Using Constraint Propagation and Domain Reduction for the Generation Phase in Declarative Modeling

O. Le Roux, V. Gaildrat and R. Caubet
IRIT, UPS, 118 route de Narbonne, 31062 Toulouse Cedex 4, France
{leroux, gaildrat, caubet}@irit.fr

Abstract

This paper presents an oriented object constraint solver based on constraint propagation and domain reduction for the generation phase in declarative modeling. The solver supports generic constraints and heterogeneous parameters via generic domains. This ensures adaptability and efficiency of the resolution process in complex cases. As an application, a declarative system for 3D-environments planning is presented.

1. Introduction

Declarative modeling is a recent paradigm (80's) in the world of computer aided design systems. It allows designers to build scenes or shapes giving only a set of properties and constraints to be satisfied, freeing users to give all the numeric details. In opposite to classical geometric modeling, it does not require a complete knowledge of objects to build at start time. Moreover, it can support non-geometric information and maintains logical links between entities or logical structure of the scene. It is a design process closer to the user's needs than classical modeling.

A declarative modeler is the combination of three parts [4]:

a. The first one is called *description module*. It defines the application language and offers the user an interface to describe the properties of scenes or shapes. It also translates high-level descriptions in an internal language easier to manipulate and process.

b. The second module can be considered as the system's kernel. It is the *generation module*. Its role is to explore search spaces to produce models that meet the description made by the user even if it is incomplete. One of the key ideas in declarative modeling is the ability of the system to find several or all solutions.

c. The third part of a declarative modeling system is the *insight module*. Many solutions can be generated according to the designer's specifications. This module permits presentation, navigation and refinement of valid models to help the user choose a solution.

In the following we only focus on the generation phase.

This paper is organized as follows. Part 2 discusses about the generation phase and related works. It explains why the search tree approach should be preferred to other techniques as much as possible. CSP[1] formalism and associated techniques are one of the best ways to implement exhaustive search, so they are presented in section 2.2. Part 3 introduces a generic constraint solver as a powerful generation tool. It is based on constraint propagation and domain reduction. This part also insists on hotspots to make generation process efficient, like domain modeling and filtering algorithms. Architecture (section 3.1) and main algorithms (section 3.2) are given. Section 3.3 describes how to extend the system at lower cost by creating more complex constraints from boolean combination of existing ones. As an application, Part 4 presents a declarative space planning system for virtual environments.

2. The generation phase

2.1. Related works

There are mainly four approaches to deal with the generation phase problem:

- Specific procedural approach: designed for a particular application; can be very efficient but is not flexible and generally difficult to extend.
- Deductive approach: it is either a rules-based system or an expert system based on an inference engine [12, 13]. The difficulties to build rules and their dependencies in relation to the application are the main drawbacks.
- Stochastic approach: it leans on metaheuristics like evolutionist algorithms or local search. It is a general

[1] CSP = Constraint Satisfaction Problem

method but it suffers from incompleteness of related algorithms. However, it is the only viable approach when the search space is too large. Examples of declarative applications that use such techniques are [6, 11].
- Search tree approach: probably one of the most flexible, more and more used in declarative modeling [1, 2, 9]. It allows a systematic exploration of search spaces and thus the generation of all the solutions either at once or in several times. The user can also control the search process by giving tree branches to prefer or to prune. One of the best ways to implement this approach is to use the CSP model developed about thirty years ago by the AI community. The generality and expressivity of this formalism allow the expression of a wide range of declarative modeling problems.

This is why, when the size of the search space is reasonable, the search tree approach based on CSP can be considered as one of the most appropriate to build a powerful and efficient generic generation tool.

2.2. The CSPs

This section introduces basic terminology and fundamental notions about CSPs.

Definition 1: a CSP
A *CSP* P = <V, D, C> is defined by:
- a finite set of variables $V = \{ v_1,\dots v_n \}$
- a set of domains $D = \{ D_1,\dots D_n \}$, where D_i is the domain associated with v_i; i.e. the only authorized assignment of v_i are values of D_i.
- a set of constraints $C = \{ C_1,\dots C_e \}$; each $c \in C$ is a subset of the cartesian product $\prod_{i \in V_c} D_i$ where $V_c \subset V$ is the set of variables of c.

