

TURTLE++ – A CIP-Library for C++

Petra Hofstedt and Olaf Krzikalla

Technical University of Berlin
ph@cs.tu-berlin.de krzikalla@gmx.net

Abstract. This article introduces the TURTLE++ library which combines constraint-based and imperative paradigms and enables in this way *constraint imperative programming* (CIP) with C++. Integrating CIP into C++ allows to exploit the powerful expressiveness of the CIP paradigm within a language widely used and accepted in practice. We discuss the main concepts and implementation and illustrate programming with TURTLE++ by means of typical examples.

1 Introduction

Using *imperative* programming languages the user describes the precise steps to compute a solution for a given problem. In contrast, in *declarative* languages the programmer simply gives a problem specification, while the computation mechanism of the language is responsible to provide solutions. In constraint programming – as a member of the declarative paradigm – constraints, i.e. formulae of first order predicate logic, are used for the problem description, specifying the problem variables, their properties and relationships. These constraints are handled by constraint solvers provided by the languages to obtain solutions.

The imperative and the constraint-based paradigm are qualified for different tasks: Temporal orders of events are best specified using imperative concepts, while constraints are better suited for search problems or partially unspecified tasks. *Constraint imperative programming (CIP)* [4] allows to combine both paradigms and thus to use their concepts simultaneously within one language.

The following example origins from [8]. Program 1.1 uses a pure imperative approach to describe a graphical element, e.g. in a user interaction, which can be dragged with the mouse inside certain borders. Pressing a mouse-button the vertical coordinate y of the graphical element is adapted within a minimum and a maximum range. This description cleanly reflects the temporal – i.e. imperative – behaviour but requires also declarative properties, like the range restrictions, to be expressed imperatively. They are ensured by explicit tests which must be performed in each pass within a while-loop as long as the button is pressed.

In contrast, Program 1.2 gives an according CIP implementation, already in the syntax of TURTLE++. The initial instantiation of y by a so-called preferred value (cf. Sect. 2.3) `mouse.y` gives an orientation for the optimization wrt. the required constraints. It will be overwritten by `border.min` resp. `border.max` in case the mouse exceeds a border. Program 1.2 is not only shorter, but expresses

Program 1.1 A user interface example in the imperative style

```
while (mouse.pressed)
{
    // message processing is left out
    int y = mouse.y;
    if (y > border.max)
        y = border.max;
    if (y < border.min)
        y = border.min;
    draw_element (fix_x, y, graphic);
}
```

Program 1.2 The user interface example using CIP

```
while (mouse.pressed)
{
    constrained<int> y = mouse.y;
    require (y >= border.min && y <= border.max);
    draw_element (fix_x, y(), graphic);
}
```

the relationship between the border-object and the y-coordinate in exactly the way a programmer would think about it.

TURTLE++ [7] is a C++ *library* which inherited the main ideas of the *language* Turtle¹ [5] but was extended for smooth integration into C++. Meanwhile TURTLE++ has been refined and extended by new constructs for a more convenient handling of value assignment (fixing/unfixing, cf. Sect. 2.4) and by a boolean solver the two of which together allow to nicely express new kinds of problem descriptions as demonstrated later on.

We think that the development of TURTLE++ may support both the spreading and further development of the CIP approach since C++ is a widely used and in practice accepted language whose semantics remain untouched by TURTLE++, while the application programmer is allowed to use the benefits of constraint programming in his or her professional work.

This paper deals with the TURTLE++ approach of integrating constraints and constraint solvers with imperative language concepts. In Sect. 2 we introduce and explain the main concepts of TURTLE++ and shortly touch its implementation. Section 3 is dedicated to programming with our library. We draw a conclusion and discuss related and future work in Sect. 4.

¹ a CIP language as well developed at the Technical University of Berlin

Program 2.1 Variables and a constraint statement

```
int x = 0;           // a normal variable
constrained<int> y;  // a constrainable variable
require (y <= x);   // a constraint statement
```

2 The Main Concepts of TURTLE++

The main concepts of the C++ *library* TURTLE++ are *constrainable variables*, *constraint statements* and *user-defined constraints*.

