Composing constraint solvers
Zoeteweij, P.

Citation for published version (APA):
This is the first of three chapters that demonstrate how OpenSolver can be configured for specific application domains. In this chapter, we introduce the facilities for constraint solving on finite domains, Booleans, and real-valued variables, as well as some domain-independent plug-ins for constraint propagation and search. These are basic facilities that are available in many other systems. Together with the previous chapter, this forms a description of OpenSolver as a basic constraint solver. Of the other two chapters in this “applications” series, Chapter 5 deals with constraints on integer interval variables. These are also available in many other systems, and could therefore have been included in this basic facilities chapter, but because of the lengthy analysis, a separate chapter was devoted to these constraints.

The research question that underlies this chapter is whether the framework that we introduced and implemented in the two previous chapters is suitable for composing five particular solving techniques that would normally be hard-wired in solvers: limited discrepancy search, best-first search, no-good recording, non-chronological backtracking, and applying domain reduction functions for decomposed arithmetic constraints in an order that respects their hierarchical relationships. These techniques are discussed in sections 4.1.2, 4.3, 4.4, and 4.5.

4.1 General-Purpose Facilities

4.1.1 Constraint Propagation

Three scheduler plug-ins are currently available in OpenSolver. These are general-purpose facilities for constraint propagation that can be used with any set of reduction operators. The schedulers apply the propagation functions of reduction operator plug-ins, so they control only the execution of propagation operators and optimization operators. Branching operators are left untouched.
A Basic Scheduler

The BasicScheduler plug-in implements the following round robin scheduling strategy, which also forms the basis for the AC-1 algorithm for computing arc consistency (see, e.g., [Dec03]): apply all operators in sequence, and keep doing so until a full sequence passes in which no variable domains are modified. This strategy can be expressed as an instantiation of Algorithm 2.1, having

\[
update(F, D, D') := \begin{cases} 
F & \text{if } D' \neq D \\
\emptyset & \text{otherwise}
\end{cases}
\]

for the select function, we would need to introduce an extra variable that maintains the sequence number of the last operator that has been applied.

The plug-in does not use the specifier string of the SCHEDULE statement, so the default scheduler is replaced by the basic scheduler using the following command in the OpenSolver input language:

```
SCHEDULE RR BasicScheduler { };
```

Variable-Based Scheduling

The variable-based scheduler maintains a queue of modified variables. It keeps removing variables off the front of the queue, and for each variable that is removed, all reduction operators that have the variable in their input scheme are applied in sequence, in the order in which they are introduced in the solver configuration script. If the application of an operator modifies the domain of a variable, this variable is enqueued, unless it already is in the queue. By default, an operator is deactivated if it signals that it has become redundant in the current branch of the search tree. As explained in Section 3.2.3, this can have significant costs in terms of storage, and the variable-based scheduler can be made to ignore this information as follows:

```
SCHEDULE RR VariableScheduler { ignore };
```

The variable-based scheduler implements the scheduling strategy described in the ILOG Solver 5.1 reference manual. A potential disadvantage of this scheduler is that if a reduction operator depends on more than one, say two, variables, and both these variables have been enqueued, the DRF will be applied twice also if no relevant changes are made in between dequeueing the two variables.

Operator-Based Scheduling

In correspondence with Algorithm 2.1, the default scheduler plug-in explicitly maintains a set of reduction operators that still need to be applied. This set is implemented as an array, containing a bit per operator that is used to mark the operator as being scheduled for application. Like the basic scheduler introduced
above, it cycles through the operators, applying those that are marked, and resetting their marks. If a variable is modified, the operators that have this variable in their input scheme are marked, but the operator-based scheduler will not reschedule idempotent DRFs that have modified the domain of a variable in their input scheme.

The operator-based scheduler maintains a counter of operators that still need to be applied. In principle, it keeps cycling through the sequence of operators until this counter reaches zero, but many aspects of this scheduler’s operation can be configured through the definition of a schedule, as explained below.

Defining a Schedule

A schedule specifies the order in which the operators are considered for application, as an alternative for cycling through the sequence. This is not the same as the order in which they are applied: actual application of a reduction operator also depends on if it has been marked. In principle, a schedule is a list of (zero-based) operator indices. These indices refer to the sequence of DRF statements in a configuration script. The scheduler traverses the list, and at the end it terminates, reporting to OpenSolver whether a fixed point was reached or not. In the latter case, the node of the search tree where the scheduler was applied remains in the set of nodes that are subject to constraint propagation, and the scheduler will be called again for this node, at a later stage.

Parts of a schedule can be enclosed in brackets, to indicate a fixed point computation. When entering a pair of brackets during the execution of a schedule, the scheduler will keep executing this subschedule until a common fixed point is reached for the operators that are pointed to from within the brackets. Typically, the entire schedule is enclosed in brackets, to indicate that the scheduler does not return control to OpenSolver before a fixed point it reached, but this facility can also be used to group several operators, and to implement priority schemes.

Two kinds of brackets can be used, for two different ways of performing the computation of the fixed point.

- Curly brackets specify that the scheduler cycles through the enclosed sequence. For example, the following statement specifies that the scheduler considers the first five operators in sequence, but does not continue with operators 5 through 9 until a fixed point of operators 10 through 14 has been reached.

```plaintext
SCHEDULER ChangeScheduler { 
    schedule = {0,1,2,3,4,{10,11,12,13,14},5,6,7,8,9} ;
}
```

1Scheduler plug-ins coordinate the computation of a fixed point of the domains extensions of DRFs (see Section 2.3.1) implemented by the propagation functions of reduction operator plug-ins. Where this does not lead to confusion, we will sometimes refer to such fixed points as fixed points of a set of reduction operators.
This fixed point is computed by cycling through the sequence 10,11,12,13,14, when considering to apply these operators. Likewise, after considering to apply operator 9, the scheduler returns to the beginning of the top-level schedule and considers to apply operator 0.

- Parentheses specify that instead of cycling, each time that a change is made, the scheduler restarts, and returns to the beginning of the sequence that they enclose. This is used for priority schemes, e.g., do not execute computation-intensive operators when less computation-intensive operators are scheduled for execution. Suppose that we have three groups of operators, with indices 0-4, 5-9, and 10-14, having increasing computational costs. The following schedule specifies that we do not execute the expensive operators before a fixed point of the easy operators has been reached, and as soon as an expensive operator modifies the domain of a variable, we first compute the fixed points of the less computation-intensive functions again.

```
SCHEDULER ChangeScheduler {
    schedule = ((({0,1,2,3,4},5,6,7,8,9),10,11,12,13,14)
}
```

In addition, the top-level schedule can be enclosed in square brackets. This specifies that the schedule is executed once, as if no brackets were used at all. Likely, this does not lead to a fixed point, but the scheduler will still signal that constraint propagation has terminated. This is useful when we want to enforce a limit on the number of applications of very expensive reduction operators.

Like the variable-based scheduler, the specifier for the ChangeScheduler plugin can be prefixed with the keyword ignore to save the memory costs of taking into account that certain DRFs have become redundant. In case both ignore and schedule are used, these are separated by a comma. The full syntax of the ChangeScheduler specifier language is given in Figure 4.1.