Each constraint can be defined:
- *extensionally*, for example giving sets of authorized tuples in the case of finite domains;
- *intentionally*, giving either a mathematical equation or a procedural method.

Note that the previous definition does not involve any restriction neither on constraints arity nor on reference sets of domains (each domain of D can be finite or continuous, ordered or not, etc.).

Definition 2: solution to a CSP
A solution is an assignment of a value to each variable from its domain such that all the constraints are satisfied.

Definition 3: partial assignment locally consistent
A partial assignment is said *locally consistent* if it doesn't break any constraint.

Definition 4: equivalence of two CSPs
Two problems P and P' are *equivalent* (P ≡ P') if and only if they have the same set of solutions.

Definition 5: neighborhood of a variable
The *neighborhood* of $v \in V$ is the set of variables denoted $\Gamma(v)$ such that $\Gamma(v) \subset V$ and $\forall w \in \Gamma(v)\ \exists c \in C\ v \in V_c$ and $w \in V_c$.

Definition 6: projection of a constraint
Let c be a n-ary constraint on $V_c = \{Vc_1,\dots Vc_n\}$. The projection of c on $W \subset Vc$, that will be denoted $c\downarrow w$, is the constraint defined on W and equals to the set of projections on W of the values of c.

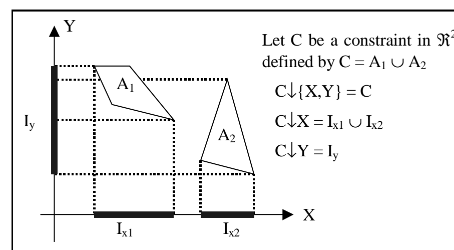

Figure 1: example of 1D projections

Definition 7: a disjunctive constraint
A constraint c on $V_c = \{Vc_1,\dots Vc_n\}$ is said *disjunctive* if and only if $\exists i \in \{1..n\}$ such as $c\downarrow Vc_i$ is non-convex.

Example: C (fig.1) is disjunctive because of the projection on X.

Definition 8: terminology
A CSP P is said to be:
- *under-constrained* if P has several solutions;
- *over-constrained* if P has no solution;
- *well-constrained* if P has one and only one solution.

Though tractable algorithms exist in some particular cases, general constraint satisfaction problems are NP-hard. Nevertheless, many techniques exhibit a good performance in practice in the average case. One of the best approaches consists in combining systematic search with constraint propagation (also called *consistency inference*). The main difficulty is to find an effective trade-off between the two ones.
- Systematic search algorithms explore the search space by testing different possible combinations of variables assignments until a complete solution is found. The most common search algorithm is the chronological backtracking.
- Consistency inference algorithms reason through equivalent problems (see definition 4): each step consists in narrowing down domains of the CSP to make it more explicit. A propagation method embedded in backtracking algorithm – so-called look-ahead scheme – limits thrashing by pruning some detected inconsistent branches of the search tree.

Declarative modeling is an interactive designing task. Consequently most of the resulting generation problems are under-constrained or over-constrained. Hard problems

(transition phase) are very singular. Due to its specificity the intractability of CSP approach must not be considered as a handicap.

To learn more about CSP, refer to [5, 7, 8].

3. A generic constraint solver

In this part, a generic oriented object constraint solver based on CSP is presented. It can be adapted for various generation problems in declarative modeling.

3.1. Architecture

The adaptability of the solver leans on genericity of both constraints and domains. Here are the main classes and the most important associated fields and methods.

The CSP class embeds the problem description:

Class CSP
Main fields
V: list<Variable>
D: list<Domain>
C: list<Constraint>
Main methods
static boolean solve(in p:CSP, in/out c:Context) *finds the first solution (see definition 2) if it exists and others by successive calls (see section 3.2)*
void addVariable(in v:Variable, in d:Domain)
void removeVariable(in v:Variable)
void addConstraint(in c:Constraint)
void removeConstraint(in c:Constraint)
void List<Domain> createBackupDomains() *makes a copy of all domains (line 10 - algorithm solve)*

Table 1: CSP class description

The Variable class describes free parameters of the generation problem:

Class Variable
Main fields
D: Domain *domain associated with this variable*
C: list<Constraint> *constraints that share this variable*

Table 2: Variable class description

The solver is able to manipulate a heterogeneous set of variables. That means several types of domains can be handled simultaneously in the resolution process. This permits to precisely model a wide range of complex generation problems (an example is presented on part 4). Its specificity requires the implementation of a generic support of domains. Such an idea has already been used; for example in [3] to design effective algorithms for a 2D isothetic[2] planning application (EAAS). Note it is a much more general approach than [1, 2, 9] which restrict all the domains to only one mathematical reference set.

