OBJECT ORIENTED CONSTRAINT SATISFACTION FOR HIERARCHICAL DECLARATIVE SCENE MODELING

Pierre-François Bonnefoi, Dimitri Plemenos
MSI Laboratory, 83 rue d'Isle, 87000 Limoges FRANCE
E-mail : bonnefoi, plemenos@unilim.fr

ABSTRACT

Declarative modeling transforms deeply the task of scene designing by adding to it flexibility, and more creativity and by being more user-friendly. However it's difficult to generalize its use, because it requires too much processing time and memory. The way to improve this is to use the most suited and efficient method from constraint logic programming, to deeply study the declarative modeler in order to exploit all useful informations that it can provide.
The defined framework must be implemented with extensibility, scalability and robustness in order to use it as base for fast implementation of new techniques, to study the influences of these techniques on the modeler process and their combination in order to achieve the best reduction of needed work for obtaining a scene.
KEYWORDS: Declarative Modeling, Constraint Logic Programming, Object Oriented Programming.

1 INTRODUCTION

Imperative modeling, or classical geometric modeling, is practiced by giving all the characteristics of the scene in order to construct it. The main drawback of this approach is to oblige the user of the geometric modeler to spend an expensive amount of time to provide these informations to the modeler. The user must have a complete knowledge about the scene and its different parts, from the coordinates of each point to the complete spatial structure of each object. No room and no support are provided in this modeler to maintain some useful information like non geometric information and very useful information like logical structure of the scene : relationships between the different parts of the scene.
A response to this failure is to define a declarative way to handle geometrical informations and to use the logical informations in order to construct the scene.
So, the user provides a description of the scene, made from properties applied on the different elements of the scene he wants to obtain and this description is used by the modeler to construct the scene satisfying the description.
This is the purpose of the so-called "Declarative Modeler" : freeing the user from giving all the details of the scene and letting him reach a more abstract level of description, combining logical information to ensure that a produced scene fulfil them, generating creative scenes from the description that could be imprecise.
This new type of modeler is studied through a special and very efficient form of declarative modeler : the declarative modeler by hierachical decomposition and its implementation, MultiFormes.
In this paper a framework to study declarative modeling is described and some encouraging results are provided. This framework is based on the use of Constraint Logic Programing and Object Oriented method in order to implement a powerful amelioration of the process of generating solution scene.

2 DECLARATIVE SCENE MODELING BY HIERARCHICAL DECOMPOSITION

2.1 Declarative modeling and modeled objects

Declarative modeling allows the user to define scenes by giving some high level properties of them and letting the solver of the modeler the task to generate all the scenes verifying the given properties.
In our approach, scenes are constituted of "objects" given basically by their bounding boxes (some other features could be associated like texture, color, …). Each one of these boxes is defined by three variables

for spatial location (X, Y and Z) and three others representing the width, the height and the depth (W, H and D) of the box.
In order to allow computation of all possibilities, the different variables characterizing the scene must take a finite number of values (a bounded set of integers for example). So, we can expect that all of the combinations of all the variables will be computed in an acceptable amount of time.

2.2 The hierarchical decomposition

In MultiFormes an improved technique of declarative modeling was initiated, so-called declarative modeling by hierarchical decomposition ([Pleme95]). This technique employs top-down description and criteria of complexity to partition the scene in various sub-scenes. When a scene is too difficult to describe with the language recognized by the modeler, it is partitioned into two or more sub-scenes easier to describe, in accordance with the logical and spatial structure of the whole scene. The same description process is applied to each of produced sub-scenes.
A scene is easy to describe if it is formulated by a small amount of properties which can actually be size properties (higher than deep, as wide as high, …) or form properties (elongated, very rounded, …). When a scene is decomposed into sub-scenes, it is possible to enounce relationships between its different sub-scenes: placement properties (centered in, put on, pasted on the left, …) or size properties (comparison of the dimensions of two sub-scenes: higher than, deeper than, …).

