AliceTalks!, an object-oriented reengineering of Alice

LAFORIA, Institut Blaise Pascal, Université Pierre et Marie curie

4 place Jussieu 75252 cedex 05

Abstract

This work related an experiment in combinatorial problems formulation and resolution using the CSP model. We argue that the object-oriented programming provides a suitable technology for this task and the use of objects allows to extend the power of problems formulation.

Constraint satisfaction programming (CSP) is a powerful paradigm for solving combinatorial problems. It provides a general and rigorous formalism which allows to state and solve a great deal of problems. This idea is exploited in the Alice system which proposes an original manner for solving CSP from a mathematical statement. The reconstruction of Alice using the object-oriented technology contributes to the system AliceTalks, which solves CSP and provides an interface that allows the user to understand the solution.

1 Introduction

Constraint satisfaction programming is a powerful paradigm for solving complex combinatorial problems, which has gained attention recently. Typical constraint satisfaction problems include : allocation problems (e.g. human resource management) , scheduling problems (e.g. scheduling vehicles on an a assembly line, scheduling trains, boats or planes), planning (e.g. human resources), etc. The notion of constraint was initially seen as an algorithmic problem, e.g. by [Mackworth 77] and [Laurire 78] who see constraint graphs as networks of relations for finite domains. Complex combinatorial problems have been studied extensively in operation research, graph theory and artificial intelligence for over two decades, leading to the elaboration of a rich theoretical framework. The main notion that came up from these works is arc-consistency [Mackworth 77]. Most existing algorithms are based on the exploitation of arc-consistency : forward-checking [Haralick & Elliott 80], full look-ahead, and various extensions (e.g. backjumping, [Prosser 93]). These mechanisms have been later incorporated into logic programming languages (Prolog III [Colmerauer 89], CHIP [Van Hentenryck 89], CLP (R) [Jaffar & Lassez 87]). More recently these mechanisms have been integrated with object-oriented languages [Puget 94], [Caseau 94] or [Avesani et al. 90], [Roy & Pachet 96].

However, most difficult problems are still out of reach, even using state of the art CSP algorithms or languages. The main reason is well known in AI : general-purpose algorithms are by definition limited, because they do not have the knowledge specific to the problem instance. The idea of exploiting knowledge about problem instances was explored initially by J.-L. Laurière in the Alice system [Laurière 78]. The system was the first to propose a precise problem formulation language and a scheme for combining formal reasoning and constraint propagation to solve combinatorial problems. Even though it was validated on few examples, and shown to be sometimes more efficient than specialized procedures [Laurière 79], this experience is difficult to share : Alice did some extraordinary things but it was difficult to anticipate its behavior, and fully understand its design. On some problems Alice would perform better than specialized procedures, and on some others it would yield poor results. Since Alice is the only system so far to have shown that a non trivial form of knowledge, the formal reasoning could be integrated successfully within constraint satisfaction mechanisms, it is important to understand precisely its essential contributions without referring to a particular implementation of the system.

We report here a project consisting in the reconstruction of Alice using the object-oriented technology. This reconstruction has one practical goal : produce a system that behaves exactly as Alice did, but in a totally transparent way, so that we can study precisely its behavior. More generally this experiment is taken as an example of a successful reconstruction of an AI system using the object-oriented technology. The system is called AliceTalks, both by conformance to a tradition of systems ending by « Talk » written in Smalltalk, and because we wanted Alice to talk, at last.

2 Alice

The ALICE system was developed by J.L. Laurière in the 70's [Laurière 76]. Alice stands for « A Language for Intelligent Combinatorial Exploration», and is both a language - based on set theory - to state constraint satisfaction problems, and a program to solve these problems. The range of problems Alice can state is roughly the set of finite domain constraints satisfaction problems, i.e. problems which consist in finding a *solution* - a collection of values for the variables - in a finite space which satisfies a given set of constraints. Alice has been used to efficiently solve real-world problems such as time scheduling, architecture design or real-world railway planning [Laurière 78].

2.1 Description of the system

A complete description of Alice may be found in [Laurière 76], [Laurière 78], as well as chapter eight of [Laurière 86]. [Laurière 96] reports recent improvements in the system. It should be noted that J. Pitrat developed a declarative version of Alice, using only production rules and meta-rules. Although this latter system should be considered more as a research prototype than a fully-fledged system, its reconstruction using only a declarative formalism (rules) shed new lights on the inner machinery of the system, such as the graph module [Pitrat 93].

A problem in Alice is stated using mathematical symbols and concepts from set theory and function theory. Alice distinguishes two types of constraints : the formal ones which rest on the domain and the simple ones which modify the domain. Then it creates a two-fold internal representation : a graph and a set of constraints. The graph represents the current state of variables of the problem and can be seen as a "compiled" representation of constraints the system considers as simple. For instance, Alice keeps the constraint `f(8)+10g(4)-f(1)f(3)+9≠0` in its formal representation, while constraints as `x=45`, `f(3)<1` or `g(1)≠g(4)` are simple enough for the system to decide to propagate them in the graph.

A general control loop based on backtracking oscillates « intelligently » between two activities : 1) brute force in which domains reductions are propagated using the graph and 2) intelligent search in which constraints are formally analyzed to generate simpler constraints,. The system loops over the constraints set until each constraint is treated, and all information is translated from the formal structure to the compiled one. The solution is finally constructed by a simple enumeration of the graph.