[2] isothetic = parallel to reference axes

The performance of the resolution process strongly depends on the domains internal representation. These representations must take into account the following considerations:

- Optimization of the ratio *filtering quality / filtering computation time*: the more a domain data structure is precise and in accordance with its mathematical reference set the more its memory cost and related filtering evaluation time are important. In case of continuous domain, combinatorial explosion due to disjunctive constraints (see definition 7) and domain splitting should be avoided by setting a lower bound for domain size.
- Intersection and union operators should be supported to allow boolean combination of constraints (see section 3.3). When using an exact representation - for example in the case of finite domains, the complementary operator should be added.

Abstract Class Domain
Main fields
V: Variable *variable associated with this domain*
Main methods
abstract boolean isEmpty() *returns true iff the domain is empty*
abstract List<Value> discretize() *returns a discrete representation of the domain (a list of values) (line 9 – algorithm solve – see section 3.2)*
Domain createBackup() *returns a clone of the Domain object*
static abstract Domain Inter(in d1, d2:Domain) *computes and returns* $d_1 \cap d_2$
static abstract Domain Union(in d1, d2:Domain) *computes and returns* $d_1 \cup d_2$
abstract void setSingleValue(in val:Value) *reduce domain to a single value*

Table 3: Domain class description

In the solver, constraints are generic. They can be defined according to the application.

Abstract Class Constraint
Main fields
V: list<Variable> *variables of this constraint*
RM : list<ReductionMethod>
Main methods
abstract boolean testConsistency(in t: Assignment) *returns true if the constraint is satisfied, false otherwise* *Assignment* t *is a list of pairs (variable, value)*
boolean reduceDomains() *constraint filtering method: applies reduction methods on free parameters (i.e. not yet instanciated); returns false iff inconsistency has been detected, (at least one domain is empty)*

Table 4: Constraint class description

A *reduction method* is an over-estimated projection function of a constraint on one of its variables (see definition 6). In the search process reduction methods are used to tighten domains.

Abstract Class ReductionMethod
Main fields
`C : Constraint`
`outVar : Variable` *variable on which the constraint is projected*
Main methods
`abstract` `Domain proj()` *the projection method of* `C` *on* `outVar`

Table 5: ReductionMethod class description

Requirements: let C_E be a constraint on $E = D_1 x \ldots x D_n$. Let F a set such that $F \subseteq E$. We denote C_F the restriction of the constraint C_E to F. The method `proj` – here written proj(C, outVar) - must satisfy the two following conditions:

$$(1): \forall i \in \{1..n\}\ C_E \downarrow V_i \subseteq proj(C_E, V_i)$$

$$(2): \forall i \in \{1..n\}\ proj(C_F, V_i) \subseteq proj(C_E, V_i)$$

3.2. Main algorithms

This section presents the main algorithms required by the solving process.

The search algorithm, called `solve`, is an improved non-recursive backtracking algorithm using both constraint propagation (line 17) and generic dynamic heuristics (`chooseVariable` on line 8 and `chooseValue` on line 15). First call to `solve` gives the first solution of the CSP (see definition 2). Successive calls enumerate all the other solutions of the problem via `Context` structure. `LocallyConsistent` (line 16) checks consistency of partial assignments (using the method `testConsistency` of constraints). This test is necessary because of possible inexactness of reduction methods.

About local consistency: in the general case the useful arc-consistency[3] cannot be achieved in practice. It is replaced by a weaker local consistency that only depends on implemented reduction methods.

