The Journal of Functional and Logic Programming

The MIT Press

Volume 1999, Article 8
28 April, 1999

ISSN 1080–5230. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493, USA; (617)253-2889; *journals-orders@mit.edu, journals-info @mit.edu*. Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://www.cs.tu-berlin.de/journal/jflp/*
- *http://mitpress.mit.edu/JFLP/*
- *gopher.mit.edu*
- *ftp://mitpress.mit.edu/pub/JFLP*

The Journal of Functional and Logic Programming is a peer-reviewed and electronically published scholarly journal that covers a broad scope of topics from functional and logic programming. In particular, it focuses on the integration of the functional and the logic paradigms as well as their common foundations.

Embedding Propagators in a Concurrent Constraint Language

Tobias Müller
Jörg Würtz

Abstract

Solving large and hard discrete combinatorial problems often requires the design of new constraints. Current constraint systems focus on either high-level modeling or efficient implementation technology. While each approach lacks the advantages of the other one, this paper describes the combination of them in the high-level concurrent constraint language Oz. We describe an interface to Oz providing abstractions to program new efficient constraints in C++, preserving the benefits of Oz for problem modeling.

While constraints and the Oz runtime system are linked through the interface, and adequate interface abstractions are supplied to implement advanced algorithmic techniques. In particular, it provides the means to reflect the validity of a constraint and to control and inspect the state of a constraint. This allows the user to solve demanding combinatorial problems, such as hard scheduling problems.

It is desirable to execute concurrent constraint programs in parallel to profit from multiprocessor architectures. We discuss how the proposed interface can be adapted to parallel execution, avoiding the recoding of constraint implementations for sequential solvers.

1 Introduction

Recently several approaches and systems have been suggested to solve discrete combinatorial problems with finite-domain constraints (e.g., [11, 39, 7, 4, 12, 17]). To solve large and hard combinatorial problems, it is often necessary to program new customized constraints and search strategies. Several approaches have been suggested in the literature.

A constraint logic programming (CLP) language like *ECL^{i}PS^e*(see [12]) is well suited to modeling constraint problems on a high level. It provides certain primitives like attributed variables to design new constraints. But many techniques to solve hard problems require destructive low-level operations, which are difficult to program efficiently in this setting.

The indexical approach (see [39, 7]) allows the user to program some new constraints. But it has no support to apply more sophisticated algorithmic techniques to implement new constraints (see also Section 8 and [28]) as, for example, required for global scheduling constraints employing edge-finding (refer to [2, 1]).

On the other hand, combinatorial problems can be tackled in a language like C++ together with a dedicated library for constraint solving (see for example, Ilog [17]). Although many programming abstractions are provided through C++ classes, it is hard for a C++ library to provide an adequate level of abstraction to program the constraint model intended to solve the actual problem.

Each approach lacks the advantages of the others. In this paper we describe the combination of these approaches by interfacing the high-level language Oz (see [34]) with an interface to program new constraints efficiently in C++. This results in the joined benefits of the combined approaches.

Oz is a concurrent constraint programming (CCP) language (see [29]), which comes with an extensive predefined constraint library. By the concepts of a constraint store and entailment, new constraints can be programmed in the language itself. Furthermore, Oz provides a means to program new search strategies (refer to [30, 31]). The inherent concurrency of the language makes it a good candidate for a parallel implementation (see [26]). Hence, this paper goes beyond a previous version [24] by proposing in Section 7 an extension of the interface to take advantage of multiprocessor platforms.

Typically, the first step to solving a difficult problem is a prototypical implementation. In the case of Oz, rapid prototyping is supported by the features of a high-level language. If the implementation does not show the expected performance, it is desirable to identify the performance-critical parts of the program that are not covered by predefined library constraints. (Oz provides a profiler for that purpose that is compatible with the proposed interface.) Then these parts should be recasted in a very efficient implementation. To this aim, an interface is provided that supports adequate abstractions to implement new constraints in C++ and ensures, by employing C++'s concept of an abstract base class, that all required definitions are provided. Thus, the application programmer can benefit from both a high-level language and an efficient C++ implementation.

A constraint is realized by a propagator. A propagator's implementation is done through an interface. The interface itself is called the constraint propagator interface (CPI) of Oz. Propagators and the Oz runtime system are linked together only by the interface abstractions. The programmer is freed from tedious tasks like suspending or resuming propagators. Furthermore, the CPI provides abstractions to hide (respectively, handle) specific features of Oz like computation spaces and equality constraints. That helps the programmer to concentrate on propagation techniques rather than on (in this context) irrelevant issues. The extension of the CPI to a parallel execution model follows this high-level approach and hides from the user, for example, the necessary functionality to deal with certain locking schemes.

The interface design can also be used for Prolog-based implementations providing for coroutining. Moreover, it can be extended by further constraint systems, as already done for finite set constraints (see [22]) or to support the implementation of new constraint systems from scratch (see [15]).

We emphasise that to prove the practicability of our approach, Oz's whole finite-domain constraint library is implemented using the CPI. The resulting constraint solver shows performance competitive to state-of-the-art finite-domain systems. It shows also competitive expressiveness and employs non-trivial algorithms for scheduling applications (refer to [42]).

The following section introduces the computation model of Oz followed by the introduction of the CPI abstractions. Section 4 explains the implementation of a propagator. The advanced expressiveness of the CPI in conjunction with a case study is discussed in Sections 5 and 6. Section 7 analyzes the extension of the CPI to exploit parallelism provided by multiprocessor hardware. The paper closes with related work, a performance evaluation, and a conclusion.

2 Computation with constraints in Oz

As a particular instance of a CCP language, we consider Oz (see [34, 32, 33]). Further, the paper focuses on *finite-domain constraints* over nonnegative integers (see [39, 41, 32, 43, 33] for details).

A distinction is made between basic and nonbasic constraints. For the purpose of this paper, a *basic constraint* takes the form $x = n$, $x = y$, or $x \in D$, where x and y are variables, n is an integer, and D is a finite domain.

A constraint $x \in D$ is called a *domain constraint*. We say a variable x is *determined* if the store entails a constraint $x = n$. Basic constraints reside in the *constraint store* C. Efficient algorithms to decide satisfiability and entailment are provided for basic constraints.

For more expressive constraints, like $x + y = z$, deciding satisfiability is not efficiently computationally feasible. Such *nonbasic* constraints are realized as *propagators*. A propagator is a computational agent that tries to narrow the domains of variables by adding appropriate basic constraints to the store. The term *constraint propagation* refers to advancing the constraint store in this way. A propagator imposing the constraint P advances the store C to the store $C \wedge B$ if $C \wedge P$ entails B and B adds new and consistent information to C. The variables a propagator is narrowing are called its *parameters*.

