that the operator is applied. The interface coordination layer keeps a copy of the root node of the search tree, in order that the specifier string need only be parsed once. On each application of the reduction operator, a fresh copy of this cached root node is made, which is updated with the current domains of the input variables. The interface coordination layer issues commands for a regular first-solution search on this modified root node. When a solution node is found, an export command (see Section 3.3.2) is given for the output variable. The solver will respond to this command via a call-back function provided by the coordination layer. In this case, the call-back function updates the domain of the output variable. If no solution is found, the domain of the Boolean variable is voided. After the first-solution search, the interface coordination layer issues the clear WDB command to reset the solver.

7.5 Experiments

In this section we describe the experiments that we performed for the applications discussed in Section 7.3.

7.5.1 Optimization: Job-Shop Scheduling

We tested the optimization technique described in Section 7.3.1 on the job-shop scheduling problem, using the plug-ins that were introduced in Chapter 6.

Program 7.1 shows an OpenSolver configuration for the approach of Section 7.3.1. The NestedSearch operator is applied through an adapter Idempotent DRF. This forces the scheduler of reduction operators to treat it as an idempotent domain reduction function. The operator-based scheduler (see Section 4.1.1), which is the default, takes idempotency of DRFs into account to avoid unnecessary applications of an operator. In this case, it prevents that the operator is applied more than once in the same node of the search tree, for the reasons discussed at the end of Section 7.3.1. Also, because all right branches are guaranteed to contain a feasible schedule, an adapter PropagateLeft is used to hide changes that are made by the FailFirst branching operator to variable bound in right branches. Hiding means suppressing the protocol for communicating domain changes, discussed in Section 3.2.2, thus preventing needless activation of the nested search in these branches.
VARIABLE bound IS IntegerInterval {119..5110};
AUX b IS Bool {0, 1};
DRF IdempotentDRF { NestedSearch { b, &bound, {
  AUX bound IS IntegerInterval {};
  AUX imakespan IS IntegerInterval {};
  VARIABLE makespan IS Activity {0, 0, 5110};
  ...  
  code for the JSSP, where makespan executes after the last activity of every job.
  ...  
  DRF BoundActivity { makespan, imakespan };
  DRF IIARule { imakespan\^1 * (1) = bound };
  DRF IIARule { bound\^1 * (1) <= imakespan };
} } };
DRF PropagateLeft { FailFirst { 0, bound } };

Program 7.1: OpenSolver code for bisection search for the minimal makespan of a JSSP

The indented code is a configuration for solving a job-shop scheduling problem, where the length of the schedule is constrained not to exceed the upper bound of the domain for the integer variable bound. It is similar to Program 6.1 on page 138. The IIARule operators effectively enforce the constraints imakespan := bound and bound ≤ imakespan. The parameter 0 in the specifier for the FailFirst plug-in specifies a bisection where the left branch is generated last. By default, the search frontier is maintained as a stack, so this results in a depth-first, leftmost-first exploration.

Table 7.1 compares the bisection optimization algorithm that we implemented by nested search with regular branch-and-bound, as described in Chapter 6, on the ten 10 × 10 JSSP instances of [AC91], which are also used in [BLPN01]. For each instance we report the number of nodes visited, and the user time in seconds, as reported by the GNU/Linux time command on a 1200 MHz Athlon PC. For the bisection search, the number of nodes applies to the nested search on the JSSP only. The search in the domain of the criterion variable is not taken into account. Either algorithm outperforms the other in half the cases, but the total running time for all ten cases is much better for the bisection search, which seems to suggest that it is more robust. While an evaluation of these approaches to optimization is beyond the scope of this section, it shows that the bisection search is a useful tool. Having implemented it using OpenSolver as a software component increases its value as a building block for solvers, because we can now
Chapter 7. Applications of Nested Search

