libieeep1788: A C++ Implementation of the IEEE Interval Standard P1788

Marco Nehmeier
Institute of Computer Science
University of Würzburg
Am Hubland
D 97074 Würzburg, Germany
Email: nehmeier@informatik.uni-wuerzburg.de

***Abstract*—In 2008 the IEEE interval standard working group P1788 has been founded to enlarge the acceptance of interval arithmetic. One of the main challenges for the working group was to cope with different approaches of interval arithmetic. Such as set based interval arithmetic or modal interval arithmetic, which have been developed during the last 50 years. For this reason a concept called *flavors* is defined by the standard. The concept specifies the behavior of the operations. Additionally, a so called decoration system is introduced to treat mathematical events like discontinuity or undefinedness of an expression for a given interval box.**

The C++ library *libieeep1788* tries to implement all features of the tentative interval standard in a clear and straightforward manner. The main goal is to have a faithful reference implementation of the standard and not to use dubious optimizations and "hacks". The main design concept of the library is to map the flavors concept one-to-one onto the implementation. To achieve this requirement the so called policy based class design is used.

In this paper we will present the key concepts of the faithful C++ library *libieeep1788* which will be accompanied by an overview of the preliminary IEEE P1788 interval standard.

I. Introduction

Floating point arithmetic is afflicted with rounding errors which can not be treated by a correction computed with paper and pencil. Further it is tedious, cumbersome and error-prone to investigate every single source of error. The computers must be enabled to check their results for correctness. This is mandatory for applications which require a certain kind of accuracy of the computations like chemical engineering and control theory as well as computer graphics and computer-aided design.

Interval arithmetic is a tool that can help in this situation. Instead of calculating with floating point numbers, i.e. approximations of real numbers, interval arithmetic computes rigorous bounds for "real" real numbers.

Note, however, that this guarantee is generally not obtained by only changing the data type from floating point to interval because dependencies between intervals lead to overestimation. Hence, new algorithms have to be defined and the underlying arithmetic has to compute rigorous bounds. This has been done in the last 50 years and we now have the interval Newton method, branch and bound algorithms for global optimization, self validating algorithms for linear and nonlinear equations, rigorous ODE solvers, and much more.

II. IEEE Interval Standard P1788

In 2008 the IEEE interval standard working group P1788 has been founded to enlarge the acceptance of interval arithmetic [1]. The presumable standard defines (set based) intervals as connected, closed, not necessarily bounded subsets of the reals:

$$[\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}$$

In this definition $\underline{x}$ can be $-\infty$, $\overline{x}$ can be $+\infty$, but the infinities never are members of an interval. In this paper we use $\mathbf{x}$ as a shorthand for $[\underline{x}, \overline{x}]$. The set of all intervals including the empty set is denoted as $\overline{\mathbb{IR}}$. The basic arithmetic operations are defined as powerset operations[1].

$$\mathbf{x} \bullet \mathbf{y} = \{x \circ y \mid x \in \mathbf{x}, y \in \mathbf{y}, \text{if it is defined}\}$$

The interval operations compute the interval hull of these sets. Continuous functions could be defined in a similar manner [2] and for discontinuous functions the hull of the continuous parts is computed. Since a computer representation of an interval uses floating point numbers for the bounds, directed rounding toward $-\infty$ or $+\infty$ is necessary to compute a true enclosure.

The enclosure of all real results of a basic operation or a function is the fundamental property of interval arithmetic and is called *inclusion property* or *Fundamental Theorem of Interval Arithmetic* (FTIA). Basically it means that an interval extension $\mathbf{f} : \overline{\mathbb{IR}}^n \to \overline{\mathbb{IR}}$ on an interval box $(\mathbf{x_1}, \ldots, \mathbf{x_n})$ contains the range of the corresponding real function $f : \mathbb{R}^n \to \mathbb{R}$ over this interval box [3]:

$$f(\mathbf{x_1}, \ldots, \mathbf{x_n}) \subseteq \mathbf{f}(\mathbf{x_1}, \ldots, \mathbf{x_n})$$

A. Flavors

One of the main challenges of the working group was to cope with different approaches of interval arithmetic which have been developed during the last 50 years [4]. On the one hand we have the classical Moore interval arithmetic [5] and on the other hand the more advanced and modern theories like set based interval arithmetic, containment sets [6], or modal interval arithmetic [7].

