Proceedings, Constraint Programming, Artificial Intelligence and Operations Research (CP-AI-OR) Workshop, London, UK, April 2001

# TOWARD A GENERAL LANGUAGE FOR THE SPECIFICATION OF CONSTRAINT SATISFACTION PROBLEMS

Abdulwahed. Abbas
Computer Science Department,
The University Of Balamand, P. O. Box 100, Tripoli, Lebanon
Email: abbas@balamand.edu.lb

Edward Tsang
Department of Computer Science,
University of Essex, Colchester
CO4 3SQ UK
Email: Edward@essex.ac.uk

**Abstract**: The past decade saw rapid progress toward prominence of constraint satisfaction technology. Many practical algorithms have been developed to solve larger and larger problems. The degree of maturity in this technology begs the support of well-established software engineering tools. This paper targets a formal specification language DEPICT 0.1 that can aid in the formulation of constraint satisfaction problems and the identification of suitable algorithms for their solutions. This paper outlines, through a few selected examples, suitable characteristics that DEPICT 0.1 should possess and concludes with some of the benefits that can arise from that.

**Keywords**: Constraint Satisfaction Problems and Languages, Formal Specifications, Type Theory, First-Order Predicate Calculus, the Z Specification Language.

## 1. BACKGROUND

*Constraint satisfaction problem*s (CSPs) - where appropriate values for problem variables are to be found, subject to given constraints - are ubiquitous in a wide variety of industrial and scientific situations. This includes many difficult and combinatorial applications such as scheduling, resource allocation, vehicle routing, channel assignment in telecommunication networks, and structure matching in bio molecular databases.

Addressing these kinds of problems often requires multi-disciplinary skills that includes, beside the specific knowledge of the domains indicated above, aspects of mathematics, traditional computer science, artificial intelligence, operations research, numerical computing, automated reasoning, as well as database theory and implementation.

Solving these kinds of problems requires the solution of fundamental research problems such as understanding how to formulate them and how to select appropriate problem solving techniques that can solve them efficiently and how to adapt or combine such techniques for variations of the same problem.

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

One may develop specialized software to solve each individual problem. However, a more productive alternative is to develop general-purpose constraint-programming languages to allow users to express their problems and use state-of-the-art constraint-solving techniques developed in support of these languages.

Examples of successful constraint-programming languages include CHIP[16, 21], ILOG Solver [15] and ECLiPSe[10]. CHIP and ECLiPSe are declarative, in the sense that they allow a natural and intuitive formulation of constraints, thus relieving the programmer from traditional low-level computing obligations. Thus, the programmer can specify *what* constraints the solution should satisfy without having to detail *how* this solution is to be reached.

This separation of concerns between the *what* and the *how* parts has considerable advantages, as it allows the programmer to concentrate on one without having a direct impact over the other. For instance, the programmer can fine-tune his description of what the program is to accomplish without worrying too much about which particular problem solving strategy is going to be followed. Furthermore, once this description is complete, the programmer can experiment through comparison with any number of problem-solving strategies to identify the best solution for that particular problem.

This separation of concerns motivated the development of OPL[22], a front-end to ILOG Solver (a C++ based library). It is also a major motivation behind the ESRA[7] language and likewise on the minds of the designers of the EaCL[13] language.

## 2. THIS PAPER

The research reported in this paper points essentially in the same general direction. It presents the preliminary structure of a formal specification language DEPICT 0.1 for CSPs. Focus herein is mainly on the *what* part, targeting mostly expressiveness. In order to achieve that, this paper resorts to the use of higher-level constructs such as functions, relations, sets and types. At the moment, there are just a few languages (e.g. CLPS[3], CONJUNCTO[8] and OZ[14]) that can formulate arbitrary constraints over sets, which is a source of enormous expressiveness.

This paper concludes with a few insights on how to link the *how* to the *what* part. In fact, our vision is that the two parts can eventually be completely separated; that is, the user will be able to describe the problem without any commitment whatsoever to any particular solving strategy. *Solver independency* is in fact a prime motivation behind the research reported in this paper, given a sufficiently expressive language and a rich variety of solvers the system can choose from, as it is the case in the CSP domain.

An early example to go into this direction (with a rather OR perspective) is the language ALICE [11], where the system is able to arrive at a rigorous solution through the analysis of a purely descriptive statement of the problem.

It is essential to note that the above is not an attempt to downplay the importance of efficiency vis-à-vis expressiveness. But so long as we are not talking about real-time applications and so long as the problem solving techniques have reasonable performance, expressiveness can only come first, as it has a greater part to play in the overall usability of the programming system.

On another note, DEPICT 0.1 (as it is currently proposed) targets mostly CSPs. The issue of whether this language can be of use in related domains (such as OR or even stochastic aspects of such problems) is open at this stage, although many of these problems can be (and have been) formulated as CSPs. Having said that, we are by no means advocating that all such problems be solved this way. However, we hope that, in the future, specifying a problem as a CSP, when it is appropriate to do so, will allow the user to reap many of the associated benefits (see last section of this paper).

## 3. DEFINING CONSTRAINT SATISFACTION PROBLEMS

A constraint satisfaction problem (CSP) is entirely specified by the triplet <A, B, C> whose components are defined as follows (definition derived from [20]):

- A: a finite set of variables $\{a_1, a_2, \ldots, a_n\}$.
- B: a collection $(B_a)_{a \in A}$, 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 values they may *simultaneously* take.

