# A Software Engineering Approach to Constraint Programming Systems

Ka Boon Kevin Ng
Advanced Application Technologies
Honeywell Automation and Control Solutions
kevin.ng@honeywell.com

Chiu Wo Choi
Dept. of Computer Science and Engineering
The Chinese University of Hong Kong
cwchoi@cse.cuhk.edu.hk

Martin Henz
School of Computing
National University of Singapore
henz@comp.nus.edu.sg

## Abstract

*Constraint programming (CP) systems are useful for solving real-life combinatorial problems, such as scheduling, planning, rostering and routing problems. The design of modern CP systems has evolved from a monolithic to an open design in order to meet the increasing demand for application-specific customization. It is widely accepted that a CP system needs to balance various design factors such as efficiency versus customizability and flexibility versus maintenance. This paper captures our experience with using different software engineering approaches in the development of constraint programming systems. These approaches allow us to systematically investigate the different factors that affect the performance of a CP system. In particular, we review the application of reuse techniques, such as toolkits, framework and patterns, to the design and implementation of a finite-domain CP system.*

## 1. Introduction

During the past decades, software engineering has been successfully applied to the development of information systems software and systems software. Information systems software are characterized by data transaction, storage and search. Systems software are characterized by hardware/software interfacing. The goal of software engineering is to improve software quality with criteria such as high maintainability and absence of faults.

In this paper, we focus on another class of software systems, namely knowledge systems software, such as decision support systems and other artificial intelligence systems. These systems are characterized by specialized software algorithms, otherwise known as engines, that drive the system. These software engines include algorithms for image recognition, constraint solving, machine learning, etc. Unlike conventional research where there is a time lag between research and application, research in knowledge systems software is on-going and continuously being transfered to industrial practice. By applying the right software engineering approaches to knowledge systems, we could strive for a smoother transition from research into actual applications.

The recent proliferation of knowledge systems happens because of several reasons. First, there are niche application areas where knowledge systems have proven to be highly effective compared to alternative techniques (e. g. , see [3]). Second, the increase in performance of personal computers has led to competitive performance without expensive capital investment. Finally, there is a movement towards open software concepts and research systems are written using directly deployable software architecture, such as C++/Java.

Differing software engineering practices in research and industry results in a gap between the quality level and technology level of knowledge systems software. The focus in research has mostly been on theoretical improvements. Hence, there is a lack of appreciation of the software engineering aspects of such systems, which in turn means that such research systems often have poorly designed software architectures. Ultimately, they are not flexible enough to adapt to new improvements in the software algorithms.

As the software industry moves towards a customer-centric market, fast deployment becomes essential. Knowledge systems are further exacerbated by the fact that customers are unfamiliar with the technology and may easily substitute one knowledge system with another without knowing the differentiating features of each system. To survive in the current marketplace, rapid development methodologies is critical in engineering such systems and these methodologies must be adaptive to new advancements in research. Careful software reuse appears to be a solution to support the effective transfer of research results into industrial practice.

In this paper, we use a case-study approach to evaluate the contributions of current software engineering concepts. We also provide insights on how to apply software engineering approaches for knowledge systems and how we could further improve the process.

In the case study, we recall the experience of developing a type of knowledge systems known as constraint programming (CP) systems. CP systems exhibit properties of a typical knowledge systems software. It is driven by two key engines, the propagation (or inferencing) engine and search engine. From an algorithmic stand-point, the algorithms behind the engines are fundamental to artificial intelligence research. Current ongoing research aims to improve CP by having more flexible modeling and improving speed performance, both of which will be important for solving real-life combinatorial problems.

From a software engineering perspective, it is interesting that CP concepts can be easily mapped into objects. Even the reuse of constraints has been widely encouraged since the early days of CP research. Hence, the CP system software provides an interesting case study on applying software engineering methodology, software reuse in particular, into knowledge systems.

The rest of the paper is organized as follows. Section 2 gives an overview of CP systems. Section 3 discusses our experience in applying different reuse techniques to the development of CP(FD) system. Section 4 discusses the lessons that we learned. In Section 5, we conclude our study.

## 2. Finite-Domain Constraint Programming

Constraint programming is a declarative programming paradigm combining local consistency algorithms and heuristic search algorithms in artificial intelligence. Given a problem, a constraint program models the specification in terms of constraints that describe the relationships between the problem variables. The CP system accepts the constraint program and finds solutions that satisfy all the stated constraints.

