Preference Programming for Configuration

Ulrich Junker
ILOG
1681, route des Dolines
06560 Valbonne
France
junker@ilog.fr

Abstract

Preference programming provides a new paradigm for expressing (default) decisions, preferences between decisions, and search strategies in a declarative and unified way and for embedding them in a constraint and rule language. Business experts can thus directly specify preferences and search directives in form of rules without needing to program search strategies as required by constraint programming based configuration tools. Preference programming allows to describe preferences between individual decisions, as well as groups of decisions and decision rules. There can be dynamic (or context-dependent) preferences, inconsistent preferences, and meta-preferences. Following [Brewka, 1989; Junker, 1997], preferences constrain the order in which decisions are made during search. It is possible to enumerate all configurations or to focus search to preferred configurations, which respect the default choices and preferences of the user.

Keywords: preferences, configuration, constraint programming, nonmonotonic reasoning, search.

1 Introduction

Configuration problems are typically specified in form of rules and constraints that are formulated in an adequate language (cf. [Felfernig *et al.*, 1999; Soininen *et al.*, 2000]) or tool (cf. [Mailharro, 1998]). Constraint-based approaches to configuration also allow to specify a global objective function. Constraint satisfaction appears to be one of the major approaches for solving configuration problems. Typically, a solution is determined by a tree search procedure that introduces new constraints during search and that activates constraint propagation in each node of the search tree. As consequence, the domains of the constrained variables are reduced and a fail occurs if one domain becomes empty.

The order of the search decisions is of high importance. Making a good decision first can lead to a better domain reduction and pruning of the search tree. If the reduced domains guide the choice of variables to be instantiated, the choice of a decision can also influence the order of the subsequent decisions. Last, but not least, the order of the decisions determines the order in which solutions are enumerated. If only one or k solutions are requested then the order of decisions influences which solutions are returned.

In constraint satisfaction, the order of search decisions is usually specified by variable and value ordering heuristics. Those heuristics specify which value assignment should be selected next. For example, the first-fail heuristics selects a variable x with a domain of smallest size. Then a value in the current domain $dom(x)$ of x is determined by a value-ordering heuristics that, for example, retains the smallest value $v := min(dom(x))$. Thus, the assignment $x = v$ has been selected as next decision.

Constraint programming allows to program arbitrary And-Or search trees by using PROLOG clauses, search goals as in ILOG SOLVER, or non-deterministic procedures as in OPL. Arbitrary constraints can be added in these search programs. Many successful applications of constraint programming involve sophisticated search programs that exploit the structure of the given problem. In particular, large-scale optimization problems does not allow a complete exploration of the search tree in the given time frame. Hence, they require sophisticated heuristics that guide the user to good solutions quickly. Unfortunately, the elaboration of those heuristics is quite tedious. Since they are implemented in a procedural fashion it is difficult to maintain and adapt them.

For configuration problems, the order of decisions appears to be important as well:

1. Certain configuration problems may be (nearly) backtrack-free if decisions are made in the right order.
2. Difficult configuration problems may require symmetry-breaking techniques that can only be applied if more specific (i.e. more substitutable) decisions are tried first.
3. Many configurations problems (e.g. product configuration) are weakly constrained and have numerous solutions, but only some of them respect given default choices and preferences of the user. In order to determine these preferred solutions, decisions have to be explored in an adequate order.

In this paper, we develop an approach called *preference programming* for expressing search strategies, default decisions, and preferences between decisions in a declarative and unified way. We enrich a given constraint and rule language by

new primitives for stating decisions and preferences. Thus, an expert of a particular configuration domain can directly specify preferences and search directives in form of rules without needing to program search strategies in a procedural way. The strategies can easily be modified by changing the preferences rules or by adding new ones. Hence, it is possible to interactively edit search behaviour. Furthermore, we can focus search to preferred solutions, which respect default choices and preferences of the user.