One of the key in Alice's intelligence is the systematic use of heuristics along the reasoning process. Heuristics are used to make choice for constraints, for variables, for values, and for algorithms. These heuristics account for a large part for the ability of Alice to adapt its general search procedure to specific problem instances [Laurière 79].

It is interesting to note that Alice is in fact an extension of the general CSP model. It identifies constraints with a tree structure well-adapted to the process of formal reasoning ; but, like any classical CSP system, it uses a graph to represent the knowledge of the problem, since every step of its resolution is aimed to reduce domains of variables.

2.2 Limitations and remarkable things about Alice

There are a number of features in Alice which make it a truly remarkable and unique system, by contrast with existing approaches in constraint satisfaction. Problems are stated using a formal, declarative language, which does not require any knowledge on constraint satisfaction from the user. Standard simple mathematical concepts like functions and equations are sufficient to express all combinatorial problems. No procedural « hacks » are introduced in the language : Alice is self contained and self sufficient in the sense that it decides by itself how to interpret problem statements. The resolution is based on an association of graph propagation and constraint symbolic manipulation. This allows Alice to avoid combinatorial explosion, by combining constraints and manipulating them in order to extract information. It is important to note that the main characteristic of formal reasoning is that it is in general terribly inefficient, but can sometimes produce information that would otherwise require a tremendous amount of enumeration to get. Alice is somehow able to draw the line between brute force and smart thinking : it will try to make clever formal reasoning only if it deems it worthwhile, and has a set of good heuristics to make this decision. Also Alice proved that a general and formal reasoning supported by good heuristics could efficiently deal with large real-world problems [Laurière 79].

Although Alice was validated on a number of examples, this experience is difficult to share : implemented in PL/I and later in C, Alice remains a black box. Too few information are accessible about the resolution strategy which is chosen and applied, the trace facility is scarce, and the system is hardly embeddable into a larger application. No extension of the system has been developed : Alice is a pioneer system which remains unique in its capacity to mix deduction and propagation of constraints, but it somehow became a « mythical » system with no direct lineage.

3 AliceTalks : revisiting Alice in an object-oriented context

AliceTalks was born from the will to turn Alice into a more open and adaptable system. This reconstruction use the object-oriented technology, for obvious reasons. This section reports on this experience, describes the out-coming system and relates what of clarity and modularity we gained in the system design.

3.1 Project context

AliceTalks was constructed in Smalltalk, with a team of six people. The overall tendency was to try to reuse as much code as possible. This decision led to the effective reuse of various frameworks :

For problem formulation and parsing, the framework *ParserGenerator* from ParcPlace has been used [ParcPlace 95]. This framework made it a lot easier to write the compiler for our system, by allowing the rules to be written in a declarative way. The formal reasoning part was implemented using a *Prolog interpreter* written in Smalltalk [Aoki 95]. Part of the constraint library was built using the constraint library of *BackTalk*, a library of constraint satisfaction algorithms in Smalltalk [Roy & Pachet 97]. The interface of AliceTalks was implemented using *HotDraw*, a framework for two-dimensional structured drawing editors [Johnson 96]. Finally, because of the large size of the project we used an application management tool to handle the problems related to collaborative work (*Application Management* [Heeg 95]).

3.2 Features

Problem formulation in Alice was often hard due to its limitation to domains with literal values. In AliceTalks, we extended the original language of Alice by providing facilities for specifying problems with complex domains. The formulation language is also enhanced to the range of problems where variables are either integers or any objects belonging to any class, and

constraints are expressed either by the send of a message to a composite variable or by mathematical expression.

Since we want to build an open system where the user could understand the solution provided by the system, we integrate an interactive interface with the system : by analogy with a Smalltalk debugger, a synchronous viewer shows a detailed chronological account of the steps. It displays the trace in a structured way and allows to filter information to get only a viewpoint on the solution.

Finally, an effort has been done to integrate the identification and use of heuristics in the system. Since number of steps of the solution are principally based on choice criteria, we developed sophisticated tools for creating or editing heuristics and linking them to point in the program. The user can also influence on search efficiency with no direct code modification.

3.3 Overall design

AliceTalks is made of six modules ; five extend the existing modules of Alice and one evolves the ability to work on the solution efficiency. The *Problem compilation* module builds the representation of the knowledge of the problem from the input description. The *Control* module controls the search of solutions using a backtrack procedure, and manages the context stack. The *Graph* module organizes and checks data about unknown variables. The *Formal reasoning* module encompasses the constraints of the problem and deduces new constraints, while it maintains the whole consistency. The *Output visualization* module shows current tree search and state of the problem while it controls process speed and breakpoints. Finally, the *Heuristic* module is in charge of connecting heuristics to the inner strategy for improving its efficiency.

Each module communicates with each other through well specified protocols (see Figure 1) : 1) *Control* loop chooses one constraint, according the *Heuristics* and gives it to *Formal reasoning* or *Graph*. 2) *Graph* accepts only simple constraints whose propagation is immediate and returns problem solutions if they exist. 3) Whenever the current context is inconsistent, *Graph* and *Formal reasoning* report the contradiction to *Control*. They exchange themselves information : *Graph* signals *Formal reasoning* when any value of variable is either forbidden or assigned ; *Formal reasoning* asks *Graph* for domain of values when it treats a constraint of its.

