GIFT: A Generic Interface for reusing Filtering Algorithms

Ka Boon Ng[1], Chiu Wo Choi[1], Martin Henz[1], Tobias Müller[2]

[1] School of Computing, National University Of Singapore, Singapore
{ngkaboon,choichiu,henz}@comp.nus.edu.sg
[2] Programming Systems Lab, Universität des Saarlandes, Germany
tmueller@ps.uni-sb.de

Abstract. Many different constraint programming (CP) systems exist today. For each CP system, there are many different filtering algorithms. Researchers and developers usually choose a CP system of their choice to implement their filtering algorithms. To use these filtering algorithms on another system, we have to port the code over. This situation is clearly not desirable. In this paper, we propose a generic C++ interface for writing filtering algorithms called GIFT (Generic Interface for FilTers). By providing the generic interface on different CP systems, we can reuse any filtering algorithms easily. A case study on reusing scheduling filtering algorithms between Mozart and Figaro further highlights the feasibility of this approach.

1 Introduction

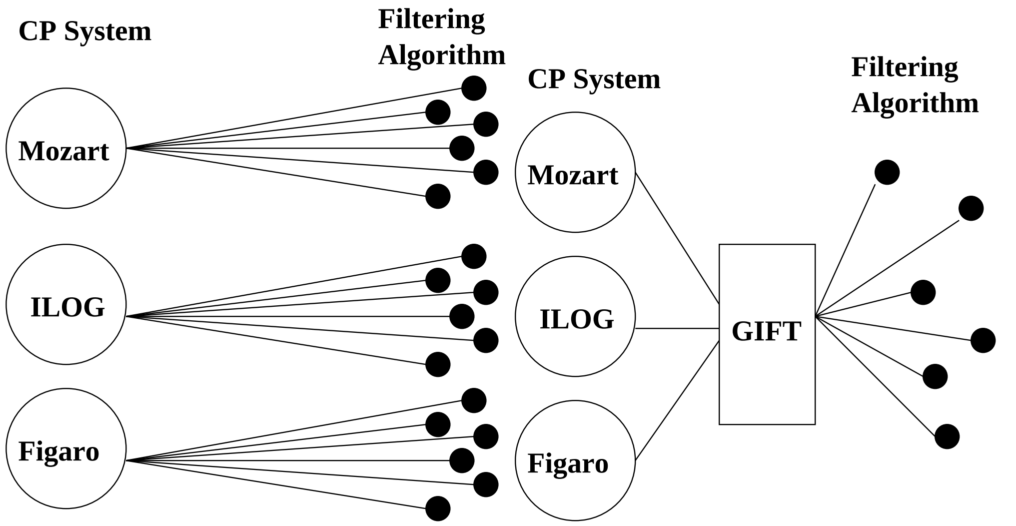
Today many programming systems and libraries make extensive use of the constraint programming technology. Examples include ILOG [ILO97b], CHIP [BSKC97], GNU Prolog [DC00], CLAIRE [CJL99], Mozart [Moz99]. A central theme of these recent CP systems is that they provide the capability for users to program their own constraints [MW97,PL95].

Meanwhile, researchers have proposed many filtering algorithms to improve constraint propagation. Examples include the all different constraints [Rég94], the task intervals constraint [CL94] and some scheduling constraints that incorporate Operations Research techniques [BPN95]. Usually, as a proof of concept, they implement these algorithms as an extension of a specific CP system. When other researchers/developers wish to implement the algorithms, they often have to program the algorithms from scratch.

As a result, we have the scenario that many programmers implement similar filtering algorithms in different CP systems. What we really like to have is a single implementation of a filtering algorithm that runs on any CP system. Figure 1 shows graphically our intention.

In Nicolas Beldiceanu and Warwick Harvey and Martin Henz and François Laburthe and Eric Monfroy and Tobias Müller and Laurent Perron and Christian Schulte, editors, *Proceedings of the Workshop on Techniques for Implementing Constraint Programming Systems - TRICS*, pages 86–100, Singapore, September 2000.

Fig. 1 Before GIFT and after GIFT



Clearly, we can have better code reuse if everyone implements filtering algorithms for a single CP system. Unfortunately, many CP systems exist for different purposes. Yet, this thought provides us with the key idea to this work. Most CP systems are extensible by C/C++ and by relying on a generic C++ interface, we can implement filtering algorithms through the use of the interface. We do not need to concern themselves with system-specific issues. What we need to do is to implement the interfaces on different CP systems.