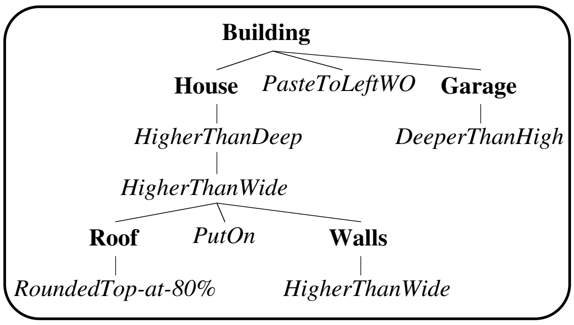

Fig.1 — An example of hierarchical description giving a tree: The hierarchical decomposition tree.

The final user description (see Fig.1) is translated by the modeler into an internal model made up of a set of Prolog-like rules and facts.
The major advantage of declarative modeling by hierarchical decomposition is the ability to specify a scene at different levels of detail. This fact:
– enables the designer to express scenes in a gradual manner,
– enforces the locality of the description (the designer's scope is focused on explaining a part of the whole scene without thinking about other parts)
– allows factorizing of the properties,
– induces relationships between a scene and its sub-scenes (inheritance of constraints for a node from its parent: the bounding box of each sub-scene is included in the bounding box of the scene from which it arises).

2.3 Some concerns about MultiFormes

- All possibilities must be explored to design a scene from its description When the designer enters a description in the modeler, he expects to obtain all occurrences of scenes filling his description.
- All of the properties in MultiFormes are translated into numeric constraints upon discrete workspace solved by exploration of all possible values. This assumption is due to the use of a set of numeric variables to characterize a scene and to a necessary decreasing of range to explore for each of these variables. Each property is translated into a set of constraints. Each constraint has to be satisfied with the values chosen for the variables appearing in the constraint.
- Only linear constraints are applied at one time (non-linear are delayed until a complete solution has been reached). After a solution has been found, then the non-linear expressions (form properties used to deform set bounding boxes) are computed.

3 CONSTRAINT SATISFACTION TECHNIQUES

Each of the properties used to express relationships between the different elements of the problem handled by MultiFormes (each sub-scene and associated variables) can be expressed by arithmetic constraints, more specifically by linear arithmetic constraints except for the shape constraints that are non-linear (this sub-scene has its top rounded at 80% for example).
The whole problem, that MultiFormes tries to resolve for obtaining solution scenes, can be thought in the constraint logic programing paradigm.

3.1 Introducing Arc-consistency

In MultiFormes, all properties are translated into sets of constraints over a discrete domain and the modeler is based on a Prolog-like process. This is the scope of the CLP (Constraint Language Programming), hence methods used in this domain could be integrated in MultiFormes.
A first proposition is to use arc-consistency, [Mackw77] [Kumar92], to enhance solving constraints, because constraints used are applied to finite ranges of values.

The expected gain is to achieve efficient interval decreasing, by filtering values of the various intervals with the information given by the constraint applied to these intervals.
The variables are viewed as nodes and a constraint applied on two variables as an arc between the two connected nodes. A graph is obtained illustrating the relationships between variables, the link established representing a way to check satisfiability of these relationships, by carefully examining all values that could be taken by the two nodes; this introduces the "arc-consistency" concept. The "node consistency" is achieved by checking correctness of unary constraints (a special case rarely encountered in MultiFormes) and extracting inconsistent values from the interval of the constrained variable.
Arc Consistency: *Arc (Vi, Vj) is arc consistent if for every value x in the current domain of Vi, there is some value y in the domain of Vj such that Vi =x and Vj =y is permitted by the binary constraint between Vi and Vj. The concept of arc-consistency is directional; i.e., if an arc (Vi,Vj) is consistent, then it does not automatically mean that (Vj,Vi) is also consistent.*
The method for achieving arc-consistency is to select each arc of the graph, and then filter any inconsistent values from each interval involved by the constraint (with one node and the other to accomplish bidirectional checking) and, in case of interval modification, propagate this modification to all other arcs affected (i.e. all the arcs sharing the altered node).