The implementation of a propagator defines the amount of constraint propagation, that is, its operational semantics. Often a complete propagator that imposes the strongest basic constraint entailed by $C \wedge P$ is computationally too expensive. Thus, weaker propagation is usually employed. For some application areas, domain-specific techniques can be exploited that lead to very good results (see also Section 9). A propagator ceases to exist either if P is entailed by C or if $C \wedge P$ is unsatisfiable.

As an example for constraint propagation, assume we have a store containing $x, y, z \in \{1, \ldots, 10\}$. The propagator for $x + y < z$ narrows the domains to $x, y \in \{1, \ldots, 8\}$ and $z \in \{3, \ldots, 10\}$ (since the other values cannot satisfy the constraint). Adding the constraint $z = 5$ causes the propagator to strengthen the store to $x, y \in \{1, \ldots, 3\}$ and $z = 5$. Imposing $x = 3$ lets the propagator narrow the domain of y to 1.

A *computation space* hosts a constraint store and a set of propagators. We first treat the case where only one computation space is given. The particularities of a hierarchy of computation spaces are described in the last paragraph of Section 3.7.

3 Extending Oz with propagators

The computational model sketched in Section 2 is realized by the Oz runtime system, which is implemented by an abstract machine (see [21]), called the *emulator*. In this section we explain the interface between the emulator and propagators. We introduce the provided CPI abstractions as consequence of the interaction between the emulator and propagators. In the following we also speak of propagators if we mean the actual implementation of the computational agents.

3.1 Overview

A propagator exists in different execution states, which are controlled by the emulator. Further, the emulator allocates resources like computation time and heap memory to a propagator. The emulator regards a propagator as an entity that requires resources (respectively, services). That allows separation of the emulator from the implementation of the propagators.

In turn, a propagator synchronises on the constraint store and may amplify it with basic constraints. The emulator *resumes* a propagator when the store has been amplified in a way for which the propagator is waiting. For example, many propagators are only resumed when the domain bounds of their parameters are narrowed. On resumption a propagator *reads* for its parameters the basic constraints that are contained in the store. In the course of constraint propagation it *writes* basic constraints to the store.

The CPI is a C++ interface and, consequently, provides abstractions as C++ classes. A propagator is implemented by an instance of a C++ class that stores, in its state, references to the propagator's parameters. Operationally, resuming a propagator means running its propagation method. Note that in the following, C++ identifiers that start with "`OZ_`" refer to CPI abstractions.

3.2 Handling a propagator

As mentioned above, the emulator regards a propagator as an opaque entity. Hence, the emulator needs a uniform way to refer to all instances of propagators. Further, the CPI must ensure that a programmer provides the minimal propagator functionality required by the emulator. The compiler should reject code that is incomplete in that sense. Technically, both requirements are realized in the interface by defining the class `OZ_Propagator` as an *abstract base class*, which is the ancestor class of all propagator classes. An abstract base class provides only the declaration (i.e., only the type signature) but not the definition for its virtual methods, which are indicated by "`=0`" after the argument list. Virtual methods allow for dynamic binding of methods. This enables the emulator to control any concrete instance of a propagator only by having a pointer of type `(OZ_Propagator*)` to it and thus completely separates propagators from the emulator.

```
enum OZ_Return {OZ_ENTAILED, OZ_FAILED, OZ_SLEEP};

class OZ_Propagator {
public:
   virtual OZ_Return propagate(void) = 0;
   virtual void updateHeapRefs(OZ_Boolean) = 0;
   ...
};
```

3.3 Imposing a propagator

Attaching a propagator instance to its parameters and introducing a reference to this instance to the emulator is called *propagator imposition*. This is done by a *header function*. Such a function is connected via the Oz foreign function C interface (see [20]) to an Oz procedure. A header function has to provide the following services.

1. A propagator is imposed as soon as its parameters are *sufficiently constrained*. For example, if a parameter that is expected to be an integer is not yet determined, the propagator should not be imposed yet. On the other hand, type errors should be detected by the header (e.g., a parameter is an atom instead of an integer).

2. It is determined on imposition what events cause a propagator to be resumed. A propagator can be resumed if a parameter is determined, the bounds of the parameter's domain are narrowed, the size of the domain is decreased, or a parameter is involved in a unification.

3. A reference to the newly created propagator instance has to be passed to the emulator.

The class `OZ_Expect` is provided for that purpose. It supplies a set of methods to test parameters to be sufficiently constrained. Further, they store the event (this is passed as extra argument to the test method) on which the propagator has to be resumed. Insufficiently constrained parameters cause the header function to be suspended such that it is resumed as soon as the parameters concerned are further constrained.

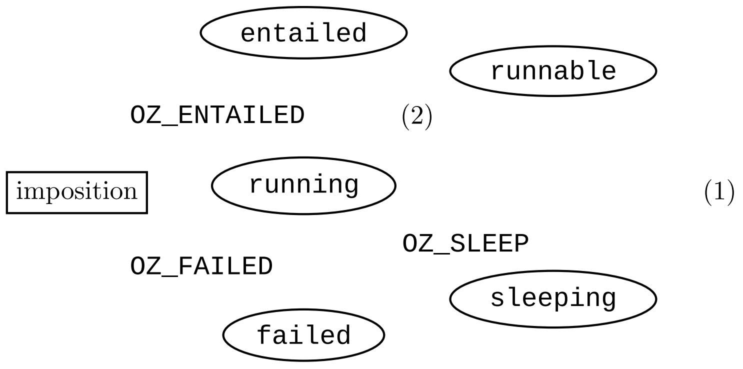

Figure 1: Execution states of a propagator

After creating a new instance of the propagator by invoking its constructor, a reference of type `(OZ_Propagator*)` is passed to `impose()` of class `OZ_Expect` to introduce the propagator to the emulator. As a side effect, `impose()` attaches *suspensions* to the appropriate *suspension lists* of the variable parameters. These parameters were previously stored in the state of the propagator by the test methods. The propagator is now *suspending* on its parameters and can be resumed if the parameters are further constrained. In fact, a variable can have several suspension lists such that the contained propagators are resumed on different events.

3.4 Scheduling a propagator by the emulator

In order to schedule propagators, the emulator maintains for each propagator an execution state that can take one of the following values: `running`, `runnable`, `sleeping`, `entailed`, and `failed`. The emulator's scheduler switches a propagator between the execution states as shown in Figure 1.