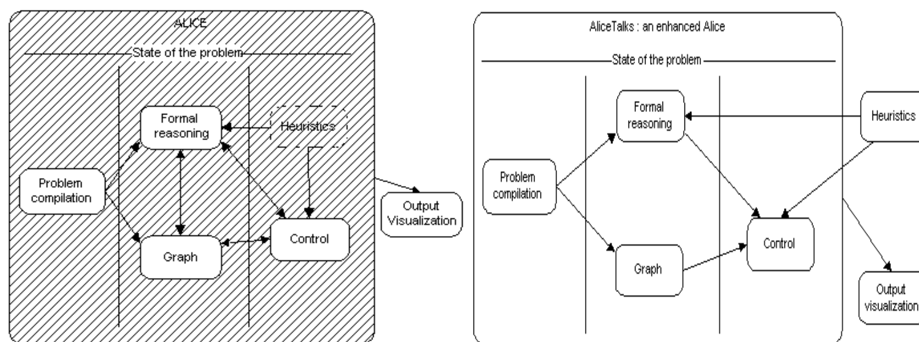

Figure 1. The division of Alice into functional modules. On the left, the initial design of Alice. On the right, the design of AliceTalks enhanced by the *Heuristics* module

The next sections will give an overview of main notions in AliceTalks ; and so, we will use the design pattern vocabulary [Gamma & al 95].

3.4 Problem formulation and compilation

Problems are formulated as the computation of functions between finite sets, which satisfy constraints. Thus, no knowledge of Smalltalk is required to state a constraint problem.

The figure shows a statement of the n-queens problem. It aims to place n point on a chessboard so that there is only one point on every row, column and diagonal. This problem contains $2(1+n^2/4 - n)$ constraints and may be very bothering to formulate. In AliceTalks, we reconstruct the high-level semantic constraints of Alice. Logical expression syntax includes quantified formula and conditional expression using the logical implication operator. With these facilities, formulating the 8-queens problem becomes very easy.

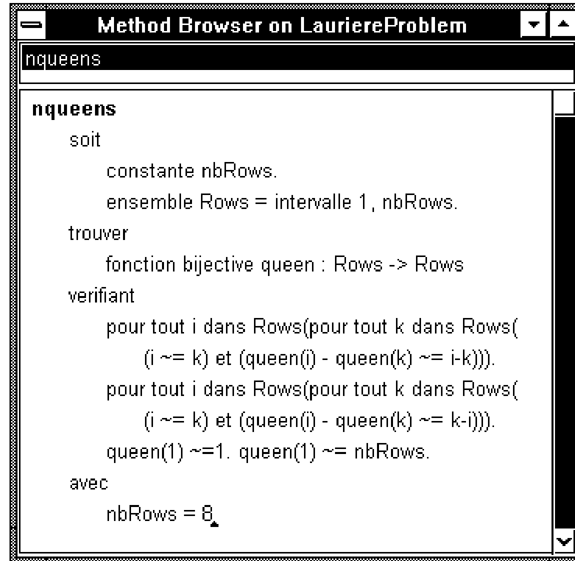

Figure 2. The formulation of the n-queens problem in AliceTalks.

The translation between the problem formulation and its internal representation requires a parser and a compiler. We used the framework *ParserGenerator* [ParcPlace 95], an advanced tool integrated in VisualWorks, and developed by ParcPlace. It allows to generate a left-to-right descending parser and a compiler for a grammar specified in a simple rule-based formalism. The out-coming syntactic tree is then analyzed by builders. First a semantic analysis checks declaration part consistency ; then construction produces the initial state of the problem. Compilation uses the *Visitor* pattern. Recall that the *Visitor* pattern is used for representing explicitly a processing to be performed on the elements of an object, usually a tree. It lets the user define a new operation without changing the classes of the elements on which it operates. Each specific builder is a *Visitor* which walks through the syntactic tree.

We show in Figure 3 the compilation overview from the problem statement to the object representation.

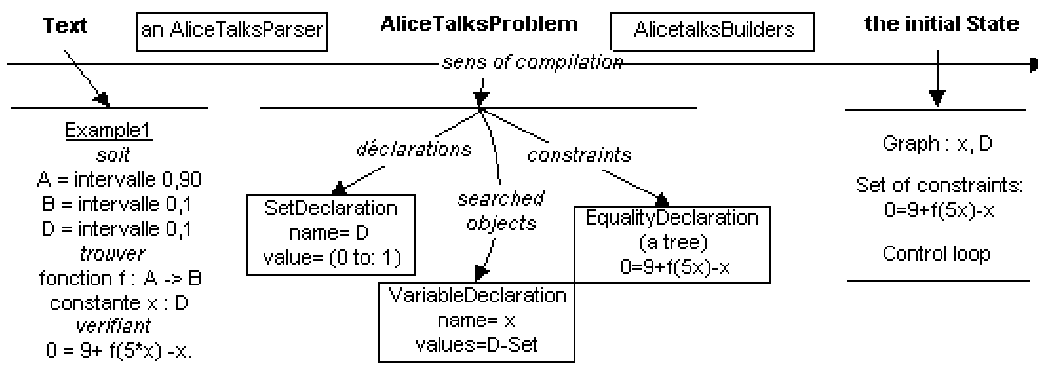

Figure 3 the Alice construction of the initial knowledge of a CSP

The visitors are organized in a hierarchy that essentially regroups their common behaviors. This allows to apply several treatments to a node by only changing the class of the visitor to use The Figure 4 shows all these features with an example.