It was quickly decided by the working group that the classical Moore arithmetic which only allows bounded and

[1]Note that monotonicity properties could be used to define the result of an interval operation or function only using the bounds of the input intervals.

nonempty intervals is not sufficient for an interval arithmetic standard that has to master all the different applications of interval arithmetic. Also containment sets was not a big topic for the working group to be part of the standard. But it was a lively debate between the advocates of set based interval arithmetic and modal interval arithmetic to decide on which theory the standard should be based on. Neither of both approaches is the magic bullet of interval arithmetic and comes with their advantages and disadvantages.

For this reason a concept called *flavors* [8] is defined by the standard which specifies the behavior of the operations. In this connection it is important that the operations (interface of an implementation) are always the same, the flavors only specify how the interval operations should behave in the required context. This is comparable to a well known design pattern called strategy pattern or policy pattern [9]. This allows to have a standard capturing different interval behaviors and theories with clearly defined extension points.

As the main condition for an individual flavor it is required that this flavor extends the classical Moore arithmetic which is commonly fulfilled by the different interval arithmetic theories. And additionally a standard compliant implementation has to provide at least one of the flavors defined in the standard. Up to now the preliminary standard contains a specification for set based interval arithmetic but an introduction of a flavor for modal intervals is planned.

B. Decorations

Typically (set based) interval arithmetic is seen as an exception free calculus [10] but effectively there are some situations like division by zero or an evaluation outside a function domain where "exceptions" can happen. Like the IEEE 754 standard for floating point arithmetic [11] the upcoming standard for interval arithmetic requires an execution without interruption. Formally, a global flag was commonly used in implementations to comply with such a requirement but nowadays multithreading is almost everywhere and the usage of global flags is burdened with a lot of drawbacks [12].

To cope with the request for an interruption free and thread safe interval standard which can treat mathematical events like discontinuity or undefinedness of an expression for a given interval box a so called *decoration* system $\mathbb{D}$ is introduced [1], [13].

Basically a decoration is one of the five properties shown in Table I which is combined with an interval to a *decorated interval*[2]. The decoration of an decorated interval then can be seen as a property storing the "history" of the computation. This "history" is realized by a propagation or quality order of the decorations which equals the order in Table I from `com` (good) to `ill` (bad) [1]. The decoration part of the result of a function is then determined by returning the worst decoration of the inputs of the function together with the decoration computed by applying Table I onto the bare intervals of the input of the function. Hence, the evaluation of interval functions with decorated intervals is an intuitional process which is driven by the propagation order of Table I. And also the construction of a decorated interval out of a bare interval

[2]Note that the interval part of an decorated interval is called *bare interval*.

TABLE I. THE SET $\mathbb{D}$ OF DECORATIONS [3]

Value	Short description	Definition
`com`	common	$\mathbf{x}$ is a bounded, nonempty subset of Dom(f); f is continuous at each point of $\mathbf{x}$; and the computed interval $f(\mathbf{x})$ is bounded.
`dac`	defined & continuous	$\mathbf{x}$ is a nonempty subset of Dom(f), and the restriction of f to $\mathbf{x}$ is continuous.
`def`	defined	$\mathbf{x}$ is a nonempty subset of Dom(f).
`trv`	trivial	always true.
`ill`	ill-formed	Not an Interval; formally Dom(f) $= \emptyset$.

$\mathbf{x}$ is managed by Table I. The decoration is simply set to `com` if $\mathbf{x}$ is nonempty and bounded, to `dac` if $\mathbf{x}$ is nonempty and unbounded, and to `trv` if $\mathbf{x}$ is empty [1]. Note that the decoration `ill` is used for marking an ill-formed interval as *NaI* (Not an Interval) which only can happen during the construction of a decorated interval using ill-formed or illegal numbers for the bounds.

