DEPICT: A High-Level Formal Language For Modeling Constraint Satisfaction Problems*

Abdulwahed Abbas, Edward Tsang
The University of Balamand, Lebanon and The University of Essex, England, U.K.
Ahmad Nasri
The American University of Beirut, Lebanon

Abstract: The past decade witnessed rapid development of constraint satisfaction technologies, where algorithms are now able to cope with larger and harder problems. However, owing to the fact that constraints are inherently declarative, attention is quickly turning toward developing high-level programming languages within which such problems can be modeled and also solved. Along these lines, this paper presents DEPICT, the language. Its use is illustrated through modeling a number of benchmark examples. The paper continues with a description of a prototype system within which such models may be interpreted. The paper concludes with a description of a sample run of this interpreter showing how a problem modeled as such is typically solved.

Keywords: Constraint Satisfaction Problems and Languages, Formal Specifications, Typed Predicate Calculus, Language Interpreter.

1 Introduction

The past decade witnessed rapid development of constraint satisfaction technologies[2]. It is a general framework within which problems (CSPs) are formulated and solved [23]. A solution consists of finding appropriate values for problem variables in associated domains so as to satisfy given constraints. Such problems are ubiquitous in a wide variety of scientific and industrial situations. They include (but certainly not restricted to) scheduling, resource allocation, vehicle routing, channel assignment in telecommunication networks, and structure matching in bio molecular databases.

Addressing such problems often requires multi-disciplinary skills [7], such as mathematics, computer science, artificial intelligence, automated reasoning, numerical computing, operations research, as well as database theory and implementation. One is also faced with fundamental difficulties when trying to formulate them and to determine appropriate techniques for their solutions.

Many of these problems[3] can be expressed as *mathematical programs*, and subsequently solved using standard efficient and robust *operations research* (OR) algorithms. However, effective mathematical programming is very difficult even for application domain experts. Moreover, solving such problems often requires amounts of time worse than polynomial in the size of the input data.

This means that efficiency with which solutions are obtained is a real issue. This issue may be tackled through specialized software developed for each individual problem. However, a more productive approach would be to develop a high-level special-purpose language (backed up by state-of-the-art constraint-solving techniques) for modeling (and also solving) such problems.

Examples of state-of-the-art constraint-based languages and systems are ECLiPSe [15] and the ILOG Solver [16] (a C++ based library). ECLiPSe is declarative, in the sense that it allows a natural and intuitive formulation of constraints which relieves the programmer from traditional low-level computing obligations. It also enables the programmer to concentrate more on *what* constraints the solution should satisfy without being overly concerned with the details of *how* these constraints are to be satisfied. This separation of concerns motivated the development of OPL [25] (a front-end to the ILOG Solver) and later on OPL++ [19].

An early language (with a rather OR perspective) that is worth mentioning here is ALICE [18]. There, a rigorous solution can be reached through the analysis of a purely descriptive statement of the problem. Along the same lines, our vision is that the user can describe the problem without any commitment to any particular solution. We project that the solving part can then be completely automated.

2 This Paper

This paper presents DEPICT, a high-level formal language specifically designed for the purpose of modeling constraint satisfaction problems. The paper also describes a prototype system within which such models may be interpreted.

Manuscript received March 9, 2007. This work was supported by Lebanese National Council for Scientific Research.

*Corresponding author. *E-mail address*: abbas@balamand.edu.lb

[2] The progress of the field can perhaps be measured by observing that many of the research projects of the field in the early 90's are now successful commercial enterprises.

[3] The general view of the subject expressed it in this introduction is heavily influenced by the work of Pierre Flener and his research team in the ASTRA [14] project, Uppsala University, Sweden. This is also in line with the general aims of the CSP research group [9], Essex University, England, U.K.

Focus herein is directed primarily toward promoting simplicity and expressiveness. For this reason, this paper resorts to the use of higher-level constructs such as functions, relations, sets and types. At the present time, only a handful of languages (e.g. CHIP [23], CLPS [8], CONJUNCTO [12], AMPL [11] and OZ [21]) are able to formulate arbitrary constraints over sets.

With the existence of the above list of titles, the question that immediately springs to mind is: why another modeling language? We give three reasons for that:

2.1 Formalism

Formalism can bring into play a full range of software engineering tools (e.g. transformation, verification, program synthesis, etc). One particular benefit of formal specifications resides in allowing a comfortable distance of the problem definition from the specific details of the implementation language and also from the large (and sometimes confusing) variety of potential solvers. This is referred to in main stream computer science as *abstraction*. We shall see that the specificity of the CSP definition makes it a particularly attractive target for formal manipulation.

2.2 Expressiveness

Formal methods are usually favored for the clarity, accuracy and the ability to spot inconsistencies in the initial statement of the programming problem. In this context, a formal expression of a problem that has a CSP appearance and that is both clear and robust should be helpful in identifying a constraint-based algorithm for its solution.