Glossary: `Context` is a structure containing five stacks:
`NIVar` = Non Instanciated Variables: stack of `Variable`
`IVar` = Instanciated Variables: stack of `Variable`
`Sol` = Solution: stack of pairs `(Variable, Value)`
`DDom` = Discretized Domains: stack of *list of* `Value`
`Bak` = Backup of domains: stack of *list of* `Domain`

[3] A constraint $c \in C$ is said arc-consistent iff $\forall V_i \in V_c$, $\forall d \in D_i$, d is supported by c. A CSP is arc-consistent iff all its constraints are arc-consistent.

```
boolean solve( in p : CSP;
               in/out c : Context )
Precondition: c is a valid Context structure
Postconditions: return either true if p has a global
  solution according to c or false if there is no more
  solution (the entire search space has been explored).
  c is a new valid context: it can be used to get next
  solution. Current solution (pairs of (var,val)) is
  returned in the field c.Sol.
1  IF |c.NIVar| = 0 THEN getNextVar ← false
2    ELSE getNextVar ← true
3  globalSolutionFound ← false
4  searchNotFinished ← true
5  WHILE NOT globalSolutionFound
6        AND searchNotFinished
7  | IF getNextVar THEN
8  | | c.IVar.push(chooseVariable(c.NIVar))
9  | | c.DDom.push(c.IVar.top.D.discretize())
10 | | c.Bak.push(p.createBackupDomains())
11 | | getNextVar ← false
12 | localSolutionFound ← false
13 | WHILE |c.DDom.top()| ≠ 0
14 |       AND NOT localSolutionFound
15 | | val = chooseValue(c.DDom.top())
16 | | IF locallyConsistent(c,val) AND
17 | |    constraintPropagation(c,val) THEN
18 | | | localSolutionFound ← true
19 | | | c.Sol.push( {c.IVar.top(), val} )
20 | | | BREAK   // exit while (line 13)
21 | | p.restoreDomains(c.Bak.top())
22 | IF localSolutionFound
23 |   THEN IF |c.NIVar|=0
24 |          THEN globalSolutionFound ← true
25 |          ELSE getNextVar = true
26 |   ELSE   // backtrack
27 |   | c.NIVar.push(c.IVar.pop())
28 |   | c.DDom.pop()
29 |   | c.Bak.pop()
30 |   | c.Sol.pop()
31 |   | IF |c.IVAR| = 0
32 |   |   THEN searchNotFinished ← false
33 IF globalSolutionFound THEN RETURN true
34   ELSE RETURN false   // no more solution
```

Algorithm 1: solve

Constraint propagation can be either limited to direct neighborhood (Forward-Checking = FC) or performed on the complete CSP (Real-Full-Lookahead = RFL) [8].

```
boolean constraintPropagtion(
                            in c : Context;
                            in val : Value )
Precondition: c is a valid Context structure
Postconditions: return false iff constraints
  propagation detects inconsistency, i.e. one domain at
  least has been emptied.
  P is an equivalent and simpler CSP.
1  Variable var = c.IVar.top()
2  var.D.setSingleValue(val)
3  FOR EACH v IN c.NIVar
4  | IF v ∈ Γ(var) THEN   // Γ=neighborhood
5  |   FOR EACH cn ∈ v.C
6  |     IF NOT cn.reduceDomains()
7  |       RETURN false
8  RETURN true
```

Algorithm 2: constraintPropagation (FC version)

3.3. Extension: combinations of basic constraints using boolean operators

Let's consider a set of basic constraints defined for a specific application. An interesting issue is: is it possible to combine them using boolean operators (AND, OR, NOT) to create *meta-constraints, i.e.* more complex constraints? In the case of homogeneous variables and extensional constraints the answer is positive and trivial:
let $C_1 = c_{\{v_{i_1} \dots v_{i_n}\}} \subset E_1 = D_{i_1} \times \dots \times D_{i_n}$ and C_2 be two extensional constraints on homogeneous variables; then:

(3): C_1 AND $C_2 = C_1 \cap C_2$
(4): C_1 OR $C_2 = C_1 \cup C_2$
(5): NOT $C_1 = \mathcal{C}_{E_1} C_1$ (complementary of C_1 on E_1)

The only task consists in implementing the three classical set operators ($\cap, \cup, \mathcal{C}$) on considered domains.