In this paper, we focus on the branch of CP called *finite-domain constraint programming* (CP(FD)), where all problem variables are restricted to a finite domain of values. Recent research shows that CP(FD) is competitive to conventional operations research techniques in solving combinatorial problems such as planning, scheduling, time-tabling, resource allocation, routing, configuration and placement problems. Here, we will highlight the important concepts in CP(FD) to facilitate our discussion. A more detail overview could be found in [13].

In CP(FD), a *variable*, also called a *logic variable*, binds to the set of possible values it can take. We call the set of possible values the *finite-domain* of the variable. The finite-domain is a subset of the predefined finite set of symbols, usually integers with pre-defined minimum and maximum, for direct representation during computation. A *constraint* specifies the relationship over a set of variables, while *constraint propagation* is the basic technique for constraint inferencing. To better understand constraint propagation, let us consider only binary constraints. Then, we can view the collection of variables and constraints as a graph called a *constraint graph*, where nodes represent variables and arcs between nodes represent a binary constraint between the variables. When a value is removed from a variable, constraint propagation inspects neighboring variables (linked by the arcs/constraints) to remove values from their finite-domain that is inconsistent with the constraints. This deductive reasoning process can be generalized to n-ary constraints. In addition to restricting the finite-domain of variables, the deductive reasoning process could infer new constraints to add to the constraint model.

Usually, constraint propagation alone is insufficient to solve a problem. *Search* complements constraint propagation. It employs a systematic process of enumerating all the possible solutions in the form of a tree, and explores the tree in a top-down fashion, usually carrying out propagation (or inferencing) at each step of the exploration. We use the term *branching* to describe the former process of creating branches to form a tree, *exploration* for the latter process of traversing the tree, and *tree search* to denote this type of search.

## 3. Reuse Techniques

Over the years, the monolithic design of CP(FD) systems has given way to a more open and modularized design. These open systems provide the interfaces for easy extension of user-defined constraints and search algorithms. However, with increasing demand for shorter development cycle and high maintainability to support ongoing research technology transfers into practice, it will be useful to apply reuse techniques to the development of CP(FD) systems. The reuse techniques that we have applied include: software toolkit, scripting, software components and framework, and design patterns.

### 3.1. Software Toolkit

A toolkit is a set of related and reusable routines designed to provide useful and general purpose functionality. The emphasis is on *code* reuse. It does not impose design restrictions on the use of them. The design of CP(FD) systems encourages the reuse of constraints provided by the system to model problems. Such a library of constraints is already a toolkit by itself.

Search is another important aspect of CP(FD). Existing CP(FD) systems support the programming of search through programming abstractions. The Mozart/Oz System [17] provides several search engines, extended in dimensions such as interaction, visualization, and optimization. However, such extensions are monolithic in design, not catering for systematic reuse. It becomes a major effort to implement a new search engine and equip it with useful facilities like visualization of search tree. A search toolkit [5] modularizes search in such a way that the different design dimensions can be implemented separately. The architecture of the toolkit allows to plug together a custom search engine by reusing the modules provided.

The search toolkit considers the following six different design dimensions:

**Memory Policy** concerns with the policy to restore the state when encountering a failure during search. It is also called the *state restoration policy* [7]. Most existing CP(FD) systems uses trailing, with the exception of Mozart/Oz which uses copying/recomputation.

**Exploration** states the order in which nodes are visited in the search tree. The most common exploration is depth-first. Other useful explorations include breadth-first, iterative deepening, and limited discrepancy.

**Interaction** dictates the way in which the user controls the exploration. The most common interactions are first-solution, all-solution, and last-solution. There are also next-solution and tracer for interactively exploring the search tree.

**Optimization** allows to modify an engine such that different pruning behaviors are achieved when searching for the most optimal solution. The two well-known optimization methods are branch-and-bound and restart.

**Visualization** is responsible for visualizing the search tree in a graphical interface. This is similar to the Oz Explorer [24]. The toolkit provides two types of visualization, the simple display only displays the search tree as it is being explored, while standard display allows to interactively explore the search tree using the tracer interaction.

**Information** enhances interaction and visualization by providing the facility for accessing the information in a given node in the tree. These include node information and edge information.