Preference programming is based on earlier work done in nonmonotonic reasoning. In [Brewka, 1989; Junker, 1997], preferences constrain the order in which default rules are applied. This idea can directly be applied to the decision making process that is performed by search procedures. A decision making process enumerates possible decisions in some (total) order $\delta_1, \ldots, \delta_n$. As discussed above, choosing a good order of the decisions is an important issue for solving a configuration problem. We therefore use preferences between decisions as constraints on the order in which decisions are made. An order $\delta_1, \ldots, \delta_n$ satisfies a preference *prefer*(δ_i, δ_j) if and only if decision δ_i precedes δ_j in the decision making process (i.e. if $i < j$). The chosen decision order only determines in which sequence the decisions are explored during search. It does not yet fix which decisions are indeed made. Given a decision order, a solution can be defined in two ways:

1. The decisions are either selected or rejected. When exploring a decision, we set up a choice point. In the left branch we make the decision whereas in the right branch we add the negation of the decision. Here, the preferences only influence the order of the choice points.
2. The decisions are selected in a greedy fashion. When exploring a decision, we check whether it is consistent w.r.t. the already made decisions. We make the decision if it is consistent. Different solutions can be obtained by different orders. Here, the preferences influence the order in which decisions are made. If the preferences describe a total order then only one solution is obtained.

The first approach allows to enumerate all solutions in a systematic way, whereas the second approach produces only the preferred solutions, i.e. those solutions that respect the given preferences. Both approaches can be used in our framework.

Preference programming allows to describe preferences between individual decisions, as well as groups of decisions, and decision rules. Decision rules are like the default rules in [Poole, 1988] and express default decisions. For example, we may write two decision rules called *look*$_D$ and *comfort*$_D$:

- *look*$_D(x)$: If x is a customer and x is a woman then the color of the proposed car should be red.
- *comfort*$_D(x)$: If x is a customer and x is a manager then the seat material of the proposed car should be leather.

In certain cases, those rules are in conflict and preferences can be used to specify which rule is preferred as proposed in [Brewka, 1989]. These preferences can depend on the context and may be introduced dynamically during decision making [Junker, 1997]. An example is

- if x is a customer and x is young then prefer *look*$_D(x)$ to *comfort*$_D(x)$.

In our approach, preferences are treated as constraints. For this reason, they can occur in implications. Furthermore, if preferences risk to be inconsistent they can themselves be introduced as decisions. Meta-preferences could then be used to specify which of the inconsistent preferences should be retracted. This shows that preference programming is able to model quite complex reasoning situations.

Preference programming also allows to model the traditional variable and value ordering heuristics. For example, a value ordering heuristics that selects the smaller values first can be described by following preferences:

- if $u < v$ prefer decision $x = u$ to decision $x = v$.

Variable ordering heuristics such as first-fail can be modelled with dynamic preferences. The essential part of a search strategy is the information which decisions are made in which order. The concepts of decision and preference allow to express this information explicitly and in a declarative and modular way.

The paper focuses on a presentation of the primitives of preference programming using examples. First of all, we describe the configuration language that is used by our system in section 2. In section 3, we introduce the basic concepts of our approach, namely decisions and preferences. Furthermore, we show how preferences constrain the decision making process and how preferred solutions can be obtained in this way. In section 4, we then show how complex decisions and preferences can be expressed in a compact and user-friendly form. In section 5, we show how classical search strategies can be embedded in the preference framework.

2 The Configuration Language

We base our discussion on a configuration language that can be seen as a first-order language where certain function and predicate symbols have a special role and a predefined interpretation (or semantics). The configuration language itself is implemented by the API of our configuration tool [ILOG, 2000]. Certain details of the API are omitted in the language in order to keep the presentation of this paper simple.

Our language is specified by different (finite) sets of symbols that are all mutually different. Logical variables are represented by symbols in a set $\mathcal{X}$. Let $\mathcal{O}, \mathcal{C}, \mathcal{A}$ be three set of symbols that denote objects, classes, and attributes. Object and class symbols are constants. Attributes symbols are unary function symbols and can be used to refer to attribute values of objects. For example `maxSpeed(car1)` denotes the maximal speed of object `car1`, whereas `engine(x)` denotes the engine of `x` where `x` is a logical variable in $\mathcal{X}$. Attributes have to be interpreted by functions mapping the objects to values, namely integers, objects, or sets of objects. Each attribute has an associated domain.

Furthermore, we use the function symbol `instances` to denote the set of objects of the classes. For example, `instances(Car)` denotes the set of instances of the class `Car`. The symbol `instances` is interpreted by a function mapping classes to sets of objects.