With the knowledge about the propagation of decorations, the introduction of decorated intervals is a big surplus for the user. Especially the renunciation of global flags and the usage of "expression local" flags simplifies the observation of necessary properties like the continuity of a function to fulfill the Brouwer's fixed point theorem [12].

C. Functions and Operations

Obviously an interval standard has to specify a generally admitted set of operations. Essentially there was a consensus about the required functionality in the working group. But in some cases like the exact dot product it was a debate on principles [14], [15], [16].

Mainly the specification captures the same functions as the IEEE 754 standard for floating point arithmetic [11], [17], of course with an interval arithmetic specific adaption, but also completely new functions were added.

1) Required operations in all flavors: With the introduction of the flavors concept it is necessary to have a common set of operations which has to be supported by all compliant flavors. Basically this are the most important arithmetic operations known form the IEEE 754 standard [11], [17]:

Basic operations: `neg, add, sub, mul, div, recip, sqr, sqrt, fma`

Case function: `case`

Power functions: `pown, pow, exp, exp2, exp10, log, log2, log10`

Trigonometric/hyperbolic functions: `sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, asinh, acosh, atanh`

Integer functions: `sign, ceil, floor, trunc, roundTiesToEven, roundTiesToAway`

Absmax functions: `abs, min, max`

As described above for the transition from a real function (or floating point function) to an interval function the convex hull of the powerset is computed.

Additionally to the common arithmetic functions the cancellative addition `cancelPlus` and subtraction `cancelMinus` are required. Thereby the function `cancelMinus`$(\mathbf{x}, \mathbf{y})$ returns a unique interval $\mathbf{z}$ such that $\mathbf{y} + \mathbf{z} = \mathbf{x}$ follows as long as $\mathbf{x}$ is as least as width as $\mathbf{y}$ [18]. `cancelPlus`$(\mathbf{x}, \mathbf{y})$ is equivalent to `cancelMinus`$(\mathbf{x}, -\mathbf{y})$.

The other required functions listed below are self-explanatory:

Set operations: `intersection, convexHull`

Constructors: `numsToInterval, textToInterval`

Numeric functions: `inf, sup, mid, wid, rad, mag, mig`

Boolean functions: `equal, subset, interior, disjoint`

2) Operations in the set based flavor: Besides the commonly required functions defined for all flavors which are all applicable onto classical Moore intervals the set based flavor adds its own additional functions. And obviously with the introduction of empty or unbounded intervals the requirement for `isEmpty` and `isEntire` is very useful. Also the function `intervalToText` as a counterpart to the commonly required `textToInterval` is a logical requirement to specify the IO functionality for set based intervals.

Another very useful function which is required for the set based flavor is `divToPair`$(\mathbf{x}, \mathbf{y})$ which delivers a tuple of two intervals[3] [19]. Simply in the case of a division by zero `divToPair`$(\mathbf{x}, \mathbf{y})$ returns the disjoint intervals of $\mathbf{x} \,/\, (\, \mathbf{y} \cap [-\infty, 0[\,)$ and $\mathbf{x} \,/\, (\, \mathbf{y} \cap]0, \infty]\,)$ otherwise the common division is performed and the second element of the tuple is left empty.

One of the most popular applications for interval arithmetic is solving constraint problems for which the so called *reverse-mode* is very feasible. The reverse interval extension φRev for a unary function φ is an interval fulfill the following requirement [1], [20]:

$$\varphi\text{Rev}(\mathbf{c}, \mathbf{x}) \supseteq \{x \in \mathbf{x} \mid \varphi(x) \text{ is defined and in } \mathbf{c}\}$$

For a binary function φ than there are naturally two reverse interval extensions φRev1 and φRev2, see [1], [20] for more details.