3.2 CLP(FD)

A specific and powerful method to enforce arc-consistency has been introduced by CLP(FD) for the CSP (Constraint Satisfaction Problem). This was originally proposed by Pascal Van Hentenryck ([Hente89], [CD96]). CLP(FD) is based on use of the ***AC-5*** algorithm and of a single primitive constraint over finite domains "X in r", where "X" is a variable that takes its values in a finite domain and "r" a range, that are expressed in a special form using indexical terms (referring infomation about a range from another variable), and/or arithmetic terms. These terms are constructed by using indexicals: *min*(X), *max*(X), *dom*(X), *val*(X), arithmetic operators (+, -, *), integer values and their combinations to generate arithmetic expressions.
The set of all primitive constraints applied is called the Store. In the store each variable appears only once: X in r1 & r2 & … & rn, where r1, r2, …, and rn are ranges and "&" denote the intersection function between ranges.
More precisely, each variable has a syntactic range (unions of ranges expressed with indexicals) and a constant range (1..10 for example). For each addition of a new constraint to a variable, the range given by the constraint is calculated: each one of the syntactic ranges is determined and their union performed to get a constant range, which is combined with the current constant range of the variable into a new current constant range. The syntactic range is also added to the variable in an unmodified form for future evaluation (when an indexical appearing in the range is changed and new constant range is obtained).
If the interval of the variable has been changed, the indexicals that could have been modified (min, max, dom or val) are searched. The primitive constraints that use the changed indexicals are checked too ("Forward Checking"scheme).
New possible modifications could be propagated ("Look Ahead" scheme), until a fix point has been reached with no more range modification, or until an empty range is produced by failing to satisfy a constraint.

3.3 Implementing a solver for the "X in r" constraint

The constraint solving is performed by adding a new primitive constraint to the store and by calculating the new range of the affected variable. The added constraint is expressed with a syntactic or constant range. In the case of a syntactic range, the constant range produced by the application of the constraint to the variable, has to be calculated. This must occur in an empty store (i.e. with no syntactic constraint in order to avoid loops through indexicals browsing). Next, the constant range is combined with the current constant range of the variable in order to produce the new constant range for the variable.
Arc-consistency alone is not sufficient to calculate a solution, so an exploration of constant range is needed. This is done by adding constraint like "X in {x}" where {x} is a singleton, to force the variable to take a single value in its domain. If the selected value fails, another value has to be tried instead. The method is divided in two phases: the first phase is propagating the modifications produced by an added constraint. The goal of the the second phase is to investigate the constant range of all variables.
A solution is obtained when all variable ranges are reduced to a single value. An error is obtained if a range becomes empty (it is detected in the first phase).
In the modeler, a trail stack must be used to memorize previous states of the range for each variable during domain exploration (for backtracking).

3.4 Translating user constraints into primitive constraints

The purpose of this translation is to reduce the needed effort for the "designer programmer" to describe new

properties to be used in the modeler and to extend its capabilities. The goal is to find a way of expressing new properties in a declarative manner.
A property is given like a set of numeric constraints, and each of these constraints has to be analyzed to produce the related primitive constraints. Each of these constraints must be in "normal form".
A primitive constraint expression must be applied to each variable appearing in the initial expression.
Thus "X=Y+Z" gives:

X	in	min(Y)+min(Z)..max(Y)+max(Z)
Y	in	min(X)-max(Z)..max(X)-min(Z)
Z	in	min(X)-max(Y)..max(X)-min(Y)

4 CONSTRAINT SATISFACTION FOR HIERARCHICAL DECLARATIVE MODELING

4.1 Decomposition of the resolution process

The resolution process is decomposed in two phases :
the first phase is responsible to handle constraints :

- adding a constraint to a variable
- evaluating the new range of the variable
- propagating the induced modifications to other variables
- checking for the entailment of the added constraint (no modified range becomes empty)

The second phase is responsible to enumerate all the possible values that could be taken by the variables.
This enumeration is performed by :

- applying a constraint to the chosen variable in order to reduce its range to a single value,
- using the first phase to check for compatibility of this value with previously instanciated variables and reducing the ranges of other non instanciated variables.