Figure 4 Example of connection between a syntactic node and two visitors. The visitor class is responsible of the processing.

3.5 Formal reasoning

The global issue of *Formal reasoning* is to empty constraints set as quickly as possible, so that the solution may be found directly in the graph.

One of the main ideas of Alice is that it generates constraints from other constraints and for that rewrites constraints in a regular form. AliceTalks's rewrite system is a set of equivalencies used as a pattern-directed program that returns as output a simplified term equal to a given input term. Rules are applied in any order to any matching sub-term until no further applications are possible. The convergence of the rules system garantees a regular expression unique for any term to simplify. In our system, a term represents an expression that appears in a constraint and may be simplified. Instead of finding a normal form - which is impossible for any expression in general, AliceTalks rewrites expressions in a regular form. This notion of rewriting logical regular expressions in order to solve problems was already introduced in Alice but without a real formalization. We'll relate the Alice's symbolic computation and deduction of constraints to the field of term-rewriting (see section 5)

3.5.1 Algebraic simplification

Normalization of Alice (understand simplification in the current vocabulary of symbolic calculus) is performed by applying rewriting rules and aimed to simplify expressions and consequently the complexity of the constraints. The rules we wanted to use, are easily represented by Prolog clauses. So, instead of twisting the rules to fit the object-oriented language, we used the *MeiProlog* in [Mei 95]. Rules are written exactly as if they formed a Prolog program (see Figure 5).

```
0 + X -> 0, !.
X + 0 -> 0, !.
X >= Y -> isConstant(X), isConstant(Y), eval(X >= Y), !.
(X + -(X))+ Y -> Y, !
(X + Y) -> (Y + X), !. si Y <order< X (*)
(X/Y + (Z/Y)) -> (Z + X)/Y, !.
```

(*)The function <order< returns true if the first argument is before the second one in the order of Alice expressions. We need to order sums for controlling commutativity and associativity of expressions [Bachmair 91].

Figure 5 Any rewriting rules used by AliceTalks for normalizing.

The framework *MeiProlog* is also well-adapted to the symbolic expression system of AliceTalks : both AliceTalks expressions and internal *MeiProlog* terms are represented by trees. The *MeiProlog* rule base requires as input a literal description which is then translated in a structure of tree whose terms exactly identify the components of the expression. The string representation is itself produced using the *Interpreter* pattern : each significant node class asks its sons for their *MeiProlog* string, and merges the results.

3.5.2 Propagation global of constraints

The second important behavior of the *Formal reasoning* module is the application of a collection of algorithms on the structure of constraints in order to deduce simpler constraints. The algorithms perform two types of operations and thus two different patterns have been used for technical design. On one hand, algorithms of coefficients simplification are well implemented using the *Interpreter* pattern, because this feature is intrinsic to constraint and also to expression. Recall that the *Interpreter* pattern applies to trees and redefines a behavior for each type of node it is significant to. On the other hand, the *Strategy* pattern provides exactly what we wanted AliceTalks to offer, as modularity and extensibility. Indeed it defines a family of algorithms, encapsulates each one and lets the algorithm vary independently from clients that use it. Thus constraint is passed as argument to algorithm main function. Each generic algorithm is implemented in a class. It is instanced whenever treatment it specifies has been chosen to simplify a constraint. Hence this instance represents one of its applications with a particular context. Using this design, elaborating new algorithms does become easy.

3.5.3 Propagation local of one constraint

The third behavior of the *Formal reasoning* applies on one constraint. The local filtering of a constraint attempts to reduce domains of values for any variables related by this one. For reasons of performance and also because we thought of reusability of frameworks, we reuse a class of BackTalk [Roy ?], a framework for satisfaction constraint with filtering.

3.6 On extending CSP formulation to objects

CSP paradigm is a general model which has the interesting property to be independent of the semantic of the constraints. Thus we extend our CSP formalism without any real difficulty. We tend also to hook a higher level of semantic on our variables and constraints. Variables have values in a set of composite objects and constraints appears with the form of the send of a Smalltalk message (see Figure 6 : A simple problem on domain of rectangles.).

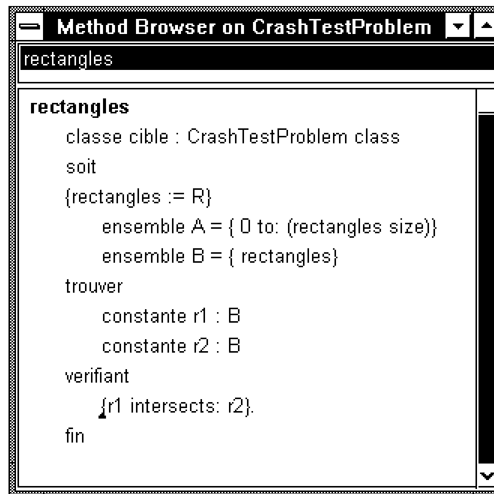

Figure 6 : A simple problem on domain of rectangles.