Integers are represented as in programming languages (and are interpreted by integers). Furthermore, there are function symbols for representing arithmetic operators such as `+`, `-`, `*`,

`/`, `abs`, `min`, `max` and set operators such as `intersect`, `union`, `card`. Hence, we can formulate *terms* such as

```
price(car1) + optionPrice(car1)
card(options(x))
```

There are also set-expressions such as sum-over-set or union-over-set, but we omit them for the sake of brevity. All these function symbols have an evident interpretation.

All predicate symbols of our configuration language have a fixed interpretation. There are symbols for representing comparison operators such as `=`, `!=`, `<=`, `<`, constraints on sets such as `subset`, `subseteq`, `in`, and other kinds of constraints. Applying a predicate symbol to terms yields an atomic formula, which we also call a primitive *constraint*. For example, we can write constraints such as

```
x in instances(customer)
sex(x) = female
color(car1) = red
```

Constraints can be combined to new constraints by using logical connectors or the all-quantifier. Existential quantifier are not used. We write logical connectors as `and`, `or`, `not`, and `if ... then ...` and the all-quantifier as `forall`. For example, we can write following constraint

```
forall x: if x in instances(Car) and
             power(engine(x)) >= 100 and
             weight(x) <= 1000
          then ABS in options(x);

forall x: if x in instances(Car) then
          engine(x) in instances(Engine)
```

A configuration problem is then specified by a set of constraints C that have no free variables. As indicated above, constant, function, and predicate symbols have a fixed interpretation except for attributes and the symbol `instances`. In order to specify a logical interpretation of C, we just have to determine a value for each term of the form $a(t)$ where a is an attribute symbol in $\mathcal{A} \cup \{\texttt{instances}\}$. We therefore map the terms $a(t)$ to constrained variables and determine a value assignment to those variables. Value assignments that satisfy all constraints in C are called *solutions* of C. The set C is called consistent iff it has a solution.

Our configuration engine [ILOG, 2000] provides constraint propagation for reducing domains of attributes, for specializing components, and for generating components if required by cardinality and resource constraints. The mechanisms are described in [Mailharro, 1998].

3 Decisions and Preferences

In this section, we introduce decisions and preferences and show how the preferences control the decision making process that is performed by search procedures.

3.1 Decisions

A (tree) search procedure for a constraint satisfaction problem is adding additional constraints to the original problem. We call such a constraint a decision. A search procedure explores different decisions in different branches of the search tree. In order to model this behaviour in an explicit way, we introduce the concept of a decision.

In our approach, a *decision* is specified by an identifying term t, for example a name, and a constraint ϕ. It is written in the form `decision` t:ϕ. We can also consider decisions as named or labelled constraints. The name belongs to a new set of symbols used for naming decisions. In later sections, we also introduce other terms for identifying decisions. Different decisions must have different identifying terms. Hence, we cannot have two decisions of same name. Some examples for decisions are:

```
decision red-d:     color(car1) = red;
decision blue-d:    color(car1) = blue;
decision black-d:   color(car1) = black;
decision leather-d:
             seatMaterial(car1) = leather;
decision cloth-d:
             seatMaterial(car1) = cloth;
```

A configuration problem is now specified by a (finite) set C of constraints and a (finite) set $\mathcal{D}$ of possible decisions.

During the decision making process, decisions are made (or executed). Executing a decision `decision` $t : \phi$ will add the constraint ϕ to the current set of constraints C. We can also describe this behavior by stating that `decision` $t : \phi$ expresses a constraint that is satisfied if and only if ϕ is satisfied. When adding `decision` $t : \phi$ to the set C we obtain ϕ as logical consequence of the resulting set of constraints.

3.2 Decision-Making Process

A decision making process explores the given decisions in a total order. As discussed in the introduction, this order influences the problem solving behaviour and is the important part of a search strategy. The order may be static, i.e. specified initially, or dynamic, i.e. depend on the decisions made. In this section, we make this order explicit, which allows to formulate constraints on it.

Suppose our configuration problem has a set $\mathcal{D}$ of n decisions. Since the order of the decisions is not necessarily known initially, we describe it by n (constrained) variables $\delta_1, \ldots, \delta_n$. The value of a variable δ_i is a decision in $\mathcal{D}$. In our example, the domain of the decision variables δ_i is the set $\{$`black-d`, `red-d`, `blue-d`, `cloth-d`, `leather-d`$\}$.