The specificity of this definition makes CSP an appealing target for the application of formal methods, and particularly a formal specification language, as this can bring into its development a full range of well-established software engineering tools, with all the benefits that arise from that (see below).

### 3.1. Why Formal Specifications?

An initial benefit of formal specifications resides in allowing a comfortable distance of the problem definition not only from the specific details of the implementation language, but also from the large variety of candidate algorithms that could possibly be used for its solution. In the traditional Software Engineering terminology, this is usually referred to as *abstraction*.

Since constraint satisfaction technology already has a wide range of applications that, in many instances, cut across disciplines, abstracting away from implementation details has an additional benefit of making constraint technology more widely accessible.

Moreover, in addition to the usual arguments made in favor of formal methods, such as clarity, accuracy and the ability to spot inconsistencies in the initial statement of the programming problem, the following remark must also be recorded. Giving a problem a CSP identity is the very first step toward determining a constraint-based algorithm for its solution. A formal expression of this problem that has a CSP appearance is surely a direct way for getting to this conclusion.

Beside *solver independency*, more arguments specific to the CSP domain can also be made in favor of formal specifications (see last section of this paper).

### 3.2. Why Another Formal Specification Language?

During the past two to three decades, research in formal methods has resulted in a wide variety of specification languages (e.g. First-Order Predicate Calculus, the Z specification language and Type Theory [1], to name but a few). Some of these have proven their usefulness to the general software engineering community in the areas of program specification, verification and synthesis. Some have also demonstrated their trustworthiness in practically oriented and safety-critical industrial-strength projects.

Given this heritage, the first question that springs to mind is why another formal specification language? A possible answer to that resides in observing that while these languages are powerful enough to deal with general software engineering projects, their generality may constitute a distraction in the CSP context. In fact, what is being addressed here is a specific class of problems with distinct formulation and requirements. Accordingly, common wisdom suggests the use of a specification language that is tailored to fit the specific needs that may arise there. In fact, this should be an enormous service done in favor of the goal of synthesizing CSP solutions from problem specifications (also see section 5).

## 4. REQUIRED FEATURES OF THE SPECIFICATION LANGUAGE

With the literature accumulating from research in formal methods (see previous section), one does not have to go very far to design a specialized language powerful enough for dealing with the CSP specification requirements. The main attributes such a language should be able to possess are:

- Declarative: e.g. logic based, to allow the symbolic description of the programming tasks.
- Abstract: to allow a comfortable distance from implementation details and solution strategies.
- Intuitive: with constructs that can directly map the concepts of the application domains.
- Expressive: to be able deal with real-world applications.

The best place to start identifying the special requirements for this language is the CSP definition itself, as given in the second section. In fact, it is enough that the language can deal with the specification of the triplet <A, B, C>; i.e., the variable

domain A, the value domains B and the constraints C. This is the line of thinking taken in [13], where the user is guided through defining the variables, domains and constraints.

### 4.1. Variable Domain Requirements

In this paper, we limit ourselves to CSPs with finite domains. A CSP specification requires variable names, identifier*s* and/or symbols. It should perhaps be mentioned here that a variable needs not necessarily be referred to explicitly through a unique name, as it may be referred to implicitly instead; through an index of an array, for instance (see examples below).

In conclusion, the notions required here seem to be enumerated types, finite sequences of names or symbols, arrays or lists of those, etc.

### 4.2. Value Domains Requirements

A value domain is a finite set of items not necessarily of the same type. Now, to be able to represent and use such domains, the following (or similar) notions are required:

- Primitive types similar to those found in conventional programming languages; e.g. arbitrary symbols or constants, Boolean, character, integer, real, etc.
- Operations over elements of these types; e.g. arithmetic operations, comparison operators, pairing, sub-ranges, arrays and lists of these, etc.
- Sets defined by enumeration as well as by comprehension.
- Functions and relations (e.g. ordering relations) over sets.
- The usual set operations; e.g. membership, subset and set equality tests, intersection, union, set difference, power set, Cartesian product, function sets, etc.

### 4.3. Constraints Requirements

A constraint is essentially a condition that may require satisfaction. As such, this is most naturally expressed as a logical formula. Thus, constraints will be specified using some 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 that are found in such contexts.

Furthermore, this notation will be augmented with a predicate definition facility in order to make the presentation more modular, and hence amplify its expressive power.

### 4.4. Further Requirements

The above requirements are of course 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. For instance, that definition seems to mention variable-dependent sets, which are very much like dependent types in Martin-Löf's Theory of Types. In order to support this feature here, programming constructs, such as conditionals and loops, will be required to return set or type values.

Furthermore, and in order to add to its power of expression, this specification language should be able to represent *generic* specifications or specification schemas. These are specifications dependent on one or more parameters of any type. This will enable the language to specify algorithm schemas, which may be used to solve variations of the same problem by simple instantiations of its parameters.

We close this section with a worthy note. Formal specification and derivation techniques usually imply uniform treatment of diverse classes of problems. As is well known, this uniformity often results in algorithms that are not adequately efficient. But since efficiency is of utmost importance for these algorithms and since heuristic information and optimization techniques is the way to boost efficiency, it is imperative to reserve a place and a role of that in our specification language (see [13]). However, this paper will not address this requirement here.

