Composing constraint solvers
Zoeteweij, P.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 3

OpenSolver: a Software Component

This chapter describes the OpenSolver software, which is used in the remainder of the thesis as an experimental platform for composing constraint solvers. OpenSolver can best be described as a coordination-enabled abstract branch-and-propagate tree search engine. It is based on the solving algorithms of Section 2.3: branching, and pruning the search tree are implemented as the application of reduction operators that modify the domains of variables. It is abstract in the sense that its functionality is determined by software plug-ins that configure the basic solving algorithms.

These plug-ins come in a number of categories, corresponding to various aspects of branch-and-propagate tree search. A separate category of plug-ins covers the coordination layer of the algorithm (Figure 3.1). This category is special in the sense that it does not correspond to one specific aspect of Algorithm 2.1 or 2.2. Instead, plug-ins in this category control the execution of the solving algorithms, and facilitate the exchange of data between a solver and its environment.

No component technology is used to implement the branch-and-propagate constraint solving, but a major design goal was that OpenSolver itself can be used as a software component in several solver cooperation schemes. This is realized through the coordination layer mechanism. In Chapters 7, 8, and 9 we will be looking at examples of larger systems, were OpenSolver is used as a software component.

3.1 Introduction

OpenSolver evolved from the DICE (DIstributed Constraint Environment) system, which we discuss in Chapter 9. DICE itself started as an implementation of the coordination-based distributed constraint solver proposed by Monfroy and Arbab [Mon00a, AM00]. The original DICE system is described in [Zoe03b]. It is a framework for distributed branch-and-propagate tree search, whose functionality is determined by plug-ins for domain types, domain reduction functions,
and branching and traversal strategies. Basically, in DICE every plug-in resides in its own process, and in [Zoe03a] we proposed an optimization that allows an arbitrary distribution of the plug-ins over a set of cooperating solvers. On the one hand, OpenSolver implements the solvers of the proposed optimization.

On the other hand, many of our experiments do not require distributed solving, and we also wanted to implement an efficient sequential constraint solver. Moreover, we wanted to be able to use the same plug-ins in DICE and in this sequential solver. Therefore we decided to develop a single application, and tailor it towards reuse as a software component in several environments. To a large extent this is realized by the coordination layer plug-in, which forms the interface between OpenSolver and its environment. One plug-in configures OpenSolver as a component solver of DICE, and another plug-in configures it as a stand-alone solver.

In addition to these two roles, the coordination layer made it very easy to implement the time-out mechanism that forms the basis of the parallel constraint solver described in Chapter 8. We also use it to implement nested search. This technique entails that the functionality of a domain reduction function involves a limited branch-and-propagate tree search. We use an almost autonomous OpenSolver instance for such a DRF, and this instance interfaces with another OpenSolver through a special coordination layer plug-in. Nested search is the topic of Chapter 7.

The remainder of this chapter is organized as follows. In Section 3.2 we describe the different categories of plug-ins related to constraint solving. Section 3.3 describes the odd one out: the category of coordination layer plug-ins. In Section 3.4 we clarify some implementation aspects, including writing new plug-ins for OpenSolver.

### 3.2 Constraint Solving Plug-ins

The following categories of plug-ins implement constraint solving:

- **domain types** that implement the domains of variables.
3.2. Constraint Solving Plug-ins

\[
\begin{align*}
\langle Configuration \rangle & \rightarrow \langle Statement \rangle ; \{ \langle Statement \rangle ; \} \\
\langle Statement \rangle & \rightarrow \langle Keyword3 \rangle \langle Identifier \rangle \text{ IS} \langle Identifier \rangle \langle Specifier \rangle \\
& \quad | \langle Keyword2 \rangle \langle Identifier \rangle \langle Specifier \rangle \\
\langle Keyword3 \rangle & \rightarrow \text{VARIABLE} | \text{AUX} \\
\langle Keyword2 \rangle & \rightarrow \text{DRF} | \text{SCHEDULER} | \text{ANNOTATION} | \text{TDINFO} \\
& \quad | \text{FRONTIER} | \text{INTERNAL} | \text{EXPLORE} | \text{EXPAND} \\
\langle Specifier \rangle & \rightarrow \{"\langle String\"\)\}
\end{align*}
\]

Figure 3.2: Syntax of the OpenSolver configuration language

- **reduction operators** that modify these domains.
- **schedulers** of reduction operators,
- **containers** of nodes of the search tree,
- **selectors** that make a selection among the nodes stored in containers,
- **annotations** that decorate the nodes of the search tree with extra information, to be used by plug-ins in some of the other categories,
- **evaluators** of nodes of the search tree; these determine whether a node of the search tree is a solution, failure, or internal node.

An OpenSolver instance is configured through a script in a simple language that has a statement for each of these categories. Program 3.1 is an example of such a script, related to one the experiments in Chapter 5.

Figure 3.2 defines the syntax of OpenSolver configuration scripts, where {...} should be read as “zero or more instances of the enclosed,” and where “{” and “}” denote the curly bracket symbols. Each statement consists of a keyword for one of the plug-in categories, plus an identifier-specifier pair. The identifier designates a particular plug-in in the category of the statement, and the specifier string is used to initialize an instance of this plug-in. For the purpose of this mechanism, every plug-in, in any category should be able to initialize itself from a specifier string. These specifier strings can be arbitrarily complex. For example, in Chapter 7 we use a plug-in that is an almost autonomous OpenSolver instance. Comparable to procedure definitions in imperative programming languages, the specifier string for this plug-in contains a full solver configuration. At the other extreme, when given the empty string as a specifier, the plug-in for domain type $\mathcal{F}$, the set of all floating-point intervals, yields a representation for the domain $[-\infty, \infty] = \mathbb{R}$.

The **VARIABLE** and **DRF** statements introduce variables and their domains, and reduction operators that operate on them. In addition to the identifier-specifier
VARIABLE x IS IntegerInterval {1..100000};
VARIABLE y IS IntegerInterval {1..100000};
VARIABLE z IS IntegerInterval {1..100000};
VARIABLE obj IS IntegerInterval {1..100000};
AUX aux_x3 IS IntegerInterval {1..100000};
AUX aux_y2 IS IntegerInterval {1..100000};
AUX aux_z3 IS IntegerInterval {1..100000};
AUX aux_x1y1 IS IntegerInterval {1..100000};

DRF IIARule { aux_x3~1 * (1) = x^3 }; 
DRF IIARule { aux_y2~1 * (1) = y^2 }; 
DRF IIARule { aux_z3~1 * (1) = z^3 }; 
DRF IIARule { aux_x1y1~1 * (1) = y * x }; 
DRF IIARule { x^3 * (1) = aux_x3 }; 
DRF IIARule { y^2 * (1) = aux_y2 }; 
DRF IIARule { z^3 * (1) = aux_z3 }; 
DRF IIARule { y^1 * (x) = aux_x1y1 }; 
DRF IIARule { x^1 * (y) = aux_x1y1 }; 
DRF IIARule { aux_x3~1 * (1) = -1*aux_y2 + 1*aux_z3 }; 
DRF IIARule { aux_y2~1 * (1) = -1*aux_x3 + 1*aux_z3 }; 
DRF IIARule { aux_z3~1 * (-1) = -1*aux_x3 + 1*aux_y2 }; 
DRF IIARule { obj^1 * (1) = 2*aux_x1y1 + -1*z }; 
DRF IIARule { aux_x1y1~1 * (-2) = -1*obj + -1*z }; 
DRF IIARule { z^1 * (1) = -1*obj + 2*aux_x1y1 }; 

DRF Optimize { +obj }; 
DRF RoundRobin { 0, x, y, z, obj }; 
SCHEDULER ChangeScheduler { schedule = 
  { 1,2,9,4,0,2,10,5,0,1,11,6,3,12,13,7,8,3,14,15 } 
};

Program 3.1: Example of an OpenSolver configuration script
pairs of the other statements, the VARIABLE statement uses an extra identifier that is interpreted as the variable's name. The AUX keyword is a variant of VARIABLE for introducing auxiliary variables. These two statements are the only ones that add plug-in instances.