4.2 Processing order

The resolution process requires an order to be defined to handle the elements involved.
So, to construct a solution scene, each variable is explored one after another until all variables are instanciated or ground.
The order of this exploration has to be carefully chosen; it greatly affects the amount of time required to determine a solution.
This order can be defined in different manners and is characterized by the elements on which it is applied. These elements could be single elements of the resolution process (a variable, a constraint, a sub-scene, a level of the hierarchical decomposition tree, …), or a set of elements (all the sub-scenes of the same level, all the variables associated to the same sub-scene, …).
Each of these manners is defined :

- by detecting similarities between elements involved in the resolution process in order to group them and define an order between these sets (the variables describing the same bounding-box, the variables from the same level of the hierarchical decomposition tree, the constraints applied on the same variable, the constraints deriving from the same property of MultiFormes, variables appearing in the same constraint, …),
- by analyzing constraints (number of constraints applied on the same variable, constraints applied on variables of the same sub-scene, on variables from different sub-scenes, …),
- by noticing meta-knowledge about the organization of the constraints induced by hierarchical decomposition of a scene : hierachical (or precedence relationships) between constraints, total independence between variables (no constraints applied between elements of two sub-trees in other place than the common ancestor node), constraints about the bounding box associated to a node of the hierachical decomposition tree and all of its children (all children bounding boxes are included in the parent bounding box).

4.3 Methods for improving resolution process

4.3.1 The rule of the "fail first"

A common rule for improving constraint resolution is to try to reach a possible failure faster ([Tsang93]). This is obtained by exploring first the minimal set of values that can be taken by an element. In fact, if for any group of elements a failure is possible, so, the minimal set of values permits to reach this failure faster.
Another way is to choose the most constrained element, in order to reach failure faster, by awaking and propagating the modifications of the biggest number of constraints.
These different heuristics could be used to define order between variables and are determined dynamically during the resolution process. The magnitude of improvement of the resolution process is limited because the elements involved are reduced to variables. They could possibly be scaled to sets of variables too, but they don't provide any useful information to group variables in sets.

4.3.2 The hierachical decomposition tree as guide for resolution process

The hierarchical decomposition tree is the first useful information in order to construct the order to explore variables. At each node of the tree (a sub-scene) are associated sets of variables, a set of local constraints (applied to the variables of the same sub-scene) and global constraints (applied on variables from different sub-scenes and expressed on the common ancestor node). The strongest law of ordering resolution is to ensure that local constraints are satisfied before checking globals constraints, and due to the decomposition nature (each bounding box, and their associated range of values, is included in the parent bouding box) the constraints associated to a node have to be satisfied before checking for the satisfaction of the constraints associated to the children nodes. So, the decomposition tree could be seen as the graph of precedence between sets of constraints.

A *depth-first search* is the natural way of exploring the values space. This search is made according to the hierarchical decomposition tree.

A more efficient way to explore values is to use an exploration *by levels* (see Fig.2) in order to obtain a sub-solution for all the nodes (in fact all variables associated to these nodes) of a level of the decomposition tree, before exploring the next level.

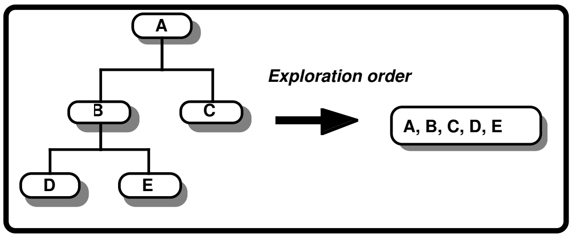

Fig.2 — The exploration order by levels.