## 5. SPECIFYING CONSTRAINT SATISFACTION PROBLEMS

A CSP solution consists of finding, for each of the variables $a \in A$, a value $b \in B_a$ satisfying all relevant constraints. That is, the set of pairs $\{<a, b>;\ a \in s\}$ must satisfy the constraint $C_s$, for each $s \in S$, where $S$ is a set of subsets of $A$. Thus, an initial attempt at specifying this problem (see [12] for a related formulation) could be:

$$(\forall\ a \in A)(\exists\ b \in B_a)C(<a, b>)$$

However, this is not sufficiently accurate since $C$ is a condition that *simultaneously* applies to all pairs $<a,b> \in A \times B_a$. A more accurate description of this simultaneity would be to require the construction of a function $f$ associating every element $a \in A$ with an element $b \in B_a$ in such a way that $C(f)$ is satisfied.

Following that, if $f/s$ is taken to be the restriction of $f$ over the subset $s$ of $A$, then $C(f)$ can be specified by:

$$(\forall\ s \in S)\ C_s(f/s)$$

In the specification language of Martin-Löf's Theory of Types [1], having dependent types and the type constructors $\Pi$ and $\Sigma$, the above specification manifests itself with surprising elegance as the type expression:

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

However, restricting ourselves to set theory and typed First-Order Predicate Calculus as a specification language, if $B$ is taken as the set $(\cup B_a)_{a \in A}$, then this specification can still be written as:

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

Note that the latter formulation seemingly loses sight of the fact that $f(a) \in B_a$ for all $a \in A$, as the initial definition of the problem suggests. At any rate, in the problems treated below and in many problems that can arise in practice, these $B_a$ will often be found bunched together in one set anyway. However, if the need arises (for the sake of efficiency, for example), separating these $B_a$'s can always be considered as a constraint, and as such this can be made a part of the constraint expression $C$ itself.

The fact, that an array is nothing more than a function from a finite ordered set of indices to a domain of values, can be used to support the use of functions here. In fact, in addition to contributing to the conciseness of the specification expression (**E**), and since $A$ is a finite set, a function here is no more than an array whose indices are variables from $A$ and whose elements are from the corresponding domains of values.

### 5.1. A Global Specification Schema

With the introduction of section **4** in mind, the expression representing a general CSP specification will take the following form:

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

Where (**E**) is the logical expression specifying the problem and $\{D_1, D_2, \ldots, D_m\}$ are the declarations (i.e. relation, function and predicate definitions, etc) that are used in (**E**) and $(P_1, P_2, \ldots, P_n)$ are the parameters that (**E**) depends on. Note that the distinction and the separation, between parameters and other kind of declarations is for the sake of formulating specification schemas. As we said above, this is a generic specification whose role is to allow for many instances of the specification through suitable instantiations of its parameters.

## 6. REMARKS ON THE NATURE OF SPECIFICATIONS

Following the above-suggested paradigm, the next section develops the specification of a few selected problems in our proposed language DEPICT 0.1 (which stands for "$\{D_i\}$ **E** $(P_i)$ In ConstrainTs"). We hope that some of the benefits of this language will already be apparent through these examples. However, before we do that, it would perhaps be instructive to add a couple of remarks about the nature of specifications and their role in the program development process.

### 6.1. The 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. In fact, these are intended to be self-standing self-contained formal expressions of the corresponding problems. As such, their complexity is

largely due to having, in explicit form, details (such as typing information, for example) that other formulations tend to keep implicit or completely ignore. Yet, these details are required in explicit form here in order to make possible further treatment and analysis. After all, this is the *raison d'être* of these expressions.

### 6.2. Specifications *vs* Implementations

As is usually defined, a specification is a way of expressing *what* is to be done without having to spell out *how* to do it. The *what* part has more or less been ignored in the past because it has less details than the *how* counterpart. Consequently, this is expected to be simpler to write down. However, this being the case relies on a couple of observations: First of all, it would depend on the expressiveness of the specification language. In fact, what may make the *what* part more cumbersome to formulate and express in general, is that it conceptually falls at a higher level of abstraction than the *how* counterpart.

Secondly, it would depend on the degree of awareness the person doing the specification has about how the problem can be solved. In fact, except in the most ideal of situations, this person cannot be totally oblivious as to how the problem is to be solved. Now since, except in the most trivial of specification languages, there is almost always more than one alternative in which to express the *what* part, the preference of one alternative over another cannot be totally separated from the degree of knowledge of how the problem is to be solved. This last observation is clearly reflected in two alternative specifications of the N-Queens problem (see below). The impact on the algorithm of the quality of a specification is discussed in Borrett and Tsang [4, 5].

## 7. SAMPLE PROBLEMS SPECIFICATIONS

The complete formal expressions corresponding to each of the problems treated below is listed in the appendix. All these expressions are written in a form that respects the global specification schema ($\Psi$) presented in section **5**.

### 7.1. The Sorting Problem

This problem has been specified many times over in as many different formal languages. Nevertheless, it will still be interesting to see its formulation in our notation when seen as a CSP.

