Available at
www.ElsevierMathematics.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Journal of Computational and Applied Mathematics 162 (2004) 79–92

JOURNAL OF COMPUTATIONAL AND APPLIED MATHEMATICS

www.elsevier.com/locate/cam

# An interval component for continuous constraints

Laurent Granvilliers

*IRIN, Université de Nantes, B.P. 92208, 44322 Nantes Cedex 3, France*

Received 29 November 2001; received in revised form 13 November 2002

**Abstract**

Local consistency techniques for numerical constraints over interval domains combine interval arithmetic, constraint inversion and bisection to reduce variable domains. In this paper, we study the problem of integrating any specific interval arithmetic library in constraint solvers. For this purpose, we design an interface between consistency algorithms and arithmetic. The interface has a two-level architecture: functional interval arithmetic at low-level, which is only a specification to be implemented by specific libraries, and a set of operations required by solvers, such as relational interval arithmetic or bisection primitives. This work leads to the implementation of an interval component by means of C++ generic programming methods. The overhead resulting from generic programming is discussed.


*Keywords:* Interval arithmetic; Consistency techniques; Constraint solving; Generic programming; C++

## 1. Introduction

Interval arithmetic has been introduced by Ramon E. Moore [9] in the late 1960s to deal with infiniteness, to model uncertainty, and to tackle rounding errors of numerical computations. In practice, rounding errors can be controlled by means of rounding modes of floating-point arithmetic, which are specified in the IEEE report [6]. Given a real function over a domain, the main algorithm defined from interval arithmetic is the verified computation of a superset of the range of the function. Interval analysis is a set of algorithms that have been extended from numerical analysis to intervals, such as solving of linear or nonlinear systems, differentiation or integration. Given a problem over the real numbers, interval computations are said to be reliable (verified) since no solution is lost.

*E-mail address:* granvilliers@irin.univ-nantes.fr (L. Granvilliers).

0377-0427/$ - see front matter 
doi:10.1016/j.cam.2003.08.022

In this framework, nonlinear systems are solved by branch-and-prune algorithms. The search space is a box defined as the Cartesian product of variable domains. Interval arithmetic is useful to check whether a box contains no solution. Inconsistent boxes are rejected. Consistent boxes are split until reaching a given precision. Local consistency techniques [2] are polynomial algorithms that may accelerate the search of solutions. These methods are more efficient than interval arithmetic alone since inconsistent parts of boxes can be pruned during the early process. Hull consistency corresponds to constraint inversion w.r.t. one variable. Box consistency is implemented by a local bisection procedure over one domain w.r.t. one constraint.

Nowadays this domain is growing since many engineering applications require reliability. The interval type recently became a first-class citizen type of Sun's Forte compilers [14], which is due to G. William Walster's work. Combinations of interval arithmetic and constraint programming algorithms have been shown to be powerful for many applications [15]. The reliability property ensures that every potential solution or decision can be reached. For instance, in computer-aided design [12], one wants that all possible concepts can be defined. In automatic control [7], a main challenge is to design robust commands.

This work is part of a component-based approach to the design of constraint programming libraries. Constraint solvers for continuous constraints are hybrid in essence, combining algorithms from artificial intelligence, interval analysis or computer algebra. The design of solvers should support their heterogeneous nature, considering that no programmer is specialist of all research domains. This can be done through the software engineering notion of component. A component is a software object encapsulating sets of functionalities, having a determined behavior, and meant to interact with other components.

In this paper we introduce the notion of component for interval arithmetic. This component should (1) implement the necessary services for constraint programming algorithms and (2) interface existing interval arithmetic libraries, for instance `Bias` [8], `Jail` [5], `MPFI` [11] or `Sun's Forte` [14]. The notion of interface supports reuse of code and allows engineers to work with their preferred tools. For this purpose we define a two-level component architecture. At low-level, there is a specification of the way to plug specific interval libraries in the constraint library. On top of this module there are other services such as relational interval arithmetic used for constraint inversion or bisection primitives, which are implemented once for all.