The advantage is to use the properties of the hierarchical decomposition tree and its bounding box combination : it is useless to explore all children nodes of a bounding box if the definition of this bounding box is not compatible with another from the same level.

The exploration by levels lets a lot of room for improvements because, for the same level, the order to explore each node is not defined.

4.3.3 The "fail first" principle applied to the hierarchical decomposition tree

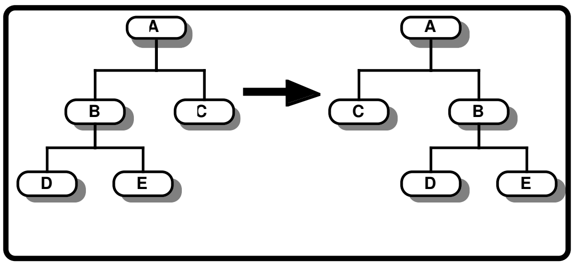

Fig.3 — Reordering conforming to the size of the sub trees.

The heuristic is defined by re-ordering the levels of this tree. The order of exploration for the different sub-levels of the same level can be chosen according to the size of the sub tree associated to each node appearing in these sub levels.

So, the node that has the best chance to finish (no more decomposition) is explored first.

4.3.4 An heuristic versus the decomposition tree

An idea to improve exploration of levels is to decompose all the sets of variables associated to each node (variables from the same sub-scene) of the level and to order them according to the number of constraints applied on them. With this heuristic, grouping is made between variables that share the biggest number of constraints (constraints between the variable and other variables from the same level).

The purpose to achieve is, in case of failure, to backtrack to the variable that seems the most suitable to obtain new possibly compatible value.

Some kind of clustering is made in order to construct the different sets of constrained variables, with probably sets of free variables (with no constraints linking the current variable with other variable from the same level). In these sets, the variables could be ordered according to the size of their domain.

4.3.5 The intelligent backtracking

The intelligent backtracking, [Kumar92], permits to avoid unnecessary work (thrashing or repeated failure due to the same reason) :

Example: the variables X, Y and Z being explored in this order, and a constraint applied on X and Z, and no constraint applied on Y and Z. If X has taken a value incompatible with any value that could be taken by Z, in a regular resolution process (chronological backtracking) when all values are tried for Z unsuccessfully, a new value is taken by Y. And all the values of Z will fail like previously. So all values of Z will be tried with each value of Y without possible success.

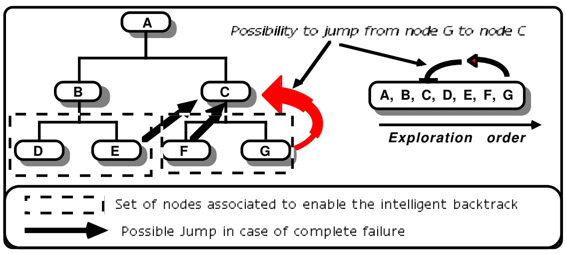

Fig.4 — Intelligent backtracking is defined by the hierarchical decomposition tree.

A better way, or more intelligent, to handle this kind of situation is to backtrack from Z to the variable that could take a value linked to Z value, by using knowledge about constraints.

So in case of complete failure (all values that could be taken by the variable induce a failure) the resolution can jump directly to the variable involved by this complete failure.