<table>
<thead>
<tr>
<th>instance</th>
<th>branch-and-bound nodes</th>
<th>time (sec.)</th>
<th>bisection nodes</th>
<th>time (sec.)</th>
<th>opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>ft10</td>
<td>238,045</td>
<td>38.87</td>
<td>321,878</td>
<td>61.38</td>
<td>930</td>
</tr>
<tr>
<td>abz5</td>
<td>385,699</td>
<td>44.84</td>
<td>586,315</td>
<td>67.23</td>
<td>1234</td>
</tr>
<tr>
<td>abz6</td>
<td>57,277</td>
<td>6.28</td>
<td>114,487</td>
<td>12.57</td>
<td>943</td>
</tr>
<tr>
<td>la19</td>
<td>319,547</td>
<td>40.80</td>
<td>228,283</td>
<td>34.47</td>
<td>842</td>
</tr>
<tr>
<td>la20</td>
<td>106,785</td>
<td>13.78</td>
<td>139,146</td>
<td>18.26</td>
<td>902</td>
</tr>
<tr>
<td>orb01</td>
<td>7,331,421</td>
<td>643.40</td>
<td>63,688</td>
<td>10.91</td>
<td>1059</td>
</tr>
<tr>
<td>orb02</td>
<td>642,645</td>
<td>63.93</td>
<td>200,350</td>
<td>28.86</td>
<td>888</td>
</tr>
<tr>
<td>orb03</td>
<td>14,031,873</td>
<td>2186.06</td>
<td>3,657,439</td>
<td>563.12</td>
<td>1005</td>
</tr>
<tr>
<td>orb04</td>
<td>179,037</td>
<td>39.12</td>
<td>278,507</td>
<td>71.62</td>
<td>1005</td>
</tr>
<tr>
<td>orb05</td>
<td>4,461,777</td>
<td>621.77</td>
<td>98,268</td>
<td>14.54</td>
<td>887</td>
</tr>
</tbody>
</table>

Table 7.1: A Comparison of Branch-and-Bound and a Bisection Search for the optimum on ten 10 × 10 JSSP instances

...combine it with other facilities, such as memory bounded LDS.

Taking the differences in clock speeds into account, our results for bisection search on these ten benchmark problems do not match the results reported in [BLPN01]. For some of the instances, performance is better, but for the majority of them it is worse. More important, though, is that the instances rank differently, which indicates that we have not been able to reproduce exactly the same heuristics.

As a further indication that our technology leads to competitive constraint solvers, Koalog Constraint Solver (KCS, [KoaA]) is reported to solve the ft10 benchmark, which is also known under the name MT10, in 11 minutes, using 293,000 backtracks. KCS is a commercially available Java library for solving combinatorial optimization problems using constraint programming or local search, and is used in industry.

7.5.2 Box Consistency

The implementation of box consistency for constraints on the reals was tested on the Broyden banded functions, a benchmark that is often used to demonstrate the advantage of box consistency over hull consistency, for example in [BMVH94].

The problem is to find the zeros of the functions

\[ f_i(x_1, \ldots, x_n) = x_i(2 + 5x_i^2) + 1 - \sum_{j \in J_i} x_j(1 + x_j) \quad (1 \leq i \leq n). \tag{7.2} \]

where \( J_i = \{ j \mid j \neq i, \max(1, i-5) \leq j \leq \min(n, i+1) \} \), and \( x_1, \ldots, x_n \in [-1, 1] \).

Every function \( f_i \) depends on the 2 to 7 variables in the set \( J_i \cup \{ x_i \} \), and for every variable that a function depends on, two reduction operators are generated: one for the left narrowing, and one for the right narrowing. Program 7.2 shows
the operator that implements left narrowing for argument $x_1$ of the function $f_3$. Variable $lx_1$ corresponds to $x'_j$ used in Section 7.3.2. It is linked to $x_1$ by the first two RIARule operators. Auxiliary variables $x_1,...,x_4$ are the input variables, and they are given their domains each time the operator is applied. The other RIARule operators evaluate $f_3$, using $fx_1,...,fx_4$ to store intermediate results. The interval for the outcome of $f_3$ is intersected with that of the variable zero, which contains only the value 0, so effectively this implements a generate-and-test search for the canonical interval in the domain of $x_1$ that contains the leftmost zero of $f_3$.

```
DRF NestedSearch { b, &x1, x2, x3, x4, {
    AUX x1 IS RealInterval {};    
    AUX x2 IS RealInterval {};    
    AUX x3 IS RealInterval {};    
    AUX x4 IS RealInterval {};    
    VARIABLE lx1 IS RealInterval {};    
    AUX fx1 IS RealInterval {};    
    AUX fx2 IS RealInterval {};    
    AUX fx3 IS RealInterval {};    
    AUX fx4 IS RealInterval {};    
    AUX zero IS RealInterval { 0 };    
    DRF RIARule { x1^-1 * (-1) <= -1*lx1 };    
    DRF RIARule { lx1^-1 * (1) = x1 };    
    DRF RIARule { fx1^-1 * (1) = 1 + lx1 };    
    DRF RIARule { fx2^-1 * (1) = 1 + x2 };    
    DRF RIARule { fx3^-1 * (1) = 2 + 5*x3^2 };    
    DRF RIARule { fx4^-1 * (1) = 1 + x4 };    
    DRF RIARule { zero^-1 * (1) = x3 * fx3 + 1
                 - lx1 * fx1 - x2 * fx2 - x4 * fx4 };    
    DRF RoundRobin { 0, lx1 };    
} };    
```
using a decomposition is that RIARule does not support brackets in the specifier string, while we want to use the syntax of formula (7.2) as a basis for the interval extension. This form appears to be less sensitive to the dependency problem than the form