Our implementation uses generic `C++` programming facilities [13], as is done for `Jail` whose interval type is parameterized by the type of interval bounds. The traits technique [10] allows one to statically specify all services demanded to interval libraries: types for interval bounds and intervals, arithmetic operations, comparisons, etc. The integration of any external interval library needs implementing the traits. The interval component is a templated class, which is parameterized by traits at compile-time. With respect to external libraries the main advantage is that high-level operations are obtained with no effort. However, the price to pay is a slight slow-down at runtime. In preliminary design of constraint solvers, we believe that this slow-down is less important than program development time, which is improved by genericity and reuse of components. The performances are discussed in Section 5.

The outline of this paper is the following. In Section 2 we introduce some notions from interval arithmetic and consistency techniques and we show the place of interval arithmetic in constraint solvers. A high-level description of services that the interval component must support is described

in Section 3. A C++ implementation of the interval component is proposed in Section 4. Finally, experimental results are discussed in Section 5.

## 2. The role of interval arithmetic

The definition of an interval component needs understanding the place of arithmetic in constraint solving methods. In this section, we present elements of interval arithmetic and consistency techniques.

### 2.1. Interval arithmetic

Let $\mathbb{F}$ be a finite set of machine-representable real numbers compactified with the infinities in the obvious way. In practice, the set $\mathbb{F}$ often corresponds to the set of IEEE floating-point numbers [6]. Given any element $a \in \mathbb{F}$, we define

$$a^- = \max\{x \in \mathbb{F} \mid x < a\} \quad \text{and} \quad a^+ = \min\{x \in \mathbb{F} \mid x > a\}.$$

Given a real number $r$, the downward and upward rounding modes are defined as follows.

$$\lfloor r \rfloor = \max\{x \in \mathbb{F} \mid x \leqslant r\}, \quad \lceil r \rceil = \min\{x \in \mathbb{F} \mid x \geqslant r\}.$$

An *interval* is a connected set of real numbers. In the following, we are interested in the set $\mathbb{F}$ of closed intervals bounded by elements of $\mathbb{F}$. Every interval $\boldsymbol{x}$ is denoted by $[\underline{x}, \bar{x}]$. The *convex hull* maps sets of real numbers to intervals using the outward rounding mode. Given $\rho \subset \mathbb{R}$, we have $\mathsf{Hull}(\rho) = [\lfloor \inf \rho \rfloor, \lceil \sup \rho \rceil]$.

Interval arithmetic operations are set theoretic extensions of the corresponding real operations. Given $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}$ and an operation $\diamond \in \{+, -, \times, \div\}$, the following property holds:

$$\boldsymbol{x} \diamond \boldsymbol{y} = \mathsf{Hull}(\{x \diamond y \mid (x, y) \in \boldsymbol{x} \times \boldsymbol{y}\}).$$

Due to monotonicity properties, these operations can be implemented by floating-point computations over the bounds of intervals. For instance, given two intervals $\boldsymbol{x} = [a, b]$ and $\boldsymbol{y} = [c, d]$, we have:

$$\boldsymbol{x} + \boldsymbol{y} = [\lfloor a + c \rfloor, \ \lceil b + d \rceil],$$

$$\boldsymbol{x} - \boldsymbol{y} = [\lfloor a - d \rfloor, \ \lceil b - c \rceil],$$

$$\boldsymbol{x} \times \boldsymbol{y} = [\lfloor \min\{ac, ad, bc, bd\} \rfloor, \ \lceil \max\{ac, ad, bc, bd\} \rceil].$$

The notion of interval extension [1] has been introduced to compute some superset of the range of a real function over a domain. Given a real function $f$ and a domain $D$, let $\boldsymbol{f}^u(D)$ denote the range of $f$ over $D$.

[1] Interval extensions are also called interval forms or inclusion functions.

**Definition 1** (interval extension). An *interval extension* of a real function $f : D_f \subset \mathbb{R}^n \to \mathbb{R}$ is a function $\boldsymbol{f} : \mathbb{F}^n \to \mathbb{F}$ such that for all $\mathbf{X}$ in $\mathbb{F}^n$,

$$\mathbf{X} \subseteq D_f \;\Rightarrow\; \boldsymbol{f}^u(\mathbf{X}) \subseteq \boldsymbol{f}(\mathbf{X}).$$

This definition implies the existence of infinitely many interval extensions of a given real function. The concept of natural extension of a real function $f$ corresponds to an interval extension obtained from a given expression of $f$, where each operation is replaced with the corresponding interval operation, each real number $r$ is replaced with $\mathsf{Hull}(\{r\})$ and each variable is replaced with an interval variable. Natural extensions are inclusion monotonic, which follows from the monotonicity of interval operations (see [1] for more details).

