Extending a Concurrent Constraint Language by Propagators

Tobias Müller and Jörg Würtz
Programming Systems Lab
Universität des Saarlandes and DFKI Saarbrücken
Postfach 15 11 50, D-66041 Saarbrücken, Germany
Email: {`tmueller,wuertz`}`@ps.uni-sb.de`

Abstract

To solve large and hard discrete combinatorial problems it is often necessary to design new constraints. Current systems either focus on the high-level modeling aspect or on very efficient implementation technology. While each approach lacks the advantages of the other one, this paper describes the combination of both approaches in the concurrent constraint language Oz. Through an interface to program new finite domain constraints efficiently in C^{++}, the benefits of a high-level language to model a problem and of an efficient implementation technology for user-defined constraints are inherited.

Constraints and the Oz runtime system are linked together only by the interface abstractions. The interface supplies adequate abstractions to implement advanced algorithmic techniques. It provides, for example, also means to reflect the validity of a constraint and to control and inspect the state of the actual implementation of a constraint. This allows to solve demanding combinatorial problems, as for instance hard scheduling problems.

The described interface is not limited to concurrent constraint languages or a particular constraint system.

1 Introduction

Over the last years several approaches and systems were suggested to solve discrete combinatorial problems with finite domain constraints [4, 19, 3, 2, 5, 8]. To solve large and hard combinatorial problems it is often necessary to program new customized constraints and search strategies. Several approaches were suggested in literature.

A constraint logic programming (CLP) language like ECL^iPS^e [5] is well-suited to model 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 [19, 3] 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 [13]).

In Jan Małuszyński, editor, *Proceedings of the International Logic Programming Symposium*, pages 149–163. The MIT Press, 1997.

On the other hand, combinatorial problems can be tackled in a language like C++ together with a dedicated library for constraint solving (see *e.g.* ILOG [8]). 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 the advantages of these approaches by interfacing the high-level language Oz [17] with an interface to program new constraints efficiently in C++. Oz is a concurrent constraint programming (CCP) language [14] which comes with a rich predefined constraint library. By the concepts of a constraint store and entailment, new constraints can be programmed in the language itself. Furthermore, Oz provides means to program new search strategies [16].

Typically, the first step to solve a demanding problem consists in a prototypical implementation. In case of using Oz, rapid prototyping is supported by the features of a high-level language. After identifying the performance-critical parts of the program which are not covered by predefined library constraints, these parts should be re-casted in a very efficient implementation. To this aim an interface is provided which supports adequate abstractions to implement new constraints in C++. Thus, the application programmer can benefit from both a high-level language and an efficient C++ implementation.

The actual implementation of a constraint through the interface is called a propagator. 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 *resp.* 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 implementation of advanced propagators is supported as follows. The validity of arbitrary constraints can be reflected into a 0/1-valued variable (also called reification) efficiently by a general and easy-to-use mechanism. The state of a propagator can be controlled and inspected by the programmer. For example, it is possible to extend the set of variables an already running propagator is constraining. Running propagators can spawn further propagators to strengthen constraint propagation. Furthermore, it is possible to keep a history of computation steps in the propagator's state. This can be used to avoid redundant computation.

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 at the DFKI for set interval constraints.

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 competitive performance to state-of-the-art finite domain systems. It shows also competitive expressiveness and employs non-trivial algorithms for scheduling applications [20].

Plan of the paper. 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 Section 5 and 6. The specific features of Oz are considered in the following Section 7. 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 [17, 16, 11]. Further, the paper focuses on *finite domain constraints* over non-negative integers (for short finite domain constraints, see [16] for details).

In Oz a distinction is made between those constraints which are basic and those which are not. 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 $x = n$. The 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 their satisfiability is not computationally tractable. Such non-basic constraints are not held in the constraint store but are realized as *propagators*. A propagator is a computational agent which tries to narrow down 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, *i.e.*, its operational semantics. Often a complete propagator which 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 which lead to very good results (see also Section 9). A propagator may also cease to exist. If it ceases to exist, either P is entailed by C, or $C \wedge P$ is unsatisfiable.

As an example for constraint propagation, assume 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 is described in Section 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 [10], called the *emulator*. In the current 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. Note that in the following we also speak of propagators if we mean the actual implementation of the computational agents.

Overview. A propagator exists in different execution states which are controlled by the emulator. Further, the emulator provides a propagator with resources like computation time and heap memory. To separate the emulator and the implementation of propagators, from the emulator's point of view a propagator is an opaque entity that requires resources *resp.* services.

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 the propagator is waiting for. For example, many propagators will only be resumed when the domain bounds of its parameters are narrowed. On resumption a propagator *reads* for its parameters the basic constraints which 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 which 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 which start with "`OZ_`" refer to CPI abstractions.