Intentionally given constraints (for example, constraints defined by a procedural method)**:** unlike the extensional previous case, classical set operators cannot be directly applied. However it is possible to preserve filtering by observing:

(6): $(C_1$ AND $C_2) \downarrow V_i = (C_1 \downarrow V_i) \cap (C_2 \downarrow V_i)$
(7): $(C_1$ OR $C_2) \downarrow V_i = (C_1 \downarrow V_i) \cup (C_2 \downarrow V_i)$
where V_i is both a variable of C_1 and C_2

Hence, (1) and (2) (see section 3.1) imply:

(8): proj (C_1 AND C_2, V_i) $\subseteq$ proj (C_1, V_i) $\cap$ proj(C_2, V_i)
(9): proj (C_1 OR C_2, V_i) $\subseteq$ proj (C_1, V_i) $\cup$ proj (C_2, V_i)

Therefore, instead of defining reduction methods for (C_1 {AND, OR} C_2), intersection and union operators can be implemented on concerned domains while generally preserving a correct filtering.

The NOT operator is more problematic. In fact proj(NOT C_1, V_i) $\subseteq$ $\mathcal{C}_{E_1}$ proj(C_1, V_i) cannot be used because when D_i has been reduced – i.e. proj(C_1, V_i) has been computed - we don't know yet if its values can be extended to global solutions or not. In conclusion: specific reduction methods must be given to negate an intentional constraint.

Heterogeneous constraints: for example if c_1 and c_2 are two unary constraints on two heterogeneous variables v_1 and v_2, then evaluating proj(C_1 AND C_2, V_1) and proj(C_1 AND C_2, V_2) consists in both computing proj(C_1, V_1) and proj(C_2, V_2). Because of independence between domains D_1 and D_2, the constraint (C_1 OR C_2) cannot be filtered. A new kind of domain - the cartesian product of D_1 and D_2 - should be added to perform a reduction.

Note: in real applications boolean combinations of heterogeneous constraints can appear when handling combination of meta-constraints.

4. An application: a declarative planning system for virtual-environments

4.1. Overview

This part presents a declarative modeler called DEM²ONS-NG[4]. A functionality of this application is to place 3D-objects – for example furniture – in a virtual environment according to a high-level description. Notice that the goal is neither to create the most efficient space planning system, nor to propose the most complete one; but only to design a generative system to illustrate solver capabilities.

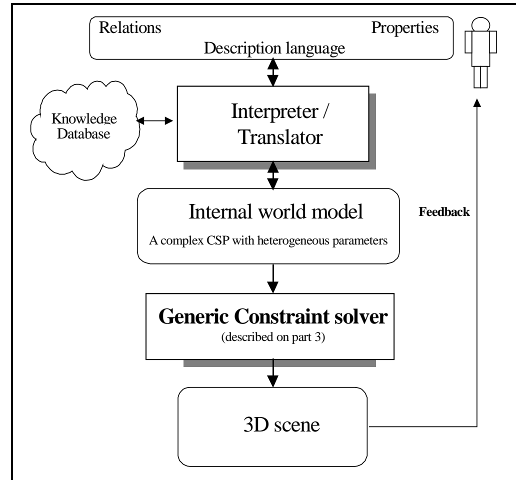

Figure 2: synoptic of DEM²ONS

As said, the underlying generation problem is a space-planning problem. Each 3D-object is identified with its local coordinate space and defined by three different kinds of parameters:

- *Position variable*: location of the local coordinate space origin in the world coordinate space. A position variable is made up of three one-dimensional values;
- *Orientation variable*: an angle in degree. Three orientation parameters (the three Euler angles) are used to orientate an object;
- *Size variable*: a one-dimensional parameter. It describes a dimension along an axis. Three size parameters are needed to define the size of a 3D-object.

[4] DEM²ONS-NG = Declarative Multimodal Modeling System – Next Generation (initial version, called DEM²ONS, was partly developed by Kwaiter in [9]).

Consequently to describe an object in the application, seven Variable objects are required: one for position, three for orientation and three for size.

For 2D space planning problems, it is proven [3] that such a representation gives good results. Here, this idea is extended to 3D. Furthermore one of the most important differences with [3] is the ability of the system to handle any orientation, not only isothetic ones. Notice also that objects can be identified with hierarchical structures of bounding boxes - i.e. a set of local coordinate spaces - for a more accurate representation.

