Object-Oriented Constraint Satisfaction Problems[1]

Timothy Hinrichs
Nathaniel Love
Michael Genesereth
{thinrich, love, genesereth}@cs.stanford.edu

Abstract

The object-oriented paradigm can be leveraged to make the definition of constraint satisfaction problems more modular, which often results in reusable subproblems and fewer encoding errors. These object-oriented constraint satisfaction problems (OOCSPs) synthesize object-oriented constructs with declarative constraints to provide a rich, solver-independent language for describing both finite and infinite domains as sets of structured objects. This paper introduces the concept of an OOCSP, describes algorithms for solving certain classes of OOCSPs, and gives an undecidability result that shows not all OOCSPs can be solved.

Stanford Logic Group Technical Report LG-2006-03

Stanford Logic Group
Computer Science Department
Stanford University
353 Serra Mall
Stanford, CA 94305

[1]Our thanks to Akhil Sahai and Sharad Singhal for providing the inspiration for this work and for their contributions in many discussions. Lyle Ramshaw, Nathaniel Love, and Charles Petrie also played pivotal roles in the development of OOCSPs. Our thanks to Hewlett-Packard for its support of the work reported herein.

1 Introduction

The intuition behind a constraint satisfaction problem (CSP) is simple: given a set of constraints, find a solution that satisfies all those constraints. Problems in the real-world are often phrased in just that way, and people who know of the existence of general CSP solvers sometimes try to use that software to solve their problems, thereby avoiding problem-specific programming. Unfortunately, not all constraint satisfaction problems fit nicely under the classic definition of a CSP: a three-tuple $<V, D_V, C_V>$, where V is the set of variables, D_V assigns to each variable a domain of values, and C_V are the constraints on the variables. For example, some problems are naturally conceptualized as a hierarchy of objects, where a solution to the problem is a top-level object that satisfies a set of constraints. Each object contained within that top-level object also must satisfy certain constraints, and so on down the hierarchy.

Many people when trying to force a hierarchical problem to fit into the classical CSP framework end up flattening the hierarchy, and in doing so throw away the structure and therefore the information inherent in that hierarchy. This information loss is not only aesthetically unpleasing, but it can make the construction of the CSP error-prone and the answer discovered by a CSP solver hard for people to understand. Moreover, it robs the solver of extra information that might help it more efficiently find a solution to the problem.

The major obstacle that inhibits phrasing a hierarchical problem as a classical CSP is that the domain values in a CSP are required to be atomic entities, e.g. a, $car137$, 18. Without the ability to assign variables to complex entities, e.g. $f(a, b, g(c))$, the answer will never be a structured object, but rather can only be interpreted as a structured object by the user after seeing the solution.

In response to this limitation, Sabin and Freuder[SF96] developed the notion of a Composite CSP. The input is almost the same as a standard CSP. The difference is that a variable can either be assigned an atomic value as in a standard CSP, or it can be assigned a solution to a subproblem. A solution to a subproblem is a variable assignment for some subset of all the remaining variables in the original problem. Composite CSPs cleverly cast a hierarchical problem in what is basically the standard input language for a CSP; consequently, all the powerful results on solving CSPs can be brought to bear on Composite CSPs. The drawback is that expressing hierarchically structured problems as a composite CSP is not always the most natural thing to do for the people who want to solve those problems. People with a background in object-oriented programming, for example, often want to employ the same features they use when programming, e.g. inheritance, encapsulation, and polymorphism, which Composite CSPs do not nicely accommodate.

Our attempt at addressing these concerns abandons the classical three-tuple definition of a CSP. By developing a new input language that has an inherent hierarchical structure, more of the beneficial features of object-oriented programming can be brought to bear on developing, maintaining, and reusing CSP specifications. Moreover, this language naturally allows infinite domains to be represented finitely, in a solver-independent manner. In this paper we introduce the Object-Oriented Constraint Satisfaction Problem (OOCSP) [2], a nonstandard CSP that synthesizes several object-oriented programming features with the classical CSP formalism. OOCSPs are more expressive than CSPs, and in fact they turn out to be undecidable in general. We demonstrate undecidability and also show decidability results for two subclasses.

This paper contains the following contributions as well as a section on related work (Section 6).

- a formal definition for OOCSPs using logic (Section 2 and 3)
- decidability results (Section 4)

[2] The term Object-Oriented Constraint Satisfaction was originally used by Paltrinieri [Pal94], but his version was no more expressive than a CSP. In this paper OOCSPs are strictly more expressive, but the name fit so well we chose to redefine it.

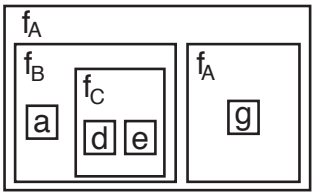

Figure 1: A simple object

- undecidability results (Section 5)

2 OOCSP Syntax: Object Grammars

Every object-oriented constraint satisfaction problem includes a set of objects and a set of object types. An object belongs to a particular type if it satisfies all the constraints associated with that type. In this paper, we represent objects symbolically and use logical sentences to represent constraints on object types.

The logical language used in what follows is a first-order language. Variables start with letters from the end of the alphabet, e.g. x, y, z. Relation constants, function constants, and object constants will be clear from context. We will often use constants of the form r_X for relation constants and constants of the form f_X for function constants.

In this paper, we assume objects can be assembled in different ways, and each assembler is labeled differently than every other assembler. For example, the object in Fig. 1 has been built using an assembler named f_A out of objects built using assemblers f_B and f_A.

Such objects can be succinctly represented as functional terms. For example, the object in Fig. 1 can be represented as the term

$$f_A(f_B(a, f_C(d, e)), f_A(g))$$

Sets of objects that share a common pattern at the root can be represented in logic by functional terms with variables. The set of all objects with three children where the root is f_A, the first child is c, and the second child has root f_B with two children of its own can be represented by the following *object template*.

$$f_A(c, f_B(x, y), z)$$

Each object can be assigned multiple types, and each type can include multiple objects. Types are represented with relation constants. The object $f_A(f_B(a, f_C(d, e)), f_A(g))$ can be assigned the type r_H by writing the statement

$$r_H(f_A(f_B(a, f_C(d, e)), f_A(g)))$$

Likewise, the object template $f_A(c, f_B(x, y), z)$ can be assigned the type r_H with the expression

$$r_H(f_A(c, f_B(x, y), z))$$

Statements like this one that assign object templates to types can be constrained so that they only apply to a subset of all the objects that match the object template. The only way to differentiate objects that match the object template is by differentiating its variables. We consider four types of constraints: type constraints, equality constraints, distinction constraints ($\neq$), and constructor constraints.