Figure 1 shows a snapshot of a custom search engine constructed with the toolkit code generator by combining limited discrepancy search with recomputation, node-level

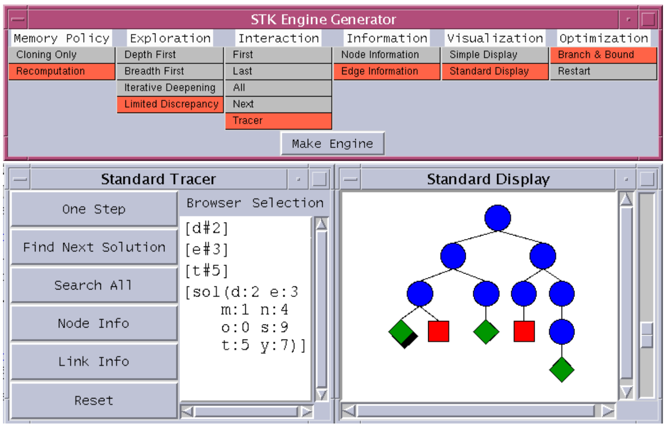



**Figure 1. Engine Generator and the Resulting Custom Search Engine**

tracing, edge information, branch-and-bound optimization, and simple display visualization. The current implementation makes use of Oz's advanced object-oriented features [12] where classes are first-class citizens. The toolkit accepts the classes that describe the different modules as parameters and generates the class definition of the desired search engine at runtime. Such a toolkit can be compared to a set of comparison functions for a sort algorithm. In this case, we have a set of parameters that characterizes the search algorithm. This technique can also be seen as generic algorithms found in generic programming [1] which is supported by the C++ STL toolkit [25]. E. g. , `sort(v.begin(), v.end(), compfunctor())`.

### 3.2. Scripting Approach

Scripting languages, like Tcl [22] and Perl [26], are very high-level languages made for "gluing" components together. The scripting approach is built on the philosophy that combining different types of software components should be made as convenient and simple as possible. This philosophy conveniently supports the requirement for rapid application development. The scripting approach requires that the components are highly reusable. As we shall see in the next section, the heart of CP(FD) is such a component-based system.

For a real-life application, building a prototype as quickly as possible for evaluation is very important. Often, such prototype design stresses on common functionalities such as user interface, database connectivity and web-enabling. The scripting approach to CP(FD) systems [20] is an ideal way for rapid prototyping a constraint application. We can show both common functionalities and problem-solving capabilities in such a constraint application.

The scripting language chosen for implementation is

```
proc nqueen {R n} {
    for {set i 1} {$i <= $n} {incr i +1} {
        set row($i) [$R newvar 1 $n]
        lappend ids "row($i)"
        lappend vars "$row($i)"
    }
    set pub [makesoln $R $ids $vars]
    set L1N ""
    set L1MN ""
    for {set i 1} {$i <= $n} {incr i +1} {
        lappend L1N  "#$i"
        lappend L1MN "#-$i"
    }
    new_distinct $R $vars
    FD_distinctoffset $R $vars $L1N
    FD_distinctoffset $R $vars $L1MN
    drawboard
    naive $R $vars $pub 0 7
}
```

**Figure 2. A generic n-queen problem formulation in Tcl**

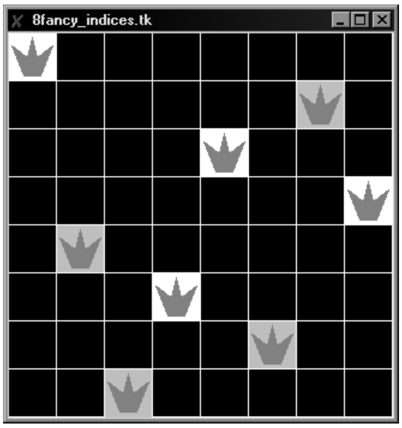



**Figure 3. A screen shot of the solved solution for 8-queens**

Tcl/Tk [22]. The Tcl scripting language provides ease of integration with different libraries, which are useful in developing constraint application. In particular, the Tcl extension, Tk, provides rich visualization support with little programming effort. The Standard Wrapper and Interface Generator (SWIG) [2] provides an easy way for interfacing Tcl and the core CP(FD) system. By providing the class definitions, SWIG automatically generates the wrapper code for the class method to interface with Tcl language internals.