2.1 Constrainable Variables

These are variables whose values can be determined by placing constraints on them. A constrainable variable is declared by giving it a **constrained** type. In Program 2.1 *y* is a constrainable variable, while *x* is a normal one.

Most of the time a constrainable variable acts like a normal variable: it can be used in expressions and as a function argument. Only in a constraint statement they differ from their normal counterparts. A normal (or imperative) variable is treated like a constant, but a constrainable (or declarative) variable acts like a variable in the mathematical sense, and the constraint solver may change its value in order to satisfy all invoked constraints.

2.2 Constraint Statements

Whenever declarative and imperative languages are combined, one of the main issues is the interaction of the integrated declarative concepts with the imperative model of time. In TURTLE++ this is solved by introducing a lifetime for constraints using the keyword **require** as shown in Program 2.1.

During program execution the run-time system manages a constraint store which contains the currently required constraints. When a **require** is reached during the execution of the program, the given constraint is added to this store and taken into account for further computations - its lifetime starts.

The **require**-statement returns a handle to manage the constraints lifetime. If the return value is ignored (as e.g. in Program 2.1), the imposed constraint exists as long as all its constrainable variables. Otherwise, the lifetime of the constraint is also limited to the lifetime of the returned constraint handle. Constraint handles are, thus, useful especially when imperative execution flow elements (e.g. loops) and constraints are used together, as shown in Program 2.2. Here, after every pass through the **while**-loop the constraint (**a >= b**) is removed. If we had instead foregone the handle and simply used a single **require**-statement, then each pass *i* of the **while**-loop would add a new constraint **a >= b_i** to the store.

Besides, constraint handles allow to explicitly overwrite a constraint (as will be shown in Program 3.1) and to manipulate its strength. Constraints can be

Program 2.2 A handle restricting the lifetime of its constraint $a \geq b$

```
constrained<int> a;
... //setup some initial constraints over a
while (not_done())
{
    int b = compute_something();
    constraint_handle<int> z = require (a >= b);
    ...
//leaving the scope of z removes (a >= b) from the store
}
```

labelled with strengths in constraint hierarchies which are taken into consideration during the solving process and which allow to deal with over- and underconstrained problems. When a constraint is annotated with a strength, e.g. **strong** or **weak**, it is added to the store with the given strength, otherwise with the strongest strength **mandatory**.

The run-time system actually does not handle a single constraint store but several sub-stores. This allows not just search over constraint disjunctions (which may appear in **require** statements as well as conjunctions, see [7]) but a more appropriate and convenient handling of independent sets of constraints. A constraint sub-store is the set of all constraints over a set of constrainable variables, which are linked. Two variables x and y are linked, if they either both appear in one constraint or if x appears in a constraint containing a constrainable variable linked to y . This allows to maintain independent sets of variables.

2.3 Obtaining Values from Constrainable Variables

The function call operator **operator>() const** is overloaded to obtain a constrainable variable value conveniently. When this operator is invoked, e.g. by

```
std::cout << a();
```

the constraint solver is started to determine a value of the appropriate variable, here a , satisfying all constraints over a . When the store is overconstrained and no value can be determined, an exception of type **overconstrained_error** (derived from **std::logic_error**) is raised.

But more often underconstrained situations occur. For this purpose **TURTLE++** supports a preferred value which can be assigned to a constrainable variable. If it turns out that, while solving a store, more than one solution exists for a certain constrainable variable, a solution closest to the preferred value (if possible for the domain under consideration) is taken. An example is given by Program 2.3: For the variable a there exist infinitely many solutions. Thus, the value 2.5 is chosen which is closest to the preferred value 3 wrt. the linear constraint ($a \leq 2.5$).