The set based flavor requires the following set of reverse functions for the corresponding (forward) functions:

For unary functions: `sqrRev, recipRev, absRev, pownRev, sinRev, cosRev, tanRev, coshRev`

For binary functions: `mulRev, divRev1, divRev2, powRev1, powRev2, atan2Rev1, atan2Rev2`

Since the comparison of two intervals are not as straightforward as the comparison of two numbers[4] the standard defines some more specific boolean functions which are required by the set based flavor, see Table II [1], [21]. In addition a function inspired by J.F. Allen who defined a relation in a temporal logic setting [22] is recommended for the set based flavor. This overlapping function compares two intervals $\mathbf{x}$ and $\mathbf{y}$ and returns one out of 16 states describing the relation between them [23]. Figure 1 illustrates the functionality of the overlapping function comparing two nonempty intervals capturing the 13 different states described in [22]. For handling of empty intervals 3 additional states `bothEmpty`, `firstEmpty` and `secondEmpty` are specified.

TABLE II. ADDITIONAL COMPARISON FUNCTIONS REQUIRED BY THE SET BASED FLAVOR.

Name	Symbol	Definition
`less`	$\mathbf{x} \leq \mathbf{y}$	$\forall_{x \in \mathbf{x}} \exists_{y \in \mathbf{y}}\, x \leq y \wedge \forall_{y \in \mathbf{y}} \exists_{x \in \mathbf{x}}\, x \leq y$
`precedes`	$\mathbf{x} \preceq \mathbf{y}$	$\forall_{x \in \mathbf{x}} \forall_{y \in \mathbf{y}}\, x \leq y$
`strictLess`	$\mathbf{x} < \mathbf{y}$	$\forall_{x \in \mathbf{x}} \exists_{y \in \mathbf{y}}\, x < y \wedge \forall_{y \in \mathbf{y}} \exists_{x \in \mathbf{x}}\, x \leq y$
`strictPrecedes`	$\mathbf{x} \prec \mathbf{y}$	$\forall_{x \in \mathbf{x}} \forall_{y \in \mathbf{y}}\, x < y$

Beside the required functions the set based flavor recommends a set of additional functions.

Recommended elementary functions: `rootn, expm1, exp2m1, exp10m1, logp1, log2p1, log10p1, compoundm1, hypot, rSqrt, sinPi, cosPi, tanPi, asinPi, acosPi, atanPi, atan2Pi`

Recommended boolean functions: `isCommonInterval, isSingleton, isMember`

Recommended slope functions: `expSlope1, expSlope2, logSlope1, logSlope2, cosSlope2, sinSlope3, asinSlope3, atanSlope3, coshSlope2, sinhSlope3`

Recommended complete arithmetic[5]*:*
`convert, completeAdd, completeSub, completeMulAccum, completeDotProduct`

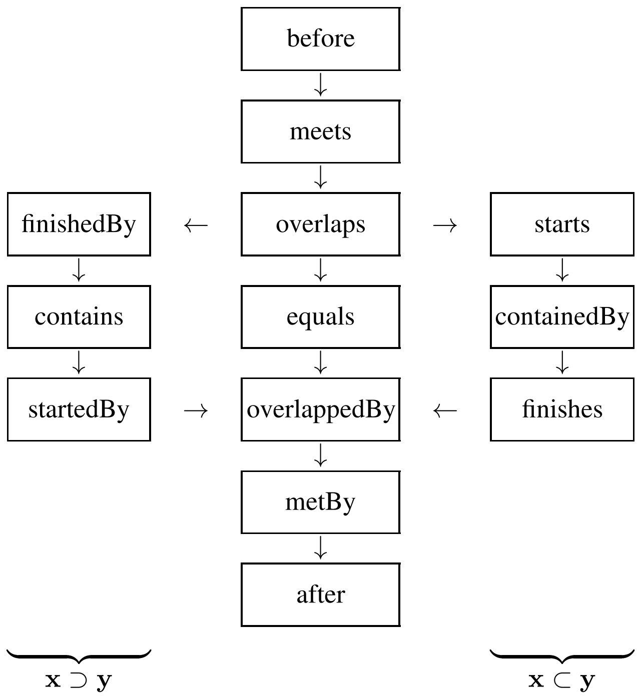

Fig. 1. Overlapping function for two nonempty intervals $\mathbf{x}$ and $\mathbf{y}$: shifting $\mathbf{x}$ from left to right

[3]`divToPair` is typically used for the interval Newton method to cope with nonhomogeneous functions.