### 2.2. Branch-and-prune algorithms

Given a nonlinear constraint system and a box representing the variable domains, branch-and-prune algorithms compute sets of boxes, the union of which enclosing the solution set of the system. Boxes are processed by pruning and branching operations. A pruning operation removes inconsistent values from a box. Branching of a box generates a set of sub-boxes. A branch-and-prune process terminates if each box is precise enough or if a given number of boxes have been computed.

In this section we not only present state-of-the-art constraint solving methods, but we focus on the required interval operations.

*Termination criterion.* Given an interval $\boldsymbol{x}$, the precision of $\boldsymbol{x}$ may be defined by the width w.r.t. Haussdorf distance,

$$\mathsf{Width}(\boldsymbol{x}) = \lceil \bar{x} - \underline{x} \rceil.$$

This operation requires access to interval bounds and a correctly rounded minus operation over $\mathbb{F}$. The result is an element of $\mathbb{F}$. Another approach consists in using the number of elements of $\mathbb{F}$ occurring in $\boldsymbol{x}$.

The precision of a box is generally defined componentwise, which uses comparisons of elements of $\mathbb{F}$.

*Constraint satisfaction.* Given $\Diamond \in \{=, <, \leqslant, \geqslant, >\}$ and two intervals $\boldsymbol{x}, \boldsymbol{y}$, the "possible" interpretation of relation $\Diamond$ is defined as follows:

$$\exists (x, y) \in \boldsymbol{x} \times \boldsymbol{y} : x \Diamond y \;\Rightarrow\; \boldsymbol{x} \Diamond \boldsymbol{y}.$$

Given a constraint $c$ defined by the expression $f(x) \Diamond g(x)$, a domain $\boldsymbol{x}$ and an interval extension $\boldsymbol{f}$ ($\boldsymbol{g}$) of $f$ ($g$), $c$ is said to be satisfied if $\boldsymbol{f}(\boldsymbol{x}) \Diamond \boldsymbol{g}(\boldsymbol{x})$ is evaluated in true. Otherwise, $\boldsymbol{x}$ contains no solution of $c$, which is guaranteed by the reliability property of interval arithmetic. In this case, $\boldsymbol{x}$ is declared to be inconsistent. Such a domain can be rejected during the branch-and-prune process.

Unfortunately, constraints may be declared satisfied even if parts of domains contain no solution. Before rejecting a box, the branching procedure must completely isolate inconsistent values. This is clearly not efficient, which motivates the use of consistency techniques to narrow down domains.

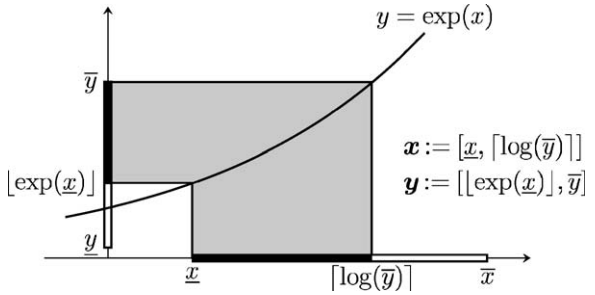


Fig. 1. Narrowing of $\boldsymbol{x} = [\underline{x}, \bar{x}]$ and $\boldsymbol{y} = [\underline{y}, \bar{y}]$ w.r.t. constraint $y = \exp(x)$.