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 which is incomplete in that sense. Technically, both requirements are realized in the interface by defining the class `OZ_Propagator` as *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, separates completely propagators from the emulator.

```
enum OZ_Return {ENTAILED, FAILED, SLEEP};

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

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 so-called *header function*. Such a function is connected via the Oz standard C interface [9] 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 which 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 shrunk, 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.

After creating a new instance of the propagator by invoking its constructor, a reference of type `(OZ_Propagator*)` is passed to the method `impose()` of `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.

Scheduling a propagator by the emulator. In order to schedule propagators, the emulator maintains for each propagator an execution state which 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.

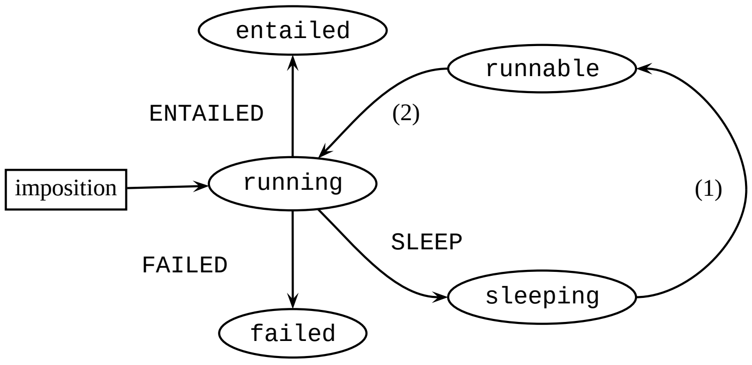

Figure 1: Execution states of a propagator

The value `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 will be ignored by the emulator until it is eventually disposed by the next garbage collection. An immediate disposal is not desirable since there may be multiple references to a propagator.

The return value `ENTAILED` indicates that the propagator detects that the constraint it implements is entailed by the constraint store, *i.e.*, 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 will be ignored until it is disposed by garbage collection.

If the propagator can neither detect inconsistency nor entailment, it returns `SLEEP`. Its execution state is set to `sleeping`.

A propagator is resumed if at least one of its variable parameters was involved in unification or its domain was further narrowed. The emulator scans the suspension lists of the concerned variables and either deletes entries where the propagator's execution state is `failed` *resp.* `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 will schedule it later on (the transition (2) from `runnable` to `running` is subject to the scheduler's policy and will be not discussed here). In fact, when the scheduler switches a propagator to `runnable` the propagator's method `propagate()` is executed.

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 difference between the current 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 the representation of the domain constraint of a parameter.

Memory management. A propagator class `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 which are reachable from the propagator's state. For example, to update a reference of the predefined type `OZ_Term` the provided function `OZ_updateHeapTerm()` has to be applied to it.

4 An Example

This section explains the constraint propagator interface of Oz by implementing the propagator for the constraint $x \leq y$.

The implementation of the $x \leq y$ propagator requires the definition of a new class inheriting from `OZ_Propagator`.

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

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.