Once the values of the decision variables are determined, the decision making process explores them in increasing order starting with δ_1. Each decision in $\mathcal{D}$ should have a unique position in this order. Hence, the δ_i's are mutually different, which is imposed by an implicit constraint:

$$\delta_i \neq \delta_j \text{ for } i \neq j \tag{1}$$

We use these constrained variables δ_i only to describe our approach conceptually. We do neither maintain a current domain for the δ_i's, nor apply domain reduction techniques to them. Constraints on the δ_i's are only checked during search, but do not propagate.

The decision making process chooses the values of the δ_i's in increasing order. Once, the i-th decision has been selected it is explored. This exploration can be done in different ways:

S1 when exploring a decision δ_i, a (binary) choice point is set up. In the left branch, the decision δ_i is added to the current set C of constraints. In the right branch, the negation of δ_i is added to C.

S2 when exploring a decision δ_i, we check whether it is consistent w.r.t. C. If yes the decision δ_i is added to C. Otherwise it is dropped. Hence, decisions are tried in increasing order and only succeeding decisions are added to C. In this approach, decisions are made whenever possible. In this respect, they behave exactly as defaults.

The first procedure chooses the values of δ_i in a greedy fashion and uses them to define a complete search tree that allows to enumerate all solutions. Since the values of the δ_i's are chosen incrementally they can differ in different branches of the search tree.

The second procedure considers alternative values for the δ_i's, but executes the decisions in a greedy fashion once the δ_i's are determined. We can enumerate all solution in this way if all orders are allowed. In the next section, we impose constraints on the order. In this case, we eliminate certain solutions and enumerate only so-called preferred solutions. On a first glimpse, it seems that the basic complexity of this approach is much worse than that of the first approach ($O(n!)$ instead of $O(2^n)$), although less solutions may be obtained if the order is constrained. Indeed, many orders produce the same solutions. In [Junker, 2000], we have shown how irrelevant orders are avoided and reduce the effort to a binary tree of depth n. Furthermore, we can prune parts of the search tree that do not contain preferred solutions.

We give a simple example for the second procedure. Suppose the set C initially contains two constraints.

```
if color(car1) = red
then seatMaterial(car1) != leather;
if color(car1) = black
then seatMaterial(car1) != cloth;
```

Since there is no constraint on the order of decisions, we can introduce them in any order. Once the order is determined, we select decisions in a greedy fashion. The first decision `black-d` succeeds. Since the color is determined the next decisions, namely `red-d` and `blue-d` fail. The fourth decision `cloth-d` is incompatible w.r.t. `black-d` and fails as well. The final decision `leather-d` succeeds:

δ_1	**black-d**	succeeds
δ_2	red-d	fails
δ_3	blue-d	fails
δ_4	cloth-d	fails
δ_5	**leather-d**	succeeds

3.3 Preferences as Constraints on the Decision Order

Preferences between decisions are a convenient way to specify which decisions should be retained if several decisions are in conflict. Following [Brewka, 1989; Junker, 1997], we use preferences of the form `prefer`(t,u) as constraints on the order of decisions. If the terms t and u refer to two decisions d^t and d^u then a preference imposes following constraint on the order $\delta_1, \ldots, \delta_n$:

$$\text{if } \texttt{prefer}(t,u),\ \delta_i = d^t,\ \delta_j = d^u \text{ then } i < j \qquad (2)$$

In our example, we use following preferences:

```
prefer(red-d, blue-d);
prefer(blue-d, black-d);
prefer(leather-d, cloth-d);
```

Decision orderings that violate those preference constraints can no longer be used to generate solutions of our configuration problem. For each solution X, we require that there is an order of decisions that satisfies the preferences and that is able to produce the solution by a selection mechanism. If S1 is used any order is able to produce any solution. If S2 is used the preferences that are implied by a solution eliminate certain orders. If we can find an admissible order such that S2 produces X then X is called a *preferred solution*. In [Junker, 1997], we showed how preferred solutions can be constructed incrementally.