*Hull consistency.* We describe the narrowing technique based on relational interval arithmetic [4], which implements the so-called hull consistency. For instance, consider a constraint $c$ defined by $y = \exp(x)$, the variable domains $\boldsymbol{x}$ and $\boldsymbol{y}$, and the associated set of solutions

$$\{(x, y) \in \boldsymbol{x} \times \boldsymbol{y} \mid y = \exp(x)\}.$$

Domains can be reduced by the following narrowing operations, which show that elementary functions and arithmetic operations need to be inverted (see Fig. 1):

$$\boldsymbol{y} := \mathsf{Hull}(\boldsymbol{y} \cap \exp(\boldsymbol{x})),$$

$$\boldsymbol{x} := \mathsf{Hull}(\boldsymbol{x} \cap \exp^{-1}(\boldsymbol{y})).$$

The definition of the inverse operation $\exp^{-1}$ requires interval constants such as the empty set, access to interval bounds, comparisons of bounds with the 0 element of $\mathbb{F}$ and the interval logarithm operation.

$$\exp^{-1}(\boldsymbol{x}) = \begin{cases} \log(\boldsymbol{x}) & \text{if } \underline{x} > 0, \\ \varnothing & \text{if } \bar{x} \leqslant 0, \\ [-\infty, \lceil \log(\bar{x}) \rceil] & \text{otherwise.} \end{cases}$$

The last case, which uses rounding of an elementary function applied over an element of $\mathbb{F}$, is more difficult.[2] An implementation based on this definition must enter into the programming details of arithmetic over $\mathbb{F}$. We see that relational interval arithmetic needs a control of rounding modes and a set of correctly rounded operations over $\mathbb{F}$ such as the logarithm.

*Box consistency.* The narrowing technique based on box consistency [3] implements constraint satisfaction and branching steps. For instance, given a constraint $c$ defined by $f(x) = 0$, an interval extension $\boldsymbol{f}$ of $f$ and a domain $\boldsymbol{x}$, the new domain $\boldsymbol{y}$ is the smallest interval included in $\boldsymbol{x}$ such that $c$ is satisfied by $[\underline{y}, \underline{y}^+]$ and $[\bar{y}^-, \bar{y}]$. In practice, the narrowing algorithm applies a dichotomous search over $\boldsymbol{x}$. This process requires creation of intervals from bounds, computation of successors and

[2] This is not always true in practice since $\log(\boldsymbol{x})$ generally returns the expected result when the interval argument contains 0.

predecessors of elements of $\mathbb{F}$ and splitting of intervals. The splitting operation bisects any interval $\boldsymbol{x}$ in two parts. The bisection point can be given by the expression

$$\lceil \underline{\boldsymbol{x}} + \lceil \lceil \bar{\boldsymbol{x}} - \underline{\boldsymbol{x}} \rceil \div p \rceil \rceil.$$

Once again, correctly rounded arithmetic operations over $\mathbb{F}$ are necessary.

## 3. Specification of arithmetic

In this section, we describe the arithmetic of constraints solvers by a set of types and operations associated with machine and interval numbers.

There is a main question: should high-level operations such as relational interval arithmetic be only specified in the interval component? To our opinion, no, since our aim is to implement these operations once for all specific libraries. As a consequence, a set of operations on machine numbers are also required.

### 3.1. Machine numbers

The set $\mathbb{F}$ of interval bounds is represented in programs by a particular data type, which is also denoted by $\mathbb{F}$ in the following. This type and associated operations must be part of the interface between the component and interval libraries. We distinguish several categories of operations.

*Constants.* The set $\mathbb{F}$ is meant to contain at least three numbers: the infinities $-\infty$ and $+\infty$ and the zero value. The infinities can be used in the creation of infinite intervals. Zero appears in the definition of many interval operations. As a consequence, these numbers are considered as predefined constants.

*Rounding.* The rounding mode for machine numbers needs to be controlled. As a consequence, the rounding downward procedure and the rounding upward procedure must be given to the interval component.

*Input/output.* Intervals may be created in programs from their bounds, which are generally represented by default types of numbers, such as `int`, `float` or `double` in the `C/C++` languages. Then, for each type $\mathbb{T}$ of numbers, two conversion operations are required:

$$\text{conversions} : \begin{cases} \text{convIn} : \mathbb{T} \to \mathbb{F}, \\ \text{convOut} : \mathbb{F} \to \mathbb{T}. \end{cases}$$

Note that conversions may not preserve values if $\mathbb{T}$ and $\mathbb{F}$ are mathematically different. In practice we believe that convIn must preserve values, in order to guarantee that the user's input is handled without loss of information.