2.3 Accessibility

The techniques and tools that are nowadays being utilized in the constraint domain are increasingly becoming out of reach of the average user. This is because of the inherent difficulty of the underlying concepts and the high financial cost involved in trying to make use of them. In this sense, the material used in the description of DEPICT and its associated interpreter is, on both counts, more easily accessible. Abstraction can also help in making constraint technology more widely usable.

3 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is entirely specified by a triplet <A, B, C> defined as follows (definition derived from [24]):

- A: a finite set of variables $\{a_1, a_2, \ldots a_n\}$.
- B: a collection of domains $(B_a)_{a\in A}$, where each B_a is a (normally) finite set of values of arbitrary types associated with the variable a.
- C: a collection of constraints $(C_s)_{s\in S}$, where S is a set of subsets of A and each constraint C_s ties together the variables of the subset s thus restricting the range of values they may take.

3.1 CSP Solving

A CSP solution associates, for each variable $a\in A$, a value $b\in B_a$ satisfying all relevant constraints. In other words, a solution is a set of pairs $\{<a, b>; a\in A \,\&\, b\in B_a\}$ *simultaneously* satisfying all relevant constraints. This simultaneity can be concisely described by a function f associating every $a\in A$ with a value $b\in B_a$ such that C(f) is true. Here, C(f) is succinctly expressed as $(\forall\, s\in S)C_s(f/s)$, where f/s denotes the restriction of the function f over the subset s of A.

This functional view of CSP solutions was first presented in [2] and later on constituted the core theme of a PhD thesis [13]. This formulation considerably contributes to the expressiveness of CSP specifications and also to the development of its associated interpreter.

In Martin-Löf's Theory of Types [1], using the type constructors Π and Σ in conjunction with dependent types, a universal expression for all CSP's is represented by the type expression:

$$\Sigma(\Pi(A, B), C)$$

However, confining ourselves to set theory and typed First-Order Predicate Calculus, if B is taken to be the set $(\cup B_a)_{a\in A}$, the above expression will have the form:

$$(\exists\, f \in A \rightarrow B)\; C(f) \qquad (\mathbf{E})$$

The latter formulation seemingly loses sight of the fact that $f(a)\in B_a$ for all $a\in A$, as stated in the definition of the problem. However, these B_a's are often found together in the same set. But when the need arises, separating the B_a's can be looked at as just another constraint and be made a part of the constraint expression of the problem.

3.2 A Generic Specification Schema

The expression representing a general CSP specification takes the following form:

$$\{D_1, D_2, \ldots, D_m\}\ \mathbf{E}\ (P_1, P_2, \ldots, P_n)^4 \qquad (\Psi)$$

Where (**E**) is the logical expression specifying the problem, $(P_1, P_2, \ldots, P_n)$ are the parameters that (**E**) depends on and $\{D_1, D_2, \ldots, D_m\}$ are declarations (e.g. constant, relation, function and predicate definitions) used to set up the context within which (**E**) can be interpreted.

The distinction between parameters and other kind of declarations allows the formulation of specification schemas. These are generic specifications that can yield many instances of the same specification through suitable instantiations of its parameters.

4 The Specification Language

The basic constructs of this language are determined through identifying the minimum requirements needed for the specification of the triplet <A, B, C>; i.e., the variables A, the domains B and the constraints C [20].

[4] The name `DEPICT` is derived from: "`{Di} E (Pi) In ConstrainTs`"

4.1 The Variables

In this paper, as in most related literature, CSPs are restricted to a finite number of variables each ranging over a finite domain. In this context, a variable can be a name, an identifier and/or a symbol. However, a variable needs not necessarily be referred to explicitly through a unique name. It can instead be referred to implicitly through an index of an array, for instance.

In conclusion, the constructs required here are: enumerated types, sub-ranges, finite sequences of names or symbols, arrays or lists of those.

4.2 The Domains

A value domain is a finite set of items. Thus, the following (or similar) notions are required for the representation and use of such domains:

- Sets defined by enumeration and by comprehension and the usual set operations: membership, subset, equality, intersection, union, difference, power set, Cartesian product, function sets and functions and relations (e.g. ordering relations) over sets
- Primitive types similar to those encountered in conventional programming languages: symbols, constants, Booleans, characters, integers, floats.
- Type operation arithmetic operations, comparison operators, pairing, sub-ranges, arrays and lists of these.

4.3 The Constraints

A constraint is essentially a condition that requires satisfaction. Since this can be directly described by a logical formula, constraints will be expressed using a form of typed predicate calculus expressions, somewhat reminiscent of the specification language of Martin-Löf's Theory of Types [1]. These come with the usual logical connectives and quantifiers. The motivations for selecting this particular language are:

- It is fairly familiar and quite accessible to the average reader.
- Its syntax and semantics are well established and understood.