When a propagator is imposed, its execution state is immediately set `running` and the scheduler allocates a time slice for its first execution. After every execution, when the constraint propagation was performed by the appropriate propagation method, the emulator evaluates the propagator's return value.

The value `OZ_FAILED` is returned if the propagator (according to its operational semantics) detects its inconsistency with the store. The emulator sets the propagator's execution state to `failed` and the computation is aborted. The propagator is ignored by the emulator until it is eventually disposed of by the next garbage collection. An immediate disposal is not desirable, since there may be multiple references to a propagator.

The return value `OZ_ENTAILED` indicates that the propagator detects that the constraint it implements is entailed by the constraint store, that is, the propagator cannot further amplify the constraint store. The emulator sets the propagator's execution state to `entailed`. It happens the same as for a failed propagator: It is ignored until it is disposed of by garbage collection.

The propagator returns `OZ_SLEEP` if it can detect neither inconsistency nor entailment. Its execution state is set to `sleeping`.

A propagator is resumed if at least one of its variable parameters is involved in unification or its domain is further narrowed. The emulator scans the suspension lists of the concerned variables and either deletes entries where the propagator's execution state is `failed` (respectively, `entailed`) or switches the execution state of the suspending propagator to `runnable`. This is indicated by transition (1) in Figure 1. Now, the scheduler takes care of the propagator and schedules it later. (The transition (2) from `runnable` to `running` is subject to the scheduler's policy and is not discussed here.) In fact, when the scheduler switches a propagator to `runnable`, the propagator's method `propagate()` is executed.

3.5 Reading and writing constraints by a propagator

A propagator stores in its state references to its parameters. Constraint propagation in the implementation consists basically of the following stages: reading basic constraints of its parameters, writing further basic constraints to the store, and resuming propagators suspending on these parameters. An instance of the class `OZ_FDIntVar` provides access to a parameter's representation in the constraint store. On construction it obtains access to a parameter's suspension lists and the parameter's finite domain representation of class `OZ_FiniteDomain`. Further, it stores a profile of the finite-domain representation. A profile consists of the current domain size and the current width, that is, the difference between the largest and smallest element of the domain. Such a profile is used by the method `OZ_FDIntVar::leave()` to decide whether propagators suspending on this parameter have to be resumed or not. Instances of the class `OZ_FiniteDomain` provide methods to access and update the representation of the domain constraint of a parameter.

3.6 Memory management

A propagator `P` derived from `OZ_Propagator` has to define a method `P::updateHeapRefs()`, since `OZ_Propagator` declares this method as a pure virtual method. This method is called by the emulator's garbage collection routine and has to ensure that all references to the emulator's heap are updated that are reachable from the propagator's state. For example, to update a reference of the predefined type `OZ_Term`, the provided function `OZ_updateHeapTerm()` must be applied to it.

3.7 Handling hierarchical computation spaces

Concurrent constraint programming requires entailment checking rather than satisfiability testing as in the case of constraint logic programming (CLP). Oz employs computation spaces to check entailment by testing if the basic constraints of one computation space are subsumed by those of another. (Remember that a computation space consists of a constraint store and a set of propagators attached to it.) Through the course of computation a hierarchy of computation spaces (forming a tree) may arise, where constraints of a superordinated space (i.e., closer to the tree's root) are visible in all its subordinated spaces (i.e., more remote from the tree's root) but not the other way around. Thus, if a *global variable* (i.e., a variable that is declared in a superordinated space) is further constrained in a subordinated space S, the changes must be memorized such that they can be undone when S is left. If a *local variable* (i.e., a variable that is declared in the current space) is further constrained by a propagator, nothing needs to be memorized. The CPI handles these issues transparently to the CPI user. For example, if a variable is constrained by a propagator, the CPI causes the emulator to resume only those propagators that suspend in the current or subordinated spaces.

4 An example

This section exemplifies the constraint propagator interface by implementing a propagator for the constraint $x \leq y$. Its implementation requires the definition of a new class inheriting from `OZ_Propagator`.

```
class LessEq : public OZ_Propagator {
private:
```

```
1  OZ_Return LessEq::propagate(void) {
2    OZ_FDIntVar x(x_ref), y(y_ref);
3    OZ_FiniteDomain * x_dom=x.getDom(), * y_dom=y.getDom();
4    if (!x_dom->lowerUB(y_dom->getMaxElem())) goto failure;
5    if (!y_dom->raiseLB(x_dom->getMinElem())) goto failure;
6    if (x_dom->getMaxElem() <= y_dom->getMinElem()) {
7      x.leave(); y.leave(); return OZ_ENTAILED;
8    }
9    x.leave(); y.leave(); return OZ_SLEEP;
10 failure:
11   x.fail(); y.fail(); return OZ_FAILED;
12 }
```

Figure 2: Method `propagate()` for the constraint $x \leq y$

```
  OZ_Term x_ref, y_ref;
public:
  LessEq(OZ_Term x, OZ_Term y) : x_ref(x), y_ref(y) {}
  virtual OZ_Return propagate(void);
};
```

The propagator stores, in its state, references to its parameters (here `x_ref` and `y_ref`). A value of the predefined type `OZ_Term` refers to a parameter in the constraint store. The constructor of the class `LessEq` initialises the state and is used in the definition of the header function imposing the propagator (see at the end of this section).

4.1 The propagation method

When the emulator switches a propagator's state to `running`, the method `propagate()` of the propagator is executed (see Figure 2). This method implements the propagation algorithm of the propagator.

To obtain access to the propagator's parameters, the instances `x` and `y` of class `OZ_FDIntVar` are created. The function `OZ_FDIntVar::getDom()` returns a pointer to the representation of the domain constraint of the parameter (through its representation as an instance of `OZ_FDIntVar`). Therefore, `x_dom` and `y_dom` refer to the finite-domain constraint representations of the respective parameters (line 3 of Figure 2).

The propagation algorithm for $x \leq y$ is straightforward. The upper bound of x's domain is constrained to be less than or equal to the upper bound of y's domain (line 4) and the lower bound of y's domain is constrained to be greater than or equal to the lower bound of x's domain (line 5). The method `lowerUB(i)` makes the upper bound of the domain less than or equal to `i` (line 4), and the method execution `raiseLB(i)` makes the lower bound of the domain greater than or equal to `i` (line 5). Both functions return the size of the resulting domain. The method `getMinElem()` (respectively, `getMaxElem()`) returns the smallest (respectively, largest) value of the domain (lines 4, 5). In the case when an empty domain is produced, the execution branches to label `failure`.