The formal specification necessarily starts from an informal definition of the problem, which goes as follows:

> Given is a domain D of values together with an ordering relation (≤) over its elements. Given also is an array A of N element of D. The task is to find a rearrangement of the elements of A that are in ascending order with respect to (≤).

The above informal definition can be formulated as a CSP. In fact, ordering can be seen as re-indexing the array elements; that is, the variables of the problem (and the corresponding values) are the indices of the array and the constraint is just the ordering condition that the re-indexed array should satisfy.

This way, the ordering task can be formally specified as follows:

$$(\exists\ f \in [1..N] \rightarrow [1..N])\ \mathrm{ORDERED}(f)$$

Note here the tidy way of encoding a permutation enabled through the use of the function f. Here ORDERED(f) is simply defined by:

$$\mathrm{ORDERED}(f) \equiv \mathrm{BIJECTIVE}(f) \wedge (\forall\ i \in [1..N-1])(A(f(i)) \leq A(f(i+1)))$$

And where BIJECTIVE(f) is used to insure that no element of the initial array is ignored from the sorting process. One definition of that could be:

$$\mathrm{BIJECTIVE}(f) \equiv (\forall\ i, j \in [1..N])\ (i \neq j \Rightarrow f(i) \neq f(j))$$

And another possibility is:

$$\mathrm{BIJECTIVE}(f) \equiv (\mathrm{SIZE}(\{f(i);\ i \in [1..N]\}) = N)$$

Note that the choice of any one these alternatives can have an impact on the problem solving strategy that can be associated with the specification, as. In fact, one way or the other, any such strategy has to deal with the predicates used to specify the constraints. This can perhaps be seen more clearly in the next specification.

### 7.2. The Map Colouring Problem

The informal definition of the problem goes as follows:

> Given a map containing a set of countries CN and a set of colours CL, associate each country of CN with a colour from CL so that no two bordering countries have the same colour.

The map can be represented by a function M that associates each element country $c \in CN$ with the set of countries bordering c: $M(c) \subseteq CN$. Additionally, the bordering relationship between two countries is simply defined as:

$$\mathrm{bordering}(c_1, c_2) \equiv (c1 \in M(c_2)) \vee (c_2 \in M(c_1))$$

Now, if f is taken to be the function that associates each country with a colour, then the constraint on f can simply be specified by the following condition:

$$\forall\ c_1 \in CN\ \forall c_2 \in CN\ \mathrm{bordering}(c_1, c_2) \Rightarrow f(c_1) \neq f(c_2)$$

### 7.3. The N-Queens Problem

The informal definition of the problem goes as follows:

> Given a strictly positive integer N, find N distinct positions of the Queen piece on an N×N chessboard, so that no Queen at any of those positions can take (or be taken from) any of the others.

**A First Specification Of The N-Queens Problem**

Starting directly from the above definition, without further analysis, the specification of the problem will rely on the following definitions:

- N : an integer that is a global parameter of the problem.
- [1..N]: a range indicating the domain of variables (i.e. N queens).
- [1..N]×[1..N]: the domain of values of each one of the above variables. That is, a value that a variable can have is a pair <i,j> from this domain.
- TAKES($Q_1$,$Q_2$) tells when a queen $Q_1$ can take another queen $Q_2$. That is, when $Q_1$ and $Q_2$ are in the same row, column or diagonal. Formally:

$$\begin{aligned}&(1^{st}(Q_1) = 1^{st}(Q_2))\\ &\vee\ (2^{nd}(Q_1) = 2^{nd}(Q_2))\\ &\vee\ (1^{st}(Q_1) - 2^{nd}(Q_1) = 1^{st}(Q_2) - 2^{nd}(Q_2))\\ &\vee\ (1^{st}(Q_1) + 2^{nd}(Q_1) = 1^{st}(Q_2)) + 2^{nd}(Q_2))\end{aligned}$$

- TAKEN(Q, S) tells when a queen Q is taken by at least one queen of the set S of queens. That is:

$$(\exists\ Q_1 \in S)\ \mathrm{TAKES}(Q_1, Q)$$

- SAFE(S) tells when no queen element of S can take any of the other queens in S. That is,

$$(\forall\ Q \in S)\ \neg\mathrm{TAKEN}(Q, S - \{Q\})$$

In the above terms, the specification will be:

$$(\exists\ S \subseteq [1..N]\times[1..N])\ ((\mathrm{SIZE}(S) = N) \wedge \mathrm{SAFE}(S))$$

**An Alternative Specification Of The N-Queens Problem**

The above formulation of the problem is, in many respects, naïve because it deliberately abandons any commitment to what the solution will look like at the end. In fact, it seems to ignore useful insights that, not only make the formulation simpler, but also render the problem-solving strategy dealing with it more efficient.

Among these insights is the fact that the size of the set S should be equal to N. In fact, having that in the declaration part (i.e. saying that S is a set of N elements saves the problem solving strategy one explicit test.

Moreover, one can observe from the start that each queen will be on a separate row. This way, we can assign each queen a distinct row ahead of time. This saves us yet another test, simply by confining ourselves to look for the column part of each queen. This also confines testing to columns and diagonals only.

Finally, since the relation $\texttt{TAKES}(Q_1, Q_2)$ is symmetric, we can restrict testing each new queen to the already allocated queens without any loss of generality.

