Component Programming and Interoperability in Constraint Solver Design

Frédéric Goualard*

Centrum voor Wiskunde en Informatica
P.O. Box 94079, NL-1090 GB AMSTERDAM
`Frederic.Goualard@cwi.nl`

Abstract. Prolog was once the main host for implementing constraint solvers. It seems that it is no longer so. To be useful, constraint solvers have to be integrable into industrial applications written in imperative or object-oriented languages; to be efficient, they have to interact with other solvers. To meet these requirements, many solvers are now implemented in the form of extensible object-oriented libraries. Following Pfister and Szyperski, we argue that "objects are not enough," and we propose to design solvers as component-oriented libraries. We illustrate our approach by the description of the architecture of a prototype, and we assess its strong points and weaknesses.

1 Introduction

From the 1980's onward, constraint programming techniques have been considered a generalization of logic programming algorithms. The use of Prolog as a hosting language for constraint solvers was then natural. The principles of *constraint logic programming* have been embodied by many systems, such as Prolog IV [2], CHIP [6], GNU Prolog (formerly clp(FD)) [5] and ECL^{i}PS^e [1], to name a few.

At first sight, Prolog seems to be the ideal host for constraint programming: it enforces declarative programming, which makes it a neat modelling language, and it offers an elegant way of handling combinatorial problems thanks to the availability of nondeterminism at the language level. Prolog, however, suffers from two flaws:

- it is far from being widely used in the software industry. As a consequence, the integration of constraint solving facilities in an application, most probably written in a different language, requires *ad hoc* bridges and additional expertise in Prolog programming from the developers;
- cooperation of different constraint solvers is the key to solve hard problems. However, a Prolog interpreter/compiler is a closed system. Adding a new solving algorithm usually demands its direct implementation into the

system, and the use of its internal data structures. At best, Prolog-based environments are extensible. They do not offer *interoperability*, that is the possibility to communicate and use software components written independently.

In the beginning of the 1990's, Puget [20] tackled the first point above by showing that it is possible to benefit from the salient features of Prolog in a mainstream programming language, namely C++. Since Puget's library Ilog Solver, many object-oriented libraries have been devised for solving constraints: INC++ [12], QDOOCS [24], OpAC [9] . . . Most of them offer original features; being libraries, all of them are easily extensible. Nevertheless, none of them addresses the second point, that is interoperability. Every library has its unique internal data structures, and consequently, reusing directly solving methods from different libraries is not possible. Yet, as said previously, the cooperation of solvers with different strengths is the key to solve hard problems [3, 16, 15]. Reimplementing all required methods in one system is too inefficient to be a viable solution; in addition, which solver engine should be used?

The problem of making cooperate different solvers has long been recognized and addressed in different ways. We present a critical overview of some of the proposed solutions in Section 3, after having introduced some basics on constraint solving in Section 2.

The lack of interoperability between libraries is not an issue limited to constraint programming. There was a time when enthusiastic foretellers were predicting that object-oriented programming would create a component market; applications would then be built by connecting arbitrary components in the same way as one uses Lego blocks. Sadly enough, Pfister and Szyperski [19] have shown that the concept of object is insufficient to ensure the creation of true components. However, solutions do exist, which make *component programming* a dream (almost) come true. We present in Section 4 the ideas underlying component programming, and we point out the advantages offered by this paradigm. In particular, we show what constraint programming can gain from component programming. To support our claim, we present in Section 5 the prototype of a C++ library that uses component programming techniques. Finally, we discuss in Section 6 the pros and cons of our approach and we consider the possible future of *component constraint programming.*

2   A Short Conspectus on Constraint Programming

Even an overview of constraint programming at large would deserve a whole paper on its own. Hence, we will only consider in this section the framework that is currently supported by the library to be described in Section 5. The reader should nevertheless keep in mind that the approach proposed in this paper is fully general.

A *Constraint Satisfaction Problem* (CSP) is defined as follows: given a finite set of variables $\mathcal{V} = \{v_1, \ldots, v_n\}$, a Cartesian product of domains $\mathbf{D} = D_1 \times \cdots \times D_n$ (with $v_i \in D_i$ for any value of i in $\{1, \ldots, n\}$), and a set of relations

$\mathscr{C} = \{c_1, \ldots, c_m\}$ between the variables in $\mathscr{V}$ (*constraints*), we seek for all the possible assignments to the variables that satisfy the conjunction $c_1 \wedge \cdots \wedge c_m$.