This will be augmented with a predicate definition facility in order to further amplify expressiveness.

4.4 Other Elements

The above elements are provided for, in one form or the other, in many existing set-based or type-based specification languages, and the account of these requirements presented here is by no means exhaustive. Having said that, this account should not deliberately ignore any crucial feature the CSP definition explicitly requires. In this respect, variable-dependent sets may be needed for representing specification schemas (see above).

5 Remarks on the Nature of Specifications

The next section develops the specification of a few selected problems in DEPICT. However, before doing that, it would perhaps be instructive to add a couple of remarks concerning the nature of specifications and their role in the program development process.

5.1 Specifications versus Implementations

A specification is understood to be a way of expressing *what* is to be done without having to say *how* to do it. As such, the *what* part is expected to have less algorithmic details than its *how* counterpart and therefore simpler to write down. However, the *what* part can actually be more cumbersome to write because it conceptually falls at a higher level of abstraction than its *how* counterpart.

Although we may wish to do without it, the *what* part is better formulated with a degree of awareness of the *how* part. In fact, except in the most ideal of situations, one cannot be totally oblivious of how the problem can be solved. This partial knowledge can only bias one form of specification over another, provided the specification language is sufficiently flexible to offer such choice [5].

5.2 Complexity of Specifications

The specification expressions treated below may look more complex than other equivalent descriptions found elsewhere in the literature, and there is a good justification for that. They are intended to be self-standing self-contained formal expressions of the corresponding problems. Their complexity is largely due to having, in explicit form, details that other formulations tend to keep implicit or completely ignore. Such details are required to be explicit here to make further treatment and analysis possible.

6 Selected[5] Problem Specifications in DEPICT

The methodology followed in developing these specifications parallel what is usually encountered in a knowledge representation lecture of an introductory course in Artificial Intelligence.

All the specifications presented here are developed following the global specification schema (Ψ) presented above. These problems are listed in increasing order of complexity. We mention here that the more complex of these specifications cannot yet be handled by the interpreter whose structure is discussed later on in this paper.

At this stage, the syntax of the language used to express these specifications may seem a little too abstract. However, this should look more concrete once enough details of the associated interpreter are presented.

6.1 The N-Queens Problem

Informal statement: given a strictly positive integer N, find N distinct positions of the Queen piece on an N×N chessboard, so that the Queen at any of those positions cannot take (or be taken from) any of the others.

A first attempt at the specification of the problem relies on the following definitions:

- N : an integer that is a parameter of the problem.
- [1..N]: a range indicating the domain of variables (i.e. N queens).

[5] A good collection of such problems may be found in [26]

- [1..N]×[1..N]: the domain of values (i.e. possible positions) of each variable.

However, a second look at the problem reveals that only the column position of each queen need to be determined, since no two distinct queens can be on the same row. Consequently, the domain of values need only be [1..N]. Moreover, since the relation of one queen taking another is symmetric and transitive, testing the position of each new queen can be restricted to only those that have already been positioned.

In conclusion, the specification requires only the following definitions:

- Q: a function that associates a row i with a column Q(i). This way, the pair <i,Q> will be sufficient to determine the position of a queen on row i.
- TAKES(i,j,Q) specifies when a queen <i,Q> can take another queen <j,Q>.
- TAKEN(i,j,k,Q) specifies when a queen <i,Q> can be taken by another in a row from the range [j..k], where $i \notin [j..k]$.
- SAFE(i,k,Q) specifies that all queens placed on the rows in the range [i..k] are safe from being taken by one another.

Given that "≡" denotes definitional equality, the complete formal specification of the problem becomes:

{TAKES(i,j,Q) $\equiv (Q(i) = Q(j)) \vee$
$(i-Q(i) = j-Q(j)) \vee (i+Q(i) = j+Q(j))$,
TAKEN(i,j,k,Q) $\equiv (\exists\, t \in [j..k])$TAKES(i,t,Q),
SAFE(i,k,Q) $\equiv (\forall\, j \in [i+1..k]) \neg$TAKEN(j-1,j,Q)**}**

($\exists\, Q \in [1..N] \rightarrow [1..N]$**) SAFE(1,N,Q)**

($N \in \tilde{N}$**)**

Here, $\tilde{N}$ is taken to denote the type of all natural numbers.

6.2 The Map Coloring Problem

Informal statement: Given a map containing a set of countries CN and a set of colors CL, associate each country of CN with a color of CL so that no two bordering countries have the same color.