The propagator cannot further amplify the store if the upper bound of x's domain is less than or equal to the lower bound of y's domain (lines 6–8), that is, $x \leq y$ is entailed by the store. The returned value `OZ_ENTAILED` signals the emulator that the propagator can be discarded. Otherwise, returning `OZ_SLEEP` keeps the propagator suspending on its parameters. The method `OZ_FDIntVar::leave()` indicates for the emulator which suspending propagators should be resumed because of the occurred propagation.

The method `OZ_FDIntVar::fail()` has to be called to perform some cleanups if the propagator is left because of a detected empty domain. The returned value `OZ_FAILED` signals the emulator that the current computation space is inconsistent.

4.2 Imposing the propagator

The header function to impose the $x \leq y$ propagator defines an instance of the class `OZ_Expect`. The following macro applications apply the test method `OZ_Expect::expectIntVarBounds()` to the first and second parameters, which cause the propagator to be imposed not before the parameters are constrained to finite domains. Additionally, it is determined that narrowing the bounds of domains resumes the propagator. A new instance of the propagator is created by calling the constructor with the first and second parameters. The application of method `impose()` keeps the propagator suspending on its parameters and introduces the propagator to the emulator.

```
OZ_C_proc_begin(lesseq, 2) {
  OZ_Expect pe;
  OZ_EXPECT(pe, 0, expectIntVarBounds);
  OZ_EXPECT(pe, 1, expectIntVarBounds);
  return pe.impose(new LessEq(OZ_args[0], OZ_args[1]));
} OZ_C_proc_end
```

5 Additional expressiveness of the CPI

This section explains the extended expressiveness of the CPI, which is desired to implement advanced propagators for demanding applications. All the discussed extensions are supported by adequate CPI abstractions that fit smoothly in the setting presented before (see [23] for details).

5.1 Taking variable equality into account

Oz provides equality between variables, that is, $x = y$, as a basic constraint. The CPI deals with equality in two ways:

1. The CPI provides abstractions to check which parameters of a propagator are equal (see Section 6 for details). This can be applied to detecting an inconsistency before variables are determined, as for the *alldiff*-constraint, which imposes the constraint that n variables must be pairwise different.

2. Different instances of `OZ_FDIntVar` associated with parameters that are equal refer to the same basic constraint. Therefore, updates to such a basic constraint are already visible via all other parameters during the propagation and such before the propagator is left and the constraints are written to the store.

To avoid superficial equality treatment, a propagator can check if an equality constraint was imposed on its parameters since the propagator's last run.

5.2 Exploiting statefulness

Oz saves complete computation spaces during a search, including changes to variables and propagators. That allows destructive modifications to a propagator's state, since if an inconsistency occurs, the propagator can be fully recovered. This feature can be used to detect what parameter has been changed since the most recent execution of the propagation method, by storing a profile (see Section 3) of the parameters before the propagation method is left. This allows the implementation of consistency algorithms like AC-5 (see [38]). Further, it can be used to store intermediate propagation results in the state that are expensive to recompute on each execution of the propagation method.

5.3 Replacing or imposing propagators while propagating

In the course of propagation, a propagator may detect that it can replace itself by a more efficient one. For example, suppose $x = y$ is added to a store of a computation space where the constraint $x + y = z$ belongs. It is more efficient to replace $x + y = z$ by $2x = z$ than to take care of equality every time propagation is done for $x + y = z$.

In scheduling applications, a propagator for a specialized scheduling constraint may deduce orderings between tasks in the course of propagation. To maintain these task orderings, propagators for constraints like $Start_{T_1} + Duration_{T_1} < Start_{T_2}$ can be imposed by the scheduling propagator with the side effect that the scheduling propagator need not care for these orderings anymore.

5.4 Encapsulated propagation

Typically, propagators write the result of constraint propagation to the store. An instance of the class `OZ_FDIntVar` allows the user, therefore, to update the basic finite-domain constraint of its associated parameter, such that the changes become visible to the store. But, for example, propagators for reified (respectively, meta-) constraints (see [14]) reflect only the validity of a constraint via a 0/1-variable to the store and not the actual result of propagation. Hence, the result of propagation is encapsulated in the propagator (i.e., not visible to the store) and only used to decide the validity of the constraint (for instance, by comparing the basic constraints in the store with the result of propagation). The CPI supports encapsulated propagation by the method `OZ_FDIntVar::readEncap()`, so that reified constraints can be straightforwardly implemented in conjunction with propagator replacement.

5.5 Attaching a propagator with a stream

The abstraction `OZ_Stream` of the CPI allows the user to attach a propagator with a stream such that the propagator is able to read and write the stream. That enables communication between a propagator and other program parts independent from finite-domain constraint propagation. This feature allows branching strategies to be guided by propagators. The propagator may suggest to an Oz procedure an ordering for tasks to be scheduled by using the shared stream. The Oz procedure may take the branching suggestion into account and may communicate the actual branching decision, back via the stream, to the propagator. The propagator in turn can use this information for the next ordering suggestion. Furthermore, it is possible to add extra parameters after a propagator is imposed to allow for propagators with dynamically increasing arity.

The discussed features have been applied to solve hard scheduling problems competitive to the state of the art (see [42, 43]). The CPI enables the implementation of so-called global constraints. It is further possible to suspend the imposition of a propagator until the store contains a certain required basic constraint, which is desired to implement, for example, an autonomous solver encapsulated in a propagator. The CPI allows the implementor of a propagator to determine the degree of propagation, for example, domain consistency for a certain constraint.

6 A case study

This section outlines the implementation of a more advanced propagator using some of the previously discussed extensions. The example used is the constraint

$$\sum_{i=1}^{n} a_i x_i + c \leq 0. \tag{1}$$

Along this example, it is shown how the state of a propagator is used to avoid redundant computation, how constraints of arbitrary arity can be handled, and how equality between variables can be used.