*Comparisons.* The implementation of relational interval arithmetic uses comparisons of interval bounds. For instance, the relational division is equivalent to the functional division if the lower bound of the quotient is strictly greater than 0. The set of operations $\{<, \leqslant, =, \neq, \geqslant, >\}$ is represented by a set of procedures specified as follows:

$$\text{comparisons} : \mathbb{F} \times \mathbb{F} \to \text{boolean}.$$

*Evaluation.* The inversion of arithmetic operations and elementary functions does not only requires interval operations, but correctly rounded operations on machine numbers.

$$\text{unary operations} : \mathbb{F} \to \mathbb{F},$$

$$\text{binary operations} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}.$$

The precise list is out of scope of this paper, since it results from a precise description of relational interval arithmetic.

### 3.2. Interval numbers

Intervals are defined by their bounds, which belong to the set $\mathbb{F}$ of machine numbers. Such a definition of the interval type is adapted to solvers based on consistency techniques and bisection-like procedures. The interval type is associated with constants and operations described below.

*Constants.* There are three obvious constants in $\mathbb{F}$: the empty set $\varnothing$, the real set $[-\infty, +\infty]$ and zero $[0,0]$. Moreover, three other constants are intensively used, for instance in differentiation algorithms: $[1,1]$, $[-1,-1]$ and $[2,2]$. Finally, the evaluation of trigonometric functions needs $\pi$ to be known, which is defined by an interval enclosing it.

*Constructors.* Given input data a constructor returns an interval enclosing it. The default constructor returns the real set. The copy constructor returns a copy of the argument. Intervals can be generated from bounds represented by numbers from $\mathbb{F}$ or string representations from the set $\mathbb{S}$.

$$\text{default} : \to \mathbb{F},$$

$$\text{copy} : \mathbb{F} \to \mathbb{F},$$

$$\text{bounds} : \mathbb{F} \times \mathbb{F} \to \mathbb{F},$$

$$\text{bounds} : \mathbb{S} \times \mathbb{S} \to \mathbb{F}.$$

*Bound operations.* We must provide access to bounds and modifications of bounds. For instance, for the lower bound, there are the two following operations.

$$\text{getLower} : \mathbb{F} \to \mathbb{F},$$

$$\text{setLower} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}.$$

*Comparisons.* Comparisons of intervals implement the "possible" interpretation. For a relation $\Diamond \in \{<, \leqslant, =, \neq, \geqslant, >\}$, a comparison $\boldsymbol{x} \Diamond \boldsymbol{y}$ succeeds if there exists a couple of real numbers $(x, y) \in \boldsymbol{x} \times \boldsymbol{y}$ such that $x \Diamond y$ is true. The corresponding functions have the following prototype.

$$\text{comparisons} : \mathbb{F} \times \mathbb{F} \to \text{boolean}.$$

*Observers.* Several operations manipulate intervals. We distinguish two kinds of functions: boolean functions checking if an interval is empty, if an interval contains a real number or if an interval is canonical, and their negations; functions computing the width of an interval, the hull, the intersection or the bisection operation in two parts.

*Evaluation.* The interval component must implement all arithmetic operations and elementary functions defined on $\mathbb{F}$. Each operation, for instance the addition, is implemented by two functions. The first one takes as input two intervals and returns the sum. The second one takes as input two intervals and adds the second argument to the first argument.

*Display.* An interval can be printed as a couple of bounds or as a midpoint associated with an interval error. Both modes are implemented. Moreover the bounds have generally to be rounded if the exact representation uses a huge number of digits. The number of output decimals is managed as a parameter of display functions. Observe that outward rounding is necessary.

## 4. Implementation of arithmetic

A generic implementation of arithmetic needs separating code for specific interval libraries from high-level operations occurring in constraint solvers. For this purpose, an interface is defined, which allows easily plugging any specific library in the interval component at compile-time. We have used the so-called C++ traits technique for interface specifications, and C++ templates for generic programming (see the next section).