This behaviour can be expended to scope with the

nodes of the hierarchical decomposition tree. This intelligent backtrack is based on the hierarchical structure of the decomposition tree and takes advantage of the independence (no constraints are applied between them) of the different sub-trees of the same node (that is the “previous node” to jump in case of complete failure).
By analyzing the hierarchical decomposition tree, some nodes are grouped in sets (see Fig.4) : all the nodes that are children of the same node. To each of these sets is associated the node where the resolution process could jump to in case of a complete failure (no complete solutions for all the nodes of the same set). In Fig.4, two sets are defined : the set of node D and E and the set of nodes F and G.
With the original kind of resolution process (by levels), a failure for the node G causes the resolution to backtrack to node F. In case of no solutions for nodes F and G, the backtrack is made to node E that is not responsible for this failure. With intelligent backtracking, a failure for nodes F and G forces the resolution process to jump to node C without having to finish to explore all remaining possibilities for nodes D and E.
The node where sets could backtrack is the last node of the previous level.

5 OBJECT ORIENTED CONSTRAINT SATISFACTION

5.1 Object oriented approach

This approach allows programmer to scope with the complexity needed to implement a robust solver. The concepts of responsibility-driven design ([CN91]) permits to reduce the complexity by increasing independence between objects from different classes, and by generalizing common behaviours shared between objects from different classes.
It’s easier to check for correctness of a part of the resolution process by checking only the methods and their code of the relevant classes. The type properties provided by objects associated to their defined behaviour make easier the construction of large and complex networks of interconnected objects with some shared protocols among them.
This approach enables many ways to watch interaction between the different object classes and to obtain statistics about the resolution (number of values tried for a set of variables before reaching a solution, …).
A class could be easily replaced by another in order to expand solver capabilities or to implement new improved techniques of resolution.
Implementation of theses techniques is easier by identifying only the involved object classes to modify, and creation of new techniques is also easier by the kind of information that could be provided by the object classes.

5.2 Object modeling of the declarative modeler

5.2.1 Behaviours and classes are associated to the different parts of the resolution process

Each entity involved in the resolution process of MultiFormes has an object class associated, and the set of the defined operations that can be used on it as part of this class. These operations rule and check access to the instance variables of objects (the indexicals for the class *Variable*), implement some behaviours (resolution protocol, enumeration protocol, passivation-activation protocol…).
This approach allows to generalize some of these behaviours and to make some classes sharing the same behaviour in order to use one instead of another, to verify that behaviour is correct and can be exported to other class or scaled to set of objects (resolution process).
The enumeration is made through a protocol defined by three methods :

- set to the first occurrence
- test of next occurrence existence
- go to next occurrence.

This protocol is implemented by the class *Enumerator* whose each object is linked to a variable. An object from this class is responsible to enumerate all possible values that could be taken by a variable. It’s also responsible to restore all modified variables by the new value of its associated variable (during the propagation phase), in their previous state.
An object *Solver* is connected to the Enumerator object in order to translate its behaviour of enumerating all values to the behaviour of searching all the solutions : the possibility to get a next occurrence could give the next solution but the non existence of a next value is similar to a failure, the reset of the solver is translated to take the first value of the range and the search for the next solution explores the different next occurrences in order to reach one that satisfies the constraints.
A class *SetSolver* is also defined to manage a set of objects *Solver*, and to explore the solutions of the combination of this set of solvers.
Each of these solvers could be an object from the class *Solver* or from the class *SetSolver* because they share the same behaviours (resolution protocol). This share ensures the correctness of the resolution protocol and its scalability, but also offers an efficient and easy way to define different groupings of these solvers, each of these groupings implementing a particular way of performing the resolution.
The object oriented approach increases the independence of the different elements of MultiFormes and simplifies the study of the links between these objects in order to construct a solution scene (see Fig.5).

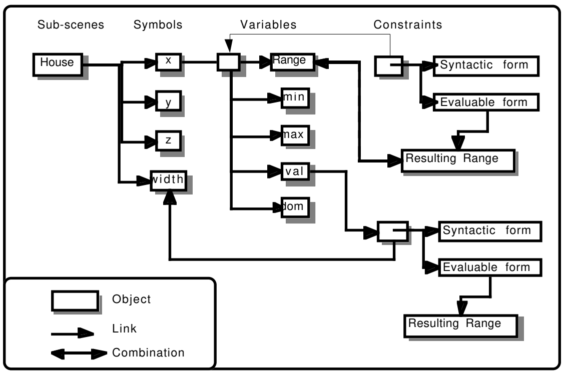

Fig.5 — The different object classes and their connection.