The operator-based scheduler allows for the composition of constraint propagation algorithms, similar to the approach proposed in [GM03]. It implements two of the three composition operators proposed there: sequence and closure. The third, decoupling, entails that several operators are evaluated independently on the same domains, and that the results are combined by intersection. The language of Figure 4.1 and the implementation of the operator could well be adapted to support this third mode of composition, but we did not need it for our experiments. Moreover, it is unlikely that the decoupling could be implemented efficiently for OpenSolver domains in general, because it would require making working copies of domains during constraint propagation, which is an expensive operation.
4.1. General-Purpose Facilities

\[
\begin{align*}
\langle \text{specifier}\rangle & \rightarrow \epsilon \\
& \mid \text{ignore} \\
& \mid \langle \text{schedule} \rangle \\
& \mid \text{ignore, } \langle \text{schedule} \rangle \\
\langle \text{schedule} \rangle & \rightarrow \text{schedule} = \langle \text{subschedule} \rangle \\
& \mid \text{schedule} = [ \langle \text{sequence} \rangle ] \\
\langle \text{subschedule} \rangle & \rightarrow ( \langle \text{sequence} \rangle ) \\
& \mid \{ "\langle \text{sequence} \rangle " \}\} \\
\langle \text{sequence} \rangle & \rightarrow \langle \text{schedule\_step} \rangle \{, \langle \text{schedule\_step} \rangle \} \\
\langle \text{schedule\_step} \rangle & \rightarrow \langle \text{Integer} \rangle \\
& \mid \langle \text{subschedule} \rangle
\end{align*}
\]

Figure 4.1: Syntax of the specifier for the DRF-based scheduler

An Assembly Language

Something that actually can go wrong with schedules is that if an index is omitted, the corresponding operator will not be executed, and a fixed point cannot be reached. In most scenarios, this leads to an infinite loop in the constraint propagation phase for the root of the search tree. This situation can easily be avoided by having the scheduler verify that all indices are present in the schedule. If not, a run-time error can be produced, or the scheduler could fall back on a default schedule for the omitted operators. More interesting than a possible solution is the fact that such situations may occur. Erroneous schedules are one example, the inter-category dependencies that we discussed in Section 3.4 are another.

Because configuring OpenSolver is error-prone, in most cases this is done by other programs. Therefore, it makes sense to consider the language of Figure 3.2 an assembly language, now used to configure our abstract branch-and-propagate tree search engine instead of a CPU. Assembler is usually generated by compilers that bridge the semantic gap to a higher-level programming language, and likewise our configuration language is usually produced by peripheral programs that complement OpenSolver to form a constraint solver with a proper user interface. As an example, for the experiments reported in Chapter 5 we used a two-stage translation of arithmetic constraints into OpenSolver configuration files. The programs involved in this translation are responsible for generating correct schedules for the operator-based scheduler. Drawing the analogy with assembly languages further, for a coherent set of plug-ins it could be considered to develop a compiler for a modeling language such as OPL [VH99].
4.1.2 Search

Basic Strategies

Recall from Section 2.3.2 that search involves branching and traversal. Basic branching strategies can be defined by a combination of a variable selection strategy and a value selection strategy. To start with the latter, as we discussed in Section 3.2.1, the variable domain type plug-ins implement basic value selection strategies. For example, in our finite domains implementation, domains can be split in several different ways. These are illustrated in Figure 4.2, for the example domain \{1,2,3,4,5,6\}.

Two reduction operator plug-ins, FailFirst and RoundRobin complement the basic value selection strategies of the domain types with a variable selection strategy to form complete branching strategies.

Fail-First. The FailFirst plug-in implements the fail-first variable selection strategy, discussed in Section 2.3.2. The specifier for this plug-in lists the variables that the strategy is applied to, preceded by an (integer) indication of the desired value selection strategy, for example:

\[
\text{DRF FailFirst \{ 0, x1, x2, \ldots \};}
\]

This specifies that from the listed variables \(x1, x2, \ldots\), we select a variable that reports the smallest domain size greater than one. If several such variables exist, the first one in the list is selected. The integer value 0 specifies how to create the subdomains for this variable, but interpretation of this value depends on the domain type of the selected variable. For example, if the selected

![Diagram](image-url)
variable is of type DiscreteDomain (see Section 4.2), then 0 specifies that the
subdomains are generated through enumeration, 1 specifies left-enumeration, 2
specifies right-enumeration, etc. As another example, for RealInterval variables
(see Section 4.5), only bisection has been implemented, but the integer argument
is used to specify the order in which the subdomains are generated. This allows
for easy switching between leftmost-first and rightmost-first traversal.

The FailFirst plug-in also implements a strategy that is sometimes referred
to as fail-last: select the variable with the largest domain. In either mode, as
an alternative, the search for the variable with the desired domain size can be
started from the middle of the list outwards. The same effect could be achieved
by a permutation of the list of variables. Fail last, and the alternative search for
the smallest or largest domain are specified by prefixing the specifier string (with
a letter l and/or m, respectively).

Round Robin. The RoundRobin plug-in tries to branch on all variables in
turn. Starting from a sequential traversal of the variables in the specifier string,
it selects the variable with domain size greater than one that has least recently
been selected. Therefore, at every node of the search tree we need to remember
the index of the variable that was split to create it. For this purpose we use an
annotation with a single integer.

Program 4.1 shows a typical configuration for round robin search. The first
line installs the integer annotation. The value 0 in its specifier string is the initial
value, which applies to the root node of the search tree. In this case, it indicates
that the first variable in the RoundRobin specifier string, x1, is to be split first.
If the IntegerAnnotation plug-in instance is not present, RoundRobin will just
select the first variable that can be split, resulting in a chronological variable
selection strategy. The specifier string for the RoundRobin plug-in itself is similar
to that for FailFirst, discussed above, so again the leading zero identifies a
specific, but domain type dependent value selection strategy.

```plaintext
ANNOTATION IntegerAnnotation { 0 };
VARIABLE x1 IS ...
VARIABLE x2 IS ...
...
DRF ...
...
DRF RoundRobin { 0, x1, x2, ... };
```

Program 4.1: Skeleton of a configuration for search based on a round robin vari-
able selection strategy
The \textit{default traversal strategy} is to maintain the search frontier as a stack, resulting in a \textit{depth-first} search. Constraint propagation runs to completion, in a single node of the search tree. Below we will discuss an alternative traversal strategy called limited discrepancy search, and a variant of this strategy that was implemented for OpenSolver.

\subsection*{Limited Discrepancy Search}

Limited discrepancy search (LDS, [HG95]), can be used when a good heuristic, in the form of a value selection strategy, is available to guide the search. The idea is that the heuristic will make only a few mistakes when assigning values to variables. When reaching the first node of the search tree that is a failure (the leftmost path in the tree), LDS first tries all alternatives that make exactly one different decision. In a binary search tree, this corresponds to all paths that follow the right branch in exactly one internal node, and the left branch everywhere else. If these new attempts all fail, LDS continues by trying two deviations from the leftmost path, and so on, gradually increasing the number of deviations, or \textit{discrepancy}, until all alternatives have been explored.

LDS can be effective for single-solution search, and for optimization schemes, such as branch-and-bound (see Section 5.9.2), where it may find better suboptimal solutions and achieve stronger pruning than a regular depth-first search. For a purely combinatorial all-solution search it will not improve on any other traversal strategy.

The straightforward implementation of LDS in OpenSolver uses an integer annotation to record the discrepancy of each node of the search tree. Just like the \texttt{RoundRobin} branching operator annotates every node of the search tree with the index of the least recently selected variable, the branching operator for LDS search could maintain the discrepancy annotation. Complemented with a container plug-in that keeps the nodes in the search frontier sorted on the basis of their (integer valued) annotation, we can explore the nodes with the smallest discrepancy first.