### 4.1. An introduction to generic programming

A *template* is a skeleton for a family of entities associated with predefined functionalities. Specific features of entities are defined as parameters of the skeleton. For instance, the family of vectors can be uniformly implemented, the only parameter being the type of elements inserted in vectors. However, for instance, vectors require elements to be assignable. How can this property be specified? For this purpose, Nathan C. Myers has introduced the *traits* technique in [10]. A traits is a specification of properties of some data type. A traits for the elements added in vectors must define at least one function `assign`, as follows:

```
template <class T>              //default traits for type T
struct ElementTraits {
  typedef T Element;
  static void assign(Element& x, const Element& e);
};
struct ElementTraits<int> {   //specialization for integers
  typedef int Element;
  static void assign(Element& x, const Element& e) {
    x = e;   //assignation for integers is obvious
  }
};
```

Vectors are parameterized by the type of elements `T`. Provided that the type `ElementTraits<T>` exists, vectors of type `Vector<T>` can be statically defined. Moreover, the skeleton of vectors does not depend on the specific properties of type `T` implemented by the traits. For instance, as shown

below, the assign function of the traits for T is called when an element of type T is inserted in a vector:

```
template <class T>
class Vector {                          //templated class Vector<T>
public:
  ...
  void insert(const T & e) {   //insertion of e in the vector
    ...
    ElementTraits<T>::assign(_vec[_last++],e);
  }
};
```

Note that the definition of vectors for a new type of elements just requires the default traits to be instantiated. Operations on vectors are implemented only once in the generic class Vector<T>. We see that a traits is a good candidate for the implementation of interfaces.

### 4.2. Interface for specific interval libraries

The interface is defined by two traits for manipulating machine and interval numbers. The first traits BoundTraits<Bo> specifies necessary functionalities for types of bounds Bo, described in Section 3.1.

```
template <class Bo>
struct BoundTraits { //default traits
  typedef Bo Real;                                  //type of bounds
  static Real toBound(const DefaultRealType& x); //cast Bo(x)
  static const Real zero;                           //0 value
  ...
  static Real nextElement(const Real& x);         //x+
  ...
  static void setRoundingDownward();               //downward rounding
  ...
  static bool eq(const Real& x,
                 const Real& y);                  //x==y ?
  ...
  static void display(ostream& os,
                      const Real& x);            //display of x
};
```

For instance, for the double type the traits BoundTraits<double> has to be implemented.

The second traits IntervalTraits<Bo,Iv,Pw> tackles the required properties of interval types (see Section 3.2). There are three parameters: Bo is the type of interval bounds (e.g., double), Iv is the interval type (e.g., BiasInterval) and Pw is an integer type that is used for power operations.

```
template <class Bo, class Iv, class Pw>
struct IntervalTraits {          //default traits for intervals
  typedef BoundTraits<Bo> RealTraits;        //traits for bounds
  typedef Bo     Real;                        //type of bounds
  typedef Iv     Interval;                    //interval type
  static const Interval realSet;              //constant [-oo,+oo]
  ...
  static Interval cons(const Real& l,
                const Real& r);        //returns [l,r]
  ...
  static Real getLower(const Interval& u);   //lower bound of u
  static void setLower(Interval& u,
                const Real& x);        //lower bound of u:= x
  ...
  static bool eq (const Interval& u,
           const Interval& v);        //u==v ?
  ...
  static bool isEmpty (const Interval& u);   //u is empty ?
  static bool contains(const Interval& u,
                const Real& x);        //u contains x ?
  ...
  static Interval hull(const Interval& u,
                const Interval& v);    //returns hull(u + v)
  static Interval inter(const Interval& u,
                 const Interval& v);   //intersection (u,v)
  ...
  static Interval add(const Interval& u,
               const Interval& v);    //returns u+v
  static void addEq(Interval& u,
              const Interval& v);      //u:= u+v
  ...
  static void display(ostream& os,
                const Interval& u);    //output of u on os
};
```

Note that the default interval traits has to be specialized for every specific interval library. The traits for machine numbers has to be implemented once for each specific type, for instance double in C/C++ languages. The effort depends on the set of functionalities provided by existing libraries.