A variable (class *Variable*) gets a name (the symbol that links it to a logical role in the scene), a range that defines the values the variable could take, a set in wich it is included (the set of the variables describing a sub-scene).
The range (class *Range*) is characterized by some indexicals (min, max, dom, val).
The variable owns a set of constraints that are applied to it. It owns a dictionary too, where "entries" are indexicals and "definitions" list of constraints to awake in case of modification of the associated indexical.
Each of these constraints (class *Constraint*) owns a link to the variable on wich it is applied, and a special form that can be evaluated :

- a collection of objects encapsulating links to entry objects of the evaluation and the operations (literally the method to call) to perform on these objects
- a range resulting of the evaluation
- a syntactic representation in order to verify the integrity of the translation from user defined property to primitive constraint.

5.2.2 Different networks of inter connected objects
The knowledge of the scene in MultiFormes is organized according different interconnection networks :

- A symbolic network to represent hierarchical information (hierachical decomposition according to logical or spatial information) where nodes are elements defining a scene,
- A relational network of the set of constraints establishing relationships between the different variables,
- A resolvable network by associating Solver's objects to previously mentioned objects and grouping them with different methods.

5.2.3 Implementation of some heuristics
The "fail first" principle is implemented by allowing an object *SetSolver* to order the elements that it solves according to the size of domain in case of *Solver* objects or size of sub-tree in case of *SetSolver* objects associated to nodes of the hierarchical decomposition tree.
The intelligent backtracking takes advantage of the ability of an object to possess a state. It is implemented by associating a SetSolver to a set of nodes, and determining wether there is a complete failure (a boolean is used to memorize the state of the solver and change of value when the first solution is encountered or when the solver is initialized) or not. A link is also associated to the SetSolver in order to provide the solver that could give a compatible sub-solution.
A powerful heuristic that is not yet implemented in the modeler is to go further with intelligent backtracking and to keep the obtained sub-solution that the resolution process jump over (the other sub-trees issued from the same node from wich the sub-tree where a complete failure is arisen is originated) and to combine them with new sub-solution for the sub-tree where the complete failure appeared, by a cartesian product (like in a parallelized version of the resolution process [BP98]).

6 RESULTS OF THE DIFFERENT HEURISTICS

In this section, our purpose is compare the influence of the implemented heuristics in generation of a test scene. It is not to compare declarative modeling to traditional modeling. Advantages of declarative modeling are well known. For example, som polyhedra generated by the declarative modeler PolyFormes [MM88] are very difficult or impossible to design by a traditional geometric modeler.

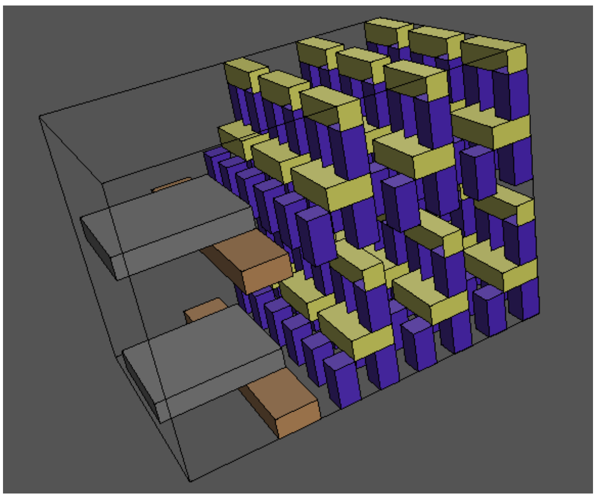

Fig.6 — A view of the first solution scene.