With these insights in mind, we can now have a better specification, relying on the following definitions:

- N : an integer that is a global parameter of the problem.
- [1..N]: a range indicating the domain of variables (i.e. queens).
- Q: a function that associates each distinct row i with a column Q(i) that is an integer in the range [1..N]. This way, the pair <i,Q>, will accurately define one position for the queen, on row i.
- TAKES(i,j,Q) tells when a queen <i,Q> can take the queen <j,Q>. Since i and j are assumed to be distinct, this can be simplified to:

$$\begin{aligned} &(Q(i) = Q(j)) \\ \vee\ &(i - Q(i) = j - Q(j)) \\ \vee\ &(i + Q(i) = j + Q(j)) \end{aligned}$$

- TAKEN(i,j,k,Q) tells when a queen <i,Q> is taken by at least one of the queens in the rows [j..k]. Noting that $i \notin [j..k]$, we get:

$$(\exists\ t \in [j..k])\ \texttt{TAKES}(i,t,Q)$$

- SAFE(j,k,Q) tells when no queen in the rows [j..k] can take any of the other queens in the same rows. That is,

$$(\forall\ i \in [j+1..k])\ \neg\texttt{TAKEN}(i,j,i-1,Q)$$

In the above terms, the specification will be:

$$(\exists\ Q \in [1..N] \rightarrow [1..N])\ \texttt{SAFE}(1,N,Q)$$

### 7.4. The Perfect Square Placement Problem

The informal definition of this problem [17] goes as follows.

> The perfect square placement problem (also called the squared square problem) is to pack a set of squares with given integer sizes into a

> bigger square in such a way that no squares overlap each other and all square borders are parallel to the border of the big square. For a perfect placement problem, all squares have different sizes. The sum of the square surfaces is equal to the surface of the packing square, so that there is no spare capacity. A simple perfect square placement problem is a perfect square placement problem in which no subset of the squares (greater than one) is placed in a rectangle.

We will need the following setting for the problem. The main square is of size (i.e. edge length) M, we can assume that to be the set $[1..M]\times[1..M]$. We assume that we have N other smaller squares whose sizes are stored in an array T; that is, each square i ($i \in [1..N]$) has a size equal to T[i]. Thus, we have the following precondition of the problem:

$$T[1]^2 + T[2]^2 + \ldots + T[N]^2 = M^2$$

Now, the problem consists of finding for each smaller square, a position <x,y> within the main square so that the condition of the problem holds. That is, all the positions <x,y> should be inside $[1..M]\times[1..M]$. Note that, each square is now completely defined by the pairs <i,p> where i is the index of where the size of this square resides in T and p is the position of this square on the main square.

In addition to the above, we will need the following auxiliary definitions:

- when two intervals [a..b] and [c..d] intersect. That is:

  $$IINTER([a..b], [c..d]) \equiv MAX(a,c) < MIN(b,d)$$

- when two squares <i,p> and <j,q> intersect. That is:

  $$SINTER(<i,p>, <j,q>) \equiv Iinter([1^{st}(p)..1^{st}(p)+T[i]], ([1^{st}(q)..1^{st}(q)+T[j]]) \wedge IINTER([2^{nd}(p)..2^{nd}(p)+T[i], [([2^{nd}(q)..2^{nd}(q)+T[j]])$$

- a function P that associates each square i with an edge length T[i], with a position <x,y> on the main square $[1..M]\times[1..M]$. This way, given an index i, the pair <i, P>, will accurately define one square on the main square; in fact, <i, P> immediately gives <i, P(i)>.

- INTER(i, j, k, P) tells when a square <i, P> overlaps at least one of the squares in the interval [j..k] of T. Noting that $i \notin [j..k]$, we get:

  $$(\exists\ t \in [j..k])\ SINTER(<i,P(i)>, <t,P(t)>)$$

- SAFE(j, k, P) tells when no square in the interval [j .. k]of T overlaps any of the other squares in the same interval. That is,

$$(\forall\ i \in [j+1..k])\ \neg INTER(i, j, i-1, P)$$

In the above terms, the specification will be:

$$(\exists\ P \in [1..N] \rightarrow [1..M]\times[1..M])\ SAFE(1, N, P)$$

The similarity between this specification and that of the N-Queens problem presented in the previous sub-section allows us to claim that, putting domain-specific heuristics aside, the same problem-solving strategy that can solve one of the problems, should perform as well on the other. This is another facet of the service that formal specification can offer this domain toward the automation of the whole process.

### 7.5. The Traffic Lights Problem

This problem [23] was originally proposed by [9] as a CSP benchmark example. It has several interesting features. It is a real-world problem, not too complex, in that it has relatively few constraints. However, each of these is a very tight higher arity (not just binary) constraint, showing that local propagation appears to be of little or no effect.

**Informal Definition**

This problem considers a four-way traffic junction with eight traffic lights. Four of these are for vehicles and can have the colours red, red-yellow, green and yellow. The remaining four are for pedestrians and can have the colours red or green. Correspondingly, there are four variables ($V_1$ to $V_4$) with domain {r, ry, g, y} and another four variables ($P_1$ to $P_4$) with domain {r, g}.

The constraints are ` odelled by quaternary constraints on the tuples ($V_i$, $P_i$, $V_j$, $P_j$) which, for $i \in [1..4]$ and $j = (i+1) \bmod 4$, allow just the corresponding values to be tuples from the set {(r,r,g,g), (ry,r,y,r), (g,g,r,r), (y,r,ry,r)}.