For example, we would like to construct a prototype to visualize the solving of the 8-queens problem. In this problem, we must place eight queens on a 8x8 chess board such that no two queens attack each other. Figure 2 gives the simple Tcl scripts formulating the constraints for the problem. The call to procedure `drawboard` draw the chess board using the canvas widget in Tk. Figure 3 is a screen shot showing a solved solution.

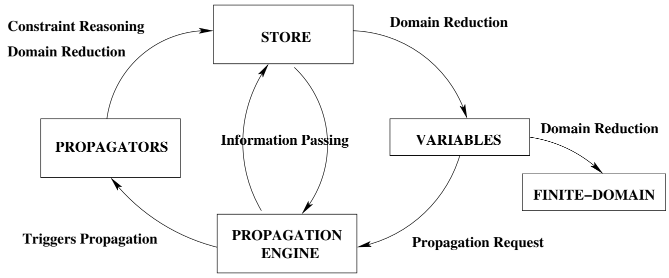



**Figure 4. System Model of Framework**

Closely related to the scripting approach is the Structured Query Language (SQL) used in relational database systems [9]. The intricate details of query search is complex but when building database applications, the application programmers are not interested in the algorithmic details, but the functionality. Similarly, in knowledge systems like CP(FD) systems, the inferencing details may be complex but the application programmers are only interested in modeling the problem.

### 3.3. Component-based framework

A component is a self-contained functional unit that has a clear interface and is highly reusable. A component-based framework describes a system as a collection of components and their interfaces. Reuse is achieved by providing an established contract on how the different components can interact. Examples of the CP framework is seen in the design of ILOG Solver [15]. However, ILOG Solver only provides an interface for extending the CP library. To have more flexibility, we conceived the idea of a component-based framework for CP(FD) called Figaro [14, 18, 6]. It is an experimental platform designed to systematically study the different aspects of CP(FD). Our CP(FD) framework allows the exploration of interesting combinations of components for customized CP(FD) systems.

The framework views the CP(FD) system as a reactive system. For simplicity, we work on a sequential CP(FD) system. Figure 4 shows the system model of the framework. The constraint store, simply called *store*, hosts *variables*, constraints called *propagators*, and *propagation engine* for performing constraint propagation. A variable represents an unknown in the problem. A variable contains a *finite-domain* that represents the set of feasible values the variable can take. When the finite-domain of a variable is reduced, the variable wakes up the propagators linked to it and places them in the propagation engine. The propagation engine propagates the woken up propagators until a fixed-point is reached. A fixed-point is reached when filtering cannot remove any further values from all the finite domains. A propagator uses its filtering rules to reduce its variables' finite-

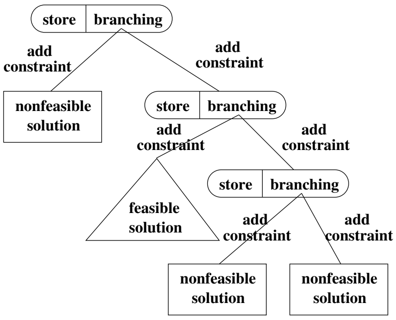



**Figure 5. Search in Framework**

domains. The domain reduction is communicated from the propagator to the store and onto the variables. The propagator can modify its form to a more simple one or replace itself with another propagator with stronger reasoning upon certain conditions. These replacements need to be communicated to the store. The store mediates the communication among three basic elements: variable, propagator and propagation engine.

Search finds solutions that satisfy the constraints of the problem. Figure 5 shows the design of search in the framework. A solution is the set of feasible assignments of value to variable without violating any of the constraints. Search systematically tries for all possible value assignments by adding constraints (e. g. , binding values to variables) along the paths of the search tree. In between search, constraint propagation reduces the search space. The store, together with the branching object, describe a search state. The *branching* object represents a choice point, which adds constraint to a store to produce the next search state. Upon adding of constraint to a store, which triggers constraint propagation, the computation state results to one of the following possible states: feasible solution, non-feasible solution, or cannot be determined, and the search carries on.

Based on the above system model, the framework identifies five major customizable components of a CP(FD) system:

**Finite-Domain** component defines the internal representation of finite-domain. Common representations include bit vector or list of intervals [4].

**Propagator** component encodes the propagation algorithm. Recent research in specialized filtering algorithms [23] provides better propagation performance for specific constraints. To improve reusability, we designed a generic interface, GIFT [19], for reusing filtering algorithms across different CP(FD) systems provided that they have a C/C++ interface for programming constraints.