Some popular methods for solving CSPs involve *local consistency* enforcement. The use of local consistency notions for solving constraints can be traced back to the observation by Fikes [7] that, given a constraint $c(v_1, v_2)$ between two variables v_1 and v_2 taking their values in the discrete domains D_1 and D_2, it is possible to discard a value a in D_1 whenever there is no value b in D_2 such that $c(a, b)$ holds. This idea is at the root of the various algorithms to enforce local consistencies, such as the Waltz algorithm [23] or AC3 [13] for computing *arc consistency* and *hyper-arc consistency*.

Definition 1 (Hyper-arc consistency). *Let $c(v_1, \ldots, v_n)$ be a constraint, $i \in \{1, \ldots, n\}$ an integer, and $\mathbf{D} = D_1 \times \cdots \times D_n$ a Cartesian product of domains. The constraint c is said* hyper-arc consistent w.r.t. $\mathbf{D}$ *and v_i if*

$$D_i = \pi_i(\rho_c \cap \mathbf{D}) \tag{1}$$

where ρ_c is the relation associated to the constraint c, and $\pi_i(\rho_1 \times \cdots \times \rho_{i-1} \times \rho_i \times \rho_{i+1} \times \cdots \times \rho_n) = \rho_i$, for ρ_k ($k \in \{1, \ldots, n\}$) unary relations. The constraint c is said hyper-arc consistent w.r.t. $\mathbf{D}$ if Eq. (1) is verified for all i in $\{1, \ldots, n\}$. A constraint system $\mathscr{C} = \{c_1, \ldots, c_m\}$ is hyper-arc consistent if each c_i ($i \in \{1, \ldots, n\}$) is hyper-arc consistent.

Enforcing hyper-arc consistency over a constraint system is done by a propagation algorithm such as the one given in Table 1, which applies a contracting operator N on each constraint in turn until reaching a fix-point. Provided the operators N fulfill some basic properties, algorithms like NC3 are confluent and terminating [11].

Table 1. The NC3 propagation algorithm

```
NC3(in {c_1,...,c_m}; in/out D = D_1 × ··· × D_n)
begin
    𝒮 ← {c_1,...,c_m}   % Constraints added to the propagation set
    while (𝒮 ≠ ∅ and D ≠ ∅) do
        c ← choose one c_i in 𝒮
        D′ ← N_c(D)   % Enforcing local consistency on c and D
        if (D′ ≠ D) then
            𝒮 ← 𝒮 ∪ {c_j | ∃x_k ∈ Var(c_j) ∧ D′_k ≠ D_k}   % Var(c_j): set of variables
            D ← D′                                          % occurring in c_j
        endif
        𝒮 ← 𝒮 \ {c}
    endwhile
end
```

Hyper-arc consistency can be expensive to compute for discrete variables since it induces holes in their domains; in addition, it is uncomputable for continuous variables. Therefore, several weakening of hyper-arc consistency have been defined: *bounds consistency* [14] where only the bounds of the domains are considered, *box consistency* [4] for continuous CSPs . . . Due to lack of space, we will not present these consistencies and the reader is referred to the references given for additional information.

3 Making Solvers Cooperate

In the following, we use the term *solver* in a broad sense, namely: a solver is any "box" that takes as input some constraints and the domains of the variables involved, and that returns as output a set of constraints that defines the same solution set as the input ones and the domains of the variables that have been possibly tightened. Hence, we consider as solvers procedures that compute redundant constraints, or that simplify them. Most solvers, however, will leave the constraint set unchanged and will simply narrow down the domains of the variables.

Solvers can be limited to some particular kind of constraints (the simplex method for linear equalities, for instance); alternatively, some costly solvers are best used in conjunction with other simpler ones (symbolic methods and interval methods, for instance [3]). The cooperation of solvers has been shown to be a key concept in solving hard problems. One way to take this fact into account is to tightly integrate several solvers in one system (see Prolog IV [2]). However, such a centralization hinders the development of new methods. Another way is to have different independent tools cooperate.