We observe that the number of variables is reduced comparing to the same problem without the object facility of statement. Therefore the CSP is clearly more understandable. [Roy & Pachet 97] have recently shown that the complexity of the object-CSP problem is reduced too. However the use of object in CSP makes us to forget Alice reasoning. There's only on treatment for object constraint : local filtering. In section 4.2, how the object-oriented technology let us to reuse one class for this operation.

4 What we gain from an object-oriented conception of Alice

The object-oriented technology properties match exactly the design of AliceTalks. Indeed Alice is described with four modules which exchange information. For an obvious preoccupation with being the most closed to the original system, we chose to divide our system in a similar way. The objet programming is also well-adapted to our experiment. Besides to be modular, OO conception let the implementers build a clear and well-defined program. We try to exploit this property for our work.

4.1 Constraints and expressions hierarchy

In a symbolic manipulation point of view, constraints have a dual nature : a constraint is both a *relation to satisfy*, and *a syntactic expression*. Each constraint has, in AliceTalks, a type which is used by the control module to decide for instance which constraint will be chosen at each cycle. The syntactic aspect is necessary for manipulating formally the constraint, and combining constraints with each other. There are therefore two different hierarchies of classes in AliceTalks : a hierarchy for representing Boolean and arithmetic expressions from a *syntactic* point of view, and a hierarchy for representing *types* of constraints. A constraint is then represented by an instance of a given constraint type, which points to an instance of the corresponding expression. Syntactic expressions are designed to provide easily functions of rewriting ; they are represented by a tree whose nodes are connectors. Constants and variables are the leaves. Figure 7 shows relative position between constraint types and expression classes on one instance.

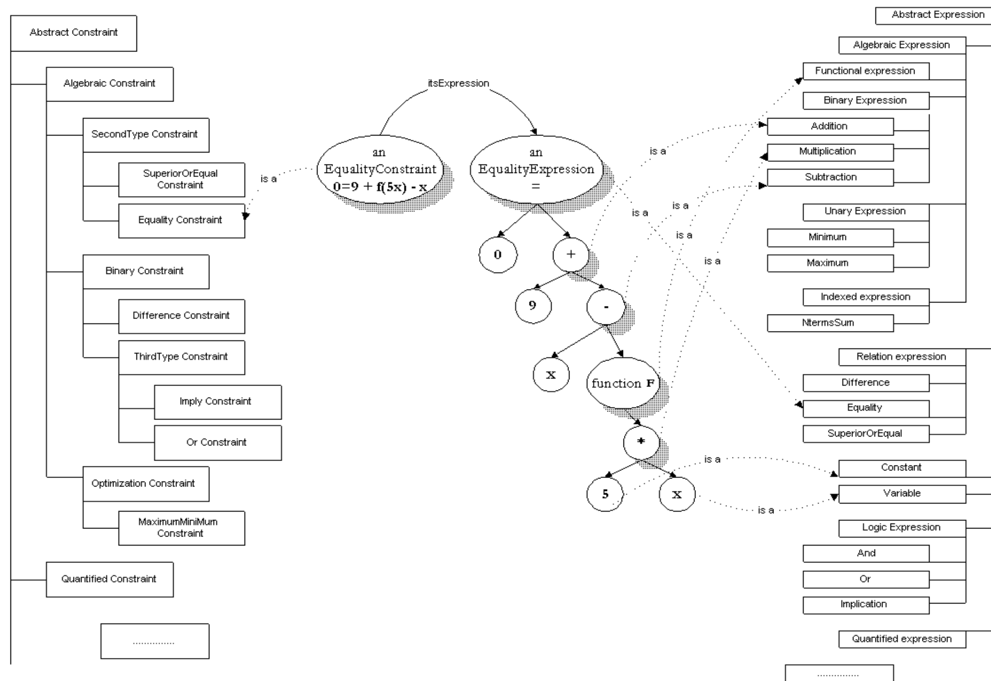

Figure 7 the equality constraint "0 = 9 + f(5*x) - x" and its syntactic expression are related to constraints types and classes of expressions.

Constraint types are organized in a hierarchy where constraints are distinguished by the nature of their operator symbol. Specific properties of a constraint type are exploited, using the *Interpreter* pattern. For instance, class `SuperiorOrEqualConstraint` redefines method for simplification because Alice proposes for this type an more clever formal algorithm than classical one.

This two-fold hierarchy allows to add new type of constraints or expression. Besides adding properties in the class, a small number a methods should be redefined to provide a specific semantic. For instance, a new type of constraint should redefined the tests of `satisfaction` and `contradiction` of the constraint. If any formal smart deduction is known it would be redefined in the class with the method `createAndAddSimplerConstraints`

Thanks to this hierarchy, we extended the system with object constraints (see section 3.6).

4.2 Reuse in symbolic computation by using patterns

As we said in section 3.5, we reuse the framework MeiProlog to provide our system a "on-line" rules-based system. We use also the strategy pattern to implement symbolic computation, as linear combination. Thus we are potentially able to reuse classes of algorithms from other systems. We practically include a class of BackTalk [Roy ?], a framework for constraint satisfaction, for local filtering.