Even if a preferred value can be seen as a weakest constraint to a certain degree, it actually isn't. It does neither impose any constraint nor influence

Program 2.3 Obtaining a value

```
constrained<double> a (3); // assigns a preferred value
require (a <= 2.5);
std::cout << a(); // prints 2.5
```

Program 2.4 When computing a value for **b** variable **a** is *implicitly fixed*

```
constrained<int> a (2), b (0);
require (a == b);
std::cout << a(); // prints 2
std::cout << b(); // also prints 2
```

calculations inside the constraint store in any other way. The concept of preferred values was mainly introduced to make programs easier and more intuitively predictable by providing a means to define the result of a transfer from a non-deterministic constrained variable to a deterministic imperative variable. In addition the concept is useful to formulate optimization problems as shown in Sect. 3.3.

2.4 Implicit Fixing

TURTLE++ delays the computation of a value for a constrainable variable until a read-action to the variable occurs. Once a value is determined for a constrainable variable, this value must be taken into account for further calculations to ensure consistency of the imposed constraints.

Consider e.g. Program 2.4. The output of **a** forces the computation of a value for this constrainable variable. This assignment must be taken into consideration in the following computation and, thus, TURTLE++ adds *implicitly* a new constraint **a == 2** to the store. This implicit addition of a constraint variable **== value** to the store with the aim to ensure consistency for further calculations is called *implicit fixing*.

Without implicit fixing the value of **b** would be evaluated to 0 and hence violate the required constraint **a == b**. Due to this important side effect the evaluation order of constrainable variables must be carefully considered, e.g. if the third and fourth lines were exchanged, both would print 0.

An implicit fix is not immediately added to the constraint sub-store but kept in a delay store inside the sub-store. If only one implicit fix exists in a sub-store, and the same variable is evaluated again, the fix is erased before the re-evaluation (and later in this process a new fix is added). E.g. in Program 2.5 in each pass of the **for**-loop the just entered value of **j** is assigned as the preferred value of **a** and is finally printed. While **a** gets implicitly fixed to **j**, **b** remains "untouched". Thus, the implicit fix of **a** can be removed in the next iteration and eventually the next value entered for **j** is printed.

Program 2.5 Implicit fixing and unfixing

```
constrained<int> a (2), b (0);
require (a == b);
for (int i = 0; i < 3; ++i)
{
    int j;
    std::cin >> j;
    a = j;
    std::cout << a();    // prints j
}
```

If more than one implicit fix exists, always all must be taken into account.

Sometimes, one wants to fix a constrainable variable *explicitly*, which is possible just by the simple statement `require (variable == value)`.

Implicit fixes must be considered carefully, especially if more than one variable is evaluated inside a loop. That's why a constrainable variable can be unfix explicitly via the member function `unfix()` for one variable and `unfix_all()` for all variables in a sub-store, resp. (see also Program 3.1).

2.5 User-defined Constraints and Dynamic Expressions

Often the declarative power of expression templates is sufficient to express the constraints in a compact and readable manner. But some constraints are so common that they deserve their own name. Such user-defined constraints can be generated using the function template `build_constraint`, which takes a constraint just like `require`, but only builds the internal representation of the given expression without adding it to the constraint store. The naming of complex static expressions enhances the readability of a program. Program 2.6² shows the definition and usage of a domain constraint.

Besides this, it is often required to create constraints dynamically. Therefore TURTLE++ provides a generic class `dynamic_expr`, which can be assigned an expression and which itself can be part of an expression. In the latter case the dynamic expression expands to its contents.

An example is given by Program 2.7. The constraint `order` is *dynamically* generated over a vector `v` during a **for**-loop where the dynamic expression `expr` is stepwise extended expanding its previous contents by new expressions.

Dynamic expressions are very useful for establishing constraints over a previously unknown number of constrained variables as in this example. Note, that this particular example could be expressed as well imperatively by a sequence of `require`-statements as shown in Program 2.8. While this is possible because we express a *conjunction* of constraints, other kinds of logic operations like *disjunction* or *implication* actually require the usage of dynamic expressions. I. e.,

² In upcoming examples we will rely on the definition of `int_c` as displayed here without explicitly repeating it again.