Below, we give the two preferred solutions of our example. The first decision is either `red-d` or `leather-d` and determines which of the two preferred solutions is obtained. For each of them, we give one of the (justifying) orders:

δ_1	**red-d**	succeeds
δ_2	blue-d	fails
δ_3	black-d	fails
δ_4	leather-d	fails
δ_5	**cloth-d**	succeeds

δ_1	**leather-d**	succeeds
δ_2	cloth-d	fails
δ_3	red-d	fails
δ_4	**blue-d**	succeeds
δ_5	black-d	fails

If a new constraint is added preferred solutions can become inconsistent and non-preferred solutions can become preferred ones. For example, if there are no blue cars with leather seats then our second solution becomes inconsistent and a new preferred solution is obtained:

δ_1	**red-d**	succeeds
δ_2	blue-d	fails
δ_3	black-d	fails
δ_4	leather-d	fails
δ_5	**cloth-d**	succeeds

δ_1	**leather-d**	succeeds
δ_2	cloth-d	fails
δ_3	red-d	fails
δ_4	blue-d	fails
δ_5	**black-d**	succeeds

4 Preference Programming

Specifying preferences between individual decisions is a tedious task. In this section, we introduce primitives for expressing sets of preferences in a more compact form.

4.1 Structured Preferences

In certain cases, several preferences can be replaced by a single one if decisions are grouped together. In our example, we want to express that all color-decisions are preferred to all seatMaterial-decisions. In order to do this, we introduce two groups of decisions. A group of decisions is specified by providing an unique identifying term (e.g. a name):

```
decision-model look-m;
decision-model comfort-m;
```

A decision model is just a set of decisions (or of other decision models). Elements of decision models are stated via containment constraints of the form `contains`(m,d). If m is a term referring to a decision model m' and d is a term referring to a decision d' then `contains`(m,d) implies that d' is a member of m'. In our example, we introduce following containment statements:

```
contains(look-m, red-d);
contains(look-m, blue-d);
contains(look-m, black-d);
contains(comfort-m, leather-d);
contains(comfort-m, cloth-d);
```

We can now introduce preferences between two decision models m_1 and m_2 meaning that all elements of m_1 are preferred to all elements of m_2:

```
prefer(look-m, comfort-m);
```

4.2 Preferences between Decision Rules

It is also possible to describe several decisions of the same form in a compact way. Instead of introducing individual decisions, we introduce so-called decision rules. The (ground) instances of such a decision rule then yield the individual decisions. Decision rules are similar to the default rules in [Poole, 1988]. We give two examples for decision rules:

```
decision-rule leather-rule(x):
  if x in instances(Customer) and
     profession(x) = manager
  then seatMaterial(car(x)) = leather;

decision-rule red-car-rule(x):
  if x in instances(Customer) and
     sex(x) = female
  then color(car(x)) = red;
```

If there are three customer Jim, Jane, and Jennifer, where Jim and Jane are manager, then these two rules represent four decisions:

```
decision d1:
      seatMaterial(car(jim)) = leather;
decision d2:
      seatMaterial(car(jane)) = leather;
decision d3: color(car(jane)) = red;
decision d4: color(car(jennifer)) = red;
```

Similar to decision models, we can express preferences between two decision rules r_1 and r_2 meaning that all instances of r_1 are preferred to all instances of r_2.

```
prefer(leather-rule, red-car-rule);
```

Alternatively, we can specify preferences between instances of decision rules. In order to refer to an instance of a decision rule with variables $x_1, \ldots, x_k$, we use a term $r(t_1, \ldots, t_k)$. Following example states that an instance of the leather-rule is preferred to an instance of the red-car-rule for the same person x.

```
forall x: if x in instances(Customer)
          then prefer(leather-rule(x),
                      red-car-rule(x));
```

Decision rules can also be added to decision models via containment-constraints.

4.3 Dynamic Preferences

Preferences can be context-dependent and thus be introduced dynamically during the decision making process. Since preferences are constraints we can use them in logical implications and thus express context-dependent preferences.

For example, we state that for young customers, the look of the car is more important than its comfort:

```
forall x: if x in instances(Customer) and
             age(x) = young
          then prefer(look, comfort);
```

For old persons, the comfort is more important than the look:

```
forall x: if x in instances(Customer) and
             age(x) = old
          then prefer(comfort, look);
```

Those examples demonstrate that quite complex preferences can be expressed by a few statements provided that a suitable categorization of decisions in form of decision models and decision rules is given.