The *Visitor* pattern has a direct application in the manipulation of the Alice Expressions. It allows to regroup common steps and features of several algorithms. When an existing algorithm has similarities with an existing one, heritage of algorithm really simplifies the definition. For instance, implementation of the evaluation of the maximum should be obvious, if it inherits from the evaluation of the minimum of an expression. Only methods for unary minus, unknowns and difference must be redefined.

4.3 Modularity in AliceTalks

This section describe briefly the other modules of AliceTalks. *Graph* and *Control* complete the inner system, whereas the modules *Heuristics* and *Output Visualization* behave as point of communication with the inner machinery of AliceTalks.

4.3.1 Graph

The *Graph* module represents the CSP knowledge and manages the simple constraints propagation. It replaces standard CSP algorithm like values propagation used by BackTalk and IlogSolver. Compilation of simple constraint simulates a total AC algorithm, limited to unary and binary-linear constraint. A simple structure of graph supports this feature. It contains as many sub-graph as functions to find in the goal of the problem. (see the example of n-queen). For each, a sub-sub-graph for each domain of function and a sub-sub-graph of values in the co-domains of the same. The *Graph* module propagates also domain reduction and detects failure of an instantiation as soon as possible. Semantically, the *Graph* module includes all potential solutions of the problem and in particular the real ones, if they exist ; that's the reason why it is in charge to enumerate solutions as soon as there is no more constraint to check.

4.3.2 Control

The *Control* module represents the control loop which coordinates *Graph* and *Formal reasoning* at top level. It makes choices intelligently and reconsiders them if it finds no solution. It pushes and pops context, using a classic backtrack strategy. That way, it develops a search tree containing problem states it has explored. Context only represents current state of the problem *i.e.* graph and constraints to satisfy. Indeed it suffices to let the control know at which point it has left the search in this branch. To save context with the slightest code, we used the BOSS (BinaryObjectStorage) mechanism. Like all searches by exploration, control loop calls a number of heuristics to decide which state to visit in the search tree it builds.

4.3.3 Heuristics

In most of the CSP solver, heuristics are determined once, by the designer himself. In AliceTalks, strategy is open and may be defined by the user. This latter can decide (in the place of the system) which heuristic will be applied at each key-point of the search procedure . So heuristics have a double location : they are involved in the deep mechanism of the program and may however be specified from a browser. To take account this feature of the system, we added the module *Heuristics*. The module proposed a real efficient tool for testing heuristics. Definition of heuristics takes on two aspects, the informal one and the structural one. Heuristic is viewed as both a function and comment. It is a calculation resource to guide the system while it 's walking through the state space of a problem. It must possess also a comment which explains its features : role, context of application, argument, result, use.... Figure 8 shows the interface for informal definition.

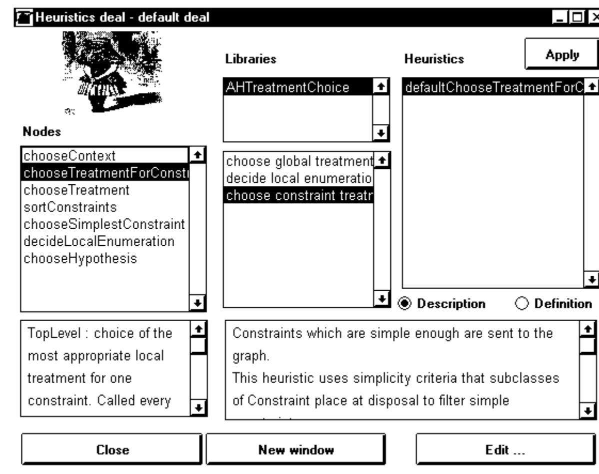

Figure 8 Interface for creating user-defined heuristics

Heuristic is first of all designed by a symbol derived from the name of the function it holds. The body is stored in a method of the library, which refers to the context of the call, like the caller module or methods of test. Cross references may be done also : heuristic may combine the results of simpler heuristics. Therefore it is obvious that the way heuristics are used depends largely on the context they are called in. In AliceTalks, every definition of heuristic is supported by a caller. It represents a main module -*Graph*, *Control* or *Formal reasoning*- and is the sole entity that allows to access local and global context from the tool interface. The introduction of a caller is also the solution of the following problem : "how writing a method without knowing explicitly where it will be executed ?". The answer is impossible unless we have a reference on an object with public methods which return significant information.

A heuristic is finally designed by its name, its body and the library it belongs. Libraries are ordered by type of action performed by their components. This concept is defined in the class `AHeuristic`. One instance of this class represents one heuristic ready to be evaluated. With this tool, creating a particular library becomes very simple : we only have to subclass the category of heuristics of the same type.

4.3.4 Output visualization

While visualization of the solution, the system adapts itself to the requirement of the users and helps them to understand the solution search. This concerns the *Output visualization* module and the tools -the *tracer* and the *filter editor* - it involves. The Figure 10 shows, on an example, the tracer when the system has just found the first solution. The tracer keeps track of the search and displays significant events. It indents traces in a list : traces with the same indentation concerns the same task. Each trace provides local context of the step or event it is supposed to signal. Global state of the problem is permanently shown since the tracer should above all show a global information on the solution.