The other statements replace plug-in instances, for which a default is readily available. The SCHEDULER keyword is used for replacing the scheduler of reduction operators. FRONTIER and INTERNAL replace the containers for storing sets of nodes of the search tree, and EXPLORE and EXPAND replace the selectors operating on them. The ANNOTATION statement specifies what information, by means of an annotation plug-in, is attached to nodes of the search tree. The default is to use no annotations. The statement for introducing a node evaluator is TDINFO, for termination detection information. This reflects OpenSolver's origin as a distributed system: establishing that distributed constraint propagation has finished is then a matter of distributed termination detection. Determining the nature of a node of the search tree is naturally combined with detecting the termination of constraint propagation.

In the remainder of this section, we discuss the different categories of constraint solving plug-ins.

### 3.2.1 Variable Domain Types

This category of plug-ins corresponds directly to the variable domain types discussed in Section 2.2.4. Plug-ins exist for the four standard domain types $\mathbb{B}$, $\mathbb{Z}$, $\mathbb{I}$, and $\mathcal{F}$ that were introduced there, and as is the case for all categories of plug-ins, new domain type plug-ins can be added to an OpenSolver installation. This is described in Section 3.4. In Chapter 6 we discuss special-purpose domain types that are introduced for solving one specific kind of combinatorial problems.

Just like in object-oriented programming an object is an instance of a class, in OpenSolver a variable domain is an instance of a variable domain type. As we shall see in Section 3.4, the plug-ins, and hence the individual domain types are actually classes, with a common base class for each category. For now it suffices to realize that being objects, the domains of variables have a state, on which a number of operations are defined, and that these operations are implemented by means of member functions (we use the C++ terminology, in other object-oriented languages member functions are called methods).

In the OpenSolver input language, variables are introduced with the following statement.

```
VARIABLE (Identifier) IS (Identifier) (Specifier)
```

The first identifier gives the variable a name, and the second identifier designates the plug-in that will be used to implement the domains that are associated with the variable, during the solving process. The specifier is a character string that represents the initial domain. It will be used to create a domain for the variable
Chapter 3. OpenSolver: a Software Component

in the root of the search tree.

Specifier strings are interpreted by a constructor for the class that implements a plug-in. For a set \( T \) and a domain type \( T \subseteq \mathcal{P}(T) \), such a constructor implements a partial function \( f : \mathcal{P}(T) \to T \) having \( f(D) = T(D) \). For example, the plug-in \( \text{RealInterval} \) implements the standard type \( \mathcal{F} \subseteq \mathcal{P}(\mathbb{R}) \), the set of all floating-point intervals. For the specifier string we can use any interval where the bounds have a finite decimal representation. This is the case for \( \frac{1}{10} \), but no floating-point representation exists for \( \frac{1}{10} \), so

\[
\text{VARIABLE } x_1 \text{ IS RealInterval } \{[-0.1, 0.1]\};
\]

will create the smallest floating-point interval that properly contains \([-\frac{1}{10}, \frac{1}{10}]\).

For variable domain types, the operations on the state include the following.

- An operation to \textit{clone} the domain. OpenSolver is a \textit{copying-based} constraint solver (see Section 4.2), and the clone operation implements the copying on the level of the variable domains.

- An operation to \textit{split} the domain into a number of subdomains. The member function for this operation has an integer argument that can be used to specify a particular method for generating these subdomains, for example enumeration, or bisection (see Figure 4.2 on page 70). The interpretation of this argument, and the value selection strategies that it encodes are specific to the variable domain types. Applying the split operation on the domain of a single variable is the primary method of branching, but more complex branching strategies, typically involving more than a single variable, are also supported.

- A member function that gives an indication of the \textit{size} of the domain. This is a non-negative integer, where 0 means that the domain is empty, i.e., a failure has been deduced. The value 1 indicates that the domain is a singleton set, and values greater than 1 are an indication that the domain can be split into a number of subdomains. In principle, the domain sizes determine the nature (solution, failure, or internal) of the nodes of the search tree, but as we shall see below, this can be overridden by a node evaluator plug-in.

Instead of \texttt{VARIABLE}, the keyword \texttt{AUX} can be used to introduce a variable. The syntax is the same, and the only effect is that a flag is set, to mark the variable as auxiliary. Node evaluator plug-ins use this information to implement the notion of auxiliary variables introduced in Section 2.2.5. For auxiliary variables, all domains except the empty set are final domains. They are not considered in distinguishing solutions from internal nodes of the search tree. Therefore we do not need to branch on auxiliary variables.
3.2. Constraint Solving Plug-ins

3.2.2 Reduction Operators

Plug-ins in the reduction operator category implement the domain reduction functions and domain branching functions of Sections 2.3.1 and 2.3.2. A hybrid form is used for optimization. In the OpenSolver configuration language, the statement for introducing a reduction operator is

\[ \text{DRF (Identifier) (Specifier)} \]

The specifier string typically contains the list of variables that the operator applies to, and some further specification of the operation that it performs on these variables. For example,

\[ \text{DRF DDNEQ \{ x1 - x2 \<\> 2 \}}; \]

enforces the constraint \( x_1 - x_2 \neq 2 \) on two discrete domain variables \( x_1 \) and \( x_2 \).

State

Reduction operator instances have a state that holds at least an internal representation of the information extracted from the specifier string. Contrary to variable domains, the state of a reduction operator is global. It applies to all nodes of the search tree. The only information about a reduction operator that is stored per node, is a flag indicating whether the operator is active or not. Reduction operators can signal to the scheduler that controls their application that they have become redundant in a certain branch of the search tree. Schedulers (see below) may use this information to avoid unnecessary application of such operators. In principle, the design of OpenSolver also allows that new, redundant reduction operators (and auxiliary variables) are added during the solving process, to be active in particular parts of the search tree only, but these facilities are not currently exploited.

Interface

Three member functions constitute the basic interface of reduction operators and the rest of the system:

- a function that reports the names of the variables that the reduction operator applies to; this information is typically extracted from the specifier string when the reduction operator is created,

- the propagation function, which is called during the constraint propagation phase, and

- the termination function, which is called upon termination of constraint propagation, during the branching stage.
Both the propagation function and the termination function take as an argument an array of pointers, whose type is the abstract base class for variable domain types. Through these pointers, the functions can reduce the domains of the variables. The OpenSolver framework takes care that the arrays of pointers correspond directly to the variable names reported by the first of the above three member functions. Also, like the domain reduction functions that they implement, reduction operators have an input scheme and an output scheme: the set of variables that trigger their application, and the set of variables that they can modify, respectively. In Section 3.4 we see how these are implemented.