6.1 Propagation rules

We assume for the presented formulas that for a given real number n, $\lfloor n \rfloor$ ($\lceil n \rceil$) denotes the largest (smallest) integer that is equal to or smaller (larger) than n. Further, the current lower (respectively, upper) bound of the domain of a variable x is denoted by $\underline{x}$ (respectively, $\overline{x}$). Resolving the inequation (1) for $a_k x_k$ yields

$$a_k x_k \leq - \sum_{i=1, i \neq k}^{n} a_i x_i - c.$$

The upper bound of the right hand side of this inequation is

$$up_k = \quad - \sum_{i=1, i\neq k, a_i>0}^{n} a_i \underline{x}_i \quad - \sum_{i=1, i\neq k, a_i<0}^{n} a_i \overline{x}_i \quad - \quad c.$$

For every k, the variable x_k is narrowed as follows until a fixed point is reached:

$$x_k \leq \left\lfloor \frac{\mathrm{up}_k}{a_k} \right\rfloor \quad \text{if}\, a_k > 0 \qquad \text{and} \qquad x_k \geq \left\lceil \frac{\mathrm{up}_k}{a_k} \right\rceil \quad \text{if}\, a_k < 0.$$

This propagator ceases to exist if the following inequation holds:

$$\sum_{i=1, a_i>0}^{n} a_i \overline{x}_i + \sum_{i=1, a_i<0}^{n} a_i \underline{x}_i \; + c \;\leq\; 0.$$

6.2 Handling vectors

The CPI provides adequate abstractions to convert data structures of Oz (like lists) into C++ data structures. In Oz, a list, a tuple, or a record is denoted as a *vector*. To allow for propagators with an arbitrary number of parameters, parameters can be structured by vectors. The inequality propagator (1) has three parameters, that is, the first parameter contains the coefficients, the second one the variables, and the third one the constant.

The vectors of coefficients and finite-domain variables are converted to C++ arrays of integers and elements, respectively, of type `OZ_Term`. The class for the propagator implementing inequality constraints stores arrays for the coefficients a_i and the variables x_i, the constant c and the current size of the arrays.

```
class GenLessEqProp : public OZ_Propagator {
  int arr_sz, c, * a;
  OZ_Term * x;
public: ...
};
```

The header checks whether the arrays have the same size and whether the parameters have the correct type. For this aim, the class `OZ_Expect` can be customized to handle more complex data structures (like arrays or matrices).

6.3 Exploiting variable equality

The CPI provides the function

```
int  * OZ_findEqualVars(int  size, OZ_Term * v)
```

to detect equal variables, in an `OZ_Term` array. It expects `v` to be an array of size `size`. Assume the application

```
int  * pa  = OZ_findEqualVars (arr_sz, x );
```

where `pa` is called the position array. The array `x` is scanned with ascending index starting from 0 to determine the values of `pa`. If `x[i]` denotes a variable and this variable occurs the first time, the value of `pa[i]` is `i`. If the variable occurs but not the first time, `pa[i]` contains the index of the first occurrence. If `x[i]` denotes an integer, `pa[i]` contains -1.

As an example, consider the constraint $2a + 3b - 4c - 5d + 4e + 8 \leq 0$ where at runtime the constraint $c = e \wedge d = 2$ is imposed. The result of checking for equal variables is as follows.

`i`:	0	1	2	3	4
`x[i]`:	a	b	c	d	e
`pa[i]`:	0	1	2	-1	2

The state of the propagator can now be updated to represent the equivalent constraint $2a + 3b - 2 \leq 0$. Thus, this simplification avoids tedious handling of equal variables in the propagation algorithm, and it improves memory consumption and runtime behaviour.

7 Extending the CPI for a parallel emulator

Since propagators are concurrent computational entities, they are well suited to be executed in parallel. In this way, the efficiency of a propagator-based constraint solver using the CPI can be improved by parallelising the CPI.

Two reasons make Oz an ideal platform for a parallel constraint solver. First, on multiprocessor platforms, concurrent computation—which is modeled in Oz by threads—can be executed by parallel workers, as described by Popov in [26]. Second, Oz provides programming constructs that allow implementation of parallel search engines, that is, search engines that exploit different branches of a search tree in parallel (see [31] for the implementation of search engines in Oz).

This section describes an adaptation of the CPI to the parallel emulator (see [26]) and the changes necessary to run propagators under the parallel CPI.

A propagator is connected to the constraint store by its parameters. As described in Section 3, constraint propagation works by reading basic constraints of parameters, writing further basic constraints to the store, and resuming propagators suspending on these parameters. In the presence of parallel workers, we have to prevent the interference of operations on the data structures of the constraint store to avoid that data structures are corrupted (respectively, become inconsistent). Hence, exclusive access to the parameters of propagators has to be ensured by locking parameters. Reading and writing constraints are very frequent operations, so that the locking/unlocking overhead becomes very expensive. Another possibility is to snapshot the parameters, propagate locally on the snapshots, and eventually merge the snapshots with the store. Although this scheme avoids frequent locking, making snapshots and merging them (which requires locking, too) imposes extra an cost for copying domains. Therefore, we took the design decision to lock all parameters of a propagator before first accessing them. In this way, the propagator gains exclusive access to its parameters and can propagate without any efficiency penalties. This has the additional advantage that the changes to the nonparallel CPI are minimal.

7.1 Locking parameters

Locking parameters by a propagator is done by *spin-locks* (see [8]), which make a locked parameter inaccessible to other propagators. An advantage of spin-locks is that the way information in the constraint store is accessed need not be changed. Locking cyclic dependent objects has to be done with care to avoid creating a deadlock. For example, suppose we have two propagators $P1$ and $P2$ both sharing as parameters the variables A and B. Assume, further, $P1$ locks A and $P2$ locks B. Then they have run into a deadlock, since now $P1$ is waiting for B to be unlocked and $P2$ is waiting for A. A solution is to impose a total order on all variables in the constraint store and to lock parameters according to this order. Assume for our example $A < B$. Now, if $P1$ has successfully locked A, then $P2$ cannot lock B anymore (since $A < B$), and $P1$ can proceed by locking B.

7.2 Optimisations

Running multiple propagators in parallel in a single space allows only for limited speed-ups, since only propagators not sharing any variables as parameters can run in parallel (due to parameter locking). To overcome this deficiency we propose two optimisations.

1. As soon as a propagator notices that one of its parameters is locked, it stops and signals the emulator that it wants to be reexecuted at a later point in time. This allows the worker executing this propagator to pick another propagator and does not go idle by waiting for a lock. Therefore, we introduce a new return value `OZ_SCHEDULED`. The appropriately adapted execution state graph is shown in Figure 3. It shows an extra transition from execution state `running` to `runnable` for the new return value.

2. Only one propagator at a time is executed per computation space (see Section 2). That avoids the problems caused by locking shared parameters. This can easily be achieved by modifying the emulator's scheduler. This scheduling strategy is only beneficial if a search explores the nodes of a search tree in parallel. Oz allows the user to implement parallel search engines. The first experimental result of the proposed scheduling strategy is given in Section 9.