For instance, for Bias, we use the following traits. The implementation of the addition shows that an intermediary data result is necessary for the call to BiasAddII. Such feature clearly involves an overhead at runtime.

```
struct IntervalTraits<double,BiasInterval,int> {
  typedef BoundTraits<double> RealTraits;     //traits for bounds
  typedef double              Real;            //type of bounds
  typedef BiasInterval        Interval;        //type of intervals
  ...
  static Interval add(const Interval& u, const Interval& v) {
    Interval result;
    BiasAddII(& result,& u,& v);
    return result;
  }
  ...
}
```

### 4.3. The interval component

The way to make interval constraints independent of arithmetic is to create an interval class CtlInterval parameterized by an interval traits. An interval of type CtlInterval is just coded by an interval from the traits. Doing so, the class represents the interface for interval arithmetic to be used in the constraint library:

```
template <class Bo, class It, class Pw>
class CtlInterval {
public:
  typedef Bo Real;
  typedef Pw Integer;
  typedef It Interval;
  static const
  CtlInterval realSet=CtlInterval(IntervalTraits::realSet);
  ...
  CtlInterval(const Real& l, const Real& r)       //constructor:
    :_I(IntervalTraits::cons(l,r)) {         //call to the trait's
  }                                                 //cons operation
  ...
private:
  typedef BoundTraits<Bo>          RealTraits;
  typedef IntervalTraits<Bo,It,Pw> IntervalTraits;
  Interval _I;     //a CtlInterval is just coded by an Interval
};
```

The rest of the implementation of the class corresponds to functionalities from the traits or high-level operations. For instance, the relational logarithm operation is described as follows:

```
void logRel(CtlInterval& u)
//relational logarithm operation of *this=[a,b] computed in u
{
  if (RealTraits::leq(this->getUpper(),
                RealTraits::zero)) {                    //b<=0
    //result: empty set
    u.setEmpty();
  }
  else if (this->contains(RealTraits::zero)) {              //a<=0<b
    //result: [-oo,log(b)]
    u.setLower(RealTraits::minusInfinity);
    u.setUpper(RealTraits::logUp(this->getUpper()));
  }
  else {                                                     //a>0
    //result: log([a,b])
    u.log(*this);
  }
}
```

For instance, intervals from `Bias` are plugged in the constraint library at compile-time by the following instantiation of the `CtlInterval` template,

```
CtlInterval<double,BiasInterval,int>,
```

provided that both traits `IntervalTraits<double,BiasInterval,int>` and `BoundTraits <double>` are already implemented. Note that several specific types of intervals with bounds of the same type `double` must share the same traits `BoundTraits<double>`.

## 5. Performances

In this section, we analyze the overhead induced by generic programming, in particular computation times w.r.t. specific libraries and the effort required to integrate specific libraries.

The results are reported in Table 1. Programs are compiled with `egcs` version `2.91.66`: with optimization (`gcc -O`) and without optimization (`gcc`). Columns with label `Interval` contain computation times in seconds for 10 million operations over the type `Interval`. Columns with label `CtlInterval` contain computation times in seconds for 10 million operations over the type `CtlInterval` parameterized by `Interval`. Ratios $R$ correspond to slow-downs when the type `CtlInterval` is used. The values of intervals $\boldsymbol{x}$ and $\boldsymbol{y}$ have been randomly defined.

Without compilation optimization, the average price to pay for genericity is about 80% ($R \approx 1.8$). If the compiler optimization option `-O` is used then the average price is only 10% ($R \approx 1.1$). We

Table 1
Comparison of performances for the interval types

| | gcc | | | gcc -O | | |
|---|---|---|---|---|---|---|
| Operation | CtlInterval | Interval | $R$ | CtlInterval | Interval | $R$ |
| $x + y$ | 3.9 | 1.1 | 3.5 | 1.3 | 0.8 | 1.6 |
| $x \times y$ | 4.9 | 2.1 | 2.3 | 2.3 | 1.8 | 1.3 |
| $x / y$ | 7.1 | 5.8 | 1.2 | 4.5 | 5.6 | 0.8 |
| $\exp(x)$ | 9.3 | 7.6 | 1.2 | 7.8 | 7.4 | 1.1 |
| $\log(x)$ | 5.4 | 6.1 | 0.9 | 3.9 | 5.9 | 0.7 |
| $x^7$ | 4.3 | 2.4 | 1.8 | 2.8 | 2.1 | 1.3 |

consider that this slow-down can be paid to enhance genericity. However, for specific operations like the addition, the price is also 80%, which is irrelevant with respect to efficiency requirements. We expect that the new generation of C++ compilers like gcc 3.0 will improve code generation to handle templated classes.