[4]E.g. does $\mathbf{x} < \mathbf{y}$ mean that $sup(\mathbf{x}) < inf(\mathbf{y})$ or is $\exists_{x \in \mathbf{x}} \exists_{y \in \mathbf{y}}\, x < y$ enough?

[5]See [1], [14] for details.

III. C++ Library libieeep1788

The C++ library *libieeep1788* tries to implement all features of the tentative interval standard in a clear and straightforward manner. The main goal is to have a faithful reference implementation of the standard and not to use dubious optimizations and "hacks".

A. Flavors and the policy based class design

The main design concept of the library is to map the flavors concept one-to-one onto the implementation. To achieve this requirement the so called policy based class design [24] which has been successfully applied to other interval libraries [25] is used. In detail the implementation is divided into two parts, the frontend or interface which specifies all the required operations and the backend or "flavors" which implements the concrete operations. These two parts are joint together using C++ templates and template metaprogramming [26]. The benefit of this approach is that the flavors are easily exchangeable and the library can be used as a toolbox for different flavors or for optimized implementations.

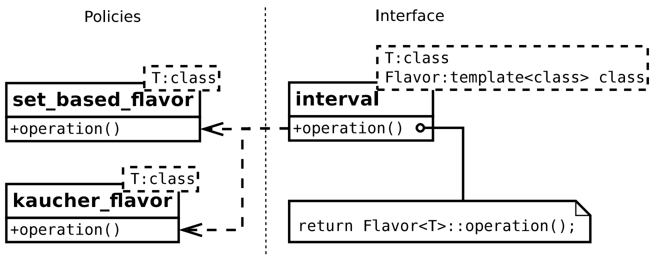

Fig. 2. Basic concept of the policy based class design for the realization of flavors

Figure 2 shows the basic concept of the policy based class design on a simplified example. The interface is realized by the generic class `interval` with its two template parameters `T` and `Flavor`. In this case `T` is the type used for the bounds of the interval, e.g. `double`. The template-template parameter `Flavor` specifies the concrete policy which implements the functionality of the flavor. All the operations or functions of the interface `interval` are only wrappers around a static call of the corresponding function of the specified policy class `Flavor`. Note that the policy class `Flavor` is distincted with the type `T`, see Fig. 2.

The benefit of the policy based class design in contrast to the classical object oriented strategy pattern [9] is the better optimization at compile time. All the information of the generic types is available at compile time. Hence, the compiler is able to smelt the the interface `interval` and the policy `Flavor` into one unique class. A function call of the interface will than directly perform the corresponding function of the policy class without the detour of a virtual function table.

Additionally the usage is quiet easy. In the following example there is first a type alias `I` defined for a set based interval with a generic bound type. Then this type alias can be used in a straightforward manner by specifying the required bound.

```
template<typename T>
using I = interval<T, set_based_flavor>;

I<double> x(1.0, 2.0), y(1.0, 1.0);

I<double> z = sqrt(x) + y;
```

B. Library design

Obviously an implementation of the library *libieeep1788* is a little more complicate then the simple example in Sec. III-A. First of all there are two basic types necessary, `interval` and `decorated_interval` which have a big similarity. The bigger part of the interface of `interval` and `decorated_interval` is identical. The only difference is that they work with `intervals` or `decorated_intervals`, respectively. An implementation with two completely distinctive types for bare and decorated intervals will lead to a lot of code duplication which is hard to handle. Especially for an implementation of a preliminary standard which tries to keep track of all revisions.

In this case the classical objectoriented programming can cause a false conclusion if the class `decorated_interval` is derived from the class `interval`. At the first sight a derivation of `decorated_interval` from `interval` is logically because a decorated interval is a bare interval with a decoration. But the danger of this approach comes with user-defined functions. E.g. a user defines a function `func` only for the type `interval`. Because of polymorphism the decoration of the decorated interval `x` will be thrown away and the function `func` will be performed with a bare interval. Then the result (bar interval) of `func` will be transformed back into a decorated interval `y`. Hence, the history of the computation is not completely tracked and could be wrong.

```
template<typename T>
using DI = decorated_interval<T,
             set_based_flavor>;
DI x = ...

DI y = func(x);
```

To avoid this problem we decided to introduce a class `base_interval` which centralizes the shared functionality and both `interval` and `decorated_interval` are derived from `base_interval`, see Fig. 3.