The problem with this implementation is that while exploring the set of nodes that have discrepancy \(n\) (called the \(n\)-th \texttt{wave}), we will also be generating nodes of a higher discrepancy \((n + 1\) and higher values in case of a non-binary value selection strategy). The size of the search frontier that we accumulate before starting the next wave is exponential in the size of the problem. To see this, consider the search tree for a problem with \(n\) binary variables. After processing all nodes of discrepancy \(d - 1\), the search frontier consists of those nodes of discrepancy \(d\) that are right branches. Let \(r_d\) denote the number of such nodes. In the worst case (no pruning), the tree consists of \(2^{n+1} - 1\) nodes, \(2^n - 1\) of which are right branches. With \(n\) binary variables, there are \(n + 1\) possible discrepancy values, so if all nodes were evenly distributed over the discrepancies, there would be \((2^n - 1)/(n + 1)\) nodes of a given discrepancy. This is not the case: there is, for example, only one node of the highest discrepancy \(n + 1\), so \((2^n - 1)/(n + 1)\)
is a lower bound for the largest $r_d$ in $r_0, \ldots, d_n$. This fraction is exponential in $n$, and as a result, even though in practice constraint propagation will prune a large part of the search space, the size of the search frontier is bounded only by an exponential function, and this implementation is too memory intensive.

**Memory-Bounded LDS**

The root of the problem with LDS is that OpenSolver explicitly maintains the search frontier. The problem is even more severe because OpenSolver is a *copying-based* solver (see also the next section) where the nodes in the search frontier are full copies of their parents, with minor modifications. LDS was originally formulated as an *iterative* algorithm: for increasing discrepancy values, a search procedure is called that performs a depth-first search, pruning all nodes that exceed the discrepancy value. This iterative algorithm does not suffer from the memory overhead, but a large amount of work is potentially repeated: for a purely combinatorial problem, the last iteration of an exhaustive search is a full depth-first exploration.

However, LDS is used primarily for constrained optimization problems. In this case, if we want to spend only a limited time on searching, and a trustworthy value selection strategy exists, iterative LDS was demonstrated to find better suboptimal solutions than backtracking [HG95]. We have not found any results in the literature indicating that iterative LDS outperforms chronological backtracking for a full best-solution search.

There are several ways to implement an iterative scheme in OpenSolver, but all of them are slightly artificial. One that we tried is based on nested search (see Chapter 7): a branching operator enumerates increasing allowed discrepancy values. For each value, by means of nested search we solve a full CSP in which a constraint has been posted that actively prunes away the nodes with discrepancy values that exceed the current maximum. For this implementation, some special-purpose reduction operators were needed to maintain the discrepancy information in regular CSP variables, in order that it becomes amenable to constraint propagation.

The experiment for which we used LDS (see Section 7.5.4) involves a full best-solution search for an optimization problem. While good solutions were found early compared to depth-first search, the stronger pruning did not outweigh the duplicate work, and overall performance did not improve. Therefore, instead of the iterative implementation we used a variant of the straightforward implementation discussed above: store the entire search frontier, and explore the nodes with the smallest discrepancy annotation first. If, as a result of branching, the size of the search frontier exceeds a certain threshold, switch to depth-first search to clean up the search frontier. Only when the size of the search frontier drops below a second, lower threshold, resume the discrepancy-based traversal.

For this scheme, which we refer to as *memory-bounded* LDS, the following
facilities are needed:

- an annotation plug-in \texttt{LDSAnnotation} that allows us to maintain both the depth and the discrepancy of a node in the search tree,
- a branching operator \texttt{Discrepancy} to annotate the nodes,
- a container \texttt{MBLDSStack} that switches between the two modes of traversal.

Program 4.2 shows a typical configuration file for memory-bounded LDS, using fail-first as a variable selection strategy. The specifier string for the \texttt{MBLDSStack}

\begin{verbatim}
FRONTIER MBLDSStack \{ 1024, 10240 \};
ANNOTATION LDSAnnotation \{ 0, 0 \};
VARIABLE x1 IS ...
VARIABLE x2 IS ...
...
DRF ...
...
DRF Discrepancy \{ FailFirst \{ 0, x1, x2, ... \} \};
\end{verbatim}

Program 4.2: Skeleton of a configuration for memory-bounded LDS search

container plug-in consists of the threshold sizes that trigger switching traversal modes. In the current implementation, these are numbers of stored nodes. The actual amount of memory occupied by a node depends on many factors, notably the number of variables. Therefore an implementation of the container that measures actual memory usage would be preferable.

Memory-bounded LDS is not as robust as iterative LDS, because potentially it could be doing a depth-first traversal most of the time. For our experiments, this did not happen, and we were able to exploit the advantages of LDS for a full best-solution search, in a copying-based solver.

Adapters

The specifier string for the \texttt{Discrepancy} plug-in contains an identifier-specifier pair for another branching operator. Internally, it actually creates this other plug-in instance, for which it serves as a wrapper. The branching is performed by the inner operator. \texttt{Discrepancy} only annotates the resulting nodes with the correct depth and discrepancy information.

Plug-ins like \texttt{Discrepancy} are called \textit{adapters}. They are used to make minor modifications to the functionality of other plug-ins, usually in the same category, or to combine the functionality of several such plug-ins. We will see more examples of adapters in this thesis.
4.2 Finite Domains

Many combinatorial problems are naturally expressed as a CSP using $\mathbb{Z}$, the set of all finite sets of integers, as a variable domain type. The variables of such CSPs are usually called \textit{finite domains} variables, because of the nature of the domains in $\mathbb{Z}$, but a more important property is that a representation exists for all possible subsets of a variable's original domain.

**Plug-ins**

The OpenSolver plug-in for finite domains is called\texttt{DiscreteDomain}. It is activated as follows:

```plaintext
VARIABLE identifier IS DiscreteDomain { specifier };
```

where \texttt{identifier} is the name of the variable, and \texttt{specifier} is a sequence of integer ranges.

The following constraints are available for \texttt{DiscreteDomain}:

- The binary disequality constraint $x - y \neq c$, where $c$ is an integer constant. It is implemented by a plug-in\texttt{DDNEQ} that works exactly as explained in Example 2.3.2 on page 19. Its use is demonstrated by Program 4.3 on page 77.

- The constraint $(x, y) \in T$, where $T$ is some set of allowable tuples. It is implemented by the\texttt{BinaryConstraint} plug-in. The following example shows its use.

```plaintext
DRF BinaryConstraint { <q1,q2> IN { <1,3>,<1,4>,<2,4>,<3,1>,<4,1>,<4,2> } };
```

The latter plug-in uses only a very crude algorithm and data structures for verifying that values in one domain are supported, through the list of allowed tuples, by a value in the other domain. Values that are not supported are removed from the domains. State-of-the-art algorithms for enforcing arc consistency try to minimize the number of \texttt{support checks}, and will be more efficient than our implementation. See notably the work of Van Dongen, e.g., [vD02].

As we mentioned at the end of Section 3.2.2, a set of reduction operators can maintain a protocol to distinguish different kinds of modifications of domains. This is used by the reduction operators for finite domains variables: \texttt{DDNEQ} and \texttt{BinaryConstraint} distinguish between the following events:

- changing the bounds of a domain
- deleting a value that is not a bound
- reducing a domain to a singleton set
Chapter 4. Applications

DDNEQ can make a change only if the value of one of the variables that it is applied to is fixed, so its application is triggered only if a domain becomes a singleton set. BinaryConstraint is triggered by all three events, and the first event, a change of the domain bounds, could be used to trigger a reduction operator that links a DiscreteDomain variable to a variable whose domain has an interval representation.

Protocols regarding modifications to variable domains can be implemented in two ways (see also Program 3.2 on page 58):

- By setting bits in an unsigned integer array element for each argument of a reduction operator. A complementary bitmask per argument is used to see if a particular operator needs to be scheduled after a domain has been modified.