Several schemes have been devised for allowing such a cooperation between systems with different inner structures. **CoSAc** [16], for instance, is a client/server architecture to manage the exchange of information among different solvers through *pipes*. The data exchanged are character strings. In the same spirit, AMPL [8] allows one to hook a new solver to the system through a very simple interface. The AMPL kernel supervises the exchange of information through files in a particular format. Both systems are interesting in that they are truly open; they allow cooperation between completely independent solvers. The amount of work to add a new solver only requires the addition of a small interface to have it understand the format used for inputs and outputs. However, the cooperation happens at a fixed level and a solver cannot obtain additional information from the internal structure of the constraint store that would allow it to speed up the solving process. In addition, the use of a primitive format for information interchange, namely text, incurs a significant slow down. As a consequence, communications must be kept to a minimum. A side effect is also that nothing can prevent a priori the connection of a solver that do not understand correctly the input format (no type safety). Lastly, these schemes do not allow distributed cooperation.

To overcome some of these problems, Ng *et al.* [17] have proposed a generic C++ interface for connecting systems such as Ilog Solver [20] and Oz [21]. The communication takes place at the lowest level, which ensures performances, and type safety is guaranteed by the language. A major inconvenience lies in that this communication scheme often requires the extension of the systems to provide some necessary hooks. On the technical level, all solvers have to be written in C++ and compiled with the same compiler to be able to communicate. In addition, the level of cooperation is once again fixed and depends on the interface specifications.

In fact, the lack of interoperability is a language issue that is not limited to constraint programming systems. For many years, it was believed that object-oriented programming would permit the development of interoperable components. Pfister and Szyperski [19] have shown that it is not the case. However, a new paradigm, *component programming*, has emerged, that allows us to meet this goal.

4 Component Programming

In an object-oriented language, communication is done by method calls between objects. Usually, an object cannot be used in isolation because it does not represent a concept and it relies on services provided by other objects. In addition, objects have to know which objects they must call to send and receive information.

By contrast, Szyperski [22] defines components as *units of deployment*. They may be composed of one or several objects; they might even be some non-object-oriented piece of code.

The communication model among components is based on *slots* and *signals*. A component whose state has been modified sends the information as a *signal* (an *event* in the Java terminology) to the other components that might be interested in it. A component that wants to be informed of the modification of another component (a *listener* in Java terminology) connects its input slot to the output slot of that component. It is important to stress that the sender does not know anything about the components listening to its signals. It even does not know whether anybody is listening at all. On the other side, the receiver does not have to know the precise type of the sender. Suffice that the sender provides the right kind of signal. Several components can listen to the same slot, and a component can listen to several slots. In addition, a component can be dynamically connected to, and disconnected from a slot. Lastly, two connected components can indifferently run on the same or on different computers; this is transparent to the communication process.

The communication model by signals and slots is already extensively used in Graphical User Interface libraries, such as Qt, the GUI layer for the KDE environment (www.kde.org), awt and Beans in Java, or Gtk, the layer for the Gnome environment (www.gnome.org). It is also part of Microsoft's *Component Object Model* [25].

There are few languages that offer a native support for component programming. However, component programming can be achieved at a reasonable cost with most object-oriented languages such as C++ and Java.

Components can be freely connected provided they agree on the kind of messages they want to exchange (type safety). They can communicate transparently in a distributed environment. The cost of communications is reasonably cheap since the communication mecanism is at the same level as the rest of the code. Lastly, protocols such as CORBA [18] allow communications between programs written in different languages.

All these qualities led us to devise aLiX (*a Library for Constraint Solving*), a C++ constraint solving library based on the component programming paradigm. The description of its architecture is the subject of the next section; The impact of this approach on constraint solver design and the new possibilities offered are discussed in Section 6.

5   The aLiX architecture

The implementation of aLiX is still a work in progress, though we already have a full system that allowed us to validate our approach. Due to lack of space, we only skim over the salient features of the aLiX architecture to show the new possibilities offered by the component programming paradigm.