Given the following definitions:

- M: a function that associates each country with the set of countries bordering it.
- BORDERING(c_1,c_2) specifies that c_1 and c_2 have a common border.
- CONFLICTING(c_1,c_2,f) specifies that two countries c_1 and c_2 are bordering each other and have the same color according to a particular coloring f of the map. Here, f is a function that associates every country c with a color f(c).

Hence, the complete formal specification of the problem becomes:

{BORDERING(c_1, c_2) $\equiv (c1 \in M(c_2)) \vee (c_2 \in M(c_1))$,
CONFLICTING(c_1, c_2,f) $\equiv$ BORDERING(c_1, c_2) $\wedge$
$f(c_1) = f(c_2)$,
SAFE(f) $\equiv \forall\, c_1 \in CN\ \forall\, c_2 \in CN\ c_1 \neq c_2 \Rightarrow$
$\neg$CONFLICTING(c_1, c_2,f)**}**

($\exists\, f \in CN \rightarrow CL$**) SAFE(f)**

($N_1 \in \tilde{N}$, $CN \in Đ(N_1)$, $N_2 \in \tilde{N}$, $CL \in Đ(N_2)$,
$M \in CN \rightarrow P(CN)$**)**

Here P(CN) denotes the set of subsets of CN and Đ(N) denotes the universe of finite domains of size N each.

6.3 The Magic-Series Problem

Informal statement: Given a natural number N > 0, find a magic series of length N; i.e., a sequence of numbers S = [k_0, k_1,…, k_{N-1}] so that k_m represents the number of occurrences of m in S.

The function to be constructed here is the sequence S itself. It is a function from the domain [0..N-1] to itself, where N is taken as a parameter to the specification. Thus, the complete formal specification of the problem becomes:

{ }

($\exists\, S \in [0..N-1] \rightarrow [0..N-1]$**)**
($\forall\, m \in [0..N-1]$**) (S(m) = SIZE{i**$\in$**[0..N-1]; S(i) = m})**

($N \in \tilde{N}$**)**

Even though the constraint expression looks simpler, this problem is more complex than the previous ones because of the higher degree of abstraction of the structures involved in its expression.

6.4 The Stable-Marriage Problem

Informal statement: given n men and n women and given also that each individual of the two sexes has an order of preference of the individuals of the opposite sex, the problem is to find a one-to-one relation that associates each man m with one woman w in such a way that, for all other women w', either m prefers w over w' or w' prefers her associated man m' over m.

The degree of difficulty of a specification largely depends on the basic building blocks and tools used to construct this specification. For this reason, we start by describing these:

- M (of men) and W (of women): two sets containing n element each.
- each element x of each of the two sets is associated with a total ordering ($\leq_x$) of the elements of the other set. The relation ($\leq_x$) represents the order of preference that x has of the individuals of the opposite sex.

Accordingly, the formal specification of the problem becomes:

{$\leq \equiv \leq_{WM} \cup \leq_{MW}$ **}**
($\exists\, h \in W \rightarrow M$**)**
$\forall\, w \in W\ \forall\, w' \in W\ (w \neq w') \Rightarrow$
$((w' \leq_{h(w)} w) \vee (h(w) \leq_{w'} h(w'))$
($N \in \tilde{N}$, $M \in Đ(N)$, $W \in Đ(N)$, $\leq_{WM} \in W \rightarrow \hat{O}(M)$,
$\leq_{MW} \in M \rightarrow \hat{O}(W)$**)**

Here, $\hat{O}(S)$ is the type of total ordering over the set S. The difficulty here clearly resides in the higher-order construct $\leq_{WM} \cup \leq_{MW}$. This is taken to mean the ordering relation over the domain M and W depending on whether its arguments are from one domain or the other.

6.5 The Timetabling Problem

This is a more elaborate example than the previous ones (details are in [3]). It addresses a general system that can generate school and university timetables. This is a real-world practical system that has been operational for several years now. This system has entirely been specified using the same paradigm and the same the language proposed in this paper. Once complete, it was surprising how few are the primitives that were actually required for the specification of a substantial task such as that.

7 A Basic Interpreter For DEPICT

Crucial as it may be for expressiveness, formal specification in a higher-level language does not, on its own, solve a given problem. This is especially true for the real-life CSPs that are encountered in practice, as these are usually associated with algorithms that take too long to execute. Their execution time is usually much worse than polynomial in the size of the input data.

7.1 Implementation Approaches

In order to solve such problems, one can translate this specification into an adequate constraint-based programming language. Alternatively, one can associate the problem description with a suitably efficient solving procedure. Accordingly, implementing a modeling language such as DEPICT may be achieved using one of the following two approaches.

The first approach consists of building a suitable translator that hooks the specification language to an existing language or solver [10], with the advantages that building a translator takes less time and the language will benefit from the proven abilities of the corresponding solver.