The effort for programming the traits should not be prohibitive. For instance, let us analyze the case for the Bias library. Machine numbers belong to the double type. Most of operations described in Section 3.1 are already developed in Bias. In particular, a portable code for rounding modes is available. Programming of the bound traits requires about a hundred lines, which mainly consists in calls to existing functions in Bias. The case for interval numbers is slightly different since a set of operations have to be developed, for instance a correctly rounded operation for computing reals from strings, a function for counting the number of floating-point numbers in intervals or a correctly rounded display function taking into account the number of decimals. The interface is made of about 200 lines. However, the more difficult functions like interval evaluation of cosine already exist.

## 6. Conclusion

Generic C++ programming is a clearly identified approach to the conception of scientific softwares. In this work, we have just implemented a universal interval type that may represent any specific interval type. Moreover requirements of constraint solvers have been taken into account in the design of arithmetic.

The next step will be to instantiate the traits for all known interval libraries. May CtlInterval be the interval library of reference in the scientific community, the specific C++ libraries being plugged in it? For this purpose, efficiency must be improved.

An interesting perspective concerns the dynamic use of interval libraries. For instance that would allow one to switch to a multi-precision library at runtime if more precision is required. Unfortunately the actual implementation does not support this feature since the traits technique is static in essence. A solution would be to create an abstract base class for all specific interval types associated with conversion mechanisms.

Several components of the constraint library have already been developed. Several traits associated with continuous constraints specify mathematical objects, like a field to be a parameter of a

polynomial ring. We believe that this approach is powerful for the development of mathematical softwares.

## Acknowledgements

The author thanks Frédéric Goualard for help on C++ programming and interval arithmetic, Martine Ceberio and the anonymous referees for valuable comments on early versions of this paper.

This work has been partly supported by the European project COCONUT (IST 2000-26063).

## References

[1] G. Alefeld, J. Herzberger, Introduction to Interval Computations, Academic Press, New York, 1983.
[2] F. Benhamou, D. McAllester, P. Van Hentenryck, CLP (Intervals) revisited, in: Proceedings of the International Logic Programming Symposium, Ithaca, NY, USA, MIT Press, Cambridge, MA, 1994, pp. 124–138.
[3] F. Benhamou, F. Goualard, L. Granvilliers, J.-F. Puget, Revising Hull and Box consistency, in: Proceedings of the International Conference on Logic Programming, MIT Press, Cambridge, MA, Las Cruces, USA, 1999, pp. 230–244.
[4] J.G. Cleary, Logical arithmetic, Future Comput. Systems 2 (2) (1987) 125–149.
[5] F. Goualard, JAIL—just another interval library, Technical Report, University of Nantes, 1999.
[6] IEEE, IEEE standard for binary floating-point arithmetic, Technical Report IEEE Std 754-1985, 1985. Reaffirmed 1990.
[7] L. Jaulin, M. Kieffer, O. Didrit, E. Walter, Applied Interval Analysis: With Examples in Parameter and State Estimation, Robust Control and Robotics, Springer, Berlin, 2001.
[8] O. Knüppel, BIAS—basic interval arithmetic subroutines, Technical Report, Technical University of Harburg, 1993. Available at http://www.ti3.tuharburg.de/knueppel/profil/index_e.html.
[9] R.E. Moore, Interval Analysis, Prentice-Hall, Englewood Cliffs, NJ, 1966.
[10] N.C. Myers, A new and useful template technique: "traits", C++ Report 7 (5) (1995) 32–35.
[11] N. Revol, F. Rouillier, The multiple precision floating-point interval library, Technical Report, ENS Lyon, 2001. Available at http://www.enslyon.fr/~nrevol/mpfi.html.
[12] D. Sam-Haroud, B. Faltings, Consistency techniques for continuous constraints, Constraints 1 (1996) 85–118.
[13] B. Stroustrup, The C++ Programming Language, Special Edition, Addison-Wesley, Reading, MA, 2001.
[14] Sun Microsystems, C++ interval arithmetic programming reference, Technical Report, Sun Microsystems, 2001. Forter Developer 6 Update 2.
[15] P. Van Hentenryck, L. Michel, Y. Deville, Numerica: A Modeling Language for Global Optimization, MIT Press, Cambridge, MA, 1997.