In this paper, we review the important functionalities that a filtering algorithm will need from a CP system in the context of finite domain constraints. By extracting these functional requirements, we identify a suitable interface between the CP system and the filtering algorithms. Instead of providing everything using one thick interface, we define a set of thin interfaces, namely CP Service Interface, CP Constraints Interface, CP Finite Domain Interface, CP Variables Interface. Filtering algorithms are mostly concerned with the latter two to help achieve local-consistency by domain reduction. The difference between finite domains and variables is that a variable links a finite domain to the actual variable in the store. Constraints and filtering algorithms have an analogous relationship in that the constraints represent meta-information about the filtering algorithms. The CP Service is an interface use for communicating information between the filtering algorithms and the CP System.

Building on the interface, we show the design of an interface for ILOG and Mozart. The interface can be extended to other CP systems readily. Despite the differences between ILOG and Mozart, both represent constraints as objects in their systems. To program our own constraints, we can extend from the pre-defined constraint classes on both systems. Thus, it means that our interface class can be a member of the extended constraint class.

As a concrete proof of concept, we present a case study on the sharing of filtering algorithms between Mozart and Figaro [HMN99]. Although the two CP systems have significant differences, we show that both systems can share filtering algorithms through our generic interface. Moreover, experimental evaluation shows that the overhead for our generic interface is minimal for solving constraint problems.

The rest of the paper is organized as follows. Section 2 reviews the filtering algorithms' design requirements of the generic interface. Section 3 outlines the interface declaration given in C++. Section 4 shows how the interface can be implemented in ILOG and Mozart. Section 5 gives an example of the implementation of a filtering algorithm. Section 6 provides a case study illustrating the reuse of scheduling propagators between Mozart and Figaro. Section 7 discusses how we can handle differences among different CP systems and sheds some light on our future work.

2 Design Requirements

This section explains how a filtering algorithm can interact with the CP system. By identifying the operations, we can provide an interface to such operations in a generic way. In addition, the interface should ensure that the filtering algorithms are loosely coupled with the underlying CP system so as to achieve platform independence.

The requirements of a filtering algorithms are as follows:

1. A filtering algorithm needs to identify variables and manipulate their respective domains. Manipulation usually comes in the form of domain reduction.
2. A filtering algorithm needs to communicate its state to the CP system. It needs to tell the CP system whether it is entailed or failed.
3. A filtering algorithm may need to communicate changes to the CP system. There may be advanced filtering algorithms that can replace the current constraint with a better constraint when they meet a certain condition or when other constraints unify two variables in the filtering algorithm.
4. A filtering algorithm may need to get information from the CP system. For example, it may want to query if two variables in the filters are referring to the same variable.
5. A filtering algorithm should be stateful. Some filter algorithms need to maintain a support graph in order to speed up filtering. For system independence, we should construct these graphs and keep them in the filtering algorithms.
6. A filtering algorithm must be able to create a copy of itself. This requirement allows search algorithm to restore the state of the store, which includes the individual filtering algorithms, to a previous state, during a backtracking search.

3 Generic Interface

This section describes the important parts of the interface, especially with respect to the requirements listed out in the previous section. We represent this set of interfaces using the following C++ classes, namely, CP Service, CP Constraint, CP Variable and CP Finite Domain.

Program 1 The CP FiniteDomain interface

```
class CPFD {
public:
    // constructors to be defined...
    int getMin() const;     // returns minimum value
    int getMax() const;     // returns maximum value
    int getValue() const;   // return a singleton value or -1

    int operator >=(const int v); // filter off values less than v
    int operator <=(const int v); // filter off values greater than v
    int operator &=(const int v); // set domain to v
    int operator -=(const int v); // takes away v from FD

    ~CPFD(); // destructor
};
```

Program 1 shows a basic interface for CP Finite Domain. Programmers can use the interface to access the domain and modify the domain. To provide backward compatibility to Mozart's propagators, we kept the overloaded operators for domains manipulation.

Program 2 shows the interface of CP Variable. These variables hold the finite domain and the identity of the variables themselves. In this simple setup, the CP Variable needs only to return its CP Finite Domain for domain accesses and modifications. Thus, the CP Variable and CP Finite Domain classes help to encapsulate system-specific details about the variables from the programmers.