Type constraints require a variable to be instantiated with an object of a specific type. Equality constraints between two variables x and y require x and y to be instantiated with two copies of the same object, i.e. their functional representations must be syntactically identical. Distinction constraints require the objects to be syntactically different. The fourth constraint that we consider is the constructor constraint, which will be discussed shortly.

For example,

$$r_H(f_A(x, y, z)) \Leftarrow r_B(x) \wedge x = y \wedge r_C(f_C(x, a, y, x))$$

states that all objects matching $f_A(x, y, z)$ are of type r_H as long as (1) x is of type r_B, (2) y is the same tree as x, and (3) $f_C(x, a, y, x)$ is an object of type r_C.

The last constraint is a constructor constraint. It says that x and y are constrained so that $f_C(x, a, y, x)$ turns out to be an object of type r_C, where r_C is a type defined elsewhere. We call this constraint a *constructor constraint* because it is analogous to what a constructor does in common object-oriented programming languages. In languages like C++ and Java, new objects are built by calling the constructor for the object type and supplying the appropriate arguments. That is, a constructor returns an object whose attributes are the arguments it was given.

For example, in C++, the line

```
p = new Pair(a,b);
```

assigns p to an object of type Pair whose values are a and b. Pair(a,b) is the constructor in this example.

Viewing a constructor as a constraint simply means that the constraint is satisfied only when the arguments given to it can be used to build an object of that class. Constructor constraints are a great source of expressive power and will be used extensively in examples; their utility will be illuminated to a greater extent later.

Rules like the one above will be called *composition rules* because they can be seen as composing larger objects out of smaller ones. It will be useful later on to have a name for a composition rule with no constraints, i.e. of the form $r_X(f_X(t_1, \ldots, t_n))$; such rules will be called *simple typing rules*. Additionally, rules stating that a particular primitive object is of a particular type, e.g. $r_X(a)$, will be called *primitive typing rules*.

Composition rules are not the only kind of rule allowed. It is natural in object-oriented languages for some types to be defined entirely in terms of other types. Here we allow new types to be defined as the intersection of other types, e.g. $r_H(x) \Leftarrow r_G(x) \wedge r_J(x)$. Such rules will be called *intersection rules*. The special case when the right hand side contains a single conjunct, e.g. $r_A(x) \Leftarrow r_B(x)$, will be called a *subtying rule*. Subtyping is what is usually done in traditional object-oriented languages.

$$r_{Vehicle}(x) \Leftarrow r_{Auto}(x)$$
$$r_{Vehicle}(x) \Leftarrow r_{Boat}(x)$$
$$r_{Vehicle}(x) \Leftarrow r_{Plane}(x)$$
$$r_{Auto}(x) \Leftarrow r_{Car}(x)$$
$$r_{Auto}(x) \Leftarrow r_{Truck}(x)$$

These two kinds of rules and their special cases are sufficient to formally define the notion of an object grammar.

Definition 1 (Object Grammar). *A Object Grammar is a six-tuple* $<N, P, T, V, S, P>$.

N : *finite set of symbols called the nodes, e.g.* f_A
P : *nonempty, finite set of symbols, called the primitives, e.g.* a
T : *nonempty, finite set of symbols, called the types, e.g.* r_A
V : *finite set of symbols, called the variables, e.g.* x, y, z
S : *an element of T, the start type*

The sets N, P, T, and V are disjoint.
R : *finite set of rules of the following three forms.*

1. $r_A(f_B(t_1, \ldots, t_n)) \Leftarrow C_1 \wedge \ldots \wedge C_k$
   $r_A \in T$
   $f_B \in N$
   $t_i \in Terms[N, P, V]$
   $$C_i \in \begin{cases} t = u, \text{ where } t, u \in V \cup L \\ t \neq u, \text{ where } t, u \in V \cup L \\ r_B(t), \text{ where } r_B \in T \text{ and } t \in Terms[N, P, V] \end{cases}$$
2. $r_A(b)$
   $r_A \in T$
   $b \in P$
3. $r_A(x) \Leftarrow r_{B_1}(x) \wedge \ldots \wedge r_{B_n}(x)$
   $r_A, r_{B_i} \in T$
   $x \in V$

$Terms[N, P, V]$ *is the set of all functional terms (object templates) produced from the nodes, primitives, and variables, respectively. No arities are placed on the function constants, but every term is finite.*

Example 1. The following example is the one used in [SF96]; we use it here both to illustrate object grammars and for the purpose of comparison. The domain of interest is automobiles, specifically cars and the possible engines for each car class. The car classes of interest correspond to primitive objects: $\{compact, midsize, fullsize, coupe, sedan\}$. Each engine corresponds to a primitive object as well, $\{4D, 4TD, 2.0L4, 2.2L4, 2.5L6, 3.0L6, 3.2V6, 4.6L8, 5.7L8\}$, and are arranged in a hierarchy.

Gasoline and diesel are subtypes of engine.

$$r_{Engine}(x) \Leftarrow r_{Gasoline}(x)$$
$$r_{Engine}(x) \Leftarrow r_{Diesel}(x)$$

Four, six, and eight cylinder are subtypes of the gasoline engine.

$$r_{Gasoline}(x) \Leftarrow r_{4cyl}(x)$$
$$r_{Gasoline}(x) \Leftarrow r_{6cyl}(x)$$
$$r_{Gasoline}(x) \Leftarrow r_{8cyl}(x)$$

Each primitive engine object is either a diesel, 4-cylinder, 6-cylinder, or 8-cylinder engine.

$$r_{Diesel}(4D)$$
$$r_{Diesel}(4TD)$$
$$r_{4cyl}(2.0L4)$$
$$r_{4cyl}(2.2L4)$$
$$r_{6cyl}(2.5L6)$$
$$r_{6cyl}(3.0L6)$$
$$r_{6cyl}(3.2V6)$$
$$r_{8cyl}(4.6L8)$$
$$r_{8cyl}(5.7L8)$$

The constraints ensure that only certain engines can be placed in certain car classes.

$$r_{Car}(f_{Car}(compact, x) \Leftarrow r_{4Cyl}(x)$$
$$r_{Car}(f_{Car}(compact, 2.5L6)$$
$$r_{Car}(f_{Car}(compact, 4D)$$

A midsize car can take any 4-cylinder or any 6-cylinder engine as well as the 4D engine.