However, the following couple of points are worth having in mind when the specification language (S) is mapped into a target language (or solver) (T):

- (S) is assumed to be a higher-level language than (T) and, thus, there is something to be gained from programming in (S) rather than in (T).
- For (T) to be a suitable end of the translation process, there must be a one to one mapping between the basic constructs of (S) and those of (T) or, at least, it must be possible to automatically generate implementations in (T) of those constructs of (S) that do not have direct counterparts in (T).

The second approach consists of mapping the constructs of the specification language directly into a suitable solver. This approach will have to exhibit distinguishing features for it to be convincing.

7.2 The DEPICT Interpreter

The interpreter presented in this paper is implemented following the second of the two approaches discussed above.

Given a specification schema (S) and a list of arguments (A) intended to replace the list of parameters (P) of (S), the interpreter returns some or all the solutions of the problem, if any. As explained above, a solution is precisely a list of values from the corresponding domains satisfying all the stated constraints of the problem.

The sequence of transformations undertaken by (S) and (A) is summarized by the following diagram:

Specification Instance → *Substitution* → *Reduction* → *Solving* → *Solutions*

Each step of this sequence will briefly be described below.

1) The Implementation Language

Several benefits can be gained from embedding the interpreter within a symbolic Language such as LISP:

- Any question concerning syntax and semantics as well as concerning the primitive constructs of DEPICT is decided by what LISP has to offer. This is also delimited by the working context of the specification language (see appendix for a summary).
- Constants, variables, functions and other data are defined and evaluated in LISP. The specification formula will be managed in the context of the accompanying list of predicate definitions.

2) Formal Logic and Substitution

Given the choice of the particular implementation language and also the choice of typed predicate calculus as a specification language, substitution of parameters, variables, function and predicate calls within the main specification expression is done symbolically.

Moreover, quantified logical formulae are substituted as follows: (Exists (x in D)(P x)) is substituted by (or (P d_1)(P d_2)…(P d_n)), and (Forall (x in D)(P x)) is substituted by (and (P d_1)(P d_2)…(P d_n)), where D = (d_1 d_2 … d_n). Clearly, this scheme is possible only because all domains are assumed to be finite. As an exception, the main function f of the specification will be left as is in the main specification expression.

3) Reduction to Clausal Form

Substitution turns the specification into a single logical expression (E) containing no variables except the constraint variables coming from A. We shall be interested in instances of those variables that exists under the form (f a). Each (f a) represents the yet unknown value from the domain B_a to be associated with the variable a.

Moreover, the only predicate or function calls that remain in this expression should be predefined in LISP. This way, when (E) is reduced to *clausal form*, it will look like: (and C_1 C_2 … C_m), where C_i is a *clause* of the form (or D_1 D_2 … D_n) and D_i is a call to a primitive LISP predicate or the negation of a call to a primitive LISP predicate. Again, no unknowns are left in D_i except those of the form (f a).

This reduction phase should be reminiscent of resolution-based theorem proving in formal logic, and therefore of the PROLOG programming language. However, compared to that, the substitution phase described above obviates the need for the unification algorithm.

4) Solving: constructing the function f

Each unknown (f a) is automatically associated with values from B_a. This association is used to decide which of those values to keep and which ones to reject, depending on the truth value of the primitive predicates D_i. This way, running through the clauses (or D_1 D_2 … D_n) along the constraint store (and C_1 C_2 … C_m) will complete the construction of the function f, which will be a solution of the problem.

Note that not all the constraint variables need to have associated values for all the constraints to be satisfied. Accordingly, the function f might turn out to be a partial function, depending on the specification being interpreted.

A sample run of the interpreter is briefly described in the appendix.

8 Conclusions and Further Work

The basic results reported in this paper can be summarized as follows:

- An elegant formal language specific for modeling constraint satisfaction problems.
- Elegance comes through adopting a so-called functional view of a CSP solution. This has the benefit of unifying the representation of the constraint variables under the umbrella of a single function name.
- Elegance also comes through adopting formalization and symbolism using typed first order predicate calculus over finite types.

8.1 DEPICT vs. Other Similar Languages

Comparing and contrasting DEPICT with other existing similar languages in this domain has already been done in [2] and [3]. We provide here a brief summary.

The first language that comes to mind is OPL [25], which is a front-end to ILOG Solver [16]; a very powerful, commercial constraint solver. While OPL is a quite rich language designed for engineers, DEPICT is designed for mathematicians and logicians who would be more at home with DEPICT's style.

In the same context, ESRA [14] is a constraint modeling language. Hnich [13] extended ESRA and introduced function variables for constraint programming. These are both expressive. By contrast, building on a formal logic, DEPICT is designed with program verification as future development in mind.