Program 2 The CP Variable interface

```
class CPFDVar {
public:
    // constructor : to be defined...
    CPFD& operator * (void) { return (*fd); } // for accessing
    CPFD* operator -> (void) { return (fd); } // finite domains

    ~CPFDVar(); // destructor
};
```

There are four types of operations in CP Service. From the filtering algorithm point of view, it allows communicating information to and from the CP system. Similarly, from the CP system point of view, it allows an additional channel to communicate information to and from the filtering algorithm. Using an interface class allows more expressivity in passing arguments and returning values. This interface fulfills requirements (2), (3) and (4) described in the previous section. By implementing the filtering algorithm as an object, we achieve (5). C++ provides an interface known as a copy constructor for every class. A copy constructor allows the programmers to create a copy of an existing object. The copy constructor naturally fulfills (6).

Program 3 illustrates an interface that can communicate the propagation result to the CP system. We use an enumerated type, `CPState`, to reflect the state of the filtering algorithms. The filtering function must return the `CPService` reference pointing to itself (i.e. `*this`). In this way, we can write more concise code without declaring any additional variables to store the return `CPService` object.

Program 3 A CP Service interface

```
enum {FAIL, ENTAIL, SLEEP} CPState;
class CPService {
public:
    // constructor : to be defined...
    CPService(const CPService &rhs); // copy constructor

    CPService& entail(); // mark state to entail
    CPService& fail();   // mark state to fail
    CPService& sleep();  // mark state to sleep
    // advance services omitted ...
    CPState getResult();  // gets the propagation result
};
```

4 Implementing Interface

This section gives some pointers on the actual implementation of the interface in ILOG and Mozart.

4.1 ILOG

In ILOG, the finite domains are represented using the `IlcIntExp` class. For most operations, there should be a one-to-one correspondence between the interface functions' declaration and ILOG's `IlcIntExp` member functions. Program 4 gives an example of the implementation of accessing and modifying the finite domain.

In this setup, the identity of the variable is not necessary. Although this interface class seems redundant in ILOG's context, it can be useful for other systems. Program 5 shows a straightforward implementation of this additional wrapper code for ILOG. It is important that the copy constructor must be properly defined because by default, the copy constructor of C++ does a member-wise copy of the class. In other words, we must copy the content of the pointer and not just the pointer only.

In ILOG, they represent the solver using the `IlcManager` class. Thus, the CP Service interface should contain `IlcManager`. In addition, to remove a constraint from the manager, we must remember the constraint object. Program 6 shows the implementation of what we just described.

Program 4 Simple ILOG's Implementation CP Finite Domain interface

```
class CPFD {
    IlcIntExp fd;
public:
    CPFD(IlcIntExp i) : fd(i) { }
    int getMin() const { return fd.getMin();} // get min value
    int operator >=(const int v) { // filter off values
        fd.setMin(v);              // less than v
        return fd.getSize();
    }
};
```

Program 5 ILOG's Implementation CP Variable Interface

```
class CPFDVar {
    CPFD* fd;
public:
    CPFDVar(IlcIntExp var) { fd = new CPFD(var); }
    // copy constructor must be properly defined
    ~CPFDVar() { delete fd; } // destructor

    CPFD& operator * (void) { return (*fd); } // for getting
    CPFD* operator -> (void) { return (fd); } // back Finite Domains
};
```

4.2 Mozart

In Mozart, they represent the finite domains using a `OZ_FiniteDomain` object. As before, there should be a one-to-one correspondence with the interface functions' declaration and the Mozart's `OZ_FiniteDomain` member functions. Program 7 gives an example of the implementation of accessing and modifying the finite domain. The striking similarity between the CP Finite Domain interface and the `OZ_FiniteDomain` should not be surprising. CP Finite Domain was originally modeled using `OZ_FiniteDomain` class as the base.

Like in ILOG, the CP Variable interface class is redundant in our simple setup. Program 8 shows an almost equivalent implementation of Mozart CP Variable interface.

Mozart's CP Service interface is slightly simpler than the ILOG's version because many things are handled directly by the Mozart's constraint solver. Program 9 is just another fancy way to return the filtering results. Though this way seems overweight in this context, this interface is useful for the ILOG system.

5 Programming Filters

This section illustrates how to write a simple inequality filter, i.e. $x \neq y$. In addition, it shows how the whole setup should work under Mozart and ILOG.