The propagation function and the termination function must supply the solver with information about variables that they change. The minimum requirement is that they set a flag for every change, but a more elaborate protocol is possible. For example it could be useful to set different flags for modifying a bound or an internal value of a domain. Per variable, a reduction operator can specify what modifications it is interested in. The reduction operators that modify a variable and the reduction operators that depend on this variable must use the same protocol for signaling such modifications. It is up to the scheduler that applies the reduction operators to exploit this information, though. An example of the use of this facility is discussed in Section 4.2.

**Interaction with Domain Types**

In Algorithm 2.1 domain reduction is realized by intersecting variable domains with the outcome of the domain reduction functions. In OpenSolver this is not implemented in such a clean way. All domain type plug-ins implement the intersection, but sometimes it is more efficient to use a different modification of the domains. A reduction operator can then perform modifications that are specific to the domain type that the operator is defined for. In other words, the operators make **assumptions** about the types of the domains that they operate on. Such assumptions are implemented by **type casting** (see also Section 3.4), as a result of which, member functions for domain specific modifications become available.

As an illustration, the DDNEQ operator of the above example operates on finite domains variables, and will typecast the argument domains to objects of the class that implements this domain type. If one of the argument domains has size 1, it can now retrieve the integer value that it contains. Instead of having to construct a domain for intersecting the other argument domain with, it will call a member function specific to the finite domains implementation that allows individual values to be removed.

**The Termination Function**

Normally, the termination function is used for branching. The creation of subproblems is implemented by this function creating subdomains for one or several
of the variables. Subdomains can be created in any way that suits the branching strategy, but in most cases the plug-ins rely on the basic splitting methods provided by the domain types.

The termination and propagation functions can cooperate to implement optimization. If after termination of constraint propagation we have not deduced a failure, while none of the variable domains can be split any further, the node of the search tree is considered to be a (possibly suboptimal) solution. The termination function can then record some information about this suboptimal solution, such as a new bound for the outcome of an objective function. The objective function can be evaluated by regular constraint propagation, and the propagation function can then enforce this new bound as a dynamic constraint on the variable that holds the outcome of the objective function.

**Classification of Reduction Operators**

Depending on the use of the propagation and termination functions, three kinds of reduction operators\(^1\) are distinguished.

- **propagation operators**, these are reduction operators that do not modify their state, and whose termination function does not create any subdomains. Propagation operators are active only during constraint propagation, and through their propagation functions, they implement the DRFs of Definition 2.3.1 on page 19.

- **branching operators**, these are reduction operators that do not modify their state, and where the propagation function does not modify the variable domains. Through their termination functions they implement the domain branching function of Definition 2.3.3 on page 23. Branching operators are active only during the branching stage.

- **optimization operators**, these are reduction operators where the termination function creates no subdomains, but modifies the state, and where the result computed by the propagation function depends on this state. They are active in both the propagation stage and the branching stage of the solving algorithm.

**3.2.3 Schedulers**

The application of the reduction operators is controlled by plug-ins in the scheduler category. In principle, there are two schedulers involved: one for the propagation stage, which applies the propagation functions of the reduction operators, and one for the termination stage, which applies the termination functions. In

\(^1\)The keyword DRF is a misnomer because it refers specifically to propagation operators, while it is also used to introduce branching operators and optimization operators.
practice, there is no need for elaborate scheduling mechanisms in the termination stage, and the latter scheduler is currently fixed to apply the termination functions once, in sequence.

In contrast, the scheduler for the propagation stage implements the constraint propagation algorithm, and is of great influence on the efficiency of the solver. The statement for modifying the propagation scheduler is

\[
\text{SCHEDULER (Identifier) (Specifier)}
\]

Contrary to the \texttt{VARIABLE} and \texttt{DRF} statements, which always extend the solver configuration, this statement \texttt{replaces} the current propagation scheduler. Currently, it is not possible to use different schedulers in different nodes of the search tree. In a distributed setting, however, each solver has its own scheduler, and depending on the constraints assigned to a solver, it may make sense to use different plug-ins here.

Schedulers have a state per node of the search tree. This state can be used to store the bookkeeping of reduction operators that still need to be applied, like the set \(G\) of Algorithm 2.1. In this sense, schedulers are similar to variable domains, and they also provide an operation for cloning the state of the scheduler. Cloning the scheduler state is interesting when branching commences before constraint propagation has reached a fixed point, and some operators are still scheduled for application.

The primary member function of a scheduler plug-in runs the constraint propagation algorithm. In addition, a scheduler provides to the framework member functions for scheduling individual variables and reduction operators. These are called when reduction operators are introduced, and when changes to variables are made outside the control of the scheduler, for example when creating new nodes of the search tree by splitting the domain of a variable.

The scheduler has access to two data structures: one that contains the problem structure (the \texttt{PStruct}) and one that contains information about the problem that is specific to a node of the search tree (the \texttt{WPStruct}, for \texttt{world} problem structure). The \texttt{PStruct} gives access to the reduction operators, and contains the dependencies between variables and reduction operators. The \texttt{WPStruct} is used mainly to keep track of which reduction operators are still active: through this data structure OpenSolver allows a scheduler to deactivate a reduction operator. If it signals after application that it will not be able to achieve further reduction, it can be deactivated in the present branch of the search tree. A scheduler plug-in can choose to use this facility or not. Reference counting, and copy-on-change are used to maintain this information. As a result, actually switching off reduction operators may involve a considerable memory overhead (bounded by the number of reduction operators times the size of the search frontier) for storing new versions of the bitmap of active operators.

Algorithm 2.1 computes a fixed point of the domain reduction functions in its input set \(F\). In contrast, a scheduler plug-in need not run to completion. It may
stop executing the constraint propagation algorithm at any point, but it needs to signal to the solver whether it wants to be reactivated later, to continue the computation of the fixed point. If this is not the case, the solver will consider that the propagation stage has finished. Otherwise it will activate the scheduler again before starting the branching stage. This can be useful even in a stand-alone constraint solver: as in the model of Monfroy and Arbab [AM00], multiple nodes of the search tree can be subject to constraint propagation, and each of these nodes can be expanded by branching. If schedulers run to completion, constraint propagation in these nodes is executed in sequence. Examples of CSPs exist where in some nodes of the search tree, constraint propagation takes much longer than in others. For such problems, one node could block the progress of search in the other nodes. By running a scheduler only for a fixed amount of time, and then passing control to other nodes before resuming the computation of the fixed point, we can benefit from concurrency in the search without having to resort to distributed processing.

3.2.4 Containers

The state of the OpenSolver search algorithm consists of three sets of nodes of the search tree (Figure 3.3). Each of these sets is implemented by a container plug-in.

- The *search frontier*, containing unexplored nodes that are pending constraint propagation. Both the original CSP, and the subproblems that are the result of applying a branching operator enter the algorithm in this set.

- A set of nodes that are subject to *constraint propagation*.

- A set of *internal nodes* where constraint propagation has terminated without deducing a failure or a solution. In these nodes, the search tree can be expanded by applying a branching operator.

In what follows, the *state of the solver* usually refers to these three sets.
Chapter 3. OpenSolver: a Software Component

Being set implementations, containers have member functions for adding and removing nodes of the search tree, and for iterating over their contents. Implementing sets of nodes of the search tree in specific, containers can be aware of some properties of the nodes that they contain, and iterate over these nodes accordingly. Such properties are implemented by *annotations* (discussed below). We will see examples of actual container implementations in Sections 4.1.2 and 4.3.