**Problem Analysis**

It seems that we have here eight variables ($V_1, P_1, V_2, P_2, V_3, P_3, V_4, P_4$), with a total of $2^{12}$ possible assignments, of which only $2^2$ are solutions. These are: (r,r,g,g,r,r,g,g), (ry,r,y,r,ry,r,y,r), (g,g,r,r,g,g,r,r) and (y,r,ry,r,y,r,ry,r)}.

However, looking at the structure of the tuples ($V_i$, $P_i$, $V_j$, $P_j$), and the tuples in the constraint set above, reveals a strong relationship between the pairs <$V_i$, $P_i$> for each i in the interval [1..4], and the pairs <$V_j$, $P_j$> for $j = (i+1) \bmod 4$. This relationship alone will make for a simpler specification and drastically

reduces the size of the search space, therefore simplifying the whole process of finding the solution.

**Problem Specification**

In fact, assuming the domains: $V = \{r, ry, g, y\}$ and $P = \{r, g\}$, we will bunch the eight variables $(V_1,P_1,V_2,P_2,V_3,P_3,V_4,P_4)$ into four pairs $<V_1,P_1>$, $<V_2,P_2>$, $<V_3,P_3>$ and $<V_4,P_4>$. Thus, we can think of the four variables as four elements in the interval $[1..4]$. Each one of these variables will take its values from the domain $V \times P$.

Now, looking at the constraint set above, we will define a new smaller more structured set S as follows:

$$S = \{<<r,r>,<g,g>>, <<y,r>,<ry,r>>\}$$

This way, given a function f from the interval $[1..4]$ to the set $V \times P$, we define the predicate SAFE(f) as follows:

$$(\forall\ i \in [1..4])\ (\forall\ j \in [1..4])\ (j = (i+1 \bmod 4)) \Rightarrow (<f(i),f(j)> \in S) \vee (<f(j),f(i)> \in S)$$

Thus, the specification of the problem will be:

$$(\exists\ f \in [1..4] \rightarrow V \times P)\ SAFE(f)$$

The clarity of this specification should simplify the task of generalising to other types of junction (e.g. five roads intersecting) as well as modelling the evolution over time of the traffic light sequence.

### 7.6. The Stable-Marriage Problem

This is a well know CSP benchmark example. As usual, we start by providing the informal definition of the problem and then proceed to formalise that in our proposed language.

**Informal Definition**

Given are n men and n women. Given also is 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 no other woman w' can simultaneously satisfy
>
> (1) m prefers w' over w and
> (2) w' prefers m over her associated man m'

**An Alternative Informal Definition**

Eliminating the negation from the above definition will make for a somewhat smoother translation to our formal language. The alternative definition goes as follows: given are $n$ men and $n$ women. Given also is 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 the for all other women w' either
>
> (1) m prefers w over w'or
> (2) w' prefers her associated man m' over m

The above alternative is obviously equivalent to the original definition of the problem.

**Formal specification**

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.

Given are two sets $M$ (of men) and $W$ (of women), each containing $n$ element. In addition to that, we associate each element $x$ of each of the two sets 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.

Now, according to the alternative informal definition using the basic building blocks specified in the previous paragraph, the specification of the Stable-Marriage problem looks pretty straightforward. In fact, the specification is fully described by the following logical formula:

$$
\begin{aligned}
&(\exists\ h \in W \rightarrow M)\\
&\quad \mathrm{BIJECTIVE}(h) \wedge\\
&\quad (\forall\ w \in W\ \forall\ w' \in W\\
&\qquad (w \neq w') \Rightarrow ((w' \leq_{h(w)} w) \vee (h(w) \leq_{w'} h(w')))
\end{aligned}
$$

### 7.7. The Timetabling Problem

This is a more elaborate example than the previous ones. It addresses a general system that can generate school and university timetables. This is a substantial real-world practical system that is currently in use for several years now. This system has entirely been specified using the same paradigm and the same notation based on the language proposed in this paper. It was surprising how few are the primitives required for the specification of a substantial task such as the timetabling program. The details of this example have been submitted for publication elsewhere [2].

## 8. DISCUSSION

The next step starting from the research described in this paper will be to make specific the constructs of the DEPICT 0.1 language. We intend to gain more experience with this language by trying it on further examples to see what other features it may still be lacking. When the language is more mature, we shall specify its semantics. This should be manageable, considering that it is based on a well-founded, well-specified formal specification language.
The existence of a good specification language can open the door to many research directions under software engineering:

- Program verification: algorithms and implementations can be verified against the formal specification more easily. This is the motivation for formal specification in software engineering.
- Problem transformation: some formulations allow the problem to be solved more efficiently as pointed out by Borrett and Tsang [4, 5]. For example, some redundant constraints are useful, but some are counter-productive for problem solving efficiency. Problem specification under a well-defined language potentially allows one to reason with the transformation more systematically.
- Software reusability: DEPICT 0.1 allows one to concentrate on formal specification of the problem. One possible research direction is to link a specification to constraint programming languages, such as ECLiPSe [10], ILOG [15] or EaCL [6, 13]. This way, constraint software engineers may use the advanced algorithms developed in these systems without having to re-implement them.
- Software Synthesis: programs (hence solutions) can be generated automatically from specifications. This is a long-term aim, but we see it as a distinct possibility in the CSP context.