The simple inequality filter ($x \neq y$) rules are:

Program 6 ILOG's Implementation CP Service interface

```
class CPService {
    CPState state;
    IlcManager M;
    IlcConstraint currCon;
public:
    CPService(IlcManager m,IlcConstraint c)
        : currCon(c), M(m) { }

    CPService& entail() {
        M.remove(currCon);
        state = ENTAIL; return *this;
    }
    CPService& fail() { state = FAIL; return *this; }
    CPService& sleep() { state = SLEEP; return *this; }

    CPState getResult() { return state; }
};
```

Program 7 Mozart's Implementation CP Finite Domain interface

```
class CPFD {
    OZ_FiniteDomain fd;
public:
    CPFD(const OZ_FiniteDomain &i) : fd(i) { }
    int getMin() const { return fd.getMinElem();} // get min value

    // filter off values less than v
    int operator >=(const int v) { return fd >= v; }
};
```

1. If x's domain is disjoint to y's domain, filter is entailed.
2. If x and y are both singletons,
   (a) and if x's singleton value $\neq y$'s singleton value , filter is entailed. (Note this step is already subsumed by step 1)
   (b) Otherwise, filter is failed.
3. If only x is singleton,
   (a) take out x's singleton value from y's domain
   (b) Filter is entailed if y's domain size is not zero. Otherwise, filter is failed.
4. If only y is singleton,
   (a) take out y's singleton value from x's domain
   (b) Filter is entailed if x's domain size is not zero. Otherwise, filter is failed.

Program 10 implements the filter rules. Recall that `*x` and `*y` are for getting the finite domains' representation of variable `x` and `y`, and the `-=` – operator takes away the right-hand side value from the left-hand side finite domain. The `&` – operator intersects

Program 8 Mozart's Implementation CP Variable interface

```
class CPFDVar {
    CPFD* fd;
public:
    CPFDVar(const OZFiniteDomain &f) { fd = new CPFD(f); }
    // copy constructor must be properly defined
    ~CPFDVar() { delete fd; } // destructor

    CPFD& operator * (void) { return (*fd); } // for getting back
    CPFD* operator -> (void) { return (fd); } // Finite Domains
};
```

Program 9 Mozart's Implementation CP Service interface

```
class CPService {
public:
    CPService() { }

    CPService& entail() { state = ENTAIL; return *this; }
    CPService& fail() { state = FAIL; return *this; }
    CPService& sleep() { state = SLEEP; return *this; }

    CPState getResult() { return state; }
};
```

two finite domains whereas function `getSize()` returns the number of elements in the domain produced by the `&` – operator.

To call the filter from ILOG, we can call the filtering function as follows.

```
void IlcDiffConstraint::propagate() {
    CPFDVar x(_x), y(_y);
    CPService s(_x.getManager(),*this);
    diffFilterObject->filter(s,x,y);
}
```

The `_x` and `_y` are `IlcIntExp` objects. After initializing the CP Variable, we can instantiate a service object with a manager and the identity of the current constraint. Finally, we can execute the filtering function by passing in the CP Service object and the CP Variable objects. There is no need to take care of the propagation result because in ILOG, when the domain of a variable becomes empty, a failure will be triggered in the manager [ILO97a].

To call the filter from Mozart, we implement the following.

```
OZ_Return DiffProp::propagate() {
    CPFDVar x(*_x), y(*_y);
    CPService s;
    diffFilterObject->filter(s,x,y);
```

Program 10 The $x \neq y$ filtering member function

```
CPService& diffFilter::filter(CPService &svc, CPFDVar &x, CPFDVar &y)
{
    if ((*x & *y).getSize() == 0)            // rule 1
        return svc.entailed();
    if (x.isSingleton() && y.isSingleton())   // rule 2
        if (x.getSingleElem() != y.getSingleElem())
            return svc.entailed();
        else
            return svc.failed();
    if (x.isSingleton())                      // rule 3
        if ((*y -= x.getSingleElem())==0)
            return svc.failed();
        else
            return svc.sleep();
    if (y.isSingleton())                      // rule 4
        if ((*x -= y.getSingleElem())==0)
            return svc.failed();
        else
            return svc.sleep();
}
```

```
    switch (s.getResult()) {
        FAIL : return OZ_FAIL;
        ENTAIL : return OZ_ENTAIL;
        SLEEP : return OZ_SLEEP;
    }
}
```

The `_x` and `_y` are `OZ_FDIntVar` objects. The implementation is similar to ILOG except that we need to postprocess the propagation result.