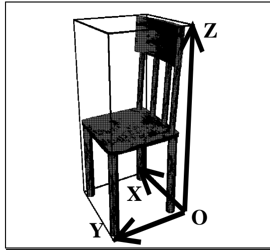

Figure 3: a 3D-object identified with its local coordinate space

4.2. About domains internal representation

As parameters are heterogeneous, a different kind of domain is assigned to each other.

Orientation variable domains: in this application, consistency tests often require a fixed orientation. So associated domains are represented by a finite list of values.

Example: D_O = { 0°; 15°; 30°; 90° }

Size variable domains: to achieve a correct filtering even processing disjunctive size constraints, a list of independent and continuous intervals is used.

Example: D_S = { [0, 100]; [150,155]; [250;300] }

Position variable domains: as position is a three-dimensional parameter, the internal representation must describe a 3D-space. Various representations, either continuous or discrete can be used; for example: a list of polyhedra, a voxel array, an octree, a list of 3D-points, a set of 3D-isothetic boxes, etc. Actually a good choice is a trade-off between accuracy of modeling and efficiency of resulting reducing algorithms. As the system must hold both equality and inequality constraints, the chosen modeling is finally a voxel array coupled with a set of *geometrical elements* (like 3D-point, lines and planes). Voxels describe volumic space generated by inequality constraints while geometrical elements are used to tighten domain in case of equality constraints. The two complementary models are maintained simultaneously. In practice each voxel owns a list of pointers towards the geometrical elements that pass through it.

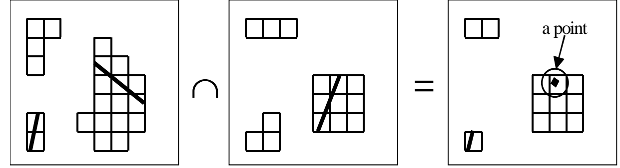

Figure 4: two position domains and the resulting intersection (2D-orthographic projection)

4.3. About constraints

The application kernel is composed of a set of basic constraints. Some examples of significant basic constraints are:

- On position parameters `(pos)`:
  - `Position-fixed (pos, value)` (unary)
- On orientation parameters `(ort)`:
  - `Orientation-fixed (ort, value)` (unary)
  - `Orientation-same-as (ort`$_1$`, ort`$_2$`)` (binary)
- On size parameters `(size)`:
  - `Size-fixed-in (size, min, max)` (unary)
  - `Size-equal-to (size`$_1$`, size`$_2$`)` (binary)
  - `Size-greater-than (size`$_1$`, size`$_2$`, k)` (binary)
- Heterogeneous ones `(obj = {pos, ort, size})`:
  - `Non-overlapping (obj`$_1$`, obj`$_2$`)` (arity is 14)
  - `In-fixed-half-space (obj, half-space)` (arity is 7) Information: a half-space is defined by a 3D-plane that splits 3D-world space in two parts.
  - `On-relative-plane (obj`$_1$`.pos, plane<obj`$_2$`>)` It constrains position parameter of `obj`$_1$ to be on a plane defined from `obj`$_2$.

Each basic constraint has a set of appropriate reduction methods to perform domain reduction during the search phase. They can be combined (see section 3.3) to get high-level constraints such as:

`object` $_{\texttt{target}}$ spatial preposition `object` $_{\texttt{landmark}}$

Examples: `obj`$_1$ `on obj`$_2$`; obj`$_1$ `in-front-of obj`$_2$.

5. Experimental results

Some experimental results are presented to illustrate what can be expected from such a system.

Test scene description: the scene is composed of 30 complex geometric objects (four walls, three tables, two chairs, a TV, two computers, shelves, books, etc.). The high-level description contains meta-constraints like *SCREEN$_1$ is on TABLE$_1$, dimensions of TABLE$_1$ are 2m x*

0.7m x 0.8m, $TABLE_2$ is against the back wall, $TABLE_1$ is to the right of TABLE2, etc.