On top of problem specification, heuristics may be formalized as well. If one can do that, then constraint software engineering (the process of formulating CSPs to developing or finding algorithms to solve them) can be done more systematically. Similar insights have been a driving force behind systems like the Kestrel Interactive Development systems (KIDS) [18, 19], which automatically synthesizes correct programs in the applicative (REFINE) language from formal specifications that are also written in REFINE. This is also the core idea in programming systems like PROLOG. This line of thinking holds even more credibility here when seeing that CSPs have a distinct logical expression, and are usually associated with a much richer variety of problem-solving strategies than just general depth-first search.

## BIBLIOGRAPHY

1. Abbas, A. *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. Abbas, A. and Tsang, E. *Constraint-based Timetabling, a case study*, submitted for review, December 2000.

3. Ambert, F., Legeard, B. and Legros, E. *Programmation en Logic avec contraintes sur ensembles et multi-ensembles héréditairement finis*, Techniques et Sciences Informatiques 15(3), pp297-328, 1996.
4. Borrett, J.E., *Formulation selection for constraint satisfaction problems: a heuristic approach*, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK, 1998
5. Borrett, J.E. & Tsang, E.P.K., *A context for constraint satisfaction problem formulation selection*, CONSTRAINTS, to appear (expected 2001)
6. Bradwell, R., Ford, J., Mills, P., Tsang, E.P.K. & Williams, R., *An overview of the CACP project: modelling and solving constraint satisfaction/optimisation problems with minimal expert intervention*, Workshop on Analysis and Visualization of Constraint Programs and Solvers, Constraint Programming 2000, Singapore 22 September 2000
7. Flener, P., Hnich, B. and Kiziltan, Z, *Compiling High-Level Type Constructors in Constraint Programming* (submitted for review). The ASTRA project, Department of Information Science, Computer Science Division, Uppsala University, Sweden. 2000.
8. Gervet, C. *Interval Propagation to reason about sets: Definition and Implementation of a practical language*. Constraints 1(3), pp 194-244, 1997.
9. Hower, W. *Revisiting global constraint satisfaction*, Information Processing Letters, 66 (1998) 41-48.
10. Lever, J., Wallace, M. & Richards, B., *Constraint logic programming for scheduling and planning*, British Telecom Technology Journal, Vol.13, No.1., Martlesham Heath, Ipswich, UK, 1995, 73-80
11. Lauriere, J. L., *ALICE: A Language and a Program for Solving Combinatorial Problems*, Artificial Intelligence, Vol. 10, 1978, pp 29 - 127
12. Mackworth, A. K. *The Logic of Constraint Satisfaction, in Constraint-Based Reasoning*, Freuder & Mackworth (ed.), MIT Press 1994.
13. Mills, P. et al., *EaCL 1.5: An Easy Abstract Constraint Optimisation Programming Language*, Technical Report CSM-324, Department of Computer Science, University of Essex, U.K., 2000.
14. Müller, T. *Solving set partitioning problems with constraint programming*. In Proceedings of PAPPACT'98, pp 313-332. The Practical Application Company, 1998.
15. Puget, J-F., *Applications of constraint programming*, in Montanari, U. & Rossi, F. (ed.), Proceedings, Principles and Practice of Constraint Programming (CP'95), Lecture Notes in Computer Science, Springer Verlag, Berlin, Heidelberg & New York, 1995, 647-650
16. Simonis, H., *The CHIP system and its applications*, in Montanari, U. & 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
17. Simonis, H. World Wide Web, CSPLib home page.
18. Smith, Douglas R. *KIDS: A Semi-Automatic Program Development System*, IEEE Transactions on Software Engineering --- Special Issue on Formal Methods, Vol. 16, No. 9, September 1990.
19. Smith, Douglas R. *KIDS: A Knowledge-Based Software Development System*, Automating Software Design, Eds. M. Lowry and R. McCartney, MIT Press, 1991, 483-514.
20. Tsang, E, *Foundations Of Constraint Satisfaction*, Academic Press, 1993.

21. Van Hentenryck, P., *Constraint satisfaction in logic programming*, MIT Press, Cambridge, 1989
22. Van Hentenryck, P. *The OPL Optimization Programming Language*, The MIT Press, 1999.
23. Walsh, T, World Wide Web, CSPlib home page.

## ACKNOWLEDGMENT

The authors are deeply indebted to Pierre Flener for reviewing a version of this paper, for his elaborate comments and for the references that helped in filling background gaps that existed in an initial version of this paper. The authors would also like to thank the workshop reviewers for enabling them to see more potential for some of the ideas presented in this paper. The idea of providing solver-independent specifications originates from Dr Carmen Gervet.

## APPENDIX

This appendix contains the complete formal expressions of all the examples treated in this paper. These are listed in the same order they are treated in the main body of the paper and presented in accordance with the general form of the CSP specification expression presented in section **6**.