- If the bitmask does not match (e.g., because it is set to all-zeros), a second check is performed to see if a DRF wants to be scheduled for a particular value of the unsigned integer. This can be used for a different encoding scheme, using an enumeration of possible cases, when the 32 bits offered by the bitmask are insufficient to encode all changes that we are interested in.

For the finite domains reduction operators we used the first implementation.

Efficiency of OpenSolver

The unavoidable benchmark problem for finite domains constraint solving is the \textit{n}-queens problem: place \textit{n} queens on an \textit{n} \times \textit{n} chess board in such a way that they do not attack each other. This can be formulated as a CSP as follows:

\[
\begin{align*}
&\{ q_i \neq q_j, q_i - q_j \neq j - i, q_i - q_j \neq i - j, \text{ for } 1 \leq i < j \leq n; \\
&q_1, \ldots, q_n \in \{1, \ldots, n\} \}
\end{align*}
\]

Any solution to the \textit{n}-queens problem will have exactly one queen per column (and row) of the chess board. This constraint is inherent to our CSP formulation, which uses a variable per column, indicating the position of the queen in that column. The constraints of the CSP state that no two queens can be on the same row or diagonal, in either direction.

Other CSP formulations for the \textit{n}-queens problem exist, notably one where the above \( \frac{3}{2}n(n + 1) \) constraints are replaced by 3 all\_different constraints, one for the row constraints, and one for both diagonals. The all\_different constraint entails that different values are assigned to all variables that the constraint applies to. It can be applied to an arbitrary number of variables, and specialized algorithms, beyond regular constraint propagation exist for enforcing it. Constraints that have these properties are sometimes called \textit{global constraints}. 
Because the specialized algorithms used for processing global constraints run counter to the compositional approach that we experiment with, no global constraints have been implemented for OpenSolver. There are, however, no limitations for doing so, and plug-ins for global constraints would make valuable additions. Typically, global constraint processing algorithms require that additional information about the CSP is maintained during constraint propagation and search. Because the state of reduction operator plug-ins is global, this information needs to be maintained elsewhere, for example in auxiliary variables of a special-purpose domain type (see Chapter 6).

While other CSP formulations of the n-queens problem exists, because of its scalability and simplicity the above formulation is perfectly suited for comparing the efficiency of the basic machinery of different solvers. For example we do not have to worry about the efficiency of the implementation of the all_different constraint. In Table 4.1 we report the results of a comparison of OpenSolver with ILOG Solver 5.1, on a SUN E450. We compare user time, as reported by the GNU/Linux time command, for counting all solutions. The configurations are similar to that of Program 4.3, and use the variable-based scheduler, which resembles the scheduling procedure that is described in the ILOG Solver manual.

```plaintext
VARIABLE q1 IS DiscreteDomain {1..4};
VARIABLE q2 IS DiscreteDomain {1..4};
VARIABLE q3 IS DiscreteDomain {1..4};
VARIABLE q4 IS DiscreteDomain {1..4};
DRF DDNEQ { q1-q2 <> 0 }; DRF DDNEQ { q1-q2 <>-1 }; DRF DDNEQ { q1-q2 <> 1 };
DRF DDNEQ { q1-q3 <> 0 }; DRF DDNEQ { q1-q3 <>-2 }; DRF DDNEQ { q1-q3 <> 2 };
... DRF DDNEQ { q3-q4 <> 0 }; DRF DDNEQ { q3-q4 <>-1 }; DRF DDNEQ { q3-q4 <> 1 };
DRF FailFirst { 0, q1, q2, q3, q4 };
SCHEDULER VariableScheduler { };
```

Program 4.3: Configuration for solving the 4-queens problem

Both solvers report the same number of solutions, failures and internal nodes, from which we can conclude that the computations are comparable. The results indicate that the efficiency of the basic procedures in OpenSolver is realistic in the sense that it is comparable to that of a successful commercial solver.
State Restoration Policy

An important aspect of a finite domains constraint solver implementation is the construction of the data structures, notably the variable domains, for the node of the search tree where search continues. This is usually referred to as the state restoration policy. While hybrid methods exist, the main options are [Sch99]:

- **Copying** When the search tree is expanded by branching, the data structures that define the current node are copied for all new nodes. These copies are then modified to construct subproblems. At potentially high memory costs, every node of the search frontier is immediately available for further exploration.

- **Trailing** Only the current node of the traversal is maintained, but all changes (deletions of values) to the domains of variables leading up to this node are registered. Backtracking is implemented by undoing changes to reach an internal node of the search tree, from which search can progress along an alternative branch. Trailing is the predominant method used in current constraint solvers.

- **Recomputation** Instead of unwinding a trail of changes, with a recomputation state restoration policy, the internal nodes are reconstructed from a shallower internal node by repeating a part of the traversal of the search tree. Iterative schemes such as LDS, which we discussed in the previous section, can be seen as a form of recomputation. Other forms of recomputation also exist that represent internal nodes in the search frontier by the branching decisions that need to be made to arrive at that node.

OpenSolver is copying-based, so the search frontier is maintained explicitly and the full data structures are available in every node. For finite domains this is the most memory-intensive option, but it can be justified because OpenSolver is a general-purpose solver. Interval computations, for example, use very modest data structures, and for that domain type, copying is a logical choice.
4.2. Finite Domains

<table>
<thead>
<tr>
<th>n</th>
<th>nodes</th>
<th>ILOG Solver time (sec.)</th>
<th>memory</th>
<th>DiscreteDomain time (sec.)</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>508,426</td>
<td>528.84</td>
<td>12M</td>
<td>523.53</td>
<td>5992K</td>
</tr>
<tr>
<td>500</td>
<td>364,754</td>
<td>3643.97</td>
<td>111M</td>
<td>4210.80</td>
<td>106M</td>
</tr>
<tr>
<td>1000</td>
<td>996</td>
<td>819.93</td>
<td>418M</td>
<td>1241.24</td>
<td>453M</td>
</tr>
</tbody>
</table>

Table 4.2: Memory consumption for large n-queens instances (first-solution)

To have an idea about the memory efficiency of our solver, we ran a number of larger instances of the n-queens problem. Because of the large search space and many solutions, we did a first-solution search on these instances. For search we still use the fail-first variable selection strategy, but now starting the search for the variable with the smallest domain from the middle of the chessboard outwards. In the ECL/PS6 tutorial [CHS+03], this strategy is shown to give better performance for this problem. Because of the quadratic number of DRFs we let the variable-based scheduler ignore the information on redundant DRFs. Table 4.2 shows the results of these experiments. Reported memory usage is the resident set size, reported by the top command.

ILOG Solver uses a combination of trailing and recomputation [Per99]. These results suggest that also regarding memory consumption, in spite of using copying as a state restoration policy, OpenSolver configured as a finite domains constraint solver is fairly efficient. The difference in running times for these large instances is partly due to using input files, which have to be parsed. ILOG Solver is a library, so no I/O is involved with setting up the large number of reduction operators involved in these instances. Another, more important factor is ignoring the redundancy of reduction operators. Probably in ILOG solver this information is subject to recomputation as well, or a more efficient representation is used, for example one based on instantiated variables. The latter solution fits the specified scheduling mechanism well, and would be easy to incorporate in OpenSolver.

The default implementation of finite domains uses a bitmap, that is copied during the cloning operation. As an experiment, we implemented a plug-in DDTrail that uses an alternative representation, where each variable domain is represented by a linked list of deleted values. Copying a domain as a part of branching consists only of replicating a pointer to the last element of this list. The list then becomes a tree, and different changes can be made in the different branches. Reference counts are used to manage the deallocation of the trails of deleted values. During constraint propagation, upon the first membership test, the bitmap is restored to avoid having to traverse the linked list to the root of the tree many times. The idea was to exploit some of the benefits of the trailing state restoration policy in a copying-based solver. This will probably work for problems with large numbers of large finite domains variables, to which little changes are made between different nodes of the search tree (i.e., many needless
copies of large bitmaps). For our experiments, this was not the case.