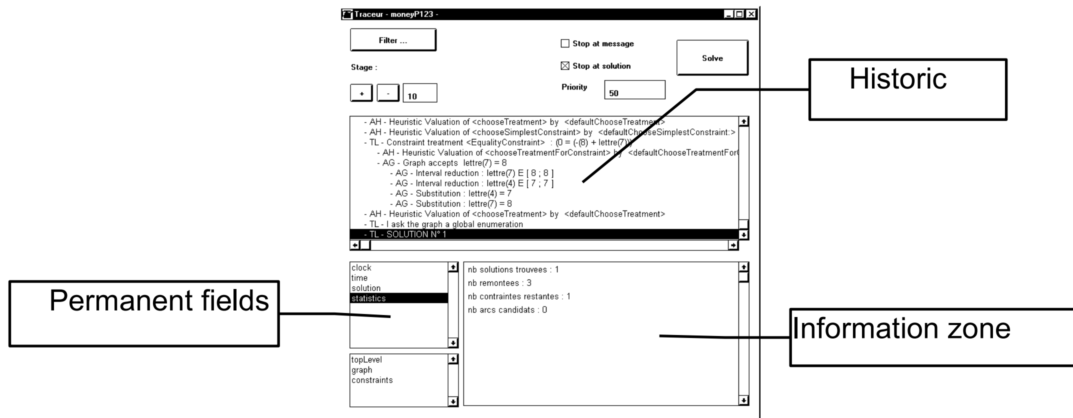

Figure 9 Interface of the tool which visualizes Output trace for the `send+more=money` problem

Thanks the trace filter editor, he can modify state of a trace and consequently avoid or set on its notification. So solution may be viewed just with the point of one type of action. For instance, Figure 9 only shown graph actions and control loop.

All modules are *clients* of the tracer. When they want to signal an important event, step or result, they sent it a message of trace. To preserve modularity and readability of the system, client and tracer should never know each other. That's why trace is reified. Distinction is made by the module which sends the trace.

4.4 Adaptation to domains

Syntax may be extended. Facilities are provided by the use of a framework and appropriate pattern. Changing the parser is done by sub-classing the compiler to inherit from ParserGenerator facilities and then adding rules. Extending syntactic (or semantic) analysis is simply done by creating a new class of syntactic node and methods in Visitor classes that concern it.

Alice problem specifications can include Smalltalk code. Code in declaration part is compiled and executed just before the construction of problem. It may besides affect global objects provided it already exists in the environment. Smalltalk block is even allowed in constraint definition. Of course this extension should be used carefully : the block must return a value significant in the constraint it appears in.

New constraint type may be added. It must be done by heritage and redefinition of methods for accessing, simplifying, properties testing and displaying.

New algorithm may be added. It may be included in a base of Strategy patterned algorithms for constraint generation. Heuristics should refer to it in order to apply it for a particular context.

5 Related and future work

Several CSP systems are today available and are even used at an industrial level. They mostly include a graphical interface for specifying CSP. This way of formulation intrinsically integrates object-oriented components since constraints relate graphical objects like square or circle. As example, we want to mention : 1) Equate [Wilk 91], which formulate constraints using mathematical-like equations where domains of variables are set of objects 2) Kaleidoscope [Freeman-Benson & Borning 92], a CSP framework mixing declarative aspect of constraint and imperative nature of OO languages.

[Roy & Pachet 97] studied in the BackTalk system, the difficulty to integrate constraint satisfaction programming and object languages for conception of problems. Objects CSP problems include composite objects and constraints on objects. The complexity of the formulation influences directly the efficiency of the resolution. Also formulating complex CSP with objects helps to divide specification into several levels of complexity, consequently improving the solution. The declaration of constraints in an object language allows to replace constraints by methods, applied on object variables.

One of the main ideas of Alice is that it performs generation of constraints and separately applies rewriting rules for simplification of constraints. A great work has been done recently on rewriting and automatic deduction. [Jouannaud 93] describes a survey of the main applications of rewriting, which is now an important sub-field of in computer science. In particular formal calculus systems use term rewriting rules to compute a normal form of any expression whenever it's possible(because it's unfortunately not in general) [Bachmair 1991]. Term rewriting is also exploited in automatic theorem (logical formula) proof as in Coq [Huet & al 95]. We'll try to relate our work on CSP to symbolic rewriting of logic and arithmetic expressions.

Our experience has shown up the difficult problem of the problem formulation into a constraint-oriented representation. In particular, problems that are generally formulated in a natural language remain not obvious to translate in a declarative way. Difficulties come from the lack of method to state correctly a problem. Initial and final state should be defined and all rules to modify the current state should be declared in the form of constraints. In order to facilitate the problem's rules formulation, our system allows high-level semantic operations such as implications in the problem statement. But more efforts can be made to improve the language of formulation of our system.

6 Conclusion

In the range of problem resolution, it is important to be able to precisely formulate the problem in a general formalism including efficient algorithms. The CSP approach appears as the most adapted for this task. So we reconstruct Alice, an AI architecture in constraint programming, using the object-oriented technology. Classic AI architecture were designed with a closed vision of the world. Things are changing, and these architectures are no longer adapted. Object-oriented reengineering allows to open these architecture, by making them adaptable to specific domains. In our CSP formulation, we can use variables with objects domains and constraints on objects. As gain from our experiment, we built a modular and open system that allows the user to understand how the former solves some problem.