The core of aLiX is composed of four main concepts (see Fig. 1):

variables. The base class for the variables offers three input slots and two output slots:

get_domain. This is the access point to modify the domain of the variable,

sharing_domain. This is an input slot used for sending information outside (cf. the comment by Szyperski [22, p. 149]). Components send a message to this slot which is used to encapsulate and retrieve the current domain of the variable,

reinit_domain. The variable reinits its domain with the one sent to this slot. The difference with the get_domain access point lies in that such an assignment is usually not considered as an event (though this can be modified by inheriting from the base class and redefining the handler of the slot),

trailing. Each time the variable is about to modify its domain, it first sends its current domain through this slot. A backtrack stack is usually connected to this slot just before enumerating the values of the variable's domain,

domain_changed. After any modification, the variable sends its new domain through this slot;

One can inherit from the variable base class to add new slots. For example, aLiX contains the class integral_variable (variable with a discrete domain), which uses three additional output slots to send messages whenever the hull of the domain, the left bound, or the right bound have been changed, and one slot to send a message upon instantiation of the variable.

constraints. The class associated to this concept is abstract and is only used to offer some facilities such as a unique identifier. Each constraint to be solved is represented by a constraint object that connects itself to the `domain_changed` (or `hull_changed`, ... for that matter) slot of all the variables involved in the relation (`get_notified` slots). It possesses also some output slots to be connected to *narrowing objects* (cnos) to create two-ways channels. A narrowing object has input slots to receive domains. It reduces these domains by enforcing some consistency and returns them through the channel. One constraint may use several narrowing operators at a time (with different tightening abilities); it can also disconnect itself from some cnos and connect itself to other cnos dynamically to choose the best suited cnos at any moment. The message exchanged with the cnos contains the domains to be tightened and a flag to be raised by a cno whenever some failure occurs. The output slot is *marshalled* in such a way that the dispatching of the domains to the other cnos listening to the slot is immediately stopped in case of failure. Whenever a constraint receives a message from a variable whose domain has been modified, it has the possibility to send some messages to the cnos to reduce immediately the domains of the other variables; alternatively, it can send a message through its `ask_for_reinvocation` slot to be managed by a scheduler (see below);

schedulers. A scheduler is an object centralizing the requests for reinvocation of the constraints. It is used to implement propagation algorithms such as AC3 [13]. However, a scheduler is fairly general in that it can handle indifferently constraints, variables, or any other object inheriting from the `schedulable` class. Such a flexibility allows us to use a constraint-oriented propagation scheme (where the propagation queue contains constraints) or a variable-oriented propagation scheme (where the propagation queue contains variable whose domain has been modified) as implemented in clp(FD) [5] by simply selecting different connection schemata;

enumerators. Once all the constraints have been added to the store, an enumerator is used to separate the different solutions and to overcome the incompleteness of the consistencies enforced. A discrete enumerator assigns to each variable every value in its domain and reinvokes the propagation process to check the consistency of the assignment. A continuous enumerator splits recursively the domains of the variables (cf. the `solve` procedure available in many CLP systems). An enumerator is also responsible for managing a backtrack stack on which are saved the domains of the variables to be recovered upon backtracking. The user can define different enumerators to select a particular heuristics for choosing the next variable to be considered, for instance.

At present, aLiX relies on the ADDL library [10] (*A Discrete Domain Library*) for representing the domains of the discrete variables. This library offers both connected domains (intervals) and domains with holes. Domains are considered as STL containers and offer STL-like iterators to retrieve values, which permits to consider connected and disconnected domains in a uniform way. Discon-
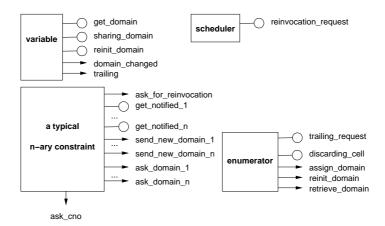
Fig. 1. Core of aLiX

nected domains use a reference-counting mecanism with copy-on-write semantics (see the C++ FAQ Lite at http://www.cerfnet.com/˜mpcline/c++-faq-lite/) to ensure the best performances.

Table 2 presents an aLiX program to solve the n queens problems. This program uses disconnected domains and a variable-oriented propagation scheme as in GNU Prolog. It implements the naive algorithm that is a direct representation of the specifications of the problem. We have also tested some other formulations proposed by Puget that use global `alldiff` constraints and a constraint-oriented propagation scheme. We have then be able to check that the coding of the alternative solutions only required a different connection schema between the constraints and the variables.