4.3 Best-First Search: the Knight’s Tour

Having introduced finite domains, and the basic facilities for search, we now study a slightly more complex combinatorial problem that requires an advanced search strategy for efficient solving. The purpose of this section is to demonstrate how such search strategies can be composed from a selection of OpenSolver plug-ins. Being of limited practical relevance yet, the problem is to move a knight piece around an $n \times n$ chess board, in such a way that all positions on the board are visited exactly once.

The problem can be formulated as a CSP as follows: with each of the $n \times n$ locations that constitute the tour we associate a variable, whose possible values are the actual positions on the board, numbered $1..n \times n$. The variables for all $n \times n - 1$ consecutive steps are constrained such that the positions that they indicate must be reachable through a knight’s move, and all variables must assume different values. As an example, for a $3 \times 3$ board, where the positions are numbered

```
7 8 9
4 5 6
1 2 3
```

and for which there obviously is no solution, the CSP is

\[
\langle (x_1, x_2), (x_2, x_3), \ldots, (x_8, x_9) \rangle \in \{(1, 8), (1, 6), (4, 9), (4, 3), (7, 6), (7, 2), (2, 7), (2, 9), (8, 3), (3, 4), (3, 8), (6, 7), (9, 2), (9, 4)\},
\]

\[
\text{all.different}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9);
\]

\[
x_1, x_2, \ldots, x_9 \in \{1, \ldots, 9\}
\]

OpenSolver can easily be configured for solving this CSP:

- for the $n^2 - 1$ “knight’s move” constraints, whose definition involves an enumeration of all positions that are linked by such a move, we can use the BinaryConstraint plug-in, and

- for the all.different constraint we can use $\frac{1}{2}n^2(n^2 + 1)$ disequalities, as we did for $n$-queens.

The search space is quite large though, and doing a basic fail-first search seems already intractable for $n = 8$ (but a better implementation of all.different could improve the situation).
Fortunately, a very good heuristic exists, which makes the problem easy to solve. It is said to have been discovered in 1823, by H.C. von Warnsdorf\textsuperscript{2}. This heuristic dictates that for the next location of the tour, we always choose the position that has the smallest number of possibilities for moving on. It is very easy to exploit this heuristic in a program written specifically for solving the knight’s tour problem. Alternatively, we could write an OpenSolver branching operator that knows about the problem, and generates subdomains for the variable holding the next location of the tour accordingly: a singleton set with the selected position, and a second subdomain containing all other alternatives. This results in a dedicated solver as well, but being based on OpenSolver, we benefit from readily available facilities like the finite domains implementation and search.

Instead of a problem-specific solution, we can aim at a more generalized approach, and try to formulate the heuristic in terms of aspects of our model of constraint solving. Observe that with enumeration value selection, the descendants of an internal node of the search tree correspond to the different alternatives for the next location of the tour. After constraint propagation, the nodes that comply with the heuristic will have the smallest total size of the variable domains. Based on this observation, we can implement the heuristic as follows.

- After branching, perform constraint propagation in all descendant nodes.
- Proceed by expanding a node for which the sum of all domain sizes is minimal.

Compared to the default, where the traversal depends on a selection from the set of nodes that are pending constraint propagation, we now make the selection from the set of nodes that are pending branching. Figure 4.3 illustrates the difference in the resulting search trees.

\textsuperscript{2}Our source of information on this problem is the website \url{http://www.delphiforfun.org/}
Implementation

The search strategy that we just described is implemented in OpenSolver by three plug-ins:

- a node evaluator `AnnotateSize` that annotates the nodes of the search tree with the sum of the domain sizes,

- a container `AnnotationOrderedStack` that always has a node with the smallest integer annotation on top,

- a selector `RestrictiveBranching` that selects a single node from the “pending branching” (Figure 3.3) set only if the set of nodes that are pending propagation becomes empty.

These three plug-ins are activated by including the four lines of Program 4.4 in the configuration script. The `ANNOTATION` statement is there just to provide an initial annotation for `AnnotateSize` to use. Note that `AnnotateSize` is another example of an adapter, this one in the node evaluator category. Internally, before annotating the node, it applies another node evaluator to determine the nature (solution, failure, or internal) of a node. Here we use the `CanonicalDomains` node evaluator, which tests for canonical domains to distinguish solutions. `CanonicalDomains` is also the default node evaluator, and therefore it normally does not occur in configuration scripts. We will see an alternative to the default in Section 4.5.

```
TDINFO AnnotateSize { CanonicalDomains {} };
ANNOTATION IntegerAnnotation { 0 };
INTERNAL AnnotationOrderedStack {};
EXPAND RestrictiveBranching{};
```

Program 4.4: Activating the plug-ins that implement Warnsdorf’s heuristic

Using `RestrictiveBranching` to select the nodes where the search tree is expanded by branching, the set of nodes that are pending propagation is emptied before new branches are created. The selected node is then split into a number of subproblems. Constraint propagation is applied in each of these nodes, and if this does not lead to a failure, the nodes are added to the “pending branching set.” Only after all descendants have been processed, and the “pending propagation” set is empty again, a new node is selected for branching. Because branching reduces the domains, with `AnnotateSize` and `AnnotationOrderedStack` this new
4.3. Best-First Search: the Knight's Tour

Figure 4.4: Knight's tour for $n = 8$

selection will be one of the most recent additions. Globally, the search is depth-first, but at every level, constraint propagation is applied in all siblings. The resulting search tree is illustrated in Figure 4.3(b).

Warnsdorf's heuristic is a form of best-first search, where a measure of quality is associated with the nodes. In this case, this measure is evaluated after constraint propagation, but this is not typical. Using the new search strategy, a solution to the knight's tour problem is found without backtracking, for all instances that we tried. Figure 4.4 shows a solution for the $8 \times 8$ problem. It is found in less than a second on our test machine, and 219 nodes are visited in the process. For $n = 18$, the search is still backtracking-free, and 1394 nodes are visited in 59 seconds. In each of these nodes, a fixed point of 52,649 DRFs is computed, 323 of which are instances of BinaryConstraint. Table 4.3 shows some additional information.

We should remark that our scheme is not a fully accurate implementation of Warnsdorf's heuristic. In each of the candidate nodes, constraint propagation may reduce the domains of variables that are more than a single step away, which could lead to different choices. Since the desired effect is obtained, we

<table>
<thead>
<tr>
<th>$n$</th>
<th>nvar</th>
<th>nDRF</th>
<th>K.M.</th>
<th>time (sec.)</th>
<th>memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>64</td>
<td>2,016</td>
<td>63</td>
<td>0.400</td>
<td>3584K</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>4,950</td>
<td>99</td>
<td>1.760</td>
<td>8932K</td>
</tr>
<tr>
<td>12</td>
<td>144</td>
<td>10,296</td>
<td>143</td>
<td>5.310</td>
<td>20M</td>
</tr>
<tr>
<td>14</td>
<td>196</td>
<td>19,110</td>
<td>195</td>
<td>13.120</td>
<td>45M</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>32,640</td>
<td>255</td>
<td>29.240</td>
<td>89M</td>
</tr>
<tr>
<td>18</td>
<td>324</td>
<td>52,326</td>
<td>323</td>
<td>58.870</td>
<td>174M</td>
</tr>
</tbody>
</table>

Table 4.3: Statistics; K.M. is the number of "knight's move" constraints
have not investigated if this occurs in practice. Although the search is backtrack-free, a large number of nodes are visited and stored. The time spent to visit the nodes is justified because we want to evaluate the heuristic using constraint propagation. A more efficient solver can be obtained by writing a dedicated branching operator, as we discussed at the beginning of this section. Storing the nodes could be avoided by writing a generic branching operator that internally performs a round of constraint propagation to evaluate the alternatives, and then generates two branches: one for the alternative that evaluates best, and one for all other alternatives. Such a branching operator would be parameterized by a set of propagation operators, and possibly a node evaluator. It would be quite similar to the operator for nested search, discussed in Chapter 7.