The purpose of all implemented techniques is to reduce the amount of time needed to construct the initial solution, to provide a good order to explore the

solution scenes and to verify that the obtained scene is a solution.
A valuable tool is added : a “Verificator” class that uses the original constraints (linear equations), instead of rewritten constraints, in order to check that values taken by variables satisfy scene description.
The description used to evaluate the efficiency of introduced heuristics defines a building with two school rooms (see Fig.6). In each of these rooms some chairs are grouped into ranges : three chairs are regurlarly interleaved in a range and there are three ranges in a room; a desk is centered on a platform in front of these chairs.
The description for each chair is the same but leads to the same amount of work and time cost (from a solution scene to another, each chair can change independently).

Method	Time needed to obtain first solution scene
Depth first search	>22 hours
by levels	11 minutes 45 secs
by levels and constraints (*see 4.3.4*)	56 minutes
by levels and constraints with clustering (*see 4.3.4*)	39 minutes 16 secs
by intelligent backtracking	2 minutes 38 secs

Table 1 — Results for the different implemented methods

The processed scene contains 1726 variables, 12372 constraints and the hierarchical decomposition tree 285 nodes.
All tests are made on DEC Personal WorkStation with OSF v4.0 (see Table 1).
The most powerful heuristics are those that are applied on the nodes of the hierarchical decomposition tree (intelligent backtracking) but the ordering of variables according to their domain size improves significantly the process resolution (see 4.3.5).

7 CONCLUSION

The choice of CLP(FD) and primitive constraints (with an unified processing from the resolution point of view) provides a useful framework. This framework allows to add new properties in the modeler in a declarative manner. It lets a lot of room of improvement : use of non linear constraints during the resolution process (with some restrictions), definition of properties that can be expressed by ranges with holes, and so on.
A powerful resolution process can be efficiently implemented in an object oriented approach. This approach provides support for implementing powerful techniques to better the resolution process.
But the best improvement obtained is due to the hierarchical decomposition of the scene description.
Not only this approach allows the designer to express more easily his scene description in the declarative modeler but provides also helpful information (constraints independence and precedence) that is required in order to take advantage of the best possible heuristics.
These heuristics have more or less influence on the resolution process and must be carefully combined and in some cases must be chosen or avoided.

REFERENCES

[BP98] Bonnefoi Pierre-François, Plemenos Dimitri, Proposals for Declarative Modeling Parallelization, in SCCG’98, April 98, Bratislava, Slovakia.

[CD96] P. Codognet, D. Diaz, Compiling Constraints in CLP(FD), *The Journal of Logic Programming* 1996:27:1-199.

[CN91] Cox Brad J., Novobilski Andrew J., *Object Oriented Programming an Evolutionary Approach,* second edition, Addison-Wesley Publishing Company, ISBN 0-201-54834-8.

[Hente89] Van Hentenryck P., *Constraint Satisfaction in Logic Programming.* Logic Programming Series, MIT Press, 1989.

[Kum92] Kumar Vipin, Algorithms for Constraint Satisfaction Problems: A survey, Appeared in *AI Magazine* 13(1):32-44, 1992.

[Mackw77] Mackworth A. K., Consistency in networks of relations. *Artificial Intelligence*, vol. 8, no 1, 1977.

[MM88] Martin D., Martin P., An expert system for polyhedra modelling. EUROGRAPHICS’88, Nice, 1988.

[Pleme91] Plemenos Dimitri, *Contribution à l’étude et au développement des techniques de modélisation, génération et visualisation de scènes - Le projet MultiFormes.* Professorial dissertation, Nantes, november 1991.

[Pleme95] Plemenos Dimitri, Declarative modeling by hierarchical decomposition. The actual state of the MultiFormes project. In GraphiCon’95, St Petersbourg (Russie), 3-7 july 1995.

[Tsang93] Tsang Edward, *Foundations of Constraint Satisfaction*, Computation in Cognitive Science, Academic Press, ISBN 0-12-701610-4.