8.2 Extensibility

There are two possible meanings of this term:

First, DEPICT is a specification language. Therefore, scalability could mean how complex the language is, and whether it can be used to specify large problems. In this context, the existence of quantifiers helps DEPICT to express complex problems in a more compact way, see examples 6.3 and 6.4 below. Without the existence of the higher-order constructs, one would be at loss of how to formally specify the problems, indeed.

Second, in as far as the scalability in the DEPICT interpreter is concerned, DEPICT does not reduce the complexity of a problem. Constraint satisfaction problems are NP-hard in nature. Some formulations may be easier to solve than others by certain heuristics [22, 24], but the complexity of the problem does not change. What can be sure is that the expressiveness of DEPCIT does not hinder problem formulation.

8.3 Further Work

The features of DEPICT enabled the development of a compact interpreter, which leaves plenty of room for embodying many of the known features of CSP solving. Correspondingly, plenty of work remains to render practical a theoretically transparent framework:

1) At the level of the specification language: there is work to be done in implementing more types in the language (relations, sets and functions) to enable it to express more complex types. We are at the moment toying with initial ideas for implementing set-based constraint expressions and also higher-order expressions of constraints involving functions and relations.
2) At the level of the interpreter: there is work to be done to make it more time and space efficient. This can follow from reducing the size of the constraint store and also from reducing the sizes of the value domains through implementing the equivalent of problem reduction and constraint propagation mechanism of traditional constraint-based solvers.
3) The uniformity with which the constraint store is represented should have some role to play within the overall automation of the process. This should have some impact on the techniques that are traditionally used to boost efficiency: e.g. parallelism.
4) The uniformity of the functional view of constraint solving will bring with it plenty of supporting (and well understood tools). These should have some role to play in the variety of complex situations that can arise in practice (e.g. default and redundant constraints).
5) This uniformity often results in algorithms that are not adequately efficient. But since efficiency is a major concern and since heuristic information and optimization techniques is a universally agreed way for boosting efficiency, it is imperative to reserve a role of that in our specification language (see [4], [6] and [17]).

We conclude the paper with a technical note. Since the current interpreter can return all possible solutions of the problem, one direction that is worth following is to add to the specification an optimization function then direct the interpreter to return the best one of the solutions accordingly.

Acknowledgment

The authors are deeply indebted to Pierre Flener for his elaborate comments and for the references that helped in filling background gaps that existed in an initial version of this paper. The idea of providing solver-independent specifications originates from Dr Carmen Gervet.

References

[1] A. Abbas, Programming With Types and Rules In Martin-Löf's Theory Of Types, *Ph.D. Thesis*, Queen Mary College, University Of London, U.K., 1987.

[2] A. Abbas and E. Tsang, Toward a General Language for the Specification of Constraint satisfaction Problems, *Proceedings of CP-AI-OR 2001 workshop*, Imperial College, London, England, April 2001.

[3] A. Abbas & E. P. K. Tsang, Software Engineering aspects of constraint-based timetabling – a case study, Information & Software Technology Journal, Vol. 46, 2004, 359-372

[4] J. E. Borrett, Formulation selection for constraint satisfaction problems: a heuristic approach, *PhD Thesis*, Department of Computer Science, University of Essex, Colchester, UK, 1998

[5] J. E. Borrett and E.P.K. Tsang, A context for constraint satisfaction problems formulation selection, *Constraints*, Kluwer Academic Publishers, Vol.6, No.4, 2001, 299-327
[6] P. Dasgupta, P. P. Chakrabarti, A. Dey, S. Ghose and W. Bibel, Solving Constraint Optimization Problems from CLP-Style Specifications Using Heuristic Search Techniques, *IEEE Transactions on Knowledge and Data Engineering archive*, Vol.14, Issue 2, 353-368, 2002.
[7] R.Detcher, Constraint Processing, *Morgan Kaufmann Publishers*, 2003.
[8] A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi, A Language For Programming In Logic With Finite Sets, *Journal of Logic Programming*,1996.
[9] cswww.essex.ac.uk/csp
[10] R. Fourer, Hooking a Constraint Programming Solver to an Algebraic Modeling Language, *CP-AI-OR'01*, Wye College, Kent, U.K., 2001.
[11] R. Fourer, D. M. Gay and B. W. Kernighan, , AMPL: A Modeling Language for Mathematical Programming *Duxbury Press / Brooks/ Cole Publishing Company*, 2002
[12] C. Gervet, Conjuncto: Constraint Logic Programming with Finite Set Domains. In *ILPS'94*, November 1994.
[13] B. Hnich, Function Variables for Constraint Programming. *PhD Thesis*, University of Uppsala, Sweden 2003.
[14] www.it.uu.se/research/group/astra
[15] www.icparc.ic.ac.uk/eclipse
[16] www.ilog.com
[17] F. Laburthe and Y. Caseau, SALSA: A Language for Search Algorithms, *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes In Computer Science, Vol. 1520, 310-324, 1998.
[18] J. L. Lauriere, ALICE: A Language and a Program for Solving Combinatorial Problems, *Artificial Intelligence*, Vol. 10, 1978, pp 29 - 127.
[19] L. Michel and P. Van Hentenryck, OPL++ A Modeling Layer for Constraint Programming Libraries, *CP-AI-OR'01*, Wye College, Kent, U.K., 2001.
[20] P. Mills, et al., *EaCL 1.5:* An Easy Abstract Constraint Optimization Programming Language, *Technical Report CSM-324*, Department of Computer Science, University of Essex, U.K., 2000..
[21] www.ps.uni-sb.de/oz2
[22] F. Rossi, P. van Beek & T. Walsh (ed.), Handbook of Constraint Programming, 2006, Elsevier pubs.
[23] H. Simonis, The CHIP system and its applications, in Montanari, U. and Rossi, F. (ed.), *Proceedings, Principles and Practice of Constraint Programming (CP'95), Lecture Notes in Computer Science*, Springer Verlag, Berlin, Heidelberg & New York, 1995, 643-646
[24] E. P. K. Tsang, Foundations of Constraint Satisfaction, *Academic Press*, 1993.
[25] P. Van Hentenryck, The OPL Optimization Programming Language, *The MIT Press*, 1999.
[26] T. Walsh, World Wide Web, *CSPlib home page*.