```
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 ENTAILED;
8    }
9    x.leave(); y.leave(); return SLEEP;
10 failure:
11   x.fail(); y.fail(); return FAILED;
12 }
```

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

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 (3).[1]

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 (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 (5). The method `lowerUB(i)` makes the upper bound of the domain less than or equal to `i` (4) and the method execution `raiseLB(i)` makes the lower bound of the domain greater than or equal to `i` (5). Both functions return the size of the resulting domain. The method `getMinElem()` *resp.* `getMaxElem()` returns the smallest *resp.* largest value of the domain (4,5). In case 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 (6–8), *i.e.*, $x \leq y$ is entailed by the store. The returned value `ENTAILED` signals the emulator that the propagator can be discarded. Otherwise, returning `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 do some cleanups if the propagator is left because of a detected empty domain. The returned value `FAILED` signals the emulator that the current computation space is inconsistent.

[1]Note that in the sequel numbers in parentheses refer to program lines in Figure 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 1^{st} and 2^{nd} parameter which causes the propagator to be imposed not before the parameters are constrained to finite domains. Additionally, it is determined that narrowing the bounds of domains will resume the propagator. A new instance of the propagator is created by calling the constructor with the 1^{st} and 2^{nd} parameter. The application of method `impose()` makes 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 which fit smoothly in the setting presented before (see [12] for details).

Taking variable equality into account. Oz provides equality between variables, *i.e.* $x = y$, as 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 detect 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 which are equal refer to the same basic constraint. Therefore, updates to such a basic constraint are already visible via all other parameters while propagating and before leaving the propagator and telling the constraints 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.

Exploiting statefulness. Because search in Oz is based on a copying-scheme, not only changes to variables are saved, but also complete computation spaces including propagators. That allows the modification of a propagator's state destructively 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 [18]. Further, it can be used to store intermediate propagation results in the state which are expensive to recompute on each execution of the propagation method.

Replacing *resp.* 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 to. It is more efficient to replace $x + y = z$ by $2x = z$ rather 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 in the course of propagation orderings betweens tasks. 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 to care for these orderings anymore.

Encapsulated propagation. Typically, propagators tell the result of constraint propagation to the store. An instance of the class `OZ_FDIntVar` allows therefore to update the basic finite domain constraint of its associated parameter, such that the changes will become visible to the store. But, for example, propagators for reified (*resp.* meta) constraints [6] reflect only the validity of a constraint via a 0/1-variable to the store. 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.

Attaching a propagator with a stream. The CPI abstraction `OZ_Stream` allows 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 back via the stream the actual branching decision 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 competive to the state-of-the-art [20]. 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 certain required basic constraint, which is desired to implement, *e.g.*, an autonomous solver encapsulated in a propagator. The CPI allows the implementor of a propagator to determine the degree of propagation, *e.g.* 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.

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 which is equal or smaller (larger) than n. Further, the current lower *resp.* upper bound of the domain of a variable x is denoted by $\underline{x}$ *resp.* $\overline{x}$. 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 in-equation is

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

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

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

This propagator ceases to exist if the following in-equation holds:

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

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 enable propagators with arbitrary arity, vectors are allowed as parameters too. The propagator for inequality (1) has three parameters, *i.e.*, 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 of type `OZ_Term`, respectively. 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 will check 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).

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`. In case the variable occurs 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 Dealing with a Hierarchy of Computation Spaces

The main difference between constraint logic programming (CLP) and concurrent constraint programming (CCP) is the replacement of satisfiability detection by entailment checking. In Oz, computation spaces are used to implement entailment checking (recall that for this paper a computation space consists of a constraint store and a set of propagators attached to it). Furthermore, computation spaces are employed to implement different search strategies in Oz (see [16] for details). Thus a hierarchy of computation spaces may arise. Because constraint stores contain only basic constraints, entailment between stores can be decided efficiently. On the other hand, propagators should be taken into account, too. To this aim a propagator should cease to exist as soon as it can detect that the constraint it is imposing is entailed (but at last if all its parameters are determined). That all propagators in a computation space have ceased to exist is a necessary condition that a space, *i.e.*, its store and the constraints the propagators are imposing, is entailed.

While the implementors of the CPI have to take care of the handling of computation spaces, their existence and the resulting extra effort are *completely transparent* for the user of the CPI. This is due to the supplied functionality by appropriate methods which hide these issues from the programmer.

The fact that Oz computation may lead to a hierarchy of computation spaces has to be taken into consideration. In case a *global variable* (*i.e.*, a variable which is declared in a super-ordinated space) is further constrained in a subordinated space, the changes must be memorized. This is because the local information must not be visible in super-ordinated spaces and must be undone when the subordinated space is left. Second, the emulator can resume only those propagators which suspend in the current or in subordinated spaces. This is because constraints of a super-ordinated space are visible in all its subordinated spaces but not the other way around (note that this does not hold for propagators).