The containers for the sets of nodes that are pending propagation and branching can be modified. The syntax is, respectively,

```
FRONTIER <Identifier> <Specifier>
```

```
INTERNAL <Identifier> <Specifier>
```

Like the SCHEDULER command, these commands replace the currently active plug-in instances. For the set of nodes that are subject to propagation, it is important that all nodes can be removed from the container efficiently, so here an implementation based on a linked list is used. There seems to be little use for other alternatives here, so this particular container is currently fixed, and cannot be changed in the configuration language.

### 3.2.5 Selectors

Selectors are used to identify the nodes that are transferred between the three sets of Figure 3.3. All nodes where constraint propagation has terminated are moved to the rightmost set automatically, so selectors are used only for transferring nodes from the search frontier, and for selecting the nodes that will be expanded by branching. The respective commands are the following.

```
EXPLORE <Identifier> <Specifier>
```

```
EXPAND <Identifier> <Specifier>
```

These commands override the currently active selectors.

Apart from the plug-in machinery, selectors offer a single operation to the OpenSolver framework. The member function for this operation takes as an argument an array of containers, and returns an array of elements of these containers. So in OpenSolver, the selectors can inspect the entire state, i.e., all three sets of nodes of the search tree of Figure 3.3. It is the responsibility of the programmer of the plug-in, and of the application or person who writes the solver configuration to ensure that nodes are selected from the correct set. For example, if OpenSolver consults the selector for identifying the nodes that must be expanded by branching, and a node from the search frontier is selected, a run-time error will occur once OpenSolver discovers that this node cannot be removed from the container of nodes that are pending branching. We see selectors at work in Section 4.3.
3.2.6 Node Evaluators

The TDINFO command changes the OpenSolver node evaluator:

TDINFO \{Identifier\} \{Specifier\}

The purpose of a node evaluator\(^2\) is to determine the nature of a node of the search tree, solution, failure, or internal node, once constraint propagation has finished. In a distributed setting, this information is collected when the coordination-layer plug-ins of the cooperating solvers try to establish termination of distributed constraint propagation. For this reason the keyword refers to the information that is collected during termination detection.

There is basically one way to establish that a node is a failure: at least one of the variable domains reports a size 0. Therefore the main purpose of a node evaluator is to distinguish between internal nodes and solution nodes, the latter corresponding to subproblems that are in solved form. This implements the test \(D_1 \in A_1, \ldots, D_n \in A_n\), for an ECSP of the form (2.2), as we discussed in Section 2.2.5. To this end, node evaluator plug-ins provide a member function that takes as an argument the array of domains of a node. In Section 4.5 we use a node evaluator to calculate solved forms of a limited precision, for variables whose domains are floating-point intervals.

After establishing the nature of a node, the plug-in instance is cloned, and attached to the node itself, in order that the nature of the node is made known to the branching operators. The termination functions of such operators can then verify properties that are hard to establish by means of constraint propagation, and may decide to fail a node yet, after it was characterized as a solution by the branch-and-propagate tree search.

In addition to establishing the nature of a node of the search tree, node evaluators may access the annotations, thus allowing extra information to be attached to a node before the branching stage commences. We will see how this facility is used in Section 4.3.

3.2.7 Annotations

Annotations are used to decorate nodes of the search tree with additional information, to be maintained and referenced by plug-ins in the other categories. The annotation of a node is set by the following command.

ANNOTATION \{Identifier\} \{Specifier\}

In addition to the basic plug-in machinery, the base class for annotation implementations requires only that annotations can be cloned. Since annotations exist

\(^2\)ILOG Solver also has a NodeEvaluator class, which implements related, but different functionality [Per99].
only for the convenience of other plug-ins, OpenSolver makes no further assumption about the functionality that they offer. This is entirely a matter of subtyping, and type casting by the plug-ins that use the annotations. We see specific uses of annotations in Sections 4.1 and 4.3, and in Chapter 8.

### 3.2.8 Putting it All Together

Before we move on to the coordination-layer plug-in, it is good to take a step back and discuss the relation and interaction between the plug-ins in the categories that we just introduced.

Generally, branch-and-propagate constraint solving starts with constraint propagation. This involves the domain type, reduction operator, and scheduler plug-ins. The **domain type** plug-ins provide representations for the domains of the variables, and the **reduction operators** inspect and modify the domains to enforce the constraints. Note that the concept of a constraint is absent in the system.

Before the start of the branch-and-propagate search, the system asks the reduction operators for the names of the variables that they want to be applied to. These names are compared to the variable names introduced by the **VARIABLE** and **AUX** statements, and the relation between the variables and the reduction operators is laid down in the **PStruct**. The actual application of the reduction operators is controlled by the **scheduler** plug-in. It is responsible for applying the reduction operators to the right variables, as specified in the **PStruct**. The scheduler plug-in applies only the propagation functions of the reduction operators, so during the constraint propagation stage, only two of the three kinds of reduction operators, namely propagation operators and optimization operators are active.

At some point, the scheduler plug-in will return control to the system, and notify that constraint propagation has finished. It now becomes important to realize that we have been working in a particular node of the search tree. At the start of the solving process, this is the root node. These nodes are data structures of the framework, but they may have been decorated with extra information in the form of an **annotation** plug-in.

When constraint propagation finishes in a node of the search tree, the system applies the **node evaluator** plug-in to this node. In any case, a node evaluator has to determine (typically by inspecting the sizes of the variable domains) whether a node is a solution, failure, or internal node of the search tree, but it may gather further information about the node, which can then be stored in its annotation.

Nodes that have been characterized as failures (dead-end leaves in the search tree) are least interesting from the perspective of the search process: these are basically just discarded. Solutions are slightly more interesting, but before a node gets the solution treatment, which may actually mean the end of the solving
process, all reduction operators are applied once more. This time, instead of the propagation functions, the termination functions of the reduction operators are applied. Termination functions are used for branching, but that concerns internal nodes of the search tree only. For solutions they can perform some last-minute tests, and decide to characterize a node as a failure yet, but they can also record some information about the solution, such as a new bound for a criterion variable in the state of the reduction operator.

Internal nodes of the search tree are the most important for the search process. They are stored in the rightmost of the three sets of Figure 3.3. Exactly how they are stored, and in what order they can be retrieved again, is determined by the container plug-ins that implement these sets. On several occasions, two selector plug-ins will examine the state of the solver (the three sets of Figure 3.3). One of these plug-ins will make a (possibly empty) selection among the internal nodes in the rightmost set. These nodes are subjected to branching, which entails that the termination functions of all reduction operators are applied in sequence. Typically, most reduction operators are propagation operators, and have inactive termination functions, and there is exactly one reduction operator that is a branching operator. This operator's termination function will create subdomains for some (typically one) of the variables. The system will generate a new node for each of these subdomains, cloning the domains of the other variables. The original internal node can now be discarded, and the new nodes are stored in the set of nodes that await constraint propagation.

The second selector plug-in selects nodes from this latter set. These nodes are moved to the middle set of Figure 3.3, where they will be subjected to constraint propagation, as we described at the beginning of this section. In case of an all-solution search or an optimization problem, the constraint solving process ends when all three sets of nodes are empty. This completes our overview of the solving process, and the roles of the different plug-ins therein. The solving process is under tight control of the coordination layer plug-in, which we will discuss next.