6 Case Study: Sharing Scheduling Filters between Mozart and Figaro

This section shows a case study for sharing scheduling filters between Mozart and Figaro. First, the differences between the two CP systems: Mozart and Figaro, is observed. Then, we show how common scheduling filters can be shared between the two different CP systems through our generic interface.

The constraint store abstraction of Figaro is know as *store* [HMN99]. It contains variables and propagators as data. Moreover, it differs from the standard way of using a direct reference to the objects of the constraint store. It uses an indirection mechanism which allows greater flexibility to describe the relation between propagators and variables. Each variable has a unique identity assigned by the store during creation. The variable `Id` together with a store uniquely identifies the variable that is being accessed. This is called indirect addressing of variables.

The computational architecture of Mozart is called a *space* and consists of a number of propagators connected to a constraint store. One particular feature is Mozart allows a tree of spaces. Spaces host threads that can concurrently run at the same time. However, variables in a space are not easily accessible as compared to a store of Figaro. The only place to gain access to these variables in a space is through the root variable. Another important difference is that Figaro's stores are accessed explicitly. On the other hand, Mozart's spaces are embedded in the Oz virtual machine and are accessed through a space register [Sch97].

The two scheduling filtering algorithms that we share between Mozart and Figaro are used for solving the disjunctive scheduling problems. Specifically, these filtering algorithms model unary resource constraints. The first one is called *disjunctive filter*. Given a set of tasks that requires a unary resource (i.e. resource of capacity one), disjunctive filter reasons that no two tasks in the set can overlap in time. The second one is called *task intervals filter* [CL94]. It adopts a concept taken from Operations Research called *edge-finding* which performs stronger propagation than the first one.

Program 11 shows the declaration for the disjunctive filter which is shared between Mozart and Figaro. We can also declare task intervals filter in the same way. The `start` variable is a vector of CP Variable objects that represent the start time of each task. Class `CPVectorVar` is a template parameter like `CPService`, because it is CP System dependent. The filter has an attribute `duration` to represent the duration of each task. The details on the implementation of the two filtering algorithms are beyond the scope of this paper. Basically, we reuse the source code of Mozart in implementing the two filtering algorithms with minor modifications. These minor modification allows us to run the two filtering algorithms on any CP system that implements our interface.

Program 11 Declaration for Scheduling Filtering Algorithms

```
template<class CPService, class CPVectorVar>
class DisjunctiveFilter : public filter {
private:
    int *duration;
public:
    diffFilter(int *dur) : duration(dur) {}
    CPService& filter(CPService &svc, CPVectorVar &start)
    {
        // ... implementation of disjunctive filter
    }
}
```

Program 12 shows how to call the disjunctive filter from Mozart. We can call the task intervals filter in the same way. The `_xs` and `_n` are space registers which contain the array of variables and its length. As we have mentioned earlier, this is the way for Mozart to gain access to the space and the variables. The class `OZ_FDIntVarVector` is a vector class for collecting variables into `start`. Class `OZ_Service` implements CP Service. The `FDIntVarIterator` object makes it possible for the `OZ_Service` object to iterate over the finite domain variables in `start`, e.g. to automatically write back domain reductions to the spaces.

Program 12 Mozart Interface To Call Disjunctive Filter

```
OZ_Return DisjunctivePropagator::propagate(void)
{
    OZ_FDIntVar _start[_n];
    for (int i = _n; i--; )
      _start[i].read(_xs[i]);
    OZ_FDIntVarVector start(_n, _start, &_xs);
    FDIntVarIterator P(_n, _start);
    OZ_Service svc(this, &P);
    DisjunctiveFilter->filter(svc,start);
    switch (s.getResult()) {
        FAIL : return OZ_FAIL;
        ENTAIL : return OZ_ENTAIL;
        SLEEP : return OZ_SLEEP;
    }
}
```

Program 13 shows how to call the disjunctive filter from Figaro. We can call the task intervals filter in the same way. Figaro uses the Standard Template Library (STL) [SL95] extensively. The STL vector class is used for `start`, rather than defining another new class. In constructing each `CPFDVar` element, we need to have both the store and variable `Id` due to indirect addressing of variables in Figaro.