**Propagation Engine** component dictates the interaction between variables and propagators to reach a fixed point. Examples include propagator-based propagation in Mozart/Oz or variable-based propagation in ILOG Solver.

**Store** component defines the memory policy for state restoration in search. Besides trailing and copying/recomputation, the framework also allows us to come up with new policies, namely lazy-copying and batch recomputation [7].

**Search** component, parameterized by the branching algorithm, defines the search aspects of CP(FD) (see Section 3.1). Common branching algorithms include a simple labeling procedure (naive enumeration of variables) and variable ordering (such as first-fail). The framework allows us to design a new framework for describing complex search scenarios [8].

The first widely used framework is the Model/View/Controller (MVC) framework [16] and this CP(FD) framework design bears semblance to the MVC framework. There are three components in the framework where the interactions involve keeping consistency among the different components. For example, the presentation layer (the view) must be made consistent with the application object (the model). Propagation to fixed point is nothing but an operation to maintain consistency between the variables and constraints.

### 3.4. Design Patterns

The use of design patterns is popularized by [10]. A design pattern describes a recurring solution to a software development problem using a systematic documentation standard. Design patterns are at a more abstract level of reuse. Each reuse of a design pattern requires a *re-implementation* and design patterns serve to capture the design considerations to facilitate the re-implementation.

A framework will usually consist of a variety of design patterns. For example, the design of the proposed framework in the previous section applies existing design patterns found in [10]. The propagation triggering mechanism makes use of the Observer pattern. The store uses the Mediator pattern to mediate the interaction between the propagators and variables during constraint propagation. However, a design pattern should not be confused with a framework. A program code is an instance of a design pattern but a program code can be part of the framework.

Beside making use of design patterns, we propose the constraint propagation as a new design pattern for reuse in different types of software applications. In industrial research, the use of constraint propagation techniques has been applied to a variety of domains such as planning [11], intelligent user-interfaces [11] and scheduling [3]. Our experience with such constraint-based approaches is that there is too much variability in the design of propagation engines. Component-based framework and software toolkit are too rigid to describe the possible propagation paradigms. It is in our view that to support reuse of constraint propagation engines, the level of abstraction should lie at the level of design patterns. Design patterns capture our experience in designing and implementing constraint propagation engines.

We describe the Propagation Engine pattern in brief and a more detailed version could be found in [21].

**Intent** Manage the many-to-many dependencies among objects so that when one object changes state, the state can be propagated with *reasonable efficiency* to direct and indirect dependents till either a fixed-point or a failure condition.

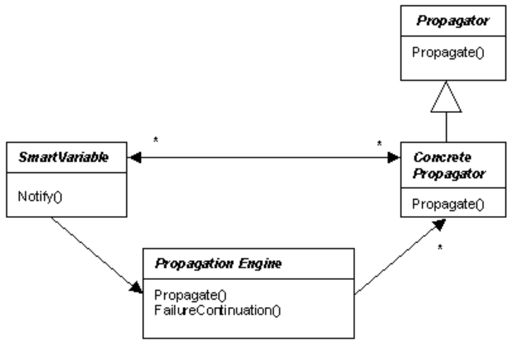
**Motivation** Managing dependencies among a set of objects is complicated. There is a need to factor in efficiency, guaranteed termination and handling of local inconsistencies. We could achieve consistency by making a monolithic system but that will decrease reusability.

For example, a user interface propagation engine propagates a set of geometrical constraints. As constraints are added, the propagation engine manages the propagation by calling the *affected* propagators one by one from a queue. When a propagation affects a variable that is linked to other propagators, the propagation engine is notified with the new propagators. It is the role of the propagation engine to schedule these propagators in the propagation queue and to detect any violations in the user interface and resolve them in the best possible way (in terms of presenting widgets in a graphical user interface).

**Applicability** Use the Propagation Engine pattern in the following situations:

- When a set of objects, whose states are dependent on one another, needs to be made consistent.
- When a monolithic algorithm is decomposed into separate classes to facilitate the reuse of parts of the algorithm, and there are strong interconnections among these subparts.

**Structure and Participants** The **PropagationEngine** ensures propagation to fixed point or failure. The **Propagator** defines the propagating interface for **ConcretePropagator**, which propagate information based on changes in the **SmartVariable** state.