The main idea behind the `base_interval` is that it has a private member `rep` representing the internals of an interval or decorated interval. The type of this member is specified by the template parameter `RepType`. Typically this representation types are tuples like `std::pair<T,T>` or `std::pair< std::pair<T,T>,decoration>` which are distincted from the typedefs `Flavor<T>::representation` or `Flavor<T>::dec_representation`, respectively. This approach has the advantage that the internal representation can be completely specified by the derived class `interval` or `decorated_interval`. The other new template parameter `ConcreteInterval` typifies the concrete type of the derived class. This "trick" allows the base class `base_interval` to create instances of its subclasses which are usually unknown to the base

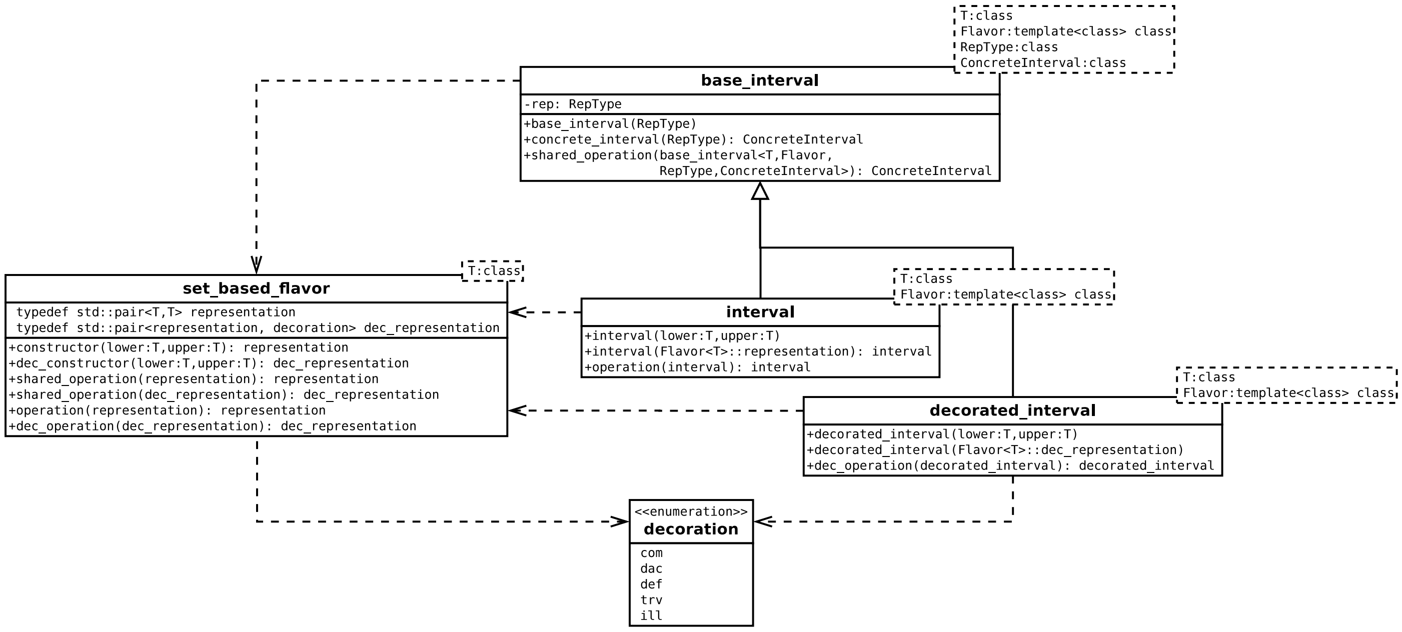
T:class
Flavor:template<class> class
RepType:class
ConcreteInterval:class
base_interval
-rep: RepType
+base_interval(RepType)
+concrete_interval(RepType): ConcreteInterval
+shared_operation(base_interval<T,Flavor, RepType,ConcreteInterval>): ConcreteInterval
T:class
set_based_flavor
typedef std::pair<T,T> representation
typedef std::pair<representation, decoration> dec_representation
+constructor(lower:T,upper:T): representation
+dec_constructor(lower:T,upper:T): dec_representation
+shared_operation(representation): representation
+shared_operation(dec_representation): dec_representation
+operation(representation): representation
+dec_operation(dec_representation): dec_representation
T:class
Flavor:template<class> class
interval
+interval(lower:T,upper:T)
+interval(Flavor<T>::representation): interval
+operation(interval): interval
T:class
Flavor:template<class> class
decorated_interval
+decorated_interval(lower:T,upper:T)
+decorated_interval(Flavor<T>::dec_representation)
+dec_operation(decorated_interval): decorated_interval
<<enumeration>>
decoration
com
dac
def
trv
ill