Program 2.6 User-defined domain constraint

```
typedef constrained<int> int_c;

constraint_solver<int>::expr dom (const int_c& x, int l, int r)
{
    return build_constraint (x >= l && x <= r);
}

int_c a, b;
require (dom (a, 0, 9));
require (dom (b, -1, 1));
```

Program 2.7 An ordered array using dynamic expressions

```
dynamic_expr<int> order (std::vector<int_c>& v)
{
    dynamic_expr<int> expr = (true);
    for (int i = 1; i < v.size(); ++i)
        expr = (expr && v[i-1] <= v[i]);
    return expr;
}

int main()
{
    std::vector<int_c> v;
    // ... populate v

    require (order (v));           // declarative style:
                                   // order is a constraint
}
```

Program 2.9 demonstrates the dynamic generation of a disjunction of constraints which ensures that at least one variable of the vector v is equal to 1.

Comparing Program 2.7 and Program 2.8 again, we think, that moreover, the usage of dynamic expressions improves the declarative expressiveness of the source code.

2.6 Constraint Solvers

TURTLE++ currently comes with two solvers: a solver for linear arithmetics based on the simplex method and a simple search-based boolean solver. Their usage is demonstrated in Sect. 3 by means of examples. Furthermore, the user can extend TURTLE++ by new solvers as described at [1].

Since the interface enables the implementation of a constraint solver responsible for more than one value type, hybrid domains are possible. Even if there is no solver for hybrid constraints available, user-defined constraints may nonetheless

Program 2.8 An ordered array without dynamic expressions

```
void require_order (std::vector<int_c>& v)
{
    for (int i = 1; i < v.size(); ++i)
        require (v[i-1] <= v[i]);
}

int main()
{
    std::vector<int_c> v;
    // ... populate v

    require_order (v);           // imperative style:
                                // require_order is a function
}
```

Program 2.9 A disjunctive dynamic constraint

```
dynamic_expr<int> contains (std::vector<int_c>& v, int x)
{
    dynamic_expr<int> expr = (false);
    for (int i = 0; i < v.size(); ++i)
        expr = (expr || v[i] == x);
    return expr;
}

int main()
{
    std::vector<int_c> v;
    // ... populate v
    require (contains (v, 1));
}
```

be hybrid (i.e. they may contain variables and operations of several domains). In this case, within their definition user-defined constraints must separate homogeneous parts, which are then forwarded to the respective solvers.

2.7 Implementation of TURTLE++ as a Library for C++

Thanks to a lot of developments in the field of generic and template programming resp. in the last decade, it became possible to define and use completely new syntactic constructs in TURTLE++ while maintaining compatibility with the original C++ language.

TURTLE++ uses the features of generic programming in C++ to a wide extent. By utilizing template mechanisms and operator overloading constraints can be used in an intuitive declarative manner. The `require` statement is implemented

as a function template which simply expects an expression object, which can contain a nearly arbitrarily complex expression. The expression object is created from the expression and both can be treated separately. So the user can either write down expressions and formulae in the usual intuitive way or build them dynamically during program execution.

Furthermore, the pure generic interface enables the user to add or adapt constraint solvers at will. This is especially important for user-defined domains and offers a wide application field for `TURTLE++`.

3 CIP with `TURTLE++`

In this section we illustrate constraint imperative programming with `TURTLE++` by means of three small but expressive examples.

3.1 User Interaction

A typical example demonstrating the advantages of the combination of imperative and constraint-based programming is user interaction, as already seen by the introductory example. We want to consider here a further example which demonstrates besides the usage of the boolean solver.

We consider a user interface for configuring a product, say a computer. The choice of certain components (motherboards, cards, chips, etc.) may be compatible with particular others or may exclude them. We will express this by boolean constraints. For simplification we consider a small setting with only three components. Their choice can be (de)activated by means of buttons `button[i]`, $i \in \{0, 1, 2\}$, while the following compatibility constraints must be ensured:

Constraint 1 At least one component must be chosen resp. its button activated.

Constraint 2 The choice of `button[0]` implies the choice of `button[1]`.

Constraint 3 Activating `button[2]` excludes the second component, i.e. the choice of `button[1]`.