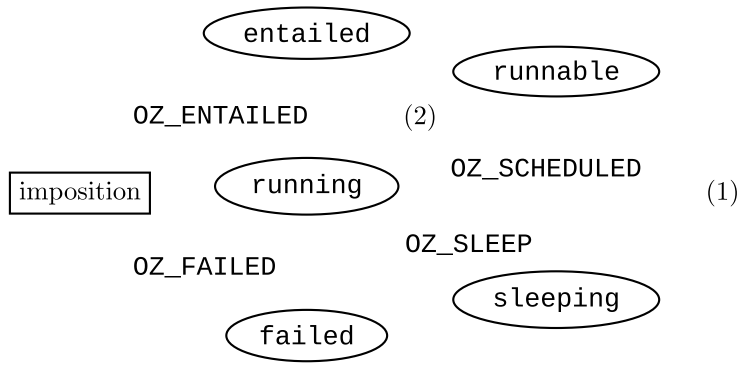

Figure 3: Execution states of a propagator in the parallel CPI

7.3 Adapting propagators to the parallel CPI

The implementation of propagators has only to be syntactically modified to allow the CPI to properly lock and unlock the propagator's parameters. The following modifications are necessary:

```
OZ_Return SomePropagator::run(void) {
   ...
   if (lockParameters())
      return OZ_SCHEDULED;
   // original propagation code
   unlockParameters();
   ...
}
```

The method `OZ_Propagator::lockParameters()` tries to lock all parameters read in and causes the propagator to return `OZ_SCHEDULED` if it finds already locked parameters. After the propagation algorithm has finished, the parameters are unlocked and suspension lists of appropriate parameters are scanned. This is done by the method `unlockParameters()` of class `OZ_Propagator`. The rest of the propagator code remains untouched. Thus, the necessary changes to adapt propagators to the parallel CPI can be neglected.

8 Related work

This section compares different constraint programming systems qualitatively rather than quantitatively with Oz (see Section 9 for benchmarks). First, we consider sequential systems and proceed by discussing parallel systems.

8.1 Sequential constraint solvers

8.1.1 Comparison with Ilog Solver

Ilog Solver [17] is a commercial C++ library that allows for solution of combinatorial problems in a constraint programming style in C++. Ilog Solver permits the user to add new constraints. Therefore, methods for the following tasks have to be implemented: posting the constraint, performing the constraint's propagation, and detecting an inconsistency. In contrast to CPI propagators, an inconsistency is signalled to the solver by a separate method rather than by a return value. There is no way to inform the solver that a constraint is entailed that disallows early discard of the constraint. The implementation of reified (respectively, meta-) constraints requires the determination of an "opposite" constraint, which is automatically imposed if the 0/1-variable is constrained to 0. The CPI provides, for that purpose, its propagator replacement features (see Section 5). Further, constraints in Ilog Solver can employ so-called demons to propagate selectively; that is, every variable is assigned a separate propagation method. This trades speeding up execution for increasing memory consumption. The CPI supports this technique, too, by modeling a demon by a separate propagator. Both systems support the implementation of global constraints effectively.

8.1.2 Comparison with *ECL^{i}PS^e* and CHIP

ECL^{i}PS^e is a Prolog-based language with a variety of extensions, in particular, a finite-domain constraint solver. *ECL^{i}PS^e* features attributed variables (see [15]) and coroutining and is extended by primitives to manipulate finite-domain variables, which allows implementation of constraints, even global ones, in *ECL^{i}PS^e* itself. The programmer has to take care of the suspension handling and propagator resumption. In contrast, the CPI abstraction `OZ_FDIntVar` fulfills this task in a self-acting way.

CHIP (refer to [11]) is the forerunner of *ECL^{i}PS^e* and provides a set of powerful built-in constraints. Nevertheless, the constraints are hard-wired and the user has to define new constraints in CHIP itself.

8.1.3 Comparison with indexicals

The so-called indexical approach (see [39]) allows the user to define new constraints by constructing them with indexicals. Indexicals are terms "x in r" where r defines how x is constrained and on what event an indexical is resumed. The indexicals that realize a single constraint exist independently of each other, that is, the constraint is not available first-class (see also [28]). Thus, algorithmic techniques employing global reasoning on all arguments of the constraint cannot be incorporated in this setting.

The first implementation of an indexical-based finite-domain solver was in clp(FD)(see [10]). The language is Prolog-based and it compiles to C code. It provides for a couple of symbolic constraints (e.g., *element*) as well as boolean constraints. Due to an indexical-based solver, it does not support global constraints.

AKL(FD) [4] implements indexicals in a concurrent constraint setting where local computation spaces are employed in so-called deep guards. Hence, there are similarities in the handling of variables and suspensions of different computation spaces. The integration of constraints is not as tight as in Oz. In AKL(FD) a single constraint cannot be used simultaneously to amplify the store and check entailment, as is possible for propagators.

8.1.4 SICStus Prolog

The finite-domain system of SICStus Prolog (see [18]) is based on the indexical implementation described in [3] using attributed variables (refer to [15]). A Prolog-based interface permits global constraints to be added. Further, the system provides for arithmetic, symbolic, and reified constraints, which are supported by library constraints implemented in C. In contrast to Oz, a finite domain allows the user also to represent negative integers.

8.2 Parallel constraint solvers

There has been a lot of work on parallelizing Prolog (for an overview see [9]). We first mention such Prolog-based systems and then proceed to systems based on the concurrent constraint (CC) paradigm.

8.2.1 Solvers based on Prolog-like languages

In [37] van Hentenryck reports on the results of *or*-parallelizing CHIP by extending the parallel logic programming system PEPSys (developed at ECRC) appropriately. To execute a Prolog program or-parallel means to explore choices in parallel. Separating the (basic) constraints in different choices is achieved by an optimized copying scheme.

There exists, for the above-mentioned ECL^iPS^e, an extension for parallel execution that inherits its parallelism from ElipSys (see [40, 27]) and its constraint-handling capabilities from CHIP (see above). ElipSys is an or-parallel system and implements *and*-parallelism by or-parallelism (as parallel ECL^iPS^e does also).

8.2.2 Solvers based on CC paradigm

A parallel implementation of Andorra-I was used as base for a finite-domain constraint solver in [13]. The implementation uses some specific features of Andorra-I (e.g., so-called updateable variables) as well as the normal variable-locking mechanism. Only one variable is locked at a time to avoid complications due to locking an entire set of variables at once (compare to Section 7).

The system GDCC was implemented on top of KL1, a committed-choice language developed at ICOT (see [35, 36]). The system allows the user to plug in constraint solvers via a so-called stream interface, which handles the synchronization between the parallel-running constraint solvers and the inference machine. There are three constraint solvers available: algebraic, boolean, and linear, all using parallel propagation algorithms.