$$2x_i + 5x_i^3 + 1 - \sum_{j \in D_i}(x_j + x_j^3)$$

The efficiency of the evaluation could probably be improved with a schedule for the operator-based scheduler that applies the operators only once.

In our opinion, being able to use RIARule both for computing hull consistency and for evaluating interval extensions of functions is a definite advantage of the design decision to separate the individual projections of constraints. The RIARule and IIARule plug-in are versatile tools for composing constraint solvers, while in combination with the programmable scheduler of Section 4.1.1, such solvers are not inherently less efficient than specialized algorithms such as HC4 for computing hull consistency.

The following results demonstrate the (well known) effect of computing box consistency for the Broyden banded functions benchmark: computation time increases linearly with the problem size. The target precision is 1.0e-8, but inside the nested search we split down to machine precision. This is realized by including in the top-level configuration two commands to replace the standard node evaluator, and to prevent branching on RealInterval variables whose domain is of the required precision:

```
TDINFO Precision { 1.0e-8 };
DRF LimitedPrecision { 1.0e-8, RoundRobin { 0, x1, x2, ... } };
```

This way, the system of equations is solved by propagation alone. The reported numbers are elapsed times in seconds, for problem instances specified by $n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.536</td>
<td>6.210</td>
<td>13.503</td>
<td>28.467</td>
<td>58.097</td>
<td>118.926</td>
</tr>
</tbody>
</table>

We have described only a basic implementation of box consistency, and there is much room for improvement. For example, we do not compute hull consistency for the decomposition used inside the nested search, but it may actually be worthwhile to do so, and propagate back the information that the functions should evaluate to zero, or in general, that the unary interval constraints should hold. Also, because the consistency check is implemented by constraint propagation, it should be easy to add propagators for the Newton reduction step described in [BMVH94]. It should be investigated in how far our approach supports the use of other interval extensions than the natural interval extension.

The code of Program 7.2 is currently generated by a program written specifically for this benchmark. We still need to extend the OpenSolver preprocessor for arithmetic constraints with an option to generate code for enforcing box
consistency. The preprocessor could then also generate the reduction operators corresponding to the Newton reduction step.

7.5.3 Box Consistency for Arithmetic Constraints on the Integers

Box consistency can also be enforced for arithmetic constraints on integer variables. In that case, a constraint $C \subseteq \mathbb{Z}^n$ on variables $x_1, \ldots, x_n$ with associated domains $D_1, \ldots, D_n \subseteq \mathcal{I}$ is called box consistent if for all $1 \leq j \leq n$

$$D_j = \text{int}(D_j \cap \{i \in \mathbb{Z} \mid \langle D_1, \ldots, D_{j-1}, \{i\}, D_{j+1}, \ldots, D_n \rangle \in C\})$$

where $C$ is an interval extension of $C$. To illustrate that this is a useful application, consider the constraint

$$x^2y^2 - 4x^2y + 4x^2 - 4xy^2 + 16xy - 16x + 4y^2 - 16y + 16 = 4$$

and ranges $x, y \in [0..10^5]$. When we solve this equation by means of decomposition into atomic constraints, according to the approach proposed in Section 5.7, the 8 solutions are found in approximately 14 sec. in a search tree of 40255 nodes. When the code for the decomposed constraint is packed in four different NestedSearch operators, for left and right narrowing of $x$ and $y$, the search tree is reduced to 39 nodes, and exploration takes less than 4 seconds.