It would be interesting to investigate the effect of Warnsdorf's heuristic on other constraint satisfaction problems, for example on randomly generated problems. Perhaps this could help us to identify the properties that a CSP must possess in order for the heuristic to work. The generic implementation described in this section facilitates such experiments. To our knowledge, the heuristic is not normally available in other general-purpose constraint solvers.

4.4 Satisfiability of Propositional Formulas

In this section we configure OpenSolver as a solver for checking the satisfiability of propositional formulas in conjunctive normal form (CNF). This problem is usually referred to as the SAT problem, and solvers for it are called SAT solvers.

A propositional formula in CNF is a conjunction of clauses, where a clause is a disjunction of literals, and literals are propositional variables (positive literals), or their negations (negative literals). An example is the formula

\[(x \lor y \lor \neg z) \land (\neg x \lor z) \land (y \lor z).\]

Possible values for the variables are true and false. A clause evaluates to true if at least one of its literals evaluates to true, where a negation of a propositional variable evaluates to true if the value of the variable is false. A formula is called satisfiable if there exists an assignment of values to variables for which all clauses evaluate to true.

Almost all complete solvers for the SAT problem descend from an algorithm known as DPLL [DLL62], which can be explained as a systematic exploration of all possible assignments, plus the following inference techniques:

- **Unit propagation** If a clause contains only a single literal, we can deduce that this literal must have the value true (the clause is said to be resolved).

- The **pure literal rule** If a variable occurs only as one form of literal, i.e., the negation of this literal does not occur in any unresolved clause, then we can set the variable such that the literal evaluates to true.
4.4. Satisfiability of Propositional Formulas

If a literal is set to true, either as part of the exploration or as a result of inference, DPLL removes all clauses in which it occurs, and removes its negation from the remaining clauses. Once we reach an empty formula, we have established satisfiability of the original formula.

The straightforward way to configure OpenSolver as a SAT solver uses a propagation operator per clause that enforces the constraint that at least one of the variables occurring as positive literals must have the value true, or at least one of the variables that occur as a negative literal must have the value false. This corresponds to the unit propagation step of the DPLL algorithm, except that OpenSolver does not directly support the modification of constraints. However, variables whose domains have been reduced to singleton sets do not trigger any further reduction, and the reduction operators for clauses that evaluate to true can deactivate themselves in any branch of the search tree. Implementation of the pure literal rule is not so straightforward, but the SAT community seems to agree that the application of this rule does not pay off (see, e.g., [ZM02]).