9 Performance evaluation

To evaluate the performance of the CPI we ran two sets of benchmarks with Oz 2.0.3 and set the results in relation to Ilog Solver 3.2, Ilog Scheduler 2.2, SICStus Prolog 3#6, and *ECL^{i}PS^e* 3.7.1 (refer to [17, 16, 5, 18, 12]). We chose Ilog for comparison because it also uses C++ as an implementation language. The finite-domain solver of SICStus Prolog and *ECL^{i}PS^e* are embedded in Prolog systems and represent the state-of-the-art of Prolog-based implementations.

Note that the scheduling propagators for benchmarking the job-shop problems are implemented using the CPI.

9.1 Propagation performance

The first set of benchmarks measures the performance of the CPI without search. Two inconsistent constraints are imposed that require a lot of propagation to detect the inconsistency.

$$\begin{array}{ll}\text{inconsistent constraint (A):} & x, y \in \{0, \ldots, 1\,000\,000\} \wedge \\ & u, v \in \{0, \ldots, 2\,000\,000\} \wedge \\ & 2x = u \wedge 2y = v \wedge u = v + 1 \\ & \\ \text{inconsistent constraint (B):} & x, y \in \{0, \ldots, 10\,000\,000\} \wedge \\ & x < y \wedge y < x \end{array}$$

Inconsistency constraint	Oz	Ilog	SICStus	*ECLⁱPS^e*	Ilog / Oz	SICStus / Oz	*ECLⁱPS^e* / Oz
(A)	24.4	26.7	35.5	198.0	1.1	1.5	8.1
(B)	29.7	28.1	28.6	136.5	0.95	0.97	4.6

Table 1: Propagation performance (ran on an Ultra Sparc 1, 170MHz, SunOs 5.5; times are in seconds)

The time is taken until the inconsistency is measured. To keep the impact of propagation algorithms minimal, we use the straightforward constraints $x < y$, $2x = y$, and $x+y = z$, which do not require sophisticated propagation techniques and reason only on the bounds of domains.

The results in Table 1 show that the propagation mechanism of the CPI is competitive to the commercial Ilog Solver library. The propagation performance of SICStus Prolog is comparable with Oz, but ECL^iPS^e performs significantly worse.

9.2 Benchmarking job-shop problems

The following benchmarks compare Oz 2.0.3 with Ilog Scheduler 2.2 for classic 10×10 job-shop scheduling benchmarks for the proof of optimality (see [1]). In both systems we used the best strategy available in the corresponding libraries.[1]

In Table 2, the entry *Fails* denotes the number of failure nodes in the search tree needed for proving optimality. The entry *CPU* denotes the runtime needed for proving optimality. The last two columns compare the runtimes between Ilog Scheduler and Oz.

To be able to solve job-shop problems we implemented special global constraints, for example, edge-finding, and obtained results similar to Ilog Scheduler. The deviation between the results is due to the different propagation algorithms (we use a variant of [19] whereas in Ilog Scheduler a variant of [25] is used) and branching strategies. The main difference between the

[1]In Oz, the capacity constraint was modeled by `FD.schedule.serialized` and the branching strategy by `FD.schedule.taskIntervalsDistP`. In Ilog, we used for the capacity constraint the provided edge-finding (with parameter 2 for the strongest pruning) and for the branching strategy `IlcSelResMinLocalSlack` to select the resource and `IlcSelFirstRCMinStartMax` to select the task to schedule first.

	Oz		Ilog		Ilog/Oz	
Problem	Fails	CPU	Fails	CPU	Fails	CPU
MT10	1 799	28.7	5 853	69.7	3.25	2.43
ABZ5	1 431	23.8	2 548	23.4	1.78	0.98
ABZ6	148	2.3	207	2.3	1.40	1.00
La19	1 066	17.7	3 786	35.1	3.55	1.93
La20	881	13.9	10 384	72.2	11.79	5.19
ORB1	7 528	120.9	3 925	42.9	0.52	0.35
ORB2	425	7.2	16 922	183.0	39.82	25.42
ORB3	22 579	339.4	16 845	211.3	0.75	0.62
ORB4	1 034	15.7	17 677	207.6	17.10	13.22
ORB5	869	14.4	3 031	27.2	3.49	1.89

Table 2: Classic 10×10 job-shop problems (ran on an Ultra Sparc 1, 170MHz, SunOs 5.5)

systems results from our use of a very effective branching strategy described in [6], which additionally provides strong propagation through the use of so-called task intervals (see [43] for a more thorough discussion).

10 Conclusion

We have presented the interface CPI, which extends the CCP language Oz with the possibility to implement efficient constraint propagators in C++. The interface abstractions are high-level enough to hide away low-level issues, like propagator resumption, from the programmer. The expressiveness of the CPI and the provided extensions (as discussed in Section 5) allow the user to easily implement, for example, complex global and reified constraints, which make the interface suitable to tackle large and hard combinatorial problems. Further, the yielded interface performance is competitive with state-of-the-art constraint programming systems. The interface design can also be applied to Prolog-based implementations providing for coroutining. The CPI is general enough to be extended by further constraint systems, as already proved for set interval constraints (see [22]).

We showed that the presented CPI can be easily parallelized with minimal effort. The most promising approach is to execute a single propagator per computation space to avoid costs otherwise imposed by parameter sharing. A first experimental implementation suggests that adequate speed-ups due to exploiting parallelism are feasible.

Acknowledgements. The authors would like to thank Martin Müller, Konstantin Popov, Peter Van Roy, Christian Schulte, and the anonymous referees for their invaluable comments on earlier versions of this paper. Additionally, we would like to thank Konstantin Popov for helping us to add the CPI to his parallel emulator.

The research reported in this paper was supported by the BMBF (FKZ-ITW-9105 and FTZ-ITW-9106), and the Esprit Working Group CCL-II (EP 22457).

Remark. A copy of the DFKI Oz 2.0 implementation featuring the CPI can be obtained from `http://www.ps.uni-sb.de/oz2/`.

References

[1] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *Operations Research Society of America, Journal on Computing*, 3(2):149–156, 1991.

[2] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.

[3] B. Carlson. *Compiling and Executing Finite Domain Constraints.* Dissertation, Computing Science Department, Uppsala University, and SICS – Swedish Institute of Computer Science, Box 311 S-751 05 Uppsala, Sweden, 1995. Uppsala Theses in Computing Science 21, and SICS Dissertation Series 18.