Program 13 Figaro Interface To Call Disjunctive Filter

```
State DisjunctivePropagator::propagate() {
    vector<CPFDVar> start;
    vector<varId>::iterator it = _start.begin();
    while (it != _start.end()) {
      start.push_back(CPFDVar(store,*it)); it++;
    }
    CPService svc;
    DisjunctiveFilter->filter(svc,start);
    switch (s.getResult()) {
        FAIL : return FAIL;
        ENTAIL : return ENTAIL;
        SLEEP : return SLEEP;
    }
}
```

The experimental evaluation uses four scheduling benchmark problems: house problem, bridge problem, ABZ6 and MT10. We measure the overhead incurred by the generic interface. Table 1 shows the results of the experiment. The label (Disj) means disjunctive propagator is used, while (TI) means task intervals propagator is used. We take the average user time of five runs as the runtime (in seconds). We run the problems on a 256 MB 400MHz Pentium II PC running Linux.

	Mozart	Mozart w/ GIFT	Figaro w/ GIFT
House (1000×) (Disj)	11.5s	12.4s	22.1s
Bridge (100×) (TI)	13.8s	14.2s	23.4s
ABZ6 (TI)	31.98s	31.98s	65.24s
MT10 (TI)	278s	279s	629.98s

Table 1. Results of Scheduling Benchmark Problem

From the results, we can observe that the overhead incurred by the generic interface is minimal. As for Mozart, the overhead incurred is almost negligible. At the time of the experiment, Figaro is still in its experimental stage. No fine tuning have been done yet. We observe that Figaro runtime is about twice of Mozart implementation based on the results of ABZ6 and MT10. From this observation, we conclude that there is no increasing blown up in runtime when we use our generic interface. The results affirms that our generic interface approach is practical in facilitating reuse of filtering algorithms among different CP Systems.

7 Discussion

This section discusses the design issues for implementing advanced propagators. It discusses also the related work and future work.

In this paper, we presented a simple interface between the CP system and the filtering algorithms. The interface cannot handle advanced filtering algorithms that can perform constraint reasoning [HS98,Mül00]. Such filtering algorithms can impose new constraints, replace itself with another constraint or even transform a set of constraints into another set of constraints. In addition, it can drop old variables from a constraint and add new variables to a constraint. In Mozart, constraints can unify two variables making them equal by reference.

Not all constraint reasoning can be implemented in this interface, but we are able to handle imposing and replacing of constraints. To do that, we can use the CP Constraint interface. This interface class consists of a number of static member functions that when called return a pointer to a new constraint object containing a particular filtering algorithm. To impose an all-different constraint, we can code:

```
svc.impose(CPConstraint::makeAllDiffConstraint())
```

We define `void dropVar(varId v)` and `void addVar(varId v)` for dropping and adding variables respectively. The member function `void equatevar(varId x, varId y)` allows us to unify the variables.

Having a specific operation like unification of variables can have subtle effects on the implementation of filtering algorithms. In Mozart, variables equality is represented directly. Such representation may not exists on other CP systems. In that case, when `equatevar` is called, the implementation can either issue a no-op or introduce an equality constraint. Doing so may decrease the strength of the propagation. More importantly, programmers should be aware of such differences and ensure that the filtering algorithms end result must be consistent in different CP systems.

By making the interface as generic as possible, we omitted some system specific features. For example, a CP system may wish to enquire information about the domain reduction rate of a filtering algorithm. Although it is easy to include this additional communicating interface between filtering algorithms and the CP system, it is clearly not desirable because of its specific nature. Moreover, nothing is stopping the programmer to modify the GIFT interface to suit her particular needs.

Although this particular work is new to the constraint programming community, the ideas of interfacing has a long history. Some recent work includes Simplified Wrapper Interface Generator(SWIG) [Bea96], which wraps C/C++ code and extend it to different scripting languages, and Java Database Connectivity (JDBC) [Sli00], which lets the user access different relational database systems using a common interface. Furthermore, [GHJV94] mentions the use of the adapter design pattern which is another name for interface and wrapper.

The future direction of this work is to focus more on the advanced filtering algorithms and identify the functionalities that these filtering algorithms need. Another problem is that even with a generic interface, extending the predefined constraint class is unnecessary tedious. It may be worthwhile to implement stub generation tools based on a simplified class definition.