Two plug-ins implement the configuration that we just outlined. Domain type Bool supports Boolean variables, and propagation operator Clause implements the constraint that at least one literal of a clause evaluates to true. The code below demonstrates the use of these plug-ins for the example problem at the beginning of this section. We could have made the Clause plug-in operate on finite domains, but for experimenting with SAT specific heuristics that are discussed below, we used a dedicated variable domain type.

```plaintext
VARIABLE x IS Bool { 0, 1 }; VARIABLE y IS Bool { 0, 1 }; VARIABLE z IS Bool { 0, 1 }; DRF Clause { x,y;z }; DRF Clause { z;x }; DRF Clause { y,z }; DRF FailFirst { 0,x,y,z };```

In their basic form, the Bool and Clause plug-ins consist of 400 lines of code, and from this perspective, little effort is required to configure OpenSolver as a SAT solver that is comparable to the DPLL algorithm. However, the DPLL algorithm is only a very basic solver, and contemporary solvers can handle vastly larger sets of problems. Complete solvers for the SAT problem, such as Chaff [MMZ+01] and GRASP [MSS96] all descend from the DPLL algorithm, but augment it with the following techniques:

- a good variable and value selection strategy,
- clause learning, and
- non-chronological backtracking.
Variable and value selection strategies for the SAT problem can be implemented in OpenSolver. As an example, we implemented the DLIS (Dynamic Largest Individual Sum) heuristic [Sil99], which entails that the search tree is always expanded by assigning the value `true` to the literal that occurs in the largest number of unresolved clauses. We implemented this heuristic by decorating each domain of type `Bool`, through the specifier string, with both the number of clauses in which it appears as a positive literal, and the number of clauses in which it appears as a negative literal. When a clause is resolved, it will ask the OpenSolver scheduler to be deactivated. At the same time, it will tell the `Bool` instances to which it applies to decrease one of these two counters, depending on the sign of the variable in that particular clause. When splitting a variable of type `Bool`, it will instantiate itself to `false` in the node that is added to the search frontier last, if the counter of negative occurrences is greater than the counter of positive occurrences, and to `true` otherwise. By default, the search frontier is managed as a stack, and the most recently added alternative is explored first. As the final element of our implementation of DLIS, uninstantiated `Bool` variables report as their size 2, plus the largest of their two clause counters, and we select a variable that reports the largest size (fail-last, see 4.1.2).

**Clause learning** is based on the observation that each time a failure is deduced, we can derive a new clause that explicitly prevents the combination of assignments that has lead to that particular node in the search tree. Not all assignments are relevant to this failure though, and a careful bookkeeping of the changes that trigger clause resolution will allow us to isolate exactly the literals that represent the contradicting assumptions made during the search. A negation of the conjunction of these literals is itself a clause. Such a clause, or in general, a constraint that corresponds to a deduced failure is also called a no-good, and maintaining these constraints is known as no-good recording. No-goods carry redundant information that is hidden too deep in the problem for the solver to exploit. As such it makes sense to add a no-good to the problem that we are trying to solve, because if it had been present in the first place, the current failure would have been prevented. Moreover, because only a fraction of all assignments that lead to the failure are actually causing it, the same failure could occur again in another part of the search tree, and adding a no-good will prevent this.

Maintaining an explanation for the assignments of truth values is straightforward. This was implemented inside the `Bool` plug-in by building alongside the search tree, for each variable, a tree of data structures with leaves for branching decisions, and internal nodes for resolved clauses, linking to the nodes for the assignments that trigger the resolution. The `Clause` plug-in was made aware of this data structure, and modified to use it for deriving a no-good upon deduction of a failure. Implementation of clause learning is impeded, though, by the lack of facilities for adding DRFs during the search. This is not a design decision, it simply has not been implemented. As a work-around, we can store the learned clauses in the domain of a special-purpose variable. All `Clause` instances
can access this variable, and a dedicated operator, which applies to all Boolean
variables, would then actually enforce the learned constraints. A problem with
this work-around is that the scheduling of the DRFs for the learned clauses then
becomes the responsibility of this last Clause instance, and is separated from the
regular scheduling mechanism.

The conflicting assumptions that lead to a failure can be stored as a no-good,
for more powerful pruning in other parts of the search tree, but analyzing the
conflict may also allow us to skip a part of the search tree that would normally
be explored by backtracking from a failure. This is the case if the most recent
assignment to a variable involved in the conflict occurs at a shallower level in the
search tree than the level directly above the conflict. Techniques for analyzing,
and exploiting conflicts to “jump” to a higher level in the search tree are known
as non-chronological backtracking, or look-back techniques. Conversely,
constraint propagation can be explained as looking forward to future branching
decisions). A comprehensive overview of non-chronological backtracking tech-
niques is given in [Dec03].

While heuristics and clause learning can be implemented, the design of Open-
Solver is unsuited for non-chronological backtracking. The fact that the search
frontier is stored explicitly makes this difficult: the nodes that would be skipped
by a backjump of more than a single level have already been created. The hierar-
chical information that is needed to identify the nodes that can be skipped after
a backjump can be maintained in annotations, but even so there is no mechanism
for discarding these nodes, and they have to be pruned away by propagation of
a no-good for the conflict. Mechanisms for backjumping can be added, but the
copying state-restoration policy used in OpenSolver is probably the worst alter-
native for a proper implementation of backjumping, because expensive copying
operations are involved in creating the nodes that may be discarded later.

The clause learning scheme that we outlined above did not lead to a significant
speedup of SAT solving. It would be interesting to investigate if more advanced
conflict analysis techniques that may deduce more powerful no-goods, and the
implementation of proper backjumping mechanisms would allow us to approach
the performance of modern SAT solvers. For the latter we would also need to
experiment with a trailing state restoration policy, which does not suffer from the
overhead of unnecessary copying.

4.5 Real Numbers

In this section we demonstrate how OpenSolver is configured for solving con-
straints on the reals. The facilities for these constraints are a modification of
those for arithmetic constraints on integer intervals, which are analyzed in more
detail in Chapter 5. Where the same notions apply, we refer to the definitions in
that chapter.
Chapter 4. Applications

The model of constraint solving that underlies this thesis, and which we described in Chapter 2, applies to combinatorial problems, where we search for combinations of elements from the domains of the variables that satisfy all constraints. In principle these domains are finite sets, but constraints on real valued variables can be handled by considering finitely many intervals of real numbers. For this purpose, in Section 2.2.4 we introduced domain type $\mathcal{F}$, containing all intervals of reals, of which the bounds are floating-point numbers.

Here we limit ourselves to arithmetic constraints that can be written as

$$p \ op \ c$$

where $p$ is a polynomial, $c$ is a constant, and $\ op \in \{=,\leq,\geq\}$ (see Section 5.3). The way that many constraint systems handle such constraints is by decomposing them into atomic constraints. For arithmetic constraints on the reals, a suitable set of atomic constraints is the following:

- addition $x + y = z$
- multiplication $x \cdot y = z$
- exponentiation $x = y^n$, for $n > 1$.
- equality $x = y$.
- disequality $x \leq y$

During the search, hull consistency (see Section 2.2.4) is then maintained for the decomposed problem. This involves the introduction of new variables. It should be noted that in general, hull consistency for the decomposed problem is weaker than hull consistency for the original problem: let $P_{\text{decomp}} = \langle C' : x_1 \in D_1, \ldots, x_n \in D_n, x_{n+1} \in D_{n+1}, \ldots, x_{n+m} \in D_{n+m} \rangle$ be the decomposition of problem $P = \langle C : x_1 \in D_1, \ldots, x_n \in D_n \rangle$. Hull consistency of $P_{\text{decomp}}$ does not imply hull consistency of $P$. The following example demonstrates this property. A detailed analysis is given in [CDR99].

**4.5.1. EXAMPLE.** The constraints $y = x^3$, $x + y = 0$ form a decomposition of $x + x^3 = 0$. The CSP $\langle y = x^3, \ x + y = 0 ; \ x \in [-1,1], y \in [-1,1] \rangle$ is hull consistent\(^3\), but $\langle x + x^3 = 0 ; \ x \in [-1,1] \rangle$ is not. \(\square\)

Constraint propagation is usually implemented by inverting the atomic constraints to isolate all variable occurrences. These inversions define projections of the domains of the other variables on the domains of the isolated variables. For example, for $x \cdot y = z$ we have

- $x = z/y$, if $y \neq 0$

\(^3\)assuming $[-1,1] \in \mathcal{F}$ and using $x + y = 0$ as a shorthand for $x + y = c$, with $c \in \{0\}.$
4.5. Real Numbers

- $y = z/x$, if $x \neq 0$
- $z = x \cdot y$

The projections follow from interval extensions of the arithmetic operations. Interval extensions form the basis of interval arithmetic, which is due to Moore [Moo66]. An interval extension of a function $f : \mathbb{R}^n \to \mathbb{R}$ is a mapping $F : \mathcal{F}^n \to \mathcal{F}$, such that for all $(D_1, \ldots, D_n) \in \mathcal{F}^n$ and $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$, $f(d_1, \ldots, d_n) \in F(D_1, \ldots, D_n)$. For example, ignoring bounds $+/-\infty$ and using an infix notation, the interval extension of the subtraction can be defined as follows

$$[a, b] - [c, d] = \text{hull}([a - d, b - c])$$

When applied to the domains of the variables in the right-hand side of the inversions in the above example, the interval extensions of the division and multiplication operations yield a set of values for the isolated variable that do not violate the constraint. The domains of these variables are then intersected by the smallest $\mathcal{F}$ interval that contains this set.

For expressions that involve more than a single operator, an interval extension can be constructed from the syntactical form of the expression as follows.

- Every constant is replaced by the smallest $\mathcal{F}$ interval that contains it,
- every variable is replaced by an interval variable, and
- every operator is replaced by the interval extension of that operator.

This is called the natural interval extension of the expression.

A problem with the natural interval extension is that multiple occurrences of the same variable are treated as if they were separate variables. Consider for example the natural interval expression of $x - x^3$, applied to the interval $[-1, 1]$. The interval extension of the exponentiation applied to this interval yields the interval $[-1, 1]$, the set of all third powers of values in the original interval. When subtracted from the interval for the other occurrence of $x$, we get $[-2, 2]$, but this interval is wider than the interval of possible outcomes of $x - x^3$. In fact, it is the interval of all possible outcomes of $x - y^3$, with $x, y \in [-1, 1]$. This is called the dependency problem of the natural interval extension. It lies at the heart of the discrepancy between hull consistency for a compound constraint, and hull consistency of the decomposition, illustrated by Example 4.5.1. The decomposition into atomic constraints entails that the natural interval extension is used. Alternative interval extensions exists, having properties that may be preferable to those of the natural interval extension, in some situations. A discussion of these is outside the scope of this thesis, and the reader is referred to the extensive literature on constraints on the reals, where [CDR99] is a good starting point.
Chapter 4. Applications

Implementation

Domain type $\mathcal{F}$ is implemented by a plug-in called RealInterval, of which we have already seen an example on page 40. The RIARule (Real Interval Arithmetic Rule) plug-in implements constraint propagation. The syntax is

$$\text{DRF RIARule } \{x^n*(m) \text{ op } p\};$$

where $x$ is a variable, $n$ is an integer, $m$ is a monomial, $p$ is a polynomial with integer coefficients, and $\text{op} \in \{=,\leq\}$. The plug-in instance will evaluate the natural interval extension of the expression $\sqrt[\text{n}]{p/m}$, and update the domain of $x$ with the hull of the resulting interval, according to operator $\text{op}$.

Recall from Section 2.2.4 that the hull of a set of reals is the smallest floating-point interval that contains the set. The plug-ins RealInterval and RIARule are implemented using the gaol library [Gou], which supports interval arithmetic based on floating-point intervals. Computing with floating-point numbers entails that potentially, a rounding error is made. In gaol, the hull of the outcome of interval arithmetic operations is calculated by outward rounding, i.e., the lower bound is rounded to the greatest floating-point number smaller than, or equal to the actual value for the lower bound, and the upper bound is rounded to the smallest floating-point number greater than, or equal to the actual value for upper bound.

The RIARule plug-in can compute more complex expressions than needed for the inversions of the atomic constraints. Formally, as soon as we allow more than a single interval arithmetic operation per projection function, more than a single rounding error can be made, and it becomes unclear what level of consistency we are computing. For example, we could allow arbitrary linear constrains as atomic constraints, as we do for the integer case in Section 5.7. Consider then the constraint $x+y+z=w$. When we evaluate $\text{hull}(D_x+D_y+D_z)$, the hull of the interval that contains all possible sums of an element from $D_x$, $D_y$, and $D_z$ each, we have three options for which two intervals to add first. Because floating-point addition is non-associative, we would be computing the hull of a decomposition that has a new variable added for either $x+y$, $x+z$, or $y+z$. Because of the accumulated rounding errors, this interval can be larger than the proper hull of $D_x+D_y+D_z$. What is worse, different inversions likely correspond to different decompositions, and the level of consistency is no longer clearly defined. Even though there is no reason to expect that this will be a problem in practice, we therefore prefer to use RIARule only for inversions of atomic constraints.

It is interesting to compare our approach with the HC4 algorithm [BGGP99], which employs a single reduction operator for a compound constraint. Internally, this operator decomposes the constraint into atomic constraints, and enforces hull consistency for this decomposition. Besides not having to introduce new variables for the decomposition, the HC4 operator is very efficient because it applies the
4.5. Real Numbers

inversions of the atomic constraints in a sequence that respects their hierarchical relationship. For example, if the constraint $2x = z - y^2$ is decomposed into

\[ t_1 = t_2 \]

\[ t_1 = 2x \]
\[ t_2 = z - t_3 \]
\[ t_3 = y^2 \]

then the HC4 algorithm first updates the domains of the (internal) variables $t_3$, $t_2$, and $t_1$ in a forward evaluation phase. Then it enforces the top level constraint $t_1 = t_2$, and traverses the decomposition in the opposite direction in a backward propagation phase to update the domain of $t_3$, and of the original variables $x$, $y$, and $z$.

Because we associate a reduction operator with every inversion of a constraint, in our approach we can also exploit these dependencies, but without having to implement a specialized, and somewhat “heavy-weight” reduction operator like HC4. Instead, we use the programmable scheduler of Section 4.1.1 to ensure that first the inversions of the forward evaluation phase are applied, and then the inversions of the backward propagation phase. The schedule for the operator-based scheduler can easily be generated automatically, along with the decomposition. This way, HC4-like functionality can be composed from readily available facilities. A disadvantage is that the decomposition has to be made explicit, but we can characterize the variables that are introduced as auxiliary variables. This way, these variables do not influence the search process, and only imply some memory overhead. The scheme discussed here is demonstrated in Section 5.9.2 in the context of arithmetic constraints on integer interval variables.

Precision

The last two plug-ins that are part of the facilities for constraints on the reals are a node evaluator and a branching operator that reduce the precision of the intervals in the solved forms. For domain type $\mathcal{F}$, the default node evaluator CanonicalDomains, which we encountered briefly in Section 4.3, implements an ECSP

\[ \langle \mathcal{C} ; x_1 \in D_1, \ldots, x_n \in D_n ; \mathcal{D}_1, \ldots, \mathcal{D}_n ; \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle, \]

having

\[ \mathcal{A}_i = [\mathcal{F}] = \{ [a,b] \mid a, b \in \mathbb{F}, a \leq b, -\exists c \in \mathbb{F} a < c < b \} \]

i.e., branching continues until the domains are canonical intervals. Sometimes, we are interested in less precise solved forms. The Precision node evaluator plug-in allows us to specify that a precision of $\epsilon > 0$ suffices:

\[ \mathcal{A}_i = [\mathcal{F}] \cup \{ [a,b] \mid a, b \in \mathbb{F}, 0 \leq b - a < \epsilon \} \]

A node evaluator only characterizes a node of the search tree as a solution, failure, or internal node. If a node is characterized as an internal node, it will be subject to branching. We want to prevent that a variable is selected for branching,
whose domain already has the required precision. There are several ways in which we can realize this:

- Through the specifier string, parameterize `RealInterval` instances with the required precision. When asked for their size, an instance representing \([a, b]\) will report 1 iff \(0 < b - a < \epsilon\).

- Alternatively, we can use an adapter for each plug-in that overrides the size reported by the plug-in with one based on the width of the interval that it represents. This would look something like