### 1. The Sorting Problem

$$\{\mathbf{BIJECTIVE(f)} \equiv \mathrm{SIZE}(f([1..N])) = N,$$
$$\mathbf{ORDERED(f)} \equiv$$
$$\mathrm{BIJECTIVE}(f) \wedge$$
$$(\forall\, i \in [1..N-1])\ (A(f(i)) \leq A(f(i+1)))\}$$

$$(\exists\, \mathbf{f} \in \mathbf{[1..N]} \rightarrow \mathbf{[1..N]})\ \mathrm{ORDERED}\mathbf{(f)}$$

$$(N \in \tilde{\mathrm{N}},\ D \in Đ,\ (\leq) \in \hat{\mathrm{O}}(D),\ A \in [1..N] \rightarrow D)$$

Where $\tilde{\mathrm{N}}$ is the domain of non-negative integers, $Đ$ is intended to mean the universe of all domains, $\hat{\mathrm{O}}(D)$ is the set of all ordering relations over the domain $D$, and $f([1..N])$ is the set $\{f(i);\ i \in [1..N]\}$.

### 2. The Map Colouring Problem

$$\{\mathbf{BORDERING}(c_1, c_2) \equiv (c1 \in M(c_2)) \vee (c_2 \in M(c_1)),$$
$$\mathbf{SAME}(c_1, c_2, \mathbf{f}) \equiv \mathrm{BORDERING}(c_1, c_2) \wedge f(c_1) = f(c_2),$$
$$\mathbf{SAFE(f)} \equiv \forall\, c_1 \in CN\ \forall\, c_2 \in CN\ c_1 \neq c_2 \rightarrow \neg \mathrm{SAME}(c_1, c_2, f)\}$$

$$\exists\, \mathbf{f} \in \mathbf{CN} \rightarrow \mathbf{CL}\ \mathbf{SAFE(f)}$$

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

Here P(CN) denotes the set of subsets of CN and Ð(N) is used to denote the universe of finite domains of size N.

### 3. The N-Queens Problem

**{TAKES(i,j,Q)** ≡
(Q(i) = Q(j))
∨ (i - Q(i) = j - Q(j)) ∨ (i + Q(i) = j + Q(j)),
**TAKEN(i,j,k,Q)** ≡ (∃ t ∈ [j..k])TAKES(i,t,Q),
**SAFE(j,k,Q)** ≡ (∀ i∈[j + 1..k])¬TAKEN(i,j,i-1,Q)**}**

**(∃ Q∈[1..N]→[1..N]) SAFE(1,N,Q)**

**(**N ∈ Ñ**)**

### 4. The Perfect Square Placement Problem

**{IINTER([a..b],[c..d])** ≡ MAX(a,c) < MIN(b,d),
**SINTER(<i,p>, <j,q>)**≡
IINTER([$1^{st}$(p)..$1^{st}$(p)+T[i]],([$1^{st}$(q)..$1^{st}$(q)+T[j]])
∧
IINTER([$2^{nd}$(p)..$2^{nd}$(p)+T[i],[([$2^{nd}$(q)..$2^{nd}$(q)+T[j]],
**INTER(i,j,k,P)** ≡
(∃ t ∈[j..k]) SINTER(<i,P(i)>,<t,P(t)>),
**SAFE(j,k,P)** ≡
(∀ i∈[j+1..k])¬INTER(i,j,i-1,P)}

**(∃ P∈[1..N] → [1..M]×[1..M]) SAFE(1,N,P)**

**(**M ∈ Ñ, N ∈ Ñ, T ∈ [1..N] → Ñ**)**

Obviously, we have to state the following preconditions of the problem:

- all small squares will fit in the main square without overlapping:

$$T[1]^2 + T[2]^2 + \ldots + T[N]^2 = M^2$$

- all small squares have different sizes for a perfect square placement:

$$(\forall\, i \in [1..N])(\forall\, j \in [1..N])\ (i \neq j) \Rightarrow T[i] \neq T[j]$$

### 5. The Traffic Lights Problem

**{V** ≡ {r, ry, g, y},
**P** ≡ {r, g},
**S** ≡ {<<r,r>,<g,g>>, <<y,r>,<ry,r>>},

```
SAFE(f) ≡
(∀ i ∈ [1..4])
    (∀ j ∈ [1..4])
         (j = (i+1 mod 4)) ⇒
              (<f(i),f(j)> ∈ S) ∨ (<f(j),f(i)> ∈ S)}

(∃ f ∈ [1..4] → V×P) SAFE(f)

( )
```

Note that, as it is stated above, this problem has an empty parameter list. However, if a generalization of this specification can be made to an arbitrary number N of traffic lights, then N will replace 4 in the specification and will therefore take its place in the parameter list.

## 6. The Stable-Marriage Problem

```
{≤ ≡ ≤WM ∪ ≤MW,
 BIJECTIVE(h) ≡ SIZE(h(W)) = N}

(∃ h ∈ W → M)
     BIJECTIVE(h) ∧
     ∀ w ∈ W ∀ w'∈ W
      (w ≠ w') ⇒ ((w' ≤h(w) w) ∨ (h(w) ≤w' h(w')))

(N∈ Ñ, M∈ Ð(N), W∈ Ð(N), ≤WM ∈ W→Ô(M), ≤MW ∈ M→Ô(W))
```

Here, Ð(N) is used to denote the universe of finite domains of size N. Also, the expression $\leq_{WM} \cup \leq_{MW}$ is taken to mean that the resulting ordering relation is over the domain M or W depending on whether its arguments are from the first or the second domain.