**Consequences** The Propagation Engine pattern has the following consequences. First, it allows flexible combinations of propagators. Second, efficiency concerns is modularized. Third, there is reflection of the propagation process. Fourth, exceptions are monitored and handled by the propagation engine.

Like the Mediator pattern, Propagation Engine pattern centralizes control. The coupling is not as loose as the Mediator pattern because a Propagator manipulates a specific class of SmartVariable. We could simplify interconnections at the expense of efficiency by incorporating this pattern with the Mediator pattern.

Like the Observer pattern, Propagation Engine pattern handles the dependencies among objects. Specifically, Propagation Engine pattern handles objects which have many-to-many relationships. Unlike Observer pattern, Propagation Engine pattern directly manages the complexity.

## 4. Discussion

Through our experience in applying reuse techniques to the development of CP(FD) systems, we learnt several important lessons. The specific insights from each approach will be covered before we give the general themes of the lessons we learnt.

### 4.1. Software Toolkit

The success of the toolkit approach shows that algorithmic behavior can be parameterized. A program is composed of code and data. By regarding code as another form of data, we are able to characterize the exploration behavior of the search algorithm. For example, breadth-first search uses a queue of nodes compared to depth-first search which uses a stack of nodes. By changing a queue to a stack, we change the algorithmic behavior of the code.

One drawback is that by imposing a set of design dimensions on an algorithm, it makes it difficult to introduce new dimensions. This difficulty arises from the fact that these design dimensions create a rigid framework for designing new algorithms. Of course, in some cases, the design may be flexible enough to handle new dimensions. In general, adding new dimensions is not trivial, and because of this reason, it may stifle creativity in searching for fundamental improvement to the algorithms. For the case of search algorithms, there has been significant research in programmable and parameterizable search algorithms, and hence, the dimensions are well-understood, justifying the design decision.

### 4.2. Scripting Approach

Extending a scripting language like Tcl to support CP(FD) allows the application programmers to spend time on other equally important aspects of the application development such as user interface, web-enabling and database connectivity. This paradigm succeeds because there are many readily available components/interfaces built for scripting languages.

Another way to look at it is to view a knowledge system containing aspects such as modeling, solution approach and computation. Modeling and solution approach can be quite independent of computation and a scripting approach brings this principle to it natural conclusion by creating a layer that does only modeling and solution approach. The invisibility of the computation aspects is best highlighted by the use of SQL. It is quite common for application programmers to be unaware of the specific query optimization technique and yet are able to create complex databases and perform equally complex queries.

Despite improvements in computer performance, efficiency is still an important issue in scripting languages. In particular, in the scripting approach, even though search (solution approach) is not a bottleneck operation, it still takes up significant time in scripting-based CP(FD) applications simply because we implement search at the scripting level. The more obvious pitfall is that like most prototype systems, it is easy to view the script application as the final system. Although scripting applications work in many cases, it is often a sub-module and may require further integration with a larger system. Hence, there may be even more speed performance issues. Another evil worth pointing out is that scripting does sometimes promote poor programming practices because of their lack of strict typing.

### 4.3. Component-based Framework

The component-based framework was a natural extension to a library of propagators implementing the different constraints. By characterizing the algorithmic behaviors of each CP(FD) aspect into modularized components, we are able to provide more expressivity in integrating the components. Though there is overhead in the design, it is surprising from some (not all) anecdotal evidences that due to the expressivity, there are improvements in speed performance. We are also able to experiment with different designs with greater ease.

As mentioned, despite improvement in expressivity and algorithm design, overheads impose a heavy penalty of the runtime performance. For example, the memory management of objects can be difficult to handle, especially when using a programming system without automatic garbage collection. However, it is possible that some of the overheads are unnecessary and can be removed [6]. Another potential failure is that the framework introduces a mechanistic process to the design of CP(FD) systems and may end up again as stifling creativity in the design of new systems.

### 4.4. Design Patterns

The necessity of the propagation engine design pattern was a result of our “failure” in creating a flexible, reusable propagation engine. Propagation engines are the bottleneck operations of a CP system and therefore, a generic design, even if possible, would be the main cause of inefficiency in the CP system. In addition, the great variability of the design of propagation engines makes it difficult to come up with a set of reusable components for propagation engines. However, from experience in developing propagation-based systems, there are some themes and common pitfalls that we often run into. It seems to make sense to document these design decisions and pitfalls to avoid in the form of a design pattern.