### 3.3 The Coordination Layer Plug-in

Recall that a major design goal was that OpenSolver can be used as a software component in several solver cooperation schemes. This is realized through the coordination layer plug-in. Every OpenSolver instance has exactly one plug-in in this category installed. This plug-in controls the branch-and-propagate tree search, and through it, the solver can exchange information with its environment. OpenSolver is almost a full application, but it has to be complemented by a coordination layer plug-in to be able to function. Even when it is used as a stand-alone constraint solver, this plug-in is responsible for aspects such as

- all I/O, in particular providing a solver configuration in the language of Figure 3.2, and dispatching solutions,
whether we search for one or all solutions, or whether we just want to count them.

The coordination layer plug-in is similar to the constraint solving plug-ins in the sense that it is activated using an identifier-specifier pair, but because its presence is required for OpenSolver to function, this pair is part of the shell command used to run the OpenSolver, for example,

```
opensolver -c SeqFile10 8queens.inp
```

starts OpenSolver as a UNIX process, using the SeqFile10 coordination layer plug-in. This coordinates it as a stand-alone, sequential constraint solver that reads the configuration script from a file, whose name is read from the specifier string. The `-c` option is followed by the identifier-specifier pair for the coordination layer plug-in. In the case of SeqFile10 the specifier string is a filename. If it should include spaces (which is not the case for SeqFile10), it has to be enclosed in quotes.

The interaction between OpenSolver and its coordination layer plug-in is via a command loop. After activating the plug-in, OpenSolver continually asks it what to do next. One of the first commands that are usually issued is the following.

**specifier.** This tells OpenSolver that a configuration specification is available. After receiving this command, OpenSolver asks the coordination layer plug-in for a pointer to a location where this specifier is stored, in ASCII byte format in the language of Figure 3.2. After processing the specifier string, OpenSolver notifies the coordination layer that the memory it occupies can be deallocated.

In total, the current version uses 20 commands that fall roughly in two categories: controlling the solving process, and interaction with the environment. Below we describe those commands that are essential for understanding the OpenSolver architecture, and its use as a software component in the other chapters.

### 3.3.1 Controlling the Solving Process

The nodes of the search tree reside in a data structure called the world database, which is essentially an array of slots that can each hold a single node. When nodes are deallocated they leave a vacant slot, which can be reused when a new node must be stored. If no vacant slot is available, new slots are created. The primary purpose of the world database is to provide a uniform node identification scheme when several OpenSolvers participate in a distributed constraint propagation algorithm. In that case, the world database of each solver contains an array of slots for each of the participating solvers. Every node has an owner: the solver that created it, and all solvers access the data structures for that node through the same slot, in the array for the solver that owns it. Now a node can uniquely be
identified by a tuple consisting of the number of the solver that owns it, and the index of the slot it occupies.

In a distributed setting, the three sets of nodes of Figure 3.3 are distributed over the cooperating solvers. All solvers maintain the three sets, but a single node can appear only in the state of one solver, i.e., the global state is distributed over the cooperating solvers. While only one solver marks a node as pending propagation, pending splitting, or being subject to propagation, all solvers may have data structures for the node, containing the local domains and information on active reduction operators.

We discuss the commands that control the solving process roughly in the order in which they occur during a regular branch-and-propagate tree search.

**schedule propagation.** Receiving this command, OpenSolver activates the propagation selector. This will identify a (possibly empty) subset of the set of nodes that are pending propagation. The nodes in this subset will be transferred to the set of nodes that are subject to constraint propagation.

**propagation.** For each node in the latter set, run the scheduler of DRFs to apply constraint propagation. When the scheduler finishes, it signals whether constraint propagation has terminated, and a flag on the node is set accordingly.

When a node is scheduled for propagation by the propagation selector, this is signaled to the coordination layer. From that point on, the coordination layer will monitor the progress of constraint propagation for the node, through the flag that we just mentioned. This monitoring takes place for all nodes that the coordination layer knows to be subject to constraint propagation, after each propagation command. It may seem counterintuitive that this is the responsibility of the coordination layer, but this is necessary in case several OpenSolver instances participate in a distributed constraint propagation algorithm, such as the one that is described in Chapter 9. In a distributed setting, the information on termination of constraint propagation is local: it applies only to the set of DRFs known to the solver that issued the propagation command, and the cooperating solvers have to combine this information in their coordination layers to obtain a global view.

As a part of a distributed termination detection algorithm, or in a stand-alone solver after having been informed that constraint propagation has terminated, the coordination layer may inquire about the status of a node by creating a new node evaluator, and asking the solver to apply it.

**evaluate.** Receiving this command, OpenSolver asks the coordination layer for a pointer to a node evaluator, and applies it to the node of the search tree for which the command is issued.
The next time OpenSolver asks the coordination layer for a command after **evaluate** was issued, the coordination layer knows that the node evaluator it created has been applied to the node of the search tree that it was interested in at that time. The node evaluator now contains information on whether the node is a solution, failure, or internal node, regarding the part of problem known to the present solver. In a distributed constraint propagation algorithm, this information must now be combined with the view of the other participating solvers. In a stand alone solver, the global status of the node is known immediately, and can be relayed to the solver with the following command.

**termination.** This command informs the solver that constraint propagation has terminated in a particular node of the search tree. In the case of distributed constraint propagation, this information is global. A node evaluator is provided by the coordination layer to indicate the status of the node.

For solutions, the solver runs the scheduler of reduction operators for the termination stage. This allows the optimization operators to update their state according to the current solution (for example, set a new bound). For failures, the coordination layer is informed that it can start the deallocation process for the node. Internal nodes are moved to the container of nodes that are pending branching.

In a distributed setting, the cooperating solvers must negotiate which solver is allowed to branch on an internal node. Only the coordination layer plug-in of the identified solver should issue the **termination** command for the node. For this purpose, a node evaluator can access information on whether a solver is configured to split a certain variable, so this decision can be made as a part of termination detection. For example, in a distributed fail-first policy, as a part of establishing termination of constraint propagation, the cooperating solvers will likely be circulating a token (see Section 9.2.3). Before forwarding the token, the solvers will issue **evaluate** commands. Through the node evaluator they can inquire about the sizes of the variable domains, and thus search for the smallest domain as a part of termination detection. Similarly, the node evaluator can determine whether a solver is able to branch on this variable, so the token is annotated with the id of the variable with the smallest domain found so far, the size of this domain, and if the token has already visited a solver that is able to split the domain of the variable, the id of this solver.

The following command initiates the actual branching.

**schedule branching.** This command activates the selector of nodes that are pending branching. This yields a subset of the nodes in the rightmost set of Figure 3.3. In each of these nodes, the scheduler of termination functions is run to actually generate subproblems. The nodes for these subproblems are added to the set of nodes that are pending propagation. The coordination layer is informed that it can start the deallocation of the parent nodes.
The Coordination Layer Plug-in

In a stand-alone solver, the process of deallocating a node of the search tree simply consists of issuing the following command to the solver.

**forget world.** This command tells the solver to deallocate the data structures for a particular node of the search tree, and to free the slot it occupies in the world database. Also any reference to it from within the containers that implement the sets of Figure 3.3 is removed.