Appendix

A. Syntax and Semantics of DEPICT: a brief summary

As it currently stands, DEPICT specification formulation is fundamentally based on typed predicate calculus over finite types. Furthermore, since the interpreter is implemented in LISP, it was natural to import all the syntactical mode of expression from there.

As a result, all questions on the semantics of the constructs used in DEPICT should be answered from the semantics of the two languages referred to above, which is both widely accessible on both counts

Furthermore, the syntax of depict is best summarized through a specification schema as follows:

(;; *predicate declarations, each possessing the following form*
(Define P (<parameter list>)
< typed predicate formula written according to LISP syntax>)

;; main *specification expression*
(Exists (F (Function (Variables …) (Domains ...)))
(< typed predicate formula written according to LISP syntax>)
))
;; *parameters list: the parameters are to be used as global*
;; variables anywhere in the predicate definitions or
;; the main specification expression
;; these parameters will be suitably instantiated before the
;; the schema is being interpreted

;; predefined or user-defined LISP predicates and functions
;; can be used anywhere in this schema.
;; In particular, Variables *and* Domains *are user-defined LISP*
;; function written to respectively return the list of variables
;; and the corresponding list of domains of the problem
;; expressions

;; the particular use of this schema is illustrated in sections
;; B and C of this appendix

B. Specifying the N-Queens Problem

(;; *local declarations*
(Define Takes (i j Q)
(or (= (Q i) (Q j))
(= (- i (Q i)) (- j (Q j))) (= (+ i (Q i)) (+ j (Q j)))))
(Define Taken (i j k Q)
(Exists (t (Interval j k)) (Takes i t Q)))
(Define Safe (i k Q)
(ForAll (j (Interval (+ i 1) k)) (not (Taken (- j 1) j k Q))))
;; *specification*
(Exists (Q (Function (Variables n) (Domains n))) (Safe 1 n Q))
;; *parameters*
((n N))
)
;; (Interval m n) returns the list (m m+1 m+2 … n) ,
assuming m ≤ n.
;; (Variables n) returns (Interval 1 n)
;; (Domains n) returns a list containing n instances
of the list (1 2 … n)

C. Interpreting the N-Queens Problem

We will now briefly describe a trace of the interpretation process of the above specification schema. This trace assumes that the associated parameter n is equal to 4. In fact, given that **Q∈[1..N] → [1..N]**:

;; *starting point*
1-**SAFE(1, 4, Q)**
;; *substitution*
2. (ForAll j∈[2..4]) (Not TAKEN(j-1, j, 4, Q))
;; *substitution*
3. (And (Not TAKEN(1,2,4,Q)) (Not TAKEN(2,3,4,Q)) (Not TAKEN(3,4,4,Q)))
;; *substitution*
4. (And (Not (Exists t ∈ [2..4]) TAKES(1,t,Q))
(Not (Exists t ∈ [3..4]) TAKES(2,t,Q))
(Not (Exists t ∈ [4..4]) TAKES(3,t,Q)))
;; *substitution*

5. (And (Not TAKES(1,2,Q)) (Not TAKES(1,3,Q))
(Not TAKES(1,4,Q)) (Not TAKES(2,3,Q))
(Not TAKES(2,4,Q)) (Not TAKES(3,4,Q)))