References

[Aoki A. 1994] Aoki A. Object-Oriented Analysis and Design Techniques. S. R. Center. Tokyo, Japan. 1994.

[Avesani et al. 1990] Avesani et al. XXX. . 1990.

[Brant J.M. 1995] Brant J.M. HotDraw. Ph. D. thesis, University of Illinois, Urbana-Champaign, 1995.

[Caseau 1994] Caseau XXX. . 1994.

[Gamma E. & Helm R. & Johnson R. & Vlissides J. 1995] Gamma E. & Helm R. & Johnson R. & Vlissides J. Design Patterns - Elements of reusable Object-Oriented Software. Addison-Wesley, 1995.

[Haralick R.M. & Helliott G.L 1980] Haralick R.M. & Helliott G.L Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* . vol 14, pp. 263-313, 1980.

[Heeg G. 1995] Heeg G. Application Management 1.2 Versioning and Packaging for ObjectWorks Smalltalk, User's manual. *Objektorientierte Systeme. Dortmund (Germany).* Heeg systems. February 1995.

[Jaffar J. & Lassez J.L. 1987] Jaffar J. & Lassez J.L. Constraint logic Programming. *Proceedings of IEEE 4th international Conference on Logic Programming, Melbourne.* M. Press, 25-29 may 1987.

[Johnson R. 1992] Johnson R. Documenting frameworks using patterns. *Proceedings of OOPSLA'92,* pp. 63-76, october 1992.

[Laurière J.L. 1976] Laurière J.L. Un langage et un programme pour énoncer et résoudre des problèmes combinatoires. Ph. D Thesis, University Pierre et Marie Curie, Paris, 1976.

[Laurière J.L. 1978] Laurière J.L. A Language and a Program for Stating and Solving Combinatorial Problems Artificial Intelligence. *Artificial Intelligence* . vol. 10, pp. 29-127, 1978.

[Laurière J.L. 1979] Laurière J.L. Toward efficiency through generality. *the sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan.* vol. 2, pp. 519-521, 1979.

[Laurière J.L. 1986] Laurière J.L. Intelligence Artificielle, Résolution de problèmes par l'homme et la machine. Eyrolles, Paris. 1986.

[Laurière J.L. 1996] Laurière J.L. Propagation de contraintes ou programmation automatique. *Laforia Institut Blaise Pascal-CNRS, University Pierre et Marie Curie* Paris. juin 1996.

[Lenat D.B. 1982] Lenat D.B. The nature of Heuristics. *Artificial Intelligence* . vol. 19 (2), pp. 189-249, 1982.

[Mackworth A.K. 1977] Mackworth A.K. Consistency in networks of relations. *Artificial Intelligence* . vol 8, pp. 99-118, 1977.

[Pachet F. & Roy P. 1995] Pachet F. & Roy P. Mixing constraints and objects: a case study in automatic harmonization. *TOOLS Europe '95,* Prentice-Hall, pp. 119-126, 1995.

[ParcPlace 1995] ParcPlace VisualWorks Advanced Tools User's Guide, release 1.2. ParcPlace-Digitalk, Sunnyvale (Ca),. 1995.

[Pitrat J. 1993] Pitrat J. Penser autrement l'informatique. Hermès, Paris. 1993.

[Prosser P. 1993] Prosser P. Domain filtering can degrade intelligent backjumping. *Proceeding of the 13th. Joint conference on Artificial Intelligence., Chambery.* pp 262-267,

[Roy P. & Pachet F. 1997] Roy P. & Pachet F. Conception de problèmes par objets et contraintes. *Journées Francophones des Langages Applicatifs-JFLA'97,,* janvier 1997.

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

[Borning A. 1994] Borning A. Principles and Practice of Constraint Programming. *2nd International Work PPCR '94, Rosario, Orcas Island, Washington, USA.* Springer-Verlag, vol. 874, 2-4 may 1994.

[Colmerauer A. 1990] Colmerauer A. An Introduction to Prolog III. *Communications of the ACM* . vol. 33 (7), 1990.

[Freeman-Benson B. & Borning A. 1990] Freeman-Benson B. & Borning A. Kaleidoscope: Mixing Objects, constraints and Imperative Programming. *Proceedings of OOPSLA/ECOOP '90, Ottawa, Canada.* ACM, SIGPLAN Notices, pp. 77-88, 1990.

[Puget J.F. 1992] Puget J.F. Programmation Par Contraintes Orientée Objet. *12th International Conference on Artificial Intelligence, Expert Systems and Natural Language,* Avignon, France, pp. 129-138, 1992.

[Van Hentenryck P. 1989] Van Hentenryck P. Constraint satisfaction in Logic Programming. MIT Press, Cambridge, Massachusetts. 1989.

[Wilk M. 1991] Wilk M. Equate : An Object-oriented Constraint Solver. *Proceedings of OOPSLA'91, Phoenix, Arizona, USA. ACM SIGPLAN Notices*, vol. 26, n°11, pp. 286-298, November 1991.

[Huet G. & Oppen D.C. 1980] Huet G. & Oppen D.C. R. Book Equations and rewrite rules : A survey. *Formal Language Theory: Perspectives and Open Problems. New York.* Academic Press ed, pp. 349-405,