8 Conclusion

There is a many-to-many relationship between CP systems and filtering algorithms. To achieve a higher degree of reuse, we should only need to implement the filtering algorithms once and be able to run it on any CP system.

The key idea to achieve this goal is to use the idea of an interface. Instead of just providing a single thick interface, we made the interface more organized and manageable by decomposing the interface into a set of interfaces, namely CP Service, CP Constraint, CP Variable and CP Finite Domain.

Based on this idea, we identified the important requirements of the filtering algorithms and designed the different interface classes. Following that, we implemented the interface in Mozart and showed how a filtering function can be called from Mozart. We also showed the design of the interface using ILOG. The case study further proved the practicality of this approach. We expect reuse of filtering algorithms will speed up development time and allow us to concentrate on other aspects of constraint programming.

Acknowledgments

We would like to thank Ong Kar Loon for implementing some of the filters.

References

[Bea96] David M. Beazley. SWIG: An Easy to Use Tool for Intergrating Scripting Languages with C and C++. In *4th Annual Tcl/Tk Workshop*, Monterey, July 1996.

[BPN95] P. Baptiste, C. Le Pape, , and W. Nuijten. Incorporating efficient operations research algorithms in constraint-based scheduling. In *Proceedings of the First International Joint Workshop on Artificial Intelligence and Operations Research*, 1995.

[BSKC97] Nicolas Beldiceanu, Helmut Simonis, Philip Kay, and Peter Chan. The CHIP system. White paper, COSYTEC SA, 1997. Ref: COSY/WHITE/002, Ver: 1.2, Rev: A.

[CJL99] Yves Caseau, François-Xavier Josset, and François Laburthe. CLAIRE: Combining sets, search and rules to better express algorithms. In Danny De Schreye, editor, *Proceedings of the International Conference on Logic Programming*, pages 245–259, Las Cruces, New Mexico, USA, 1999. The MIT Press, Cambridge, MA.

[CL94] Yves Caseau and Fran cois Laburthe. Improved CLP scheduling with task intervals. In *Proceedings of the International Conference on Logic Programming*, pages 369–383, 1994.

[DC00] Daniel Diaz and Philippe Codognet. The GNU prolog systems and its implementation. In *ACM Symposium on Applied Computing*, Como, Italy, 2000.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1994.

[HMN99] Martin Henz, Tobias Müller, and Ka Boon Ng. Figaro: Yet another constraint programming library. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, 1999. held in conjunction with ICLP'99.

[HS98] Warwick Harvey and Peter Stuckey. Constraint representation for propagation. In Jean-François Puget and Michael Maher, editors, *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming(CP98)*, Lecture Notes in Computer Science, pages 235–249, Pisa, Italy, October 1998. Springer-Verlag.

[ILO97a] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *ILOG Solver 4.0, Reference Manual*, 1997.

[ILO97b] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *ILOG Solver 4.0, User Manual*, 1997.

[Moz99] Mozart Consortium. The Mozart Programming System. Documentation and system available via WWW from `http://www.mozart-oz.org`, 1999.

[Mül00] Tobias Müller. Promoting constraints to first-class status. In *First International Conference on Computational Logic*, London, UK, 2000. to appear.

[MW97] Tobias Müller and Jörg Würtz. Extending a concurrent constraint language by propagators. In Jan Małuszyński, editor, *Logic Programming: Proceedings of the 1997 International Symposium*, pages 149–163, Long Island, NY, USA, 1997. The MIT Press.

[PL95] Jean-François Puget and Michel Leconte. Beyond the Glass Box: Constraints as Objects. In *Proceedings of the International Symposium on Logic Programming*, pages 513–527, 1995.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the AAAI 12^{th} National Conference on Artificial Intelligence*, pages 362–367. AAAI Press, 1994.

[Sch97] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Principles and Practice of Constraint Programming—CP97, Proceedings of the Third International Conference*, Lecture Notes in Computer Science 1330, pages 519–533, Schloss Hagenberg, Linz, Austria, October/November 1997. Springer-Verlag, Berlin.

[SL95] Alexander Stepanov and Meng Lee. *The Standard Template Library*. Hewlett Packard, 1995. STL has since been incorporated into the C++ standard, ISO/IEC 14882-1998.

[Sli00] Carol Sliwa. Java databse connectivity. *ComputerWorld*, 2000. Available at `http://www.computerworld.com/home/features.nsf/all/991213qs`.