Program 3.1 implements this setting. In lines 2-4 we require the three constraints given above. Line 5 initially activates `button[0]`. Within a while loop (lines 8-25) the following actions are performed: The output (line 10) of the current configuration by activated or deactivated buttons on the interface yields to an initial computation (resp. a recomputation in later passes) of values for the buttons consistent with the constraints 1, 2 and 3 and that of line 5. In line 11 we await a button click, which generates a value for the variable `i`: 1 for `button[0]`, 2 for `button[1]`, or 3 for `button[2]`. Then the constraint handle `click_button` is assigned a new constraint representing the last button click (line 17), where the variable `button[i-1]` stands for the clicked button. This variable is unfixed in line 18 and all other value assignments for constrainable variables linked with `button[i-1]` (this concerns `button[i%3]` and `button[(i+1)%3]`) are weakened setting their strength to `weak` (line 19) to hold them as stable as possible.

Program 3.1 Configuration of three components

```
1 constrained<bool> button[3];
2 require (button[0] || button[1] || button[2]); // constraint 1
3 require (button[0] -> button[1]);           // constraint 2
4 require (button[2] -> !button[1]);          // constraint 3
5 constraint_handle<bool> click_button=require (button[0] == true);
6 bool done = false;
7 int i;
8 while (!done)
9 {
10     ... // output button configuration here
11     ... // get button click on i here
12     switch (i)
13     {
14         case 1:
15         case 2:
16         case 3:
17             click_button = require (button[i-1]==!button[i-1]());
18             button[i-1].unfix();
19             button[i-1].fix_all(weak);
20             break;
21         default:
22             done = true;
23             break;
24     }
25 }
```

3.2 Puzzles

It is possible, of course, to withdraw to one paradigm and use pure imperative (by simply leaving out constrainable variables and constraints) or mostly declarative programs. A standard example for the latter are crypto-arithmetic puzzles, like the SEND-MORE-MONEY problem given in Program 3.2. The problem is described (and computed) by constraints (lines 2-8)³ and imperative constructs only service for I/O (lines 9-11).

With the help of dynamic expressions it is also possible to create such puzzles dynamically at run-time so that the user provides the participating words, like SEND, MORE and MONEY which are then composed into the puzzle. An example is given by the function `example_dynamic_puzzle` at [1].

3.3 Optimization

Optimization is one of the main uses of constraint programming. In TURTLE++ preferred values for given expressions allow optimization in a simple way without

³ Note that we apply the user-defined constraint `dom` from Program 2.6 here.

Program 3.2 A crypto-arithmetic puzzle

```
1 constrained<int> s, e, n, d, m, o, r, y;
2 require (dom (s, 1, 9));
3 ...
4 require (dom (y, 0, 9));
5 require (alldifferent (s,e,n,d,m,o,r,y));
6 require (
7     s*1000 + e*100 + n*10 + d
8     + m*1000 + o*100 + r*10 + e
9     == m*10000 + o*1000 + n*100 + e*10 + y);
10 std::cout << "s:" << s();
11 std::cout << "e:" << e();
12 ...
```

Program 3.3 A knapsack problem: Implicit optimization by preferred values

```
1 void knapsack()
2 {
3     typedef constrained<double> double_c;
4     double capacity = 100.0;
5     double_c a, b, c;
6     double_c packed (capacity);
7     require (packed <= capacity);
8     require (a == 1.0 || a == 0.0);
9     require (b == 1.0 || b == 0.0);
10    require (c == 1.0 || c == 0.0);
11    require (a * 50.0 + b * 40.0 + c * 30.0 == packed);
12    std::cout << "used:" << packed() ;
13    std::cout << "a:" << a();
14    ...
15 }
```

the need of special library functions. While this approach mainly works for small and simple optimization problems, for more sophisticated problems one may use constraint hierarchies as well provided by TURTLE++.

Our simple optimization pattern consists of four steps: Consider the rather primitive knapsack problem in Program 3.3.

In step one we declare a constrainable variable and assign to it an absolute minimum or maximum border as preferred value (line 6). In step two we define the constraints (lines 7-10) wrt. which the objective function (line 11) will be optimized (step three). Finally (step four), by reading **packed** first (i.e. before all other constrainable variables in this sub-store) a value assignment closest to the given preferred value is calculated. The implicit fixing also immediately limits the other constrainable variables to values at the searched optimum.