Fig. 3. Sketch of a simplified *libieeep1788* class design

class. Therefore, the class `interval<T,Flavor>` is publicly derived from `base_interval<T,Flavor, Flavor<T>::representation,interval<T,Flavor >>` and `decorated_interval<T,Flavor>` is publicly derived from `base_interval<T,Flavor, Flavor<T>::dec_representation, decorated_interval<T,Flavor>>`.

Basically, in the view of the class design, the library has to handle three different tasks:

1) Creation of new instances of type `interval` or `decorated_interval`: If a new instance of the type `interval` is requested by the call of the constructor `interval(lower,upper)` the two parameters `lower` and `upper` are passed to the function `Flavor<T>::constructor` which returns an initialized instance of the type `Flavor<T>::representation`. This representation is then used to call the constructor of the base class `base_interval` to initialize the private member `rep`. The creation of decorated intervals follows the same procedure, only `Flavor<T>::dec_constructor` and `Flavor<T>::dec_representation` are used[6]. Therefore, the whole representation and creation of an interval is controlled by the flavor. The classes `base_interval`, `interval`, and `decorated_interval` are only skeletons and all the functionality is implemented by the policy class `Flavor`.

2) Call of a shared operation: The call of a shared operation is almost similar to the creation of intervals. The difference is that the shared operation is implemented in the base class `base_interval` which uses polymorphism to work with subclasses like `interval` or `decorated_interval`. If a shared operation, e.g `shared_operation` in Fig. 3, is called with a parameter of a subclass like `interval` the internal representation `rep` is then passed to the corresponding function `Flavor<T>::shared_operation`. Note that in Fig. 3 there are two implementations of `Flavor<T>::shared_operation`. One working with `representation` and one working with `dec_representation`. Hence, the polymorphism of C++ is used to choose the right function automatically. The result of the function `Flavor<T>::shared_operation` is than used to create the right return type for the shared operation by invoking the function `concrete_interval` from the class `base_interval` which is than forwarded to the constructor of `ConcreteInterval`.

3) Call of an interval or decorated interval specific operation: The call of an interval or decorated interval specific operation follows the same procedure as the construction of intervals or decorated intervals. A specific operation like `dec_operation` is called and the internal representation `rep` is passed to the corresponding function `Flavor<T>::dec_operation`. The returned instance of the representation type is than used to initialize the return type of the operation. Specific operations for the type `interval` are implemented in the same manner.

C. Set based flavor implementation

At the moment an implementation of the set based flavor is under heavy development to keep track with the progress of the working group. This implementation is based on MPFR [27] to achieve the main requirement for a faithful reference implementation.

IV. Conclusion

In this paper we presented the key concepts of the preliminary IEEE P1788 interval standard as well as the faithful C++ library *libieeep1788*. The main goal of this library is to have

[6]Note that decoration is determined from the two parameters `lower` and `upper` according to the specification in Sec. II-B.

a straightforward implementation of the IEEE P1788 interval standard. Special attention was turned on the realization of the flavors concept. In this case the policy based class design was a perfect match offering flexibility and adaptability. Hence, our approach allows to have different flavor implementations and could also be used as a toolbox for other developers.