$$r_{Car}(f_{Car}(midsize, x)) \Leftarrow r_{4cyl}(x)$$
$$r_{Car}(f_{Car}(midsize, x)) \Leftarrow r_{6cyl}(x)$$
$$r_{Car}(f_{Car}(midsize, 4D))$$

A fullsize car can take any 6-cylinder engine, any diesel engine, the 2.2L4, or the 4.6L8.

$$r_{Car}(f_{Car}(fullsize, x)) \Leftarrow r_{6cyl}(x)$$
$$r_{Car}(f_{Car}(fullsize, x)) \Leftarrow r_{Diesel}(x)$$
$$r_{Car}(f_{Car}(fullsize, 2.2L4))$$
$$r_{Car}(f_{Car}(fullsize, 4.6L8))$$

A coupe can take either a 3.2V6 or a 4.6L8, and a sedan takes only the 5.7L8 engine.

$$r_{Car}(f_{Car}(coupe, 3.2V6))$$
$$r_{Car}(f_{Car}(coupe, 4.6L8))$$
$$r_{Car}(f_{Car}(coupe, 5.7L8))$$

If we make the start symbol r_{Car}, an OOCSP solver would try to find one of the possible car configurations, e.g. $f_{Car}(midsize, 2.0L4)$.

3 OOCSP Semantics

The Object Grammar semantics are exactly the same as the semantics of logic programming, where there happens to be no negation, i.e. minimal model semantics over Herbrand models. We give those semantics below, in terms of objects and object types.

An object is part of an Object Grammar whenever it corresponds to a functional term in that grammar.

Definition 2 ($Objects(G)$**).** *Let $G =<N, P, T, V, S, R>$ be an Object Grammar. The set of objects for G is defined as follows.*

- *Each member of P (a primitive object) is in $Objects(G)$.*
- *If $t_1, \ldots, t_n$ are in $Objects(G)$ and f_X is in N then $f_X(t1, \ldots, tn)$ is in $Objects(G)$.*
- *$Objects(G)$ is the minimum set satisfying the above.*

An object can only be of a certain type if there is a set of production rules that force it to be of that type.

Definition 3 ($Types$**).** *Let $G =<N, P, T, V, S, R>$ be an Object Grammar. The types of objects in G are defined inductively as follows.*

- *Each rule $r_A(t)$, where t is an object template, ensures type r_A includes all instances of t. If t is a primitive, it is of type r_A.*
- *If t is an object of types $r_{B_1}, \ldots, r_{B_n}$ the rule $r_A(x) \Leftarrow r_{B_1}(x) \wedge \cdots \wedge r_{B_n}(x)$ ensures type r_A includes t.*
- *Suppose v is a variable assignment for the variables in the object template e to objects. Consider the rule $r_A(e) \Leftarrow C_1 \wedge \ldots \wedge C_n$. Suppose v satisfies all the C_i constraints:*
  - *$(t = u)[v]$ is satisfied iff $t[v]$ is the same object as u[v].*
  - *$(t \neq u)[v]$ is satisfied iff $(t = u)[v]$ is not satisfied.*
  - *$r_H(t)[v]$ is satisfied iff $t[v]$ is an object of type r_H.*

  Then type r_A includes the object $e[v]$.
- *Types are defined by the minimum set satisfying the above.*

We can view an Object Grammar as a machine that accepts some set of objects – those objects of type S; consequently, it is natural to define the language of an Object Grammar as exactly that set of objects in type S.

Definition 4 ($Language(G)$). *Let $G =<N, P, T, V, S, R>$ be an Object Grammar. The language of G is the set of all objects of type S.*

Object grammars serve as the input to an OOCSP, just as the triple $<V, D_V, C_V>$ serves as the input to a CSP. Solving an OOCSP amounts to finding an object of a particular type. More precisely, the solution to an OOCSP for a particular object grammar is any tree whose type is the start symbol, i.e. any tree in the language of the grammar. This may not be the definition one might expect; an OOCSP solution could have been defined as an assignment of variables to objects of the appropriate types.

If what is desired is an assignment for the variables $\{v_1, \ldots, v_n\}$ with types $\{t_1, \ldots, t_n\}$, one can construct a new type r_S and use the following rule to define it.

$$r_S(f_S(v_1, \ldots, v_n)) \Leftarrow t_1(v_1) \wedge \cdots \wedge t_n(v_n)$$

Then, add this to the object grammar and make the start type r_S. Thus the definition for an OOCSP solution given here encompasses the standard definition in a natural way.

Definition 5 ($OOCSP$). *An object-oriented constraint satisfaction problem consists of an object grammar G. The set of solutions is $Language(G)$.*

Example 2. Suppose in the context of utility computing, we were interested in modeling computing resources for the purpose of constructing e-Commerce sites. Computers, C, are built out of an architecture, A, some amount of memory, M, and a hard drive H. We can model computers with the following rules.

$$r_C(f_C(x, y, z)) \Leftarrow r_A(x) \wedge r_M(y) \wedge r_H(z)$$
$$r_A(x86), r_A(sparc)$$
$$r_M(500MB), r_M(1GB)$$
$$r_H(40GB), r_H(80GB), r_H(160GB)$$

Servers, S, are built out of a computer, an operating system, O, and an IP address, I. The WinNT operating system requires an x86 architecture, and Solaris requires SPARC.

$$r_S(f_S(f_C(x86, x, y), winnt, z)) \Leftarrow$$
$$r_M(x) \wedge r_H(y) \wedge r_I(z)$$
$$r_S(f_S(f_C(sparc, x, y), solaris, z)) \Leftarrow$$
$$r_M(x) \wedge r_H(y) \wedge r_I(z)$$

For simplicity, we will use the natural numbers to represent IP addresses. The natural numbers, N, are represented as the height of a tree, i.e. encoded using the successor function.

$$r_N(0)$$
$$r_N(f_N(x)) \Leftarrow r_N(x)$$

Then the IP addresses can be defined as the set of natural numbers: $r_I(x) \Leftarrow r_N(x)$.