;; *reduction*

6. (And (Q(1)≠Q(2)) (1-Q(1) ≠ 2-Q(2)) (1+Q(1) ≠ 2+Q(2))
(Q(1)≠Q(3)) (1-Q(1) ≠ 3-Q(3)) (1+Q(1) ≠ 3+Q(3))
(Q(1)≠Q(4)) (1-Q(1) ≠ 4-Q(4)) (1+Q(1) ≠ 4+Q(4))
(Q(2)≠Q(3)) (2-Q(2) ≠ 3-Q(3)) (2+Q(2) ≠ 3+Q(3))
(Q(2)≠Q(4)) (2-Q(2) ≠ 4-Q(4)) (2+Q(2) ≠ 4+Q(4))
(Q(3)≠Q(4)) (3-Q(3) ≠ 4-Q(4)) (3+Q(3) ≠ 4+Q(4)))

;; *solving*

7. Q(1) ≠ Q(2) gives the solutions
<Q(1), Q(2)> = {<1,2>,<1,3>,<1,4>, <2,1>,<2,3>,<2,4>,
<3,1>,<3,2>,<3,4>, <4,1>,<4,2>,<4,3>}
Same for Q(1) ≠ Q(3), Q(1) ≠ Q(4), Q(2) ≠ Q(3), Q(2) ≠ Q(4) and Q(3) ≠ Q(4)

;; *solving*

8. 1-Q(1) ≠ 2-Q(2) gives the solutions
<Q(1), Q(2)> = {<1,1>,<1,3>,<1,4>,<2,1>,<2,2>,<2,4>,
<3,1>,<3,2>,<3,3>,
<4,1>,<4,2>,<4,3>,<4,4>}
Same for 1-Q(1) ≠ 3-Q(3), 1-Q(1) ≠ 4-Q(4), 2-Q(2) ≠ 3-Q(3), 2-Q(2) ≠ 4-Q(4) and 3-Q(3) ≠ 4-Q(4)

;; *solving*

9. (1+Q(1) ≠ 2+Q(2)) gives the solutions
<Q(1), Q(2)> = {<1,1>,<1,2>,<1,3>,<1,4>,<2,2>,<2,3>,
<2,4>,<3,1>,<3,3>,<3,4>,<4,1>,<4,2>,
<4,4>}
Same for 1+Q(1) ≠ 3+Q(3), 1+Q(1) ≠ 4+Q(4), 2+Q(2) ≠ 3+Q(3), 2+Q(2) ≠ 4+Q(4) and 3+Q(3) ≠ 4+Q(4)

;; *solutions*

10. The only compatible solutions that are left for <Q(1), Q(2), Q(3), Q(4)> are {<2,4,1,3>, <3,1,4,2>}

Abdulwahed M. Abbas is an Associate Professor in the Department of Computer Science in the University of Balamand (Tripoli, Lebanon). He took his BSc Degree in Pure Mathematics from the Lebanese University in Beirut in 1979, an MSc Degree in Computer Studies from the University of Essex (England) in 1982 and a PhD in Computer Science from Queen Mary College (London) in 1987. He spent six years teaching Computer Science at the University of Queensland (Australia). He helped founding the Department of Computer science at the University of Balamand, where he currently is. His major research interests are: Computer Aided Geometric Design (CAGD) and Constraint Satisfaction Problems (CSP) and Languages.

Edward P. K. Tsang is a Professor in Computer Science at University of Essex. He is also the Deputy Director of Centre for Computational Finance and Economic Agents. Edward Tsang has broad interest in artificial intelligence, including heuristic search, computational finance and economics, constraint satisfaction, combinatorial optimization, scheduling and evolutionary computation.

Ahmad H. Nasri is a professor in computer graphics and chair of the computer science department, American University of Beirut, Lebanon. He received a BS degree in Mathematics from the Lebanese University (Lebanon) in 1978, a qualifying degree in Computer Science from Essex University, U.K., in 1982, and a PhD degree in Computer Graphics from the University of East Anglia, U.K., in 1985. He was a research visitor at MIT, Arizona State University, Purdue University, Brigham Young University, Seoul National University (Korea), Cambridge University (UK), City Hong Kong University, and Bremen University (Germany). He is a member of the Board of directors of the Lebanese National Council for Scientific Research. He is also on the editorial board of the International Journal of CAD/CAM, the Journal of Shape Modeling, the Computer-Aided Design and Applications Journal, and the Lebanese Scientific Journal. He has served on the PC committee of several international conferences. With Malcolm Sabin he co-edited a special issue of the Journal of Visual Computer on Subdivision surfaces, 2002. Since 1982 he has been involved in promoting subdivision surfaces and its use in computer graphics, geometric modeling and animation. His research interests include Recursive Subdivision in Animation, Computer Graphics, Geometric Modeling, Data Visualization, and the use of Computer Graphics in Education and digital arts.