6 Discussion and Perspectives

We have tested the current version of aLiX on several standard benchmarks such as the n queens problem. It appears that our library is roughly three times slower than GNU Prolog 1.2.1 and three times faster than ECL^{i}PS^e 5.1.0 on the same formulations of the problems. We have however identified several points where optimization might dramatically speed-up the computation. All the tests have been performed on discrete CSPs. Addition of components for solving continuous CSPs is one of our priority for the near future. We believe that the overhead incurred by the more elaborate communication scheme will be less acute for this kind of problems since the time spent in the narrowing objects is likely to overwhelm the one spent in communications.

Though still at an early stage, we have been able to check the versatility of our approach by writing different formulations for the same benchmarks and

```
// Constraint narrowing operator for the variable x
class out_of : public cno {
public:
   out_of(fd_variable& x, fd_variable& y, int c) {
     connect(x.get_domain,send_domain);
     connect(x.sharing_domain,ask_domain_x);
     connect(y.sharing_domain,ask_domain_y);
     cst=c;
   }
   bool invoke() { // Called when y is instantiated
     // Retrieving current domain of x
     ask_domain_x.share(dom_x);
     ask_domain_y.share(dom_y);
     // x <- x\{d,d+cst,d-cst}
     ((dom_x %= d) %= d+cst) %= d-cst;
     send_domain.send(dom_x); // Sending new domain to x
     return dom_x.is_not_empty();
   }
output_slots:
   output_slot<fd_domain> send_domain;
   output_share_slot<fd_domain> ask_domain_x,
ask_domain_y;
private:
   fd_domain doma_x, dom_y;
   int cst;
};

// {x<>y, x<>y+i, x<>y-i} in one constraint
class diff3 : public constraint {
public:
   diff3(fd_variable& x, fd_variable& y, int i) {
     scheduled.insert(scheduled.begin(),&x);
     scheduled.insert(scheduled.begin(),&y);
     x.on_instantiation(new out_of(y,x,i));
     y.on_instantiation(new out_of(x,y,i));
```

```
   }
};

int main() {
   int n; cout << "N? "; cin >> n;

   Vector<fd_variable*> x(1,n);
   fifo_scheduler store;

   for (int i=1;i<=n;++i) {
     x.insert(x.begin(),
             new fd_variable(fd_domain(1,n)));
   }
   enumerator_round_robin<fd_domain>
     inst(store,x.begin(),x.end());

   for (int i=1;i<=n;++i) {
     int k=1;
     for (int j=i+1;j<=n;++j) {
       store.post(*(new diff3(*(x[i]),*(x[j]),k)));
       ++k;
     }
   }
   if (store.run()) {
     if (inst.first_solution()) {
       // [Display domains]
       while (inst.next_solution()) {
         // [Display domains]
       }
       cout << "No more solutions" << endl;
     } else {
       cout << "No solution" << endl;
     }
   }
}
```

Table 2. The n queens problem formulated in aLiX

changing the propagation scheme by mere reorganization of the connections between the objects.

Slots offer natural hooks for spying any event occurring inside a solver engine. There is a large number of potential uses of this for writing non-intrusive debuggers or animations of the solving of some problems for educational purpose, for instance. Suffice for the applications to connect themselves to the appropriate slots without interfering in any way with the solver.

The current version of aLiX uses ad hoc classes for implementing the signal/slot mechanism. We are in the process of replacing the use of these classes by the library sigc++ (libsigc.sourceforge.net). This library is the one used in Gtk for connecting components. An extension of it already exists to allow communications in a distributed environment. Another extension is planned in the near future to support the CORBA protocol. By using sigc++, aLiX will offer both the possibility to solve constraints in a distributed environment in a transparent way, and the ability to use programs implementing narrowing operators written in a language different from C++.

The use of an *Object Request Broker* greatly augments interoperability but it incurs a non-negligeable cost. As a consequence, we believe that its use should be only considered when the time spent in actual computation in the components is large enough to make the communication time insignificant. Such a situation is most likely to arise when components implement narrowing operators for global constraints or costly processes such as factorization of the expression of the constraints, or computation of a Gröbner basis, for instance.

The interoperability offered by the component programming approach can be used at various levels. However, the most visible one seems to lie in the possible plugging of third party narrowing objects (be they real objects or complete applications) onto constraints. At present, narrowing objects have to use the same arithmetic library as the one used by constraints since messages between them are domains. It is however possible to allow a more stringent separation by exchanging messages containing *OpenMath code* (www.openmath.org) or MP packets (http://www.symbolicnet.org/areas/protocols/mp.html), for instance.