An e-Commerce site is built out of a network connection and a list of servers, L, each of which has a unique IP address. The empty list of servers is represented by nil, and the object $f_L(x, y)$ is a list of servers if x is a server and y is a list of servers.

$$r_L(nil)$$
$$r_L(f_L(x, y)) \Leftarrow r_S(x) \wedge r_L(y)$$

Requiring that the list of servers all have unique IP addresses can be accomplished in a similar way. Every server in a list with zero or one elements has a unique IP address. If the first server in a list has a different IP address than every other server in the list, and the rest of the list has unique IP addresses, then so does the entire list.

$$r_D(nil)$$
$$r_D(f_L(x, nil))$$
$$r_D(f_L(f_S(x, y, z), f_L(f_S(u, v, w), t))) \Leftarrow$$
$$z \neq w \wedge r_D(f_L(f_S(x, y, z), t))$$
$$\wedge\, r_D(f_L(f_S(u, v, w), t))$$

The definition for an e-Commerce site is then a network connection, W, and a list of servers that is also of type r_D.

$$r_E(f_E(x, y)) \Leftarrow r_W(x) \wedge r_L(y) \wedge r_D(y) \,\square$$

4 Decidable OOCSPs

In the next two sections we will be investigating various syntactic classes of object grammars. Those classes are defined by the kinds of constraints allowed in the composition rules. All the classes we consider include basic type constraints, e.g. $r_A(f_A(x)) \Leftarrow r_B(x)$. We will use the abbreviation OG to represent the class of tree grammars that includes composition rules with only type constraints and intersection and primitive typing rules. If OG is subscripted with $=$, the composition rules are allowed to include equality constraints as well. Likewise distinction and constructor constraints will be indicated by subscripting OG with $\neq$ and c, respectively.

This section gives algorithms for solving OOCSPs for inputs $OG_=$ and $OG_{=,\neq}$ under some restrictions. The first restriction requires there be only one assembler used for all the objects in a particular type. That is, every composition rule is of the following form.

$$r_A(f_A(\ldots)) \Leftarrow \ldots$$

We will call the class of composition rules that include this connection between types and assemblers the *one-to-one composition rules*. All the results in this section will concern object grammars where every composition rule is one-to-one.

The one-to-one composition rules in $OG_=$ grammars are thus of the following form.

$$r_A(f_A(t_1, \ldots, t_n)) \Leftarrow r_{B_1}(e_1) \wedge \ldots \wedge r_{B_n}(e_m) \\ \wedge u_1 = v_1 \wedge \ldots \wedge u_k = v_k$$

Each e_i is either a variable or a primitive object, each t_i is an object template, and each u_i, v_i is either a variable or a primitive object.

Given an object grammar where all the composition rules are one-to-one $OG_=$ rules, we want to be able to determine whether there is an object of a particular type. The algorithm we use is a variant of the marking algorithm for determining whether a context-free-grammar is empty. (The connection between OOCSPs and context-free grammars is addressed in Section 6.)

Recall there are two types of rules: the composition rules and the intersection rules. The first step in this algorithm is to eliminate the equality constraints from the composition rules, which can be accomplished by repeatedly applying the usual algorithm for equality elimination. Replace variables by what they are equal to. Then remove tautologous equality constraints and remove rules containing unsatisfiable equality constraints.

For example, if at some point the constraint $a = b$ appears where a and b are primitives, remove the rule containing the constraint since it will never be satisfied. If the constraint $a = a$ appears, remove just the constraint since it is always satisfied. At the end of this process the composition rules remaining have no equality constraints.

The second step is to remove the type constraints on primitive objects within composition rules. Consider any of the type constraints $r_{B_i}(e_i)$. If e_i is a primitive, we can check whether r_{B_i} is satisfied by first computing the types as determined by the intersection rules and primitive typing rules in a bottom-up fashion [Ull89], taking advantage of the function-free nature of those rules. Store the results in primitive typing rules, e.g. if we find a is of type r_C then add the rule $r_C(a)$. Then for each $r_{B_i}(e_i)$ in a composition rule, remove it if it is satisfied; otherwise, remove the entire rule.

After removing the type constraints on primitives, the only remaining constraints in the composition rules are type constraints on variables; the next step is to ensure that no variable appears in more than one typing constraint. Multiple type constraints on the same variable constraint that variable to take on an object in the intersection of the two types. We can remove such constraints within composition rules and handle them at the same time we handle the other intersection rules. For any set of type constraints $r_{B_1}(x), \ldots, r_{B_n}(x)$ on the variable x, introduce a new type r_B and replace $r_{B_1}(x), \ldots, r_{B_n}(x)$ with $r_B(x)$ in the composition rule. Then include the intersection rule $r_B(x) \Leftarrow r_{B_1}(x) \wedge \ldots \wedge r_{B_n}(x)$.

For example, the composition rule

$$r_A(f_A(x, y)) \Leftarrow r_B(x) \wedge r_C(y) \wedge r_D(y) \wedge r_E(y)$$

can be replaced by

$$r_A(f_A(x, y)) \Leftarrow r_B(x) \wedge r_F(y) \\ r_F(y) \Leftarrow r_C(y) \wedge r_D(y) \wedge r_E(y)$$

The last preprocessing step removes the intersection rules. To do this, we will rely on the fact that the composition rules are one-to-one composition rules. Besides primitive objects, every object of type r_A produced by such one-to-one composition rule is of the form $f_A(\ldots)$. With this in mind, it is easy to see that the intersection of two distinct types is empty (except for primitives) unless there is a subtyping rule relating the two types.

Continuing the example, the intersection rule

$$r_F(x) \Leftarrow r_C(x) \wedge r_D(x) \wedge r_E(x)$$

can only add complex trees to the r_F type if there are subtyping rules that give r_C, r_D, and r_E common subtypes. The following rules ensure they have the common subtype r_H.

$$\begin{aligned} r_C(x) &\Leftarrow r_H(x) \\ r_D(x) &\Leftarrow r_H(x) \\ r_E(x) &\Leftarrow r_I(x) \\ r_I(x) &\Leftarrow r_H(x) \end{aligned}$$

Suppose that for the intersection rule $r_A(x) \Leftarrow r_{B_1}(x) \wedge \ldots \wedge r_{B_n}(x)$, the types $r_{B_1}, \ldots, r_{B_n}$ share the subtypes $r_{C_1}, \ldots, r_{C_k}$. Then the intersection rule can be replaced by a set of subtyping rules:

$$\begin{aligned} r_A(x) &\Leftarrow r_{C_1}(x) \\ &\vdots \\ r_A(x) &\Leftarrow r_{C_k}(x) \end{aligned}$$

Note that because we have already computed all the types for all the primitive objects and stored that information as primitive typing rules, no types are lost by removing this intersection rule. Thus the intersection rules can be replaced by a set of subtyping rules.