In the distributed case, deallocating a node is less straightforward, though. The reason is that for processing nodes that descend from it, we may still need to clone some of the domains. This is done the first time that a solver learns about a particular node, as a part of constraint propagation, but this may well be after the solver that created the node has initiated the deallocation of the parent. Therefore deallocating a node involves counting the number of descendants that have been created. Only when this matches the total number of generated branches, the parent node can be deallocated. This tally is kept in the solver, and can be verified by the coordination layer.

**more work?** After receiving this command the solver reports to the coordination layer whether the sets of nodes that are pending branching and pending propagation are empty or not. In the former case, the traversal of the search tree has finished. In the latter case, the coordination layer can make the solver continue the search by issuing more **schedule branching** and **schedule propagation** commands.

The following command sequence coordinates a basic branch-and-propagate tree search by a stand-alone solver, where propagation runs in a single node, the scheduler of DRFs runs to completion, and internal nodes are split immediately.

```
specifier
schedule propagation
propagation
evaluate
termination
schedule splitting
forget world
more work?
quit
```

repeated until the solver answers negative to the **more work?** question, or the coordination layer decides to break out of the loop, for example if the solver responds to the **termination** command with a notification that it has accepted a solution.

### 3.3.2 Interaction with the Environment

The **specifier** command, which we discussed at the beginning of this section, falls in the category of commands for regulating the interaction of an OpenSolver instance and its environment. The configuration specifications that it passes to
the solver typically come from a file, or have been submitted by another program such as a calculator front-end. The coordination layer plug-in knows where to get these specifications, which varies for different situations where OpenSolver is used as a component solver.

In addition to the specifier command, and some commands for generating textual representations of solutions, and nodes of the search tree in general, the following commands are of importance for the way OpenSolver is used as a component solver in the rest of this thesis.

flush. The solver supplies to the coordination layer plug-in a configuration for each of the nodes of the search tree stored in the sets of Figure 3.3.

clear WDB. The solver empties the world database, i.e., it forgets what it was doing, and enters the initial state.

A node is essentially defined by the domains of the variables, and the reduction operators that are active. Like all plug-ins, variables and reduction operators can generate a textual representation of themselves, in the language of Figure 3.2, by which they can be re-created in another solver. This facility is used by the flush command. The flush command is typically followed by clear WDB.

When a new variable is introduced, OpenSolver will ask the coordination layer plug-in if the variable is meant to be exported or not. In a distributed constraint propagation algorithm, exported variables are those that appear in two or more solvers. Changes to the domains of such variables must be communicated. This is initiated by the following command.

pending sends. The solver responds to this command with the identifiers of the variables that have been marked for export, and whose domains have been modified since the last pending sends command was issued.

The following command takes care of the actual communication:

export. The solver supplies a pointer to the data structure that represents the domain of a given variable, in a given node of the search tree.

The pending sends and export commands could be combined for the purpose of distributed constraint propagation, but they have been left separated for the use of exported variables in the operator for nested search, discussed in Chapter 7. After the solver has responded to an export command, the coordination layer can actually send the modified domain to another solver. This message has to be tagged with the node of the search tree, to which the information applies. Because of the world database, this tag need only contain the two integers that uniquely identify a node during its lifetime.

The default mechanism for communicating domain updates is the text-based representation that is used for the flush command. This has the benefit of being
machine independent. For example, we do not have to worry if integers are stored as big-endians or little-endians. To avoid the overhead of string manipulation, a coordination layer plug-in can make assumptions about the variable domain type plug-ins that are used. For example, it can rely on the assumption that all domains that it will ever export, will be able to write themselves to a binary representation. This may of course give rise to problems in a heterogeneous computing environment, but when the environment is homogeneous with respect to, say, integer representation, the string manipulation can easily be avoided. We have not experimented with this, but in a shared memory environment, it may even be feasible to use pointers as representations of the domains, through which they can then be accessed directly.

When receiving an incoming domain update, the coordination layer reconstructs the variable domain. Using the default mechanism, this involves interpreting an identifier-specifier pair. The resulting domain is then passed to the solver via the following command:

**update.** The solver computes the intersection of the domain of a variable in a particular node of the search tree, and a domain supplied by the coordination layer. The solver assumes that the domains are instances of the same plug-in. If the intersection is smaller than the original domain, the intersection is used as the new domain for the variable. The change is made known to the scheduler of reduction operators for the propagation phase, and the variable is marked as changed for the purpose of the **pending sends** command.

We conclude the description of the coordination layer by clarifying its name.

### 3.3.3 Coordination

In computer science, **coordination** refers to the orchestration of the interaction among the various active entities involved in a software system [Arb98]. As Gernter and Carriero observed, coordination is a ubiquitous aspect of computing: even the simplest programs interact with their users to exchange input and output [GC92]. Furthermore, on the level of programming languages, the sequential execution of program statements can be seen as a particular form of coordination of computing entities.

While coordination can be recognized in all software systems, it is of particular relevance for **concurrent systems**, where several interdependent computations overlap in time. Examples of such systems are

- **parallel** systems, where concurrency is introduced for reducing the turn-around time of a computation by distributing the workload over several hardware processors, and
Chapter 3. OpenSolver: a Software Component

- **distributed** systems, whose state is distributed over several computing entities that execute concurrently.

**Coordination languages** are languages for programming the interaction between the active entities of concurrent systems. Such languages are based on a particular **coordination model**, i.e., a set of assumptions on how these entities interact. Examples of coordination languages, models, and architectures are:

- Linda [CG89], a coordination language based on a coordination model where processes communicate by injecting and consuming tuples from a shared tuple space.

- The Manifold [Arb] language, which implements the Idealized Worker Idealized Manager model, and depends on channel-based communication between processes (see also Chapter 9).

- The Discrete Time TOOLBUS [BK98], a coordination architecture for the integration of software components (tools). The integration is specified through a script that describes all possible interactions between the tools, and the coordination model supports explicit specification of the timing behavior of systems.

- Reo [Arb02], a channel-based coordination model, wherein complex coordinators, called connectors, are compositionally built out of simpler ones (see also Chapter 8).

Sometimes a distinction is made between **endogenous** and **exogenous** coordination. The former means that coordination is realized through operations **within** the entity that is being coordinated. The OpenSolver command loop, described in Section 3.3.1, is an example of endogenous coordination. Conversely, exogenous coordination is coordination from **without**: the constructions that regulate the interaction are outside the interacting entities, and the computation code is separated from the coordination code. In this classification, Linda is an endogenous coordination language, and Manifold, TOOLBUS, and Reo are based on exogenous coordination models.

Although the field originated in the area of parallel and distributed computing, coordination now also manifests itself as an approach to component-based software engineering. From this point of view, a software component is an autonomous system with well defined behavior that has its own thread of control. Systems that are built from such components will be concurrent systems, and their composition requires a form of exogenous coordination. TOOLBUS and Reo are designed specifically for component-based software engineering.
3.4 Implementation

Coordination and the Coordination Layer

We already pointed out that the OpenSolver command loop can be seen as a form of endogenous coordination, but the term “coordination layer” is primarily meant to emphasize that this software layer makes OpenSolver amenable for various forms of exogenous coordination.