Solver configuration is:

Voxel size (for position parameter domains)	20 cm
Step for position parameters	20 cm
Step for orientation parameters	15°
Step for size parameters	10 cm
Size of the room (in meters)	4.4 x 3.4 x 2.5

Translation of the description in the internal representation gives:

Number of variables	210
Number of basic constraints	245

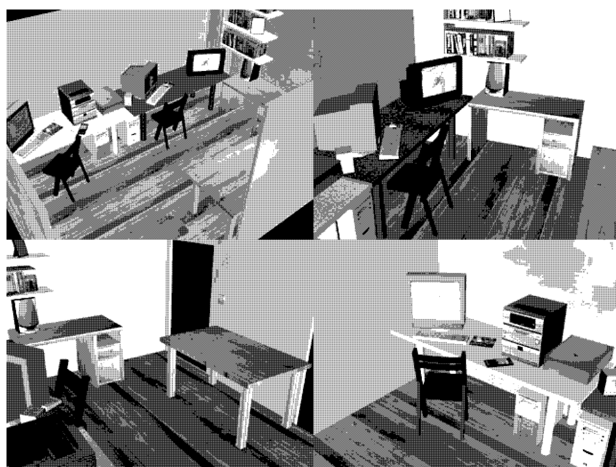

Figure 5: four views of a scene generated by the application that meets the description

Time to get the first solution – the first valid scene according to the given constraints – is approximately 3 seconds. In this example, there are too many solutions to generate them all. However, some other solutions can be obtained (required computation time is generally < 1 second).

Note: the implementation of our application is in Java (JDK 1.3) / OpenGL and runs under Windows 2000© on a PIII600 / 256Mb. With a good implementation in C++ and some heuristics adjustments, computation time should probably be highly restricted.

6. Conclusion and future works

A generic constraint solver based on classical CSP techniques for the generation phase in declarative modeling has been presented in this paper. The use of generic constraints and heterogeneous parameters (via generic domains) allows the adaptation of this tool to various generation problems while preserving correct performances.

Future works concern the improvement of existing generation tools, the design of new ones, and other aspects of declarative modeling like declarative lighting and shading.

References

[1] P-F Bonnefoi, D. Plemenos. Object Oriented Constraint Satisfaction for Hierarchical Declarative Scene Modeling. WSCG'99, Plzen, Czech Republic, February 2-12, 1999.

[2] L. Champciaux. Declarative Modelling : Speeding Up Generation. Proceedings of CISST'97, p120-129, Las Vegas, Nevada, July 1997.

[3] P. Charman. Solving Space Planning Problems Using Constraint Technology. Technical report CS 57/93, Institute of Cybernetics - Estonian Academy of Sciences, 1993.

[4] C. Colin, E. Desmontils, J.Y. Martin, J.P. Mounier. Working Modes with a Declarative Modeler. Compugraphics'97, Villamoura, Portugal, GRASP, 15-18 Dec. 1997, pp 117-126

[5] R. Dechter, F. Rossi, Constraint Satisfaction, Survey ECS, March, 2000.

[6] S. Donikian, G. Hegron. The Kernel of a Declarative Method for 3D Scene Sketch Modeling. GRAPHICON'92, Moscow, Russia, September 1992.

[7] C. Freuder, R. Dechter, B. Selman, M. Ginsberg, and E. Tsang, Systematic Versus Stochastic Constraint Satisfaction. Panel, IJCAI-95, pp. 2027-2032.

[8] V. Kumar. Algorithms for Constraints Satisfaction Problems: A Survey, The AI Magazine, Vol. 13, Number 1, p32-44, 1992.

[9] G. Kwaiter, V. Gaildrat, R. Caubet . DEM²ONS: A High Level Declarative Modeler for 3D Graphics Applications. CISST'97, p149-154, Las Vegas., Nevada, March 1997.

[10] G. Kwaiter, V. Gaildrat . Modelling with Constraints : A Bibliographical Survey. International Conference on Information Visualisation, IV'98 , London UK. IV'98, July 1998.

[11] S. Liège, G. Hégron. An Incremental Declarative Modeling Applied to Urban Layout Design. WSCG'97, Plzen, Czech Republic, Febrary 97.

[12] P. and D. Martin. Declarative Generation of a Family of Polyhedra, GRAPHICON' 93. St Petersbourg, September 1993.

[13] D. Plemenos. Declarative Modeling by Hierarchical Decomposition. The Actual State of the MultiFormes Project, GRAPHICON'95, St Petersbourg, Russia, July 1995.