Those subtyping rules can be removed by turning our one-to-one composition rules into non-one-to-one composition rules. First compute the transitive closure on the subtyping rules, e.g. the rules $r_A(x) \Leftarrow r_B(x)$ and $r_B(x) \Leftarrow r_C(x)$ yield the rule $r_A(x) \Leftarrow r_C(x)$.

Next replace each subtyping rule $r_A(x) \Leftarrow r_B(x)$ with a new set of composition rules for r_A. For each composition rule $r_B(f_B(\ldots)) \Leftarrow \ldots$, insert the rule $r_A(f_B(\ldots)) \Leftarrow \ldots$. This transformation breaks the one-to-one-ness of the composition rules, but that property was only necessary for handling intersection.

To summarize, given a $OG_=$ grammar with one-to-one composition rules, one can simplify it by (1) removing equality constraints, (2) removing the type constraints on primitives, (3) removing intersection within composition rules, and (4) removing the intersection rules. This results in a set of rules all of the following form.

$$r_A(f_C(t_1, \ldots, t_n)) \Leftarrow r_{B_1}(x_1) \wedge \ldots \wedge r_{B_m}(x_m)$$

Constructing a tree out such rules can be accomplished using a variant of the well-known marking algorithm for determining emptiness in a CFG [Sip96].

Algorithm 1 (Marking) Input: $<N, P, T, V, S, R>$.

1. Remove equality, leaf type constraints, and intersection.
2. Number the rules and run the basic marking algorithm.
   (a) For each rule for type r_A numbered k without any unmarked constraints in the body, use k to mark all the unmarked r_A type-constraints in the rule bodies.
   (b) If no rule for S has been marked and progress has been made, goto step (a).
   (c) If no rule for S has been marked, return NIL.

(d) Choose a marked rule for S and apply it to S. For each type constraint in the body, apply the production rule corresponding to the number that constraint is marked with. Repeat until no type constraints remain. If the resulting object template includes no variables, return it. Otherwise, fill in every variable with any one of the primitives. Return the resulting object.

Example 3. As an example, consider the following set of production rules, with the start symbol is r_S.

$$\begin{array}{ll} 1. & r_S(f_S(x,y)) \Leftarrow r_A(x) \wedge r_B(x) \\ 2. & r_A(a) \\ 3. & r_B(f_B(x)) \Leftarrow r_C(x) \\ 4. & r_C(f_C(d)) \end{array}$$

Begin by marking rule (2) and (4) as they have no unmarked constraints. Also mark the type constraint r_A in rule (1) with a 2 and r_C in rule (3) with a 4. No r_S rule has been marked, so continue. Since every type constraint in rule (3) has been marked, mark rule (3) itself and then mark the r_B constraint in rule (1) with a 3. Again no rule for r_S has been marked, so repeat. This time, every constraint in rule (1) has been marked, which means the rule itself must be marked, after which the loop exits. This results in the following marked production rules.

$$\begin{array}{ll} 1.^M & r_S(f_S(x,y)) \Leftarrow r_A^2(x) \wedge r_B^3(y) \\ 2.^M & r_A(a) \\ 3.^M & r_B(f_B(x)) \Leftarrow r_C^4(x) \\ 4.^M & r_C(f_C(d)) \end{array}$$

Because a rule for r_S is marked, construct an object by first applying rule (1). $r_S(f_S(x,y)) \Leftarrow_1 r_A(x) \wedge r_B(x)$. Then apply rule (2) to the r_A constraint, followed by rule (3) to the r_B constraint, and finally rule (4) to the r_C constraint that resulted from applying rule (3). This yields the object $f_S(f_A(a), f_B(f_C(d)))$. □

Theorem 1 (Soundness, Completeness, and Termination of Marking). *Marking takes as input an object grammar $G =<N, P, T, V, S, R>$ in $OG_=$ with one-to-one composition rules. It outputs an object in G of type S if and only if one exists. It returns NIL if no such tree exists.*

The next algorithm we give handles object grammars with both equality and distinction ($\neq$) constraints. As we shall see, distinction constraints require a fundamental change to the algorithm. Step (2) of Marking can be replaced by a restricted form of Ullman's bottom up evaluation [Ull89], where only a certain number of objects of each type are computed. Once this bound is reached, all the rules that produce an object of that type are discarded. Eventually no new objects can be produced or an object of the requested type has been found. For completeness, we will require every composition rule to be *fully safe*, i.e. every variable in the head must occur in the body, and every variable in the body must occur in the head.

Algorithm 2 (Bottom-up) Input: $<N, P, T, V, S, R>$

1. Remove equality, primitive type constraints, and intersection rules.
2. Run altered bottom-up evaluation.
   (a) k = max number of $\neq$ constraints in any rule.
   (b) Build a bin b_A for each type r_A.

(c) For each rule without a body $r_A(t)$, place t in b_A.

(d) If $|b_A| \geq k$ remove the rules for type r_A.

(e) Use bottom-up evaluation to produce new trees.

(f) If no new trees were produced, return NIL. If there is some tree in the bin for S, return it. Otherwise repeat the last two steps.

(g) Return NIL.

Theorem 2 (Soundness, Completeness, and Termination of Bottom-up). *Bottom-up takes as input an object grammar $G =<N, P, T, V, S, R>$ in $OG_{=,\neq}$ with one-to-one composition rules. If at step (2), the composition rules are fully safe, it outputs an object in G of type S if and only if one exists. It returns NIL if no such tree exists.*

5 Undecidable OOCSPs

In this section, we explain how to encode Diophantine equations within OG_c, making OOCSPs for OG_c formally undecidable.

Let $P(x_1, \ldots, x_n)$ be an arbitrary polynomial with integral coefficients and positive, integral exponents over the variables $x_1, \ldots, x_n$. It is well known that finding a solution to $P(x_1, \ldots, x_n) = 0$ where each x_i is an integer is undecidable. We will show how to encode a polynomial over the natural numbers in OG_c; encoding polynomials over the integers is straightforward but tedious. There can be no algorithm solving all OOCSPs since if there were, this transformation could be applied, resulting in a decision procedure for Diophantine problems.