3.4 Implementation

OpenSolver is an object-oriented application with an extendable class hierarchy. Defining the basic structure of a branch-and-propagate constraint solver, OpenSolver can be categorized as an object-oriented framework [Deu83, JF88]. It employs the typical inverse control mechanism: in principle, OpenSolver calls member functions on objects of the classes that configure it, not the other way around. This mechanism is also known as the Hollywood principle: “Don’t call us, we’ll call you.” [DGS03]

The classes that implement the problem-specific aspects of the solving process are called “plug-ins” for two reasons:

- to avoid using the word *component*, emphasizing that OpenSolver has not been implemented using any form of component-based software engineering, and
- to emphasize that in addition to being an object-oriented framework, OpenSolver is also a stand-alone application that can be configured for different tasks and environments.

For programming new plug-ins, OpenSolver can further be categorized as a white-box framework: the programmer needs to understand the implementation of the framework in order to be able to program for it. For some categories of plug-ins, this is limited to understanding the class interface, but especially for programming a coordination layer plug-in, it must be known how OpenSolver reacts to the various commands that can be issued.

As an example of an extendable class in OpenSolver, Program 3.2 shows part of the definition of the abstract class for reduction operator plug-ins.

- Member functions *Compute* and *Termination* are the propagation function and termination function, respectively. Their *arguments* parameter passes an array of pointers, pointing to the domains of the variables that the reduction operator applies to.

- Member functions *Mask* and *Activate* implement protocols for communicating changes to variable domains, as discussed at the end of Section 3.2.2. They also define an operator’s input scheme.
• Member functions \texttt{CanReduce} and \texttt{CanSplit} characterize an operator as a propagation operator, branching operator, or optimization operator, and define its output scheme.

• The member function \texttt{Idempotent} is used to tell a scheduler plug-in that the propagation function is idempotent. Recall that a function \( f \) is called idempotent if \( f(f(x)) = f(x) \), for all \( x \) in the domain of \( f \). If an idempotent DRF modifies the domain of a variable in its input scheme, this DRF need not be scheduled again.

\begin{verbatim}
class ReductionOperator : public OpenSolverPlugIn
{
public:
    static ReductionOperator *Factory(
        CoordinationLayer *coordinationLayer,
        const char *type,
        const char *spec);
    virtual ~ReductionOperator() ;
    virtual const std::vector<char*> &VariableNames( ) const =0;
    virtual int Compute( Domain **arguments, unsigned int *changes,
        bool *deactivate );
    virtual int Termination( TDInfo *tdInfo, Domain **arguments,
        Annotation *annotation, bool *deactivate );
    virtual bool CanSplit( int i ); // i==-1: do you want to be called
        // on termination?
        // i>=0: can you split argument i?
    virtual bool CanReduce( int i ); // i==-1: do you want to be called
        // during propagation?
        // i>=0: can you reduce argument i?
    virtual unsigned int Mask( int i );
        // mask for local_changes[i]
    virtual bool Activate( int i, unsigned int c );
        // If Mask[i] does not match,
        // do you want to be activated if
        // variable i has been modified
        // such that local_changes[i] == c?
    virtual bool Idempotent() const;
};
\end{verbatim}

Program 3.2: Part of the definition of the abstract class for reduction operators

While as an open-ended, extendable system, OpenSolver is a white-box frame-
work, it aims at providing a black-box framework for composing branch-and-propagate constraint solvers. Plug-ins can be combined with minimal knowledge of the implementation of the system and the plug-ins. The intention is to establish a set of atomic plug-ins, from which many different solvers can be composed. New plug-ins are added only when the desired functionality cannot be realized by composition of the readily available facilities, or when this cannot be done efficiently. OpenSolver is approached as a black-box framework through the configuration language of Figure 3.2.

Plug-in System

The identifiers for Plug-ins that are available in an OpenSolver installation can be used in the configuration language of Figure 3.2. This is realized by a static member function per plug-in category that contains a large conditional statement to couple the identifiers to the constructors of the classes that implement the plug-ins. This is the first member function in Program 3.2. For example, the code on page 40 would pass the identifier RealInterval and the specifier following it to this function. If a plug-in class for RealInterval is available, an object of this class is created by calling the constructor for this class, passing the specifier string \([-0.1, 0.1]\) as an argument.

Currently, adding a plug-in to an OpenSolver installation involves modifying the actual C++ code for the static member function, by adding a branch to the conditional statement. For example, for RealInterval, the following lines were added:

```c++
#include "RealInterval.h"
...
else if ( !strcmp( type, "RealInterval" ) )
    res = new RealInterval( spec, &syntax_error );
```

Also, the makefile must be modified in order that the new class is compiled and linked, and the application must be rebuilt to take these modification into account. All of these tasks can easily be automated, and a plug-in system based on a source code distribution is straightforward.

Ideally we would not have to relink, or modify code for adding plug-ins. This can be realized by dynamic linking of the object code for plug-ins. A potential problem here is that current C++ compilers do not adhere to a single standard for encoding class member names (name mangling). There are ways around this problem, for example we could provide a C interface. An interesting option is to implement this interface by a general-purpose class in each of the plug-in categories, whose only purpose is to take on board C code that has been compiled separately.
Chapter 3. OpenSolver: a Software Component

Figure 3.4: Category-dependencies, or assumptions, implemented by type casting. Arrows are drawn from the class that makes the assumption to the class whose objects are being cast.

Type Casting

In Section 3.2.2 we already mentioned that it is very common for a plug-in to make assumptions about the plug-ins with which it works in conjunction. The OpenSolver framework accesses all plug-in through the abstract classes for their categories. From this point of view, every variable domain is an object of class `Domain`. However, a reduction operator for finite domains will assume that the variable domains to which it is applied are actually instances of a subclass of `Domain` that provides specific operations, such as deleting integer values. Such assumptions are implemented by *type casting*, and the interaction between reduction operators and variable domains is just one of many situations where this happens. Figure 3.4 gives an overview of all cases were plug-ins make assumptions about other plug-ins.

It is the responsibility of the user or, more likely, the software that generates a solver configuration in the language of Figure 3.2, to ensure that the correct plug-ins are used. Violating an assumption leads to undefined behavior. It would be easy to implement a type checking mechanism that gives a proper run-time error in case incompatible plug-ins are used. Because checking the types is a potentially costly operation, we could limit this check to the root node of the search tree.

Lines of Code

To give an idea of the amount of code that is involved, OpenSolver itself, without any plug-ins consists of roughly 5,000 lines of C++ code. The plug-ins used for the experiments in this thesis comprise some 17,500 lines, including a few hundreds of lines of lex and yacc specifications, and 3,500 lines of peripheral programs and scripts were used.
3.4.1 Software Composition

It is tempting, but wrong to characterize the composition of branch-and-propagate constraint solvers, as supported by OpenSolver, as component-based software engineering. This would suggest that we aim for reuse of the plug-ins, while instead, we aim for reuse of the design of the solver. This is the case for object-oriented frameworks in general. In the first place, most plug-ins are of an inherent simplicity that would not make them valuable components. The challenge in designing a configurable constraint solver is in the definition of the interfaces, rather than in programming, for example, atomic arithmetic constraints. Secondly, instead of incorporating the more complex operators as third party components, OpenSolver promotes the composition of such operators from basic facilities, if possible. In the next chapters we will see several examples of this. Apart from this, external code can likely always be taken on board by wrapping it up as a plug-in. In the case of an autonomous application, OpenSolver could communicate with this application through a plug-in that acts as a proxy, although we have made no particular provisions for this form of software composition.