7.5.4 Shaving

Shaving appears to be efficient only for larger problems. For small instances, like the ones we used in Section 7.5.1, the effort spent on shaving outweighs the benefit of the reduced search space. A similar experience is reported in [Zho97]. In this reference, the shaving, or bound trimming is on the domains of the variables that determine the processing order of the activities, so this technique is comparable to that of [VHPP00] mentioned at the end of Section 7.3.3.

An example of a larger problem instance, for which shaving is essential, is swv01. The optimum for this instance was first found by Perron [Per99], using a combination of limited discrepancy search, shaving, and parallel search. As we discussed in Section 4.1.2, LDS in OpenSolver is memory-intensive, and for this reason we used memory bounded LDS, which we introduced in that same section.

Using memory bounded LDS and the implementation of shaving described in Section 7.3.3, we were able to prove optimality for swv01 in approximately 100 hours on a 2000 MHz Athlon XP processor. Without shaving, by that time the current best solution is nowhere near the optimum, and finding it within an acceptable multiple of the already long running time seems unlikely.
7.6 Discussion and Concluding Remarks

We have shown that in a branch-and-propagate tree search solver, three powerful pruning techniques from optimization, analysis of nonlinear functions, and constraint-based scheduling can be expressed as applications of a generic operator for nested search. In our implementation, each instance of this operator is itself an almost autonomous OpenSolver instance. This has the advantage that all facilities of the framework are available for specifying the nested search. A disadvantage is the overhead of using a general-purpose solver for very specific search problems, and dedicated implementations of box consistency and shaving will likely always be more efficient.

An evaluation of the efficiency of our implementation of box consistency and shaving is currently missing. Because this also depends on, for example, the interval arithmetic library that we use, this would require the implementation of dedicated operators for these techniques. There are still opportunities to improve the efficiency of our solution, though. For example, we plan to extend the set of commands of Section 3.3 with a command for first-solution search, to bypass the command loop for a longer period. Often, however, strong consistency notions like box consistency and shaving determine whether a problem can be solved or not. In such cases, being able to experiment with enforcing strong consistency notions is of greater importance than the actual efficiency.

Our approach promotes the composition of constraint solvers as low-level co-operations of basic solvers. As we discussed, this has further advantages: it avoids duplicate code in the solver implementation, and techniques carry over to other domains than those for which they were originally conceived. From this point of view it is desirable to have a small set of basic operators and combinators, that can be used to realize a wide range of constraint solving techniques. It seems reasonable that compositionality comes at the cost of some computational overhead for the framework.

There is an analogy between our operator for nested search and procedures in procedural programming languages. The input and output variables can be seen as by-value and by-reference parameters, respectively, and the OpenSolver input for the local CSP can be seen as a procedure body block. It would be interesting to investigate this analogy further, perhaps to define some notion of a parameterized reduction operator. This way we may be able to avoid duplicate code for activating plug-ins, and ease the rewriting. To illustrate that this is a useful facility, for the Broyden banded function, the code for each function (as shown in Figure 7.2) is repeated up to 14 times, with minor differences for left or right narrowing, and for the particular variable that we want to update. The OpenSolver input for $n = 320$ contains over 120,000 lines, with roughly the same number of plug-in instances created.

If we apply the shaving mechanism to arithmetic constraints, we achieve a notion of consistency known as \textit{3B consistency} [Lho93]. Also shaving itself can
be nested, which results in a technique known as double shave [MS96]. Both these techniques can be expressed as further applications of the generic operator for nested search. Other facilities that could be implemented using nested search are the squash operator of ECL'PS*, and the absolve operator of the CLIP system [Hic01].

The operator defined in Section 7.2 performs a single solution search, and in our applications we use this for updating the bounds of interval variables. For finite domains variables, it is attractive to have a variant that searches for all solutions. We can then use enumeration instead of bisection, and use the union of the domains for each variable as the result of the reduction operator. A potential application is to enforce singleton arc consistency (SAC, [DB97]). This notion of local consistency entails that for every value in the domain of every variable, the CSP can be made arc consistent if that value is assigned to the variable. To implement SAC using the proposed all-solution variant of our operator for nested search, we would need one such operator per DRF variable. This operator then searches in the domain of its variable for arc solved forms of the ECSP obtained by taking the full outer CSP, and making all other variables auxiliary. The output of the operator is the union of these arc solved forms, projected on the variable that it is applied to.

It is also possible to define box consistency (and the other applications discussed here) in terms of this all-solution variant, but then all internal zeros would be lost because of the interval representation. Therefore the first-solution search is a more natural and efficient implementation of these techniques.