The natural numbers are represented using the analog of the successor function, f_N. 0 is zero, $f_N(0)$ corresponds to one, $f_N(f_N(0))$ corresponds to two, and so on. Addition (r_A) and multiplication (r_M) operate on these f_N trees. Below, we encode addition and multiplication using the usual identities: $x + y = z$ implies $(x + 1) + y = (z + 1)$ and $x * y = z$ implies $(x + 1) * y = z + y$.

$$
\begin{aligned}
&r_A(f_A(0, y, y)) \Leftarrow r_N(y)\\
&r_A(f_A(f_N(x), y, f_N(z))) \Leftarrow r_N(x) \wedge r_N(y) \wedge r_N(z)\\
&\qquad r_A(f_A(x, y, z))
\end{aligned}
$$

$$
\begin{aligned}
&r_M(f_M(0, y, 0)) \Leftarrow r_N(y)\\
&r_M(f_M(f_N(x), y, z)) \Leftarrow r_N(x) \wedge r_N(y) \wedge r_N(z)\\
&\qquad r_M(f_M(x, y, w)) \wedge r_A(f_A(y, w, z))
\end{aligned}
$$

In exponentiation (r_E), the first argument is the exponent, the second the base, and the third the base raised to the exponent. The recursive portion of the definition is structurally identical to that of multiplication.

$$
\begin{aligned}
&r_E(f_E(0, y, f_N(0)))\\
&r_E(f_E(f_N(x), y, z)) \Leftarrow r_N(x) \wedge r_N(y) \wedge r_N(z)\\
&\qquad r_E(f_E(x, y, w)) \wedge r_M(f_M(y, w, z))
\end{aligned}
$$

Using this machinery, every polynomial can be encoded as an object grammar.

Example 4. Consider the monomial x^3y^2. We will build a new type, r_P, for objects with three fields so that if x is the first field and y is the second field, the third field is x^3y^2.

$$r_P(f_P(x,y,t)) \Leftarrow r_N(x) \wedge r_N(y) \wedge r_N(t) \wedge \\ r_E(f_E(f_N(f_N(f_N(0))),x,z) \wedge \\ r_E(f_E(f_N(f_N(0)),y,w) \wedge \\ r_M(f_M(z,w,t)$$

If this were the polynomial of interest, we could construct an OOCSP to solve the equation $x^3y^2 = 0$ by introducing a new type r_S, and making it the start symbol.

$$r_S(f_S(x,y)) \Leftarrow r_N(x) \wedge r_N(y) \wedge r_P(f_P(x,y,0)) \; \square$$

The summation of monomials can be encoded in a similar fashion, as can setting that sum to zero. The full encoding requires nothing outside TG_c. In fact, it appears that limiting every composition rule to two constructors is sufficient for undecidability.

Example 5. Consider the production rule for r_P given above. Notice it includes three constructor constraints. By inventing a new type r_{P_1} and introducing new composition rules, we can express r_P using only two constructors per rule.

$$r_M(f_M(x,y,t)) \Leftarrow r_N(x) \wedge r_N(y) \wedge r_N(z) \wedge \\ r_E(f_E(f_N(f_N(f_N(0))),x,z) \wedge \\ r_{P_1}(f_{P_1}(z,y,t))$$

r_{P_1} simply defines the result of multiplying its first argument, z, with y^2.

$$r_{P_1}(f_{P_1}(z,y,t)) \Leftarrow r_N(z) \wedge r_N(y) \wedge r_N(t) \wedge \\ r_E(f_E(f_N(f_N(0)),y,w) \wedge \\ r_M(f_M(z,w,t)$$

This transformation appears to turn a rule with an arbitrary set of constructors with an arbitrary number of new variables into a set of rules each of which has at most two constructors with no more than one new variable.

Theorem 3 (Undecidability of OG_c). *The class of OOCSPs that take as input OG_c is undecidable.*

A stronger claim appears to hold as well: the class of OOCSPs where every composition rule includes no more than two constructors is undecidable.

6 Related Work

The undecidability result in the last section relies on recursion. OOCSPs without recursion are no more expressive than standard CSPs because they can always be flattened and then written as a CSP. Additionally, every CSP can be expressed as a non-recursive OOCSP, as evidenced below.

Consider the standard input to a CSP: a set of variables, their domains, and constraints among the variables: $<V, D_V, C_V>$. Suppose the variables V consist of $\{v_1, \ldots, v_n\}$. The corresponding non-recursive OOCSP includes composition rules that state the contents of each domain. For example, assume D_{v_1} is defined as $\{e_1, \ldots, e_m\}$. Then we include the following m rules.

$$r_{D_{v_1}}(e_1)$$
$$\vdots$$
$$r_{D_{v_1}}(e_m)$$

Suppose there are c constraints, and each constraint C_{V_i} is represented as a table of allowable combinations of values for the variables in the tuple of variables V_i. Each such table can be represented as its own object class. For example, assume C_{V_1} includes m tuples of values, and each tuple constrains k variables.

$$\begin{aligned} & r_{C_{V_1}}(f_{C_{V_1}}(e_{11}, e_{12}, \ldots, e_{1k})) \\ & \vdots \\ & r_{C_{V_1}}(f_{C_{V_1}}(e_{m1}, e_{m2}, \ldots, e_{mk})) \end{aligned}$$

We can then define the class of objects where every object in the class is a solution to the CSP. In the head are all those variables in V. The constraints on the body ensure each variable is instantiated with a value from the appropriate domain and that all those instantiations satisfy the required constraints C_V.

$$r_S(f_S(v_1, \ldots, v_n)) \Leftarrow r_{D_{v_1}}(v_1) \wedge \cdots \wedge r_{D_{v_n}}(v_n) \wedge r_{C_{V_1}}(f_{C_{V_1}}(V_1)) \wedge \cdots \wedge r_{C_{V_c}}(f_{C_{V_c}}(V_c))$$

Suppose the object found is $f_S(t_1, \ldots, t_n)$. The CSP variable assignment corresponding to this OOCSP object is then $\{v_1/t_1, \ldots, v_n/t_n\}$.

Certain kinds of CSPs with infinite domains can be represented as OOCSPs. The encoding for the natural numbers, for instance, was illustrated in Section 5. But OOCSPs are strictly less expressive than the class of CSPs with infinite domains. For example, a CSP over the real numbers cannot be expressed as an OOCSP. But for the class of infinite domains that can be encoded as OOCSPs, the need for hard-coded knowledge of those domains within solvers is obviated, giving a broader range of users the ability to change those domains.