4.4 Inconsistent and Meta Preferences

The preferences that are obtained from the different statements given above risk to be inconsistent. A set of preferences is inconsistent if there is no total order of the decisions that respects all of them. For example, the following preferences are cyclic and therefore inconsistent. In this case, there is no preferred solution:

```
prefer(look, comfort);
prefer(budget, look);
prefer(comfort, budget);
```

In order to make preference statements robust, preferences should be introduced themselves as decisions as following example shows:

```
decision rule p1(x):
  if x in instances(Customer) and
    playBoy in characteristics(x)
  then prefer(look, comfort);

decision rule p2:
  if x in instances(Customer)  and
    age(x) = old
  then prefer(comfort, look);
```

It is important that preferences between decisions have a higher priority than the decisions to which they are applied. A simple way to achieve this is to create two decision models m^p and m^d where the first one contains the preferences and the second one contains the decisions. Furthermore, we specify that m^p is preferred to m^d by a hard preference.

In order to resolve conflicts between preferences, we can introduce meta-preferences, i.e. preferences on preferences, such as `prefer(p1, p2)`.

5 Search Strategies as Preferences

We briefly describe how typical search strategies of constraint programming such as value and variable ordering heuristics can be modelled as preferences. A search strategy essentially indicates which decisions are made in which order.

For example, a value ordering heuristics that assigns smaller values of a variable first can be modelled by following preferences on value-assignments:

- prefer decision $a(t) = u$ to $a(t) = v$ if $u < v$

Modelling these preferences directly requires the generation of all decisions of the form $a(t) = v$ and is too costly in practice. Instead, we associate each constrained variable $a(t)$ with a decision set that represents the assignment decisions $a(t) = v$ in an implicit form. Only the best instances (w.r.t. given preferences) of this decision set are created in each state. The statement `assign`$(a(t))$ creates such a decision set for $a(t)$. Examples are:

```
assign(color(car1));
assign(seatMaterial(car1));
```

The corresponding decision sets are identified by the terms `color(car1)` and `seatMaterial(car1)`. This enables us to describe preferences between these decisions sets as follows:

```
prefer(color(car1), seatMaterial(car1));
```

Similar to the assignment decisions, the preferences of a value ordering heuristics are also represented in a compact from. We define a preference set for a decision set $a(t)$ by specifying an order of the domain of $a(t)$. An example is:

```
preferValues(color(car1),
             increasingOrder(color-dom));
```

Variable order heuristics are modelled by (dynamic) preferences between the decision sets of the constrained variables that are contained in a decision model. We represent them by a preference set that is supplied with a variable ordering.

```
preferVariables(look-m, minSizeFirst);
```

For the sake of brevity, we omit a detailled discussion of the description of domain orderings and variable orderings. Preference sets are then used to dynamically generate the best instances of their decision sets. For example, we can introduce a preference set that reproduces the behaviour of the popular first-fail-heuristics.

6 Conclusion

In this paper, we developed a declarative language for preference programming that provides a rich expressiveness for stating decisions and preferences. It captures default rules, dynamic preferences, and search strategies and is able to deal with inconsistent preferences. During decision making, the preferences impose constraints on the order of decisions and thus influence the search tree. Hence, preference programming allows to program search strategies in a declarative way and to edit them interactively. Furthermore, preference-based search [Junker, 2000] can be used to determine only preferred solutions, which respect the default choices and preferences of the user. Both features are useful for web-based configuration with a high user interaction.

Preferences programming is based on concepts elaborated in nonmonotonic reasoning (cf. e.g. [Brewka, 1989]). In [Junker, 1993], we already implemented a preference programming approach in form of the nonmonotonic rule-based system EXCEPT II V4. This work inspired [Brewka, 1994; Delgrande and Schaub, 2000], but there have been certain problems with inconsistent preferences. In [Junker, 1997], we showed how to avoid paradoxical situations caused by cyclic interactions between decisions and preferences and provided a clear semantics. Another major step is preference-based search (PBS) [Junker, 2000] that allows to focus search to preferred solutions. Based on these technical advances, we have now been able to improve the original preference programming approach and to embed it into constraint programming.