References

[1] IEEE P1788, "IEEE Interval Standard Working Group - P1788." [Online]. Available: http://grouper.ieee.org/groups/1788/

[2] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, *Applied Interval Analysis*, 1st ed. Springer, Sep. 2001.

[3] J. Pryce and C. Keil, "P1788/D8.4, Draft Standard For Interval Arithmetic," IEEE Interval Standard Working Group - P1788, February 2014, in [1].

[4] T. Sunaga, "Theory of an interval algebra and its application to numerical analysis [reprint of res. assoc. appl. geom. mem. 2 (1958), 2946]," *Japan Journal of Industrial and Applied Mathematics*, vol. 26, no. 2-3, pp. 125–143, 10 2009.

[5] R. Moore, *Interval analysis*, ser. Prentice-Hall series in automatic computation. Prentice-Hall, 1966.

[6] J. D. Pryce and G. F. Corliss, "Interval Arithmetic with Containment Sets," *Computing*, vol. 78, no. 3, pp. 251–276, 2006.

[7] E. Kaucher, "Interval Analysis in the Extended Interval Space IR," in *Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis)*, ser. Computing Supplementum, G. Alefeld and R. Grigorieff, Eds. Springer Vienna, 1980, vol. 2, pp. 33–49.

[8] J. Pryce, "Motion 36 - Flavors," IEEE Interval Standard Working Group - P1788, June 2012, in [1].

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[10] U. Kulisch, *Computer Arithmetic and Validity: Theory, Implementation, and Applications*, 2nd ed., ser. De Gruyter Studies in Mathematics. Berlin, Germany: De Gruyter, 2013.

[11] IEEE 754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*. New York, NY, USA: IEEE, Aug. 2008.

[12] M. Nehmeier and J. Wolff von Gudenberg, "filib++, expression templates and the coming interval standard," *Reliable Computing*, vol. 15, no. 4, pp. 312–320, Jul. 2011.

[13] J. Pryce, "Motion 42 - Decoration System," IEEE Interval Standard Working Group - P1788, December 2012, in [1].

[14] U. Kulisch, "Motion 9 - exact dot product," IEEE Interval Standard Working Group - P1788, October 2009, in [1].

[15] J. Pryce, "Motion 45 - dot product," IEEE Interval Standard Working Group - P1788, June 2013, in [1].

[16] U. Kulisch, "Motion 50 - edp without ca," IEEE Interval Standard Working Group - P1788, September 2013, in [1].

[17] J. Wolff von Gudenberg, "Motion 10 - Elementary Functions," IEEE Interval Standard Working Group - P1788, October 2009, in [1].

[18] N. Hayes, "Motion 12 - inner addition and subtraction," IEEE Interval Standard Working Group - P1788, March 2010, in [1].

[19] J. Wolff von Gudenberg, "Motion 43 - divPair," IEEE Interval Standard Working Group - P1788, April 2013, in [1].

[20] M. Nehmeier, "Motion 11 - Basic Reverse Arithmetic Operations," IEEE Interval Standard Working Group - P1788, January 2010, in [1].

[21] U. Kulisch, "Motion 13 - Comparison Relations," IEEE Interval Standard Working Group - P1788, August 2010, in [1].

[22] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26/11/1983, pp. 832–843, 1983.

[23] M. Nehmeier and J. Wolff von Gudenberg, "Interval comparisons and lattice operations based on the interval overlapping relation," in *Proceedings of the World Conference on Soft Computing 2011 (WConSC'11)*, May 2011. [Online]. Available: http://wwwi2.informatik.uni-wuerzburg.de/publications/paperNehmeierWConSC11.pdf

[24] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[25] M. Nehmeier, "Interval arithmetic using expression templates, template meta programming and the upcoming c++ standard," *Computing*, vol. 94, pp. 215–228, 2012, 10.1007/s00607-011-0176-6. [Online]. Available: http://dx.doi.org/10.1007/s00607-011-0176-6

[26] T. Veldhuizen, "Using c++ template metaprograms," *C++ gems*, pp. 459–473, 1996.

[27] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, pp. 13:1–13:15, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468