Many components of the OOCSP formalism appears in different contexts in the literature. Because this paper focuses on constraint satisfaction, it is that literature we review. Object-oriented constraint satisfaction was a term first coined by Paltrinieri[Pal94, Pal95]. His version is no more expressive than a CSP with finite domains, but because our version is in general undecidable, the OOCSPs in this paper are clearly more expressive. Dynamic Constraint Satisfaction Problems [MF90] allow the values of special variables to affect the number of regular variables in a solution. But those special variables must be recognized as such by the DCSP solver. In OOCSPs, the number of variables is never given explicitly; if some class of objects does not satisfy a particular constraint but another class does, the second class will be used, which may have a different number of attributes, i.e. variables. Free logic has also been used to allow variance in the number of variables in a solution[BD91], but there the domains are treated as atomic values–just the limitation OOCSPs were invented to overcome. Hierarchical constraint satisfaction in another seemingly related topic that allows one to define a hierarchy of constraints. Those with the highest precedence must be satisfied; the others represent a ranked set of preferences [BFBW92]. Such a scheme is orthogonal to OOCSPs, since every constraint must be satisfied in an OOCSP. Hierarchical domain CSPs arrange domain values into a hierarchy of values, but those values are atomic leaving the size of expressible domains finite. [HM83].

In 2004, Bodirsky did his dissertation on constraint satisfaction in infinite domains and considered the case of constructing a tree out of a set of constraints in a particular tree description language. [Bod04] That tree description language constrains trees by forcing the existence of nodes that satisfy a set of constraints. It allows three possible constraints between two nodes x and y: x must be an ancestor of y, x and y must be equal, and x must be included in a subtree which is ordered to the left of the subtree that includes y. Tree descriptions in OOCSPs have more structure, allowing and requiring constraints to be placed on particular subtrees. Bodirsky's constraints require the existence of nodes with a particular relationship. OOCSPs require particular nodes to have a particular relationship.

Representing domains as structured objects is not a new idea within the CSP community and has appeared in Meta CSPs [Fre92] and Composite CSPs[SF96]. Meta CSPs decompose a CSP into subproblems, where each metavariable

can be assigned any solution to the given subproblem. Those solutions can be viewed as structured objects. OOCSPs build in this notion of assigning variables to solutions of subproblems and in fact reify subproblems (as types), allowing a subproblem to be defined in terms of the subproblem itself (recursive type definitions). In a Composite CSP, regular variables can be assigned solutions to subproblems of the original CSP. These solutions take the form of a set of variable assignments, where each variable is assigned either an atomic value or a solution to part of the remaining problem. However, composite CSPs cannot be defined recursively, thus ensuring the maximum depth of object embedding (or when viewed as a tree, the maximum height of the tree) is bounded by the number of variables in the problem. Without built-in infinite domains, e.g. the reals or integers, a composite CSP allows only definitions of finite domains. Employing these built-in domains allows one to construct infinite domains of structured objects, but those objects are still bounded by a fixed height. For instance, the set of all binary trees cannot be defined using composite CSPs since there is no upper bound on the height of the objects in that set.

Finally, OOCSPs have a strong relationship to context-free grammars (CFGs). Recall that a CFG consists of a set of production rules and a start symbol, e.g.

$$S \rightarrow AbDD$$
$$A \rightarrow a$$
$$D \rightarrow D$$
$$D \rightarrow d$$
$$D \rightarrow e$$

Capital letters denote nonterminal symbols and lowercase letters denote terminals. Only a nonterminal can occur on the left hand side of a rule. The nonterminals denote internal nodes in a tree, and terminals denote leaves. S is historically used as the start symbol for a grammar, meaning that the trees of interest are rooted at S.

Trees in a CFG correspond to objects in an OOCSP. Nonterminals in a CFG correspond to object types. Terminals in a CFG correspond to primitive objects. Each production rule in a CFG corresponds to a composition rule of a particular type

$$r_A(f_A(t_1, \ldots, t_n)) \Leftarrow r_{B_1}(x_1) \wedge \cdots \wedge r_{B_n}(x_n)$$

where each t_i is either a primitive object or a variable, and each variable t_i is distinct from every other variable t_i. For example, the rule $S \rightarrow AbDD$ says that any object whose first attribute is of type A, second attribute is the value b, and third and fourth attributes are of type D is an object of type S.

Notice that in the first rule, $S \rightarrow AbDD$, any tree rooted at D can be used for either of the attributes of type D on the right-hand-side. A CFG does not allow constraints to be written which would require the two Ds to be the same, hence its context-freeness. OOCSPs on the other hand do allow constraints to be added to production rules.

Additionally, while a CFG does not allow a type hierarchy to be defined without duplicating production rules, OOCSPs allow types to be defined as the intersection of other types.

Formally, the definition for a CFG is very related to the definition for an OOCSP. Recall that a context-free grammar is a four-tuple $<N, T, P, S>$.

N : finite set of symbols, called the nonterminals
T : finite set of symbols, disjoint from N, called the terminals
S : an element of N, the start symbol
P : finite set of rules of the form
$A \rightarrow \gamma_1 \ldots \gamma_m$, where $\gamma_i \in N \cup T$ and $A \in N$

To summarize, one can think of an OOCSP as a CFG with constraints on its production rules. A CFG with constraints on its production rules is reminiscent of a context sensitive grammar. Comparing OOCSPs to context sensitive grammars is the subject of future work.

The fact that every object in an OOCSP corresponds to a tree in a CFG is noteworthy. It means that objects are acyclic, e.g. there is no direct representation of a circular list in OOCSPs (though by introducing a new object that represents memory, we could encode circular lists in the same way we represent circular lists in a computer's memory: using pointers).

Besides relationships to CSPs and CFGs, other related topics include OO programming languages that provide native constructs for solving CSPs [Cas94, RP97] and natural language generation, which uses CFGs heavily and often employ constraints. OOCSPs (in a different formal incarnation) can be applied to configuration management, as illustrated in [HLP$^+$04].

7 Conclusion

Many real-world problems are modeled naturally with a hierarchy of objects and a hierarchy of object classes. The OOCSP formalism attempts to marry components of the object-oriented paradigm with declarative constraints.

The OOCSP formalism differs from OO programming languages in that inter-attribute relationships are expressed declaratively. It differs from the CSP formalism by shifting focus from variable assignments to objects and object types. That shift in focus is not simply a reconceptualization of the problem; it adds expressive power.

One of the benefits of OOCSPs is the ability to finitely encode infinite sets of objects in a natural way. Infinite CSPs can be used to encode a much larger array of problems than CSPs restricted to finite domains, which is both a strength and a weakness. The more interesting the class of problems, the harder it is to solve them.