Preference programming thus is based on a logical approach for treating preferences. Preferences are themselves elements of the logical language and can be satisfied or violated. It is not only possible to reason with preferences, but we can also reason about preferences. In this respect, preference programming differs to approaches such as Valued CSP's or Semi-ring-based CSP's [Bistarelli *et al.*, 1999]. These approaches represent preference information by truth values that are assigned to the tuples of constraints. Since these truth assignments are not themselves expressed by constraints it is not evident how they could depend on the context or how they could be relaxed.

Furthermore, valued CSP's and semi-ring CSP's use algebraic operations to determine the truth value of a solution. It is thus possible to compare two solutions and to define a preference relation and preferred solutions. However, the algebraic operations impose a certain structure on the preference relation (either lattices or semi-rings) that limits the flexibility of the approach. Future work is needed for a more detailled comparison of those approaches.

The features on preference programming are part of the constraint programming based configuration tool ILOG JCONFIGURATOR. Practical examples and benchmarks are in preparation.

Future work will be dedicated to two challenging topics. On the one hand, we want to use preference programming for encoding preference relations as used in decision theoretical approaches [Boutilier *et al.*, 1997; Doyle and Thomason, 1999]. The key problem is to identify relevant preferences between conjunctions of constraints. Furthermore, we want to use PBS for symmetry-removal based on substitutability [Freuder, 1991].

Acknowledgements

This paper is based on earlier work done with Gerd Brewka. The article profited from numerous discussions with Olivier Lhomme. For helpful remarks, I also want to thank Xavier Ceugniet, Daniel Mailharro, and Jean-François Puget.

References

[Bistarelli *et al.*, 1999] Stefano Bistarelli, Ugo Montanari, Francesca Rossi, Thomas Schiex, Gérard Verfaillie, and Hélène Fargier. Semiring-based CSPs and Valued CSPs: Frameworks, properties, and comparison. *Constraints*, 4(3):199–240, 1999.

[Boutilier *et al.*, 1997] C. Boutilier, R. Brafman, C. Geib, and D. Poole. A constraint-based approach to preference elicitation and decision making. In *AAAI Spring Symposium on Qualitative Decision Theory*, Stanford, 1997.

[Brewka, 1989] G. Brewka. Preferred subtheories: An extended logical framework for default reasoning. In *IJCAI-89*, pages 1043–1048, Detroit, MI, 1989.

[Brewka, 1994] G. Brewka. Reasoning about priorities in default logic. In *AAAI-94*, 1994.

[Delgrande and Schaub, 2000] J. Delgrande and T. Schaub. Expressing preferences in default logic. *Artificial Intelligence*, 123:41–87, 2000.

[Doyle and Thomason, 1999] Jon Doyle and Richmond H. Thomason. Background to qualitative decision theory. *AI Magazine*, 20(2):55–68, 1999.

[Felfernig *et al.*, 1999] A. Felfernig, G. Friedrich, and D. Jannach. UML as domain specific language for the construction of knowledge-based configuration systems. In *Proc. SEKE'99*, pages 337–345, Kaiserslautern, 1999.

[Freuder, 1991] Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *AAAI-91*, pages 227–233, Anaheim, CA, 1991.

[ILOG, 2000] ILOG. Ilog Configurator. Reference manual and User manual. V2.0, ILOG, 2000.

[Junker, 1993] U. Junker. Dynamic generation of assumptions and preferences. In N. Bidoit, editor, *Actes des Vièmes Journées du Laboratoire d'Informatique de Paris Nord*, Villetaneuse, 1993.

[Junker, 1997] U. Junker. A cumulative-model semantics for dynamic preferences on assumptions. In *IJCAI-97*, pages 162–167, Nagoya, 1997.

[Junker, 2000] U. Junker. Preference-based search for scheduling. In *AAAI-2000*, pages 904–909, Austin, Texas, 2000.

[Mailharro, 1998] Daniel Mailharro. A classification and constraint based framework for configuration. *AI-EDAM: Special Issue on Configuration*, 12(4), 1998.

[Poole, 1988] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:27–47, 1988.

[Soininen *et al.*, 2000] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Unified configuration knowledge representation using weight constraint rules. In *ECAI-2000 Workshop on Configuration*, pages 79–84, Berlin, 2000.