The usage of preferred values for constrainable variables depends on the variables domain and the existence, form, and complexity of an optimization

function of the concerning solver. Accordingly, a solver may allow to optimize wrt. a certain optimization function or a number of variables.

4 Conclusion and Future Work

TURTLE++ arose from the Turtle *language* [5] and was adapted and extended for its smooth integration as *library* into C++ to increase the acceptance of declarative concepts by users of an imperative language.

One main difference between TURTLE++ and Turtle is the time point for the computation of values for constrainable variables. While in Turtle this happens during the execution of the `require` statement, in TURTLE++ we use the more flexible but also more subtle approach of delayed bindings, implicit fixing and unfixing. This gives the user a more fine-grain control over the constraints and allows, in combination with the introduction of constraint handles, to redefine constraints or to readjust their strength during computation. Moreover TURTLE++ provides dynamic expressions to create constraints dynamically at run-time.

Other constraint libraries for imperative languages are e.g. ILOG [2] for C++ and `firstcs`[6] and `Koalog`[3] for Java. As opposed to TURTLE++ libraries like ILOG mostly do not utilize syntactic extension mechanisms present in the implementation language to the same extent, e.g. for constraint statements.

TURTLE++ attempts seamless integration of declarative concepts into an imperative language and thus clearly differs from the other libraries which explicitly separate the constraint-based parts from the imperative language constructs. This becomes most obvious when mixing declarative and imperative particles like in Program 3.1. We do not consider a declarative sub-procedure inside an imperative main program or explicitly invoke a solver. Instead, the declarative computation is integrated into the imperative one directly.

While `firstcs` and `Koalog` are FD-constraint libraries, TURTLE++ allows the handling of linear optimization problems and boolean constraints. Finite domain constraints are currently treated by user-defined constraints and the existing solvers only (e.g. see Program 3.2). Since TURTLE++ can be easily extended with new solvers, one interesting area of future research would be the integration of an FD-solver including search and the computation of sets of solutions (which is currently not supported because not compelling for linear optimization problems). The integration of search using backtracking relatives however, must be considered very carefully because they strongly interfere with the imperative approach of TURTLE++. One possibility (as mainly chosen by the other libraries) is again the separation of both paradigms by encapsulating the declarative parts. A tighter integration in the style of TURTLE++ is, thus, a challenging aim.

Furthermore, we intend to investigate the enhancement of TURTLE++ with new concepts like the extension of `if`-constructs with (user-defined) constraints. This would allow to also check relations between partially and non-instantiated constrainable variables but would require an entailment test on the condition. This way, in addition to the *a-priori* declaration of constraints using the `require` construct the user would be provided with an *a-posteriori* test of constraints.

References

1. TURTLE++. <http://people.freenet.de/turtle++/>. last visited 2006-08-14.
2. ILOG. ILOG Web Site. <http://www.ilog.com>. last visited 2006-08-14.
3. Koalog Constraint Solver (v3.0) Tutorial. <http://www.koalog.com/resources/doc/jcs-tutorial.pdf>, 2005. last visited 2006-08-14.
4. B.N. Freeman-Benson. *Constraint Imperative Programming*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1991. Technical Report 91-07-02.
5. M. Grabmüller and P. Hofstedt. Turtle: A Constraint Imperative Programming Language. In *Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, number XX in Research and Development in Intelligent Systems. Springer, 2003.
6. Matthias Hoche, Henry Müller, Hans Schlenker, and Armin Wolf. firstcs - A Pure Java Constraint Programming Engine. In *International Workshop on Multiparadigm Constraint Programming Languages – MultiCPL*, Kinsale, Ireland, 2003.
7. Olaf Krzikalla. Constraint Imperative Programming with C++. In *International Workshop on Multiparadigm Constraint Programming Languages – MultiCPL*, Kinsale, Ireland, 2003.
8. G. Lopez. *The Design and Implementation of Kaleidoscope, a Constraint Imperative Programming Language*. PhD thesis, University of Washington, 1997.