In contrast, OpenSolver itself forms a versatile software component. Through the coordination layer plug-in it can be adapted to various computing environments. Because it is a stand-alone application instead of a library, it poses no restrictions on the programming languages used in these environments. In addition, the configuration language allows for external manipulation both of solvers and CSPs. We will see examples of the use of OpenSolver as a software component in Chapters 7, 8, and 9.

We are convinced that modules or classes are the right units of composition for realizing a branch-and-propagate constraint solver. As we will argue in Chapter 9, using coordination languages here involves too much overhead. Modules implementing abstract data types might allow for an easier implementation of a plug-in system, because they avoid the problem with name mangling that we discussed on page 59. They may also be a bit more efficient because some overhead is involved with calling virtual functions. However, we used C++ classes because of the language support for defining interfaces. Although we did not perform experiments, we expect that with modern compilers, the overhead for calling virtual functions is limited, and the implementation of a plug-in system based on dynamic loading was not a priority in our experimental setup. However, for a commercial system, C modules, with some rigorous scheme to enforce compliance with interfaces for the different categories of components, may be preferable.

3.4.2 Comparison with Other Systems

Several other object-oriented approaches to constraint programming exist. Most of them have extendable class hierarchies with inverse control, and can therefore be classified as object-oriented frameworks as well. To put our work into context,
we compare OpenSolver to the following systems.

**ILOG Solver.** A commercially available C++ library for constraint programming. Some of its features are support for integer, floating-point, and set variables, an extensive collection of built-in constraints, and support for both tree search and local search [Ilo01]. The class hierarchies for constraints and search procedures can be extended. ILOG Solver is part of the ILOG Optimization suite, in which it can cooperate with the CPLEX mathematical programming engine.

**Koalog Constraint Solver (KCS).** A commercially available JAVA library for constraint solving on Boolean, integer and set domains, supporting tree search and local search [KoaA, KoaB]. The library can be extended with new constraints, search strategies, and solvers.

**Elisa.** A C++ library that offers a framework for integrating solvers in applications [El04]. The facilities offered by the 1.0.3 version of the system are focused on solving constraint on the reals through branch-and-propagate tree search, and include a large collection of consistency algorithms for such constraints. In addition, the Elisa class hierarchy can be extended with respect to many aspects of constraint solving, including domain types, constraints, reduction operators, and local search. Elisa is distributed under the GNU Lesser General Public License.

**Disolver.** A C++ library that offers a constraint-based optimization engine, with support for parallel and distributed (DisCSP) search [Ham05]. New constraints, and tree search and local search procedures can be defined by users. Disolver is reported to have been used for solving large industrial problems.

**Figaro.** A C++ library for finite domains constraint solving [HMN99]. To our knowledge, Figaro is the only system that is configurable with respect to state restoration policy (see Section 4.2).

**EasyLocal++.** A C++ library that provides a framework for realizing local search algorithms [DGS03].

**Localizer++.** A C++ library for local search [MVH01]. It is intended to make the facilities that are usually found in modeling languages (such as Localizer [MVH00]) available inside a mainstream programming language, to facilitate the integration in larger software projects. The library is extendable with respect to constraints, invariants (a modeling tool), and search strategies.

Despite the extendable class hierarchies. ILOG Solver, KCS, Disolver, and Localizer++ are primarily object-oriented toolkits, meaning that some specific
3.4. Implementation

functionality (in this case, constraint solving) is made available to the developers who use the toolkit. Instead of a constraint solving toolkit, OpenSolver is a configurable branch-and-propagate tree search engine, implementing only the core algorithms of such toolkits. As such, the aspects of constraint solving that are configurable in OpenSolver are at a lower level than those that are configurable in toolkits. For example, new constraints can be added to the toolkits, while the notion of a constraint does not exist in OpenSolver: it only knows about reduction operators. Also the toolkits typically provide high-level modeling facilities such as arrays. In our case, these would be provided by a modeling environment that uses OpenSolver as its solving engine.

Being a configurable branch-and-propagate tree search engine, OpenSolver gives control over aspects of constraint solving that have hard-wired solutions in toolkits. Especially, toolkits typically fix the implementation of the domain types and the constraint propagation algorithms. This makes a constraint solver that uses OpenSolver as its solving engine a much more flexible system. Because it aims at black-box composition of constraint solvers through a configuration language, altering some aspects of the solving engine of an OpenSolver-based system should only rarely involve C++ programming. Instead, such a system could offer a menu of available options, which the user can combine at will. In this respect, OpenSolver is quite similar to the Elisa library, which is also highly extendable and configurable. To reflect that the system is a white-box framework that aims at black-box composition of solvers, the Elisa distribution contains both a programmer’s manual and a user’s manual.

Figaro is configurable on one very low-level aspect of a constraint solving (tree search) algorithm, namely the state restoration policy [CHN01]. In OpenSolver, the state restoration policy is fixed (see Section 4.2), and in retrospect, this limits its applicability for search tasks that rely on a specific, and different policy (see Section 4.4). However, the state restoration policy is closely related to the variable domain type implementation, and making both aspects configurable is not straightforward, and the impact should carefully be assessed.

Finally, the approach of EasyLocal++ is also similar to ours, in the sense that it is a framework, aiming at reuse of the basic solving algorithm structures. Apart from implementing a different solving algorithm (local search instead of tree search), EasyLocal++ is a white-box framework, where composition of a local search solver always involves C++ programming.

While similarities exist with each of these systems, OpenSolver is unique in providing a highly configurable and versatile solving engine, that is also an autonomous application. Aiming for black-box composition has led to an inherently linguistic approach, where every plug-in needs to be able to interpret and generate textual specifiers. This gives possibilities for external manipulation of CSPs and solver configurations that we have not found in any other system. An API would be a valuable extension, and we already use a rudimentary form of such an interface in Chapter 7, but this would not change the autonomous nature of
the system. Building a system around OpenSolver is primarily a matter of exogenous coordination. We will see examples, and discuss further possibilities of this approach in Chapters 8 and 9.

3.5 Summary

In this chapter we introduced OpenSolver, an open-ended constraint solver that is based on branch-and-propagate search. We discussed it from two perspectives:

Composing constraint solvers. This is a matter of black-box composition, by selecting a combination of plug-ins in seven categories that correspond directly or indirectly to the aspects of branch-and-propagate search identified in the summary of the previous chapter. OpenSolver is an autonomous application. By means of a plug-in in a special, eighth, category for its coordination layer, it can be adopted to different software environments.

Implementation. OpenSolver is an object-oriented framework: it provides only basic mechanisms and data structures. The plug-ins, which are implemented as subclasses of abstract classes of the framework, determine the functionality. Because the coordination layer plug-in has tight control over the solving process, this facilitates several forms of exogenous coordination, allowing that OpenSolver is used as a software component for composing constraint solvers beyond a sequential branch-and-propagate tree search.

If we return to the concluding remarks at the end of the previous chapter, we now have an implementation of the model of constraint solving described there, with some additional features such as the coordination layer. Our next goal is to evaluate this implementation, from both the perspective of efficiency and the perspective of composing constraint solvers, on a number of standard and more advanced constraint solving techniques. This is done in the following six chapters, where Chapter 7 is a turning point, because there we move from composing constraint solvers within the OpenSolver framework to composing constraint solvers from several OpenSolver instances. At that point, our attention shifts from the seven categories of constraint solving plug-ins to the coordination layer mechanism.