The implicit use (based on our repeated experiences in implementing propagation-based systems) of the design pattern has been effective in the past in developing new propagation-based systems, but it is hard to determine that if the experience can be transferred to another programmer through documentation. Interestingly, the use of the existing design patterns has been helpful in the implementation of the CP(FD) systems and based on this knowledge, we can hope that documenting the propagation engine pattern down can lead to the same kind of benefits.

### 4.5. General Themes

From the specific lessons, we gather some general themes in developing knowledge systems and list them below.

1. We can generally divide the algorithmic behavior into coherent subunits. For instance, we split CP(FD) systems into two parts, namely constraint propagation and search. These two parts are further subdivided into

different functional units or components in our framework. Such a divide-and-conquer approach makes the system more maintainable by localizing the impact to the system for changes made during ongoing research. It also makes the system more adaptable to changes.

2. The architecture design of knowledge systems can be decomposed into several layers. The kernel operation in CP(FD) systems is the propagation engine. Search sits on top of the propagation engine and there is an application layer that sits on top of search. We describe the propagation engine using design patterns, search using toolkit and a scripting approach to provide an application front-end. Such a layering approach allows us to systematically study the different issues involved in the development of a CP(FD) system using different software engineering concepts.

3. Using known design patterns makes system-specific issues reusable, proving that design patterns is applicable in knowledge systems. In our CP(FD) framework, casting the store object as the Mediator pattern makes state restoration in search more reusable. Beside existing state restoration policies, we were able to derive new state restoration policies. However, the implementation cannot achieve the level desired in the "original" implementation. For instance, we can only implement trailing at a coarse level of granularity [7] in comparison to the conventional trailing. This instance is a trade-off against level of details in the use of a software framework.

4. Although it is not discussed in detail in the previous subsections, memory management deserves more attention, even for automatic garbage collection systems. We implement each functional units in the framework as objects. However, the complex interaction among these objects during search makes memory management rather difficult to handle (or predict, if we are using a automatic garbage collector). Effective memory management among these objects requires considering a macro perspective. Having the objects use its own local memory management policies may cause memory problems such as memory leaks or pointing to null objects. These problems are the consequences of lacking coordination in memory management.

5. We discover that reusable operations is not a silver bullet to the problems involved in the development of CP(FD) systems and most likely, knowledge systems. There are four known problems.

   First, as we strive to partition the system into localized functional units, there are still times where subtle changes in the algorithm features may lead to a significant re-implementation of most of the operations. For instance, the different schemes to support the logical operators to combine constraints, can be completely different from each other Switching from one scheme to another would require a lot of re-implementation effort.

   Second, the rigidity of some of the framework or toolkit limits creativity in software design. Programmers usually end up thinking in the constraints of the framework or toolkit. As mentioned in the first reason, even if the programmers has the creativity, it is often a tedious process to exercise that creativity.

   Third, the overheads created by software engineering may be crucial to the implementation of knowledge systems. These systems often require efficient space and time performance. However, as practitioners, we should be able to make good estimates on the final acceptable performance based on known historical trends, such as Moore's Law, and hence, the level of software engineering we should apply.

   Fourth, well-designed system framework is not a substitute for good programming. Especially for the scripting approach, it is very easy to create an unmanageable system if we do not program with discipline either using structured programming or object-oriented programming concepts.

   Some answers to the problems above may depend on the use of design patterns. Rather than providing rigid frameworks or toolkits, we use design patterns to articulate the thoughts of the framework and toolkit design. A design pattern also allows the programmer to decide the level of abstraction needed. It remains an open question if knowledge system design patterns could help us shorten the development time when we need to re-implement parts of the system.

   Even with design patterns, it is not surprising that the fourth problem mentioned above still remains. More self-awareness on the need of having good programming practice rather than relying on processes or frameworks may be the best option we can take in overcoming the problem.

## 5. Conclusion

Knowledge systems usually come out of research but these systems are applicable to the real world. Unfortunately, software engineering has not been a priority in research. Hence, there is a disparity in terms of quality between knowledge system software and other system software. Applying software engineering techniques to knowledge systems bears the potential of significantly improving their quality. Here, we concentrated on reuse techniques for constraint programming systems, a typical class of knowledge systems. Our research in this area covers toolkit, scripting, component-based framework and design patterns approaches. We showed how these approaches are applied, highlighted their impact on software reuse and provided evidences and suggestions for the potential benefits of applying these techniques in industrial practice.