[4] B. Carlson, M. Carlsson, and S. Janson. The implementation of AKL(FD). In *Proceedings of the International Logic Programming Symposium*, pages 227–241. The MIT Press, 1995.

[5] M. Carlsson, G. Ottoson, and B. Carlson. An open-ended finite domain constraint solver. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer-Verlag, 1997.

[6] Y. Caseau and F. Laburthe. Disjunctive scheduling with task intervals. LIENS Technical Report 95-25, Laboratoire d'Informatique de l'Ecole Normale Superieure, 1995.

[7] P. Codognet and D. Diaz. Compiling constraints in `clp(FD)`. *The Journal of Logic Programming*, 27(3):185–226, 1996.

[8] J. A. Crammond. *Implementation of committed choice logic languages on shared memory multiprocessors*. PhD thesis, Heriot-Watt University, May 1988.

[9] J. C. de Kergommeaux and P. Codognet. Parallel logic programming. *ACM Computing Surveys*, 26(3):295–336, September 1994.

[10] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In *Proceedings of the International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, 1993. MIT Press.

[11] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988. Institute for New Generation Computer Technology.

[12] ECRC and International Computers Limited and IC-Parc. *ECL^iPS^e, User Manual Version 3.7*, February 1998.

[13] S. Gregory and R. Yang. Parallel constraint solving in Andorra-I. In *Proceedings of the International Conference on Fifth Genration Computer Systems*, pages 843–850. Institute for New Generation Computer Technology, 1992.

[14] M. Henz and J. Würtz. Using Oz for college timetabling. In E. K. Burke and P. Ross, editors, *Practice and Theory of Automated Timetabling, First International Conference, Selected Papers, Edinburgh 1995*, volume 1153 of *Lecture Notes in Computer Science*, pages 162–178. Springer-Verlag, 1996.

[15] C. Holzbaur. *Specification of Constraint Based Inference Mechanisms through Extended Unification.* PhD thesis, Technisch-Naturwissenschaftliche Fakultät der Technischen Universität Wien, October 1990.

[16] Ilog, URL: `http://www.ilog.com/`. Ilog Scheduler *2.2, User Manual, 1996*, 1996.

[17] Ilog, URL: `http://www.ilog.com/`. Ilog Solver *3.2, User Manual, 1996*, 1996.

[18] Intelligent Systems Laboratory. *SICStus Prolog User's Manual.* SICS Research Report, Swedish Institute of Computer Science, URL `http://www.sics.se/isl/sicstus.html`, 1997.

[19] P. Martin and D. B. Shmoys. A new approach to computing optimal schedules for the job shop scheduling problem. In *International Conference on Integer Programming and Combinatorial Optimization, Vancouver*, pages 389–403, 1996.

[20] M. Mehl, T. Müller, K. Popov, C. Schulte, and R. Scheidhauer. DFKI Oz user's manual. DFKI Oz documentation series, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[21] M. Mehl, R. Scheidhauer, and C. Schulte. An abstract machine for Oz. In *Programming Languages, Implementations, Logics and Programs, Seventh International Symposium, PLILP'95*, volume 982 of *Lecture Notes in Computer Science*, pages 151–168, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[22] T. Müller and M. Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, September 1997.

[23] T. Müller and J. Würtz. The constraint propagator interface of DFKI Oz. DFKI Oz documentation series, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1997.

[24] T. Müller and J. Würtz. Extending a concurrent constraint language by propagators. In Jan Małuszyński, editor, *Proceedings of the International Logic Programming Symposium*, pages 149–163. The MIT Press, 1997.

[25] W. P. M. Nuijten. *Time and resource constrained scheduling.* PhD thesis, Technical University Eindhoven, 1994.

[26] K. Popov. A parallel abstract machine for the thread-based concurrent language Oz. In Inês de Castro Dutra, Vítor Santos Costa, Fernando Silva, Enrico Pontelli, and Gopal Gupta, editors, *Workshop On Parallism and Implementation Technology for (Constraint) Logic Programming Languages*, 1997.

[27] S. D. Prestwich. Elipsys programming tutorial. Technical Report ECRC-93-3, ECRC, January 93.

[28] J.-F. Puget and M. Leconte. Beyond the glass box: constraints as objects. In *Proceedings of the International Logic Programming Symposium*, pages 513–527. The MIT Press, 1995.

[29] V. A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, School of Comp. Sc., Carnegie-Mellon University, Pittsburgh, CA, 1989.

[30] C. Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, Leuven, Belgium, July 1997. The MIT Press.

[31] C. Schulte. Programming constraint inference engines. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, Schloss Hagenberg, Linz, Austria, October 1997. Springer-Verlag.

[32] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In A. H. Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 134–150, Orcas Island, Washington, USA, 1994. Springer-Verlag.

[33] C. Schulte, G. Smolka, and J. Würtz. Finite Domain Constraint Programming in Oz – A Tutorial. DFKI Oz documentation series, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1998.

[34] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

[35] S. Terasaki, D. J. Hawley, H. Sawada, K. Satoh, S. Menju, T. Kawagishi, N. Iwayama, and A. Aiba. Parallel constraint logic programming language GDCC and its parallel constraint solvers. In *Proceedings of the International Conferecne on Fifth Genration Computer Systems*, pages 330–346, 1992.

[36] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference MAchine. *Computer Journal*, 33(6):494–500, December 1990.

[37] P. Van Hentenryck. Parallel constraint statisfaction in logic programming: Preliminary results of CHIP within PEPSys. In Giorgio Levi and Maurizio Martelli, editors, *Logic Programming: Proceedings of the 6th International Conference (Lisbon)*, pages 165–180. MIT Press, 1989.

[38] P. Van Hentenryck, Y. Deville, and C. M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[39] P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Constraint processing in cc(FD). Technical report, Brown University, 1991.

[40] A. Veron, K. Schuerman, M. Reeve, and L. Li. Why and how in the ElipSys or-parallel CLP system. In *Proceedings of PARLE 93*, pages 291–302. Springer Verlag, 1993.

[41] M. Wallace. Practical applications of constraint programming. *Constraints*, 1(1&2):139–168, 1996.

[42] J. Würtz. Oz Scheduler: A workbench for scheduling problems. In M. G. Radle, editor, *Eighth International Conference on Tools with Artificial Intelligence*, pages 149–156, Toulouse, France, 1996. IEEE, IEEE Computer Society Press.

[43] J. Würtz. *Lösen kombinatorischer Probleme mit Constraintprogrammierung in Oz.* PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, January 1998. In German.