  ```
  VARIABLE x IS LimitedPrecisionRealInterval { 1.0e-8, 
     RealInterval { [-1.0, 1.0] } };
  ```

- Make it the responsibility of the branching operator. This operator would then have to know it is dealing with `RealInterval` instances, in order that it can inquire about the width of the interval they represent. However, this means that we would have to re-implement the variable selection strategies offered by the existing branching operators for the specific case of `RealInterval` variables.

In Section 7.3.2 we see a situation where the same `RealInterval` instance (or actually, a clone of it) is used in two cooperating solvers that use different precisions. This is all but prevented by the first two of the above alternatives, and for this reason, we chose to make it the responsibility of the branching operator yet. We can avoid re-implementing the general-purpose variable section strategies by using an adapter:

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

Internally, the branching function of the adapter passes the array of domain pointers to the branching function of the inner reduction operator, but before doing so, it inspects the width of all domains. The pointers to those domains that already have the required precision are replaced by a pointer to a dummy domain of size 1, and will not be split.

This concludes the discussion of the facilities for constraints on the reals. We will see an example application in Section 7.3.2.

## 4.6 Conclusions

In this chapter we described the plug-ins for solving constraints on domain types \(\mathbb{Z}, \mathbb{B}\), and \(\mathcal{F}\). and we introduced some basic facilities for constraint propagation and search. These facilities allowed us to investigate how several existing solving strategies can be realized in OpenSolver through composition. Our conclusions are the following.
An implementation of iterative limited discrepancy search inside OpenSolver is possible, but slightly artificial in the sense that some facilities are not used as intended. A non-iterative variant of limited discrepancy search can be realized by an adapter that annotates the nodes of the search tree with their discrepancy values, and a container that keeps the search frontier sorted on their annotations. To limit memory usage, we switch between LDS and depth-first traversal, depending on the size of the search frontier.

Best first-search can be realized by annotating the candidate nodes for continuing the exploration with a measure of the likelihood that they contain a solution, according to some heuristic. We demonstrated this for Warnsdorf's heuristic for solving the knight's tour problem. In this case the likelihood is evaluated after constraint propagation in the candidate nodes, which involves a limited amount of breadth-first traversal. The actual evaluation is done by a node evaluator plug-in. The traversal strategy is realized by a combination of a container and a selector plug-in.

Of the techniques used in SAT solving, clause learning (no-good recording) can be implemented, but storing the full search frontier and using a copying state restoration policy makes OpenSolver unsuited for non-chronological backtracking.

One of the standard scheduler plug-ins supports programmable schedules. This allows us to optimize constraint propagation by taking into account knowledge about how the reduction operators interact, and about their computational complexity. As an example, if reduction operators implement a decomposition of a polynomial constraint into atomic arithmetic constraints, we can apply them in an order that respects their hierarchical relationship. Normally this is hard-wired in heavy-weight reduction operators that implement algorithms like HC4 of Granvilliers et al. [BGGP99]. In OpenSolver, the same effect can be realized through composition.

While the efficiency of some plug-ins can be optimized further, we have shown that OpenSolver is not inherently less efficient than systems that are used in practice. In particular, we compared performance with that of ILOG Solver on the $n$-queens problem, a benchmark that is well suited for testing the basic machinery of solvers.

Having discussed the implementation of domain types $Z$, $B$, and $F$, in the next chapter we turn our attention to the one remaining standard domain type of Chapter 2: that of the integer intervals.

---

4The coordination layer mechanism offers further possibilities for implementing iterative schemes, though.