OOCSPs include three types of rules: composition rules, primitive typing rules, and intersection rules. When the composition rules are one-to-one and are limited to type and equality constraints, the OOCSP can be solved by a marking algorithm similar to that for determining whether a CFG is empty. Distinction ($\neq$) constraints require an algorithm that actually builds a small set of trees for each type. This bottom-up evaluation is a generalization of marking; hence, it can be used for solving OOCSPs that include type constraints, equality constraints, and distinction constraints.

OOCSPs that allow two or more constructors per composition rule are sufficiently expressive to encode Diophantine equations, thus making the class formally undecidable. One topic for future work is to determine whether restricting composition rules to one constructor results in decidability. On this topic, two items are noteworthy. First, addition can be expressed using one constructor constraint, but multiplication appears to require two. Second, Presburger arithmetic, which includes addition but not multiplication, is decidable, but Peano arithmetic, which includes both, is not. This hints that limiting rules to one constructor may result in a decidable fragment of OOCSPs.

8 Appendix: Planning Example

Example 6. Suppose we wanted to use OOCSPs to do planning in the context of the blocks world. Here we show how to encode that domain without knowing the blocks that exist. This formulation of blocks world has three fluents: $on(x, y)$, $clear(x)$, and $table(x)$, where the variables represent blocks.

$$r_{Fluent}(f_F(on(x,y))) \Leftarrow r_{Block}(x) \wedge r_{Block}(y)$$
$$r_{Fluent}(f_F(clear(x))) \Leftarrow r_{Block}(x)$$
$$r_{Fluent}(f_F(table(x))) \Leftarrow r_{Block}(x)$$

There are two actions, which are inverses: $st(x,y)$ and $unst(x,y)$. $st(x,y)$ moves block x which must be clear onto block y, which must also be clear. $unst(x,y)$ moves block x, which must be clear, onto the table.

$$r_{Action}(f_A(st(x,y))) \Leftarrow r_{Block}(x) \wedge r_{Block}(y)$$
$$r_{Action}(f_A(unst(x,y))) \Leftarrow r_{Block}(x) \wedge r_{Block}(y)$$

A state in a world with an unknown number of blocks must accommodate a fluent list of arbitrary length. A fluent list is either nil or a fluent and a fluent list.

$$r_{FluentList}(nil)$$
$$r_{FluentList}(f_L(x,y)) \Leftarrow r_{Fluent}(x) \wedge r_{FluentList}(y)$$

A list of actions is identical to a list of fluents, but with $Fluent$ replaced by $Action$. Each action has associated with it a list of preconditions and effects, which are also lists of fluents. The $st(x,y)$ action is shown below. $unst(x,y)$ is similar.

$$r_{PreEff}(r_E(f_A(st(x,y)), [clear(x), clear(y)], [on(x,y)]))$$

The left child of $PreEff$ is the action. The middle child is the list of preconditions, and the right child is the list of effects. For legibility, lists have been written as Prolog sequences.

Checking whether a particular action sequence changes the initial state into the final state is built upon computing the result of executing a single action in a state. The left child of r_{Exec} is the initial state; the middle child is the action to be executed, and the right child is the resulting state. If the preconditions of the action are not met in the initial state, there is no r_{Exec} object with that initial state and action.

$$r_{Exec}(f_X(x,y,u)) \Leftarrow r_{PreEff}(r_E(y,yp,ye)) \wedge r_{Subset}(f_B(yp,x)) \wedge r_{SetDifference}(f_D(u,x,ye))$$

The class $r_{Subset}(f_B(x,y))$ represents the relationship $x \subset y$. The class $r_{SetDifference}(f_D(x,u,y))$ represents the relationship $x - u = y$.

Executing a sequence of actions amounts to walking over that sequence and computing the state update after each action. r_{Result}'s left child is the initial state; its middle child is the action sequence, and its right child is the resulting state, once again using Prolog sequence notation in place of OOCSP lists.

$$r_{Plan}(f_P(x, nil, x))$$
$$r_{Plan}(f_P(x, [y|z], w)) \Leftarrow r_{Exec}(f_X(x,y,u)) \wedge r_{Plan}(f_P(u,z,w))$$

Now, posing a planning problem requires giving the set of blocks that exist, e.g.

$$r_{Block}(f_B(a))$$
$$r_{Block}(f_B(b))$$

and defining the class that requests a plan to achieve the final state from the initial state. In this example, the two blocks, a and b start out on the table. In the final state, a is on top of b.

$$r_S(f_S(x)) \Leftarrow r_{PlanResult}(f_R([clear(a), clear(b), table(a), table(b)]), x, y) \wedge r_{Subset}(f_B([on(a,b)], y))$$

Here we give two objects that will achieve the given final state, and when the start symbol is r_S, these objects are therefore in the language of the grammar.

$$f_L(st(a,b), nil)$$
$$f_L(st(a,b), f_L(unst(a,b), f_L(st(a,b), nil)))$$

References

[BD91] J. Bowen and D.Bahler. Conditional existence of variables in generalized constraint networks. *AAAI*, 1991.

[BFBW92] A. Borning, B. N. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5:223–270, 1992.

[Bod04] M. Bodirsky. *Constraint Satisfaction with Infinite Domains*. PhD thesis, Humboldt University in Berlin, 2004.

[Cas94] Yves Caseau. Constraint satisfaction with an object-oriented knowledge representation language. *Applied Intelligence*, 4(2):157–184, 1994.

[Fre92] E. Freuder. Constraint solving techniques. *Constraint Programming*, 131:51–74, 1992.

[HLP+04] T. Hinrichs, N. Love, C. Petrie, L. Ramshaw, A. Sahai, and S. Singhal. Using object-oriented constraint satisfaction for automated configuration generation. *DSOM*, 2004.

[HM83] W. Havens and A. Mackworth. Representing knowledge of the visual world. *IEEE Computer*, 16(10):90–96, 1983.

[MF90] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. *AAAI 90*, pages 25–32, 1990.

[Pal94] Massimo Paltrinieri. Some remarks on the design of constraint satisfaction problems. *Workshop on Principles and Practice of Constraint Programming*, pages 299–311, 1994.

[Pal95] Massimo Paltrinieri. A visual environment for constraint programming. *IEEE Symposium on Visual Languages*, 1995.

[RP97] Pierre Roy and Francois Pachet. Reifying constraint satisfaction in smalltalk. *Journal of Object-Oriented Programming*, 10(4):43–51, 1997.

[SF96] D. Sabin and E. Freuder. Configuration as composite constraint satisfaction. *AAAI Configuration Workshop*, pages 28–36, 1996.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. Brooks Cole, 1996.

[Ull89] Jeffrey Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989.