## References


[1] M. Austern. *Generic Programming and the STL*. Addison-Wesley, Reading, MA, 1998.

[2] D. 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.

[3] M. S. Boddy. Temporal reasoning for planning and scheduling in complex domains: Lessons learned. In A. Tate, editor, *Advanced Planning Technology Technological Achievements of the ARPA/Rome Laboratory Planning Initiative*, 1996.

[4] M. Carlsson. Finite domain constraints in sicstus prolog. Presentation Slides, Invited Talk at CICLOPS 2001: Colloquium on Implementation of Constraint and LOgic Programming Systems, a post-conference workshop of ICLP 2001/CP 2001, 2001. available at `http://www.cs.nmsu.edu/~complog/conferences/iclp01/ciclops2001.ppt`.

[5] T. Y. Chew, M. Henz, and K. B. Ng. A toolkit for constraint-based inference engines. In E. Pontelli and V. S. Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, Lecture Notes in Computer Science 1753, pages 185–199, Boston, MA, 2000. Springer-Verlag, Berlin.

[6] C. W. Choi. *Advanced Components for Finite Domain Constraint Programming*. Master's thesis, School of Computing, National University of Singapore, 2002.

[7] C. W. Choi, M. Henz, and K. B. Ng. Components for state restoration in tree search. In T. Walsh, editor, *Principles and Practice of Constraint Programming—CP 2001, Proceedings of the Seventh International Conference*, Lecture Notes in Computer Science 2239, pages 240–255, Cyprus, 2001. Springer-Verlag, Berlin.

[8] C. W. Choi, M. Henz, and K. B. Ng. A compositional framework for search. In *Proceedings of CICLOPS 2001: Colloquium on Implementation of Constraint and LOgic Programming Systems, a post-conference workshop of ICLP 2001/CP 2001, appears as Technical Report NMSU-CSTR-003/2001*, Laboratory for Logic, Databases, and Advanced Programming, New Mexico State University, 2001. available at `http://www.cs.nmsu.edu/TechReports/`.

[9] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, MA, 6th edition, 1995.

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

[11] R. P. Goldman, K. Z. Haigh, D. J. Musliner, and M. Pelican. MACBeth: A Multi-Agent Constraint-Based Planner. In *2000 AAAI workshop "Constraints and AI Planning"*, pages 11–17, 2000.

[12] M. Henz. *Objects for Concurrent Constraint Programming*. The Kluwer International Series in Engineering and Computer Science, Volume 426. Kluwer Academic Publishers, Dordrecht / Boston / London, 1998.

[13] M. Henz and T. Müller. An overview of finite domain constraint programming. In *Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies Within IFORS*, Singapore, 2000.

[14] M. Henz, T. Müller, and K. B. Ng. Figaro: Yet another constraint programming library. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, Las Cruces, New Mexico, USA, 1999. held in conjunction with ICLP'99.

[15] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *ILOG Solver 5.0, Reference Manual*, 2000.

[16] W. R. LaLonde and J. R. Pugh. *Inside Smalltalk, Volume II*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[17] Mozart Consortium. The Mozart Programming System. Documentation and system available from `http://www.mozart-oz.org`, Programming Systems Lab, Saarbrücken, Swedish Institute of Computer Science, Stockholm, and Université catholique de Louvain, 1999.

[18] K. B. Ng. *A Generic Software Framework For Finite Domain Constraint Programming*. Master's thesis, School of Computing, National University of Singapore, 2001.

[19] K. B. Ng, C. W. Choi, M. Henz, and T. Müller. GIFT: A generic interface for reusing filtering algorithms. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000, appears as Technical Report TRA9/00*, School of Computing, National University of Singapore, 55 Science Drive 2, Singapore 117599, Sept. 2000. available at `http://techrep.comp.nus.edu.sg`.

[20] K. B. K. Ng. A scripting approach to finite domain constraint programming. Honours Year Project Report, School of Computing, National University of Singapore, 1999.

[21] K. B. K. Ng and C. W. Choi. The propagation engine design pattern. draft, 2002.

[22] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Maassacusetts, 1994.

[23] J.-C. 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.

[24] C. Schulte. Oz Explorer: A visual constraint programming tool. In L. Naish, editor, *Proceedings of the International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press, Cambridge, MA.

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

[26] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly, Cambridge, second edition, 1996.