Acknowledgements

References

[1] A. Aggoun, D. Chan, P. Dufresne, E. Falvey, H. Grant, A. Herold, G. Macartney, M. Meier, D. Miller, S. Mudambi, B. Perez, E. Van Rossum, J. Schimpf, P. A. Tsahageas, and D. H. de Villeneuve. ECL^iPS^e *3.5*. European Computer Industry Research Centre (ECRC).

[2] F. Benhamou, P. Bouvier, A. Colmerauer, H. Garetta, B. Giletta, J.-L. Massat, G. A. Narboni, S. N'Dong, R. Pasero, J.-F. Pique, Touraïvane, M. Van Caneghem, and É. Vétillard. *Le manuel de Prolog IV*. PrologIA, June 1996.

[3] F. Benhamou and L. Granvilliers. Combining local consistency, symbolic rewriting and interval methods. In *Procs. of AISMC-3*, volume 1138 of *LNCS*, pages 144–159. Springer-Verlag, 1996.

[4] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Procs. of ILPS'94*, pages 124–138, Ithaca, NY, November 1994. The MIT Press.

[5] P. Codognet and D. Diaz. Compiling constraints in `clp(fd)`. *Journal of Logic Programming*, 27(3):185–226, 1996.

[6] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Procs. of FGCS'88*, pages 683–702, 1988.

[7] R. E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[8] D. M. Gay. Hooking your solver to AMPL. Technical Report 97-4-06, Lucent Technologies, Bell Labs Innovations, April 1997.

[9] F. Goualard. *Langages et environnements en programmation par contraintes d'intervalles*. Phd thesis, Institut de Recherche en Informatique de Nantes, Université de Nantes, July 2000.

[10] F. Goualard. *ADDL User Manual v. 1.0a*. Centrum voor Wiskunde en Informatica, 1.0a edition, March 2001. To be released.

[11] H.-W. Güsgen and J. Hertzberg. Some fundamental properties of local constraint propagation. *Artificial Intelligence*, 2(36):237–247, 1988.

[12] E. Hyvőnen, S. De Pascale, and A. Lehtola. Interval constraint satisfaction tool INC++. In *Procs. of the 5th ICTAI*. IEEE Press, 1993.

[13] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 1(8):99–118, 1977.

[14] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[15] P. Marti and M. Rueher. A distributed cooperating constraints solving system. *International Journal on Artificial Intelligence*, 4(1–2):93–113, June 1995.

[16] É. Monfroy, M. Rusinowitch, and R. Schott. Implementing non-linear constraints with cooperative solvers. In K. M. George, J. H. Carroll, D. Oppenheim, and J. Hightower, editors, *Procs. of the ACM Symposium on Applied Computing, SAC '96*, pages 63–72. ACM, 1996.

[17] K. B. Ng, C. W. Choi, M. Henz, and T. Müller. GIFT: A generic interface for reusing filtering algorithms. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, É. Monfroy, T. Müller, L. Perron, and C. Schulte, editors, *Procs. of TRICS'2000*, 2000.

[18] The Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification*, October 1999.

[19] C. Pfister and C. Szyperski. Why objects are not enough. In *Procs. of CUC'96*, July 1996.

[20] J.-F. Puget. A C++ implementation of CLP. In *Proceedings of the Singapore Conference on Intelligent Systems (SPICIS'94)*, 1994.

[21] G. Smolka. The definition of Kernel Oz. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*, pages 251–292. Springer-Verlag, 1994.

[22] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[23] D. L. Waltz. Understanding line drawings of scenes with shadows. In P. H. Winston, B. Horn, M. Minsky, and Y. Shirai, editors, *The Psychology of Computer Vision*, chapter 2, pages 19–91. McGraw-Hill, New York, 1975.

[24] K. Wiese, S. Nagarajan, and S. D. Goodwin. QDOOCS, a C++ class library for solving binary constraint satisfaction problems. Tech. report CS-96-08, Dept. of Computer Science, Univ. of Regina, Saskatchewan, September 1996.

[25] S. Williams and C. Kindel. The Component Object Model: A technical overview. *Dr. Dobb's Journal*, December 1994.