If a *local variable* (*i.e.*, a variable which is declared in the current space) is further constrained, the old domain need not be memorized. If a *global variable* is further constrained, it is bound to a fresh local variable. The old domain of the global variable is memorized (trailed) and the new domain is attached to the local variable. Furthermore, the suspension entries of a global variable which contain suspensions in the current or subordinated spaces are taken over. Propagators provide this functionality by the method `leave()` automatically, freeing the user from this task. Note that this technique avoids the usual time stamping where one ensures that a variable is trailed only once (see *e.g.* [2]).

8 Related Work

This section compares different constraint programming systems qualitatively rather than quantitatively with Oz (see Section 9 for benchmarks).

Comparison with ILOG SOLVER. ILOG SOLVER [8] is a commercial C^{++} library that allows to solve 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 which disallows early discard of the constraint. The implementation of reified (*resp.* meta) constraints requires to determine 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, *i.e.*, every variable is assigned a separate propagation method. This trades speeding up execution against increasing memory consumption. The CPI supports this technique too by modelling a demon by a separate propagator. Both systems support the implementation of global constraints effectively.

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

CHIP [4] 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.

Comparison with indexicals. The so-called indexical approach [19] 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 which realize a single constraint exist independently of each other, *i.e.*, the constraint is not available first-class (see also [13]). Thus, algorithmic techniques employing global reasoning on all arguments of the constraint cannot be incorporated in this setting.

AKL(FD) [2] 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 be used for entailment checking as is possible for propagators.

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 *resp.* ILOG SCHEDULER 2.2 [7]. We have chosen ILOG to compare with because they use also C++ as implementation language. Note that also the scheduling propagators for benchmarking the job-shop problems are implemented using the CPI.

The performance of Oz for small-size applications, like n-queens, is rather average due to the constant extra cost imposed by the very expressive first-class search facilities of Oz [15].

Propagation performance. The first set of benchmarks measures the performance of the CPI without search. Inconsistent constraints are imposed such that it requires a lot of propagation to detect the inconsistency. The time is taken until the inconsistency is detected. To keep the impact of propagation algorithms minimal we used 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.

Inconsistent constraint	ILOG SOLVER 3.2 (sec)	Oz 2.0.3 (sec)	ILOG / Oz
$x, y \in \{0, \ldots, 1\ 000\ 000\} \wedge$ $u, v \in \{0, \ldots, 2\ 000\ 000\} \wedge$ $2x = u \wedge 2y = v \wedge u = v + 1$	26.71	24.35	1.09
$x, y \in \{0, \ldots, 10\ 000\ 000\} \wedge$ $x < y \wedge y < x$	28.09	29.65	0.95

Benchmarks ran on a Ultra Sparc 1, 170MHz, SunOs 5.5.

Table 1: Propagation performance

The results in Table 9 show that the propagation mechanism of the CPI is competitive with that of ILOG SOLVER. Comparing the quality of the propagation algorithms used for the supplied library constraints of ILOG SOLVER *resp.* Oz is beyond the scope of this paper.

Benchmarking job-shop problems. The following benchmarks compare Oz 2.0.3 with ILOG SCHEDULER 2.2 for classical 10x10 job-shop scheduling benchmarks for the proof of optimality [1]. In both systems we used the best strategy available in the corresponding libraries.[2]

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 run time needed for proving optimality. The last two columns compare the run times between ILOG SCHEDULER and Oz.

To be able to solve job-shop problems we implemented special global constraints, *e.g.* edge-finding, and obtained results similar to ILOG SCHEDULER. The deviation between the results is due to the different propagation algorithms and

[2]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.

branching strategies.[3]

	Oz		ILOG		ILOG/Oz	
Problem	Fails	CPU	Fails	CPU	Fails	CPU
MT10	1 795	29.08	5 853	69.7	3.26	2.40
ABZ5	1 431	24.62	2 548	23.4	1.78	0.95
ABZ6	148	2.15	207	2.3	1.40	1.07
La19	1 066	18.43	3 786	35.1	3.56	1.90
La20	881	14.40	10 384	72.2	11.79	5.01
ORB1	7 533	124.93	3 925	42.9	0.52	0.34
ORB2	425	7.42	16 922	183.0	39.82	24.66
ORB3	22 590	345.28	16 845	211.3	0.75	0.61
ORB4	1 034	16.00	17 677	207.6	17.10	12.98
ORB5	871	14.84	3 031	27.2	3.48	1.83

Benchmarks ran on a Ultra Sparc 1, 170MHz, SunOs 5.5.

Table 2: Classical 10x10 job-shop problems

10 Conclusion

We have presented the interface CPI which extends the CCP language Oz by 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 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 also general enough to be extended by further constraint systems, as already proved for set interval constraints.

Acknowledgements. The authors would like to thank Martin Müller, Peter Van Roy, Christian Schulte, and the anonymous referees for their invaluable comments on earlier versions of this paper.

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

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

[3]A more thorough comparison (also with the Claire system) can be found through `http://www.ps.uni-sb.de/~wuertz/Benchmarks/schedulingBenchs.html`

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

[4] 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.

[5] ECRC. *ECL^{i}PS^e, User Manual Version 3.5.2*, December 1996.

[6] 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.

[7] ILOG, URL: `http://www.ilog.com`. ILOG SCHEDULER *2.2, User Manual, 1996*, 1996.

[8] ILOG, URL: `http://www.ilog.com`. ILOG SOLVER *3.2, User Manual, 1996*, 1996.

[9] M. Mehl, T. Müller, K. Popov, R. Scheidhauer, and C. Schulte. DFKI Oz user's manual. DFKI Oz documentation series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1997.

[10] M. Mehl, R. Scheidhauer, and C. Schulte. An abstract machine for Oz. In *Programming Languages, Implementations, Logics and Programs, Seventh International Symposium, PLILP'95*, Lecture Notes in Computer Science, pages 151–168, Utrecht, The Netherlands, 20–22 September 1995. Springer Verlag.

[11] T. Müller and J. Würtz. A survey on finite domain programming in Oz. In *Notes on the DFKI-Workshop: Constraint-Based Problem Solving, To appear as Technical report D-96-02*, Kaiserslautern, Germany, 1996.

[12] T. Müller and J. Würtz. The constraint propagator interface of DFKI Oz. DFKI Oz documentation series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1997.

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

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

[15] C. Schulte. Programming constraint inference engines. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, number 1330 in Lecture Notes in Computer Science, pages 520–534. Springer-Verlag, 1997.

[16] 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.

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

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

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

[20] 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.

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