Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2003), *17*, 13–29. Printed in the USA.
Copyright © 2003 Cambridge University Press 0890-0604/03 $16.00
DOI: 10.1017/S089006040317103X

Preference programming: Advanced problem solving for configuration

ULRICH JUNKER[1] AND DANIEL MAILHARRO[2]
[1]ILOG SA, 06560 Valbonne, France
[2]ILOG SA, BP 85, F-94253 Gentilly Cedex, France

(RECEIVED March 15, 2002; ACCEPTED October 13, 2002)

Abstract

Configuration problems often involve large product catalogs, and the given user requests can be met by many different kinds of parts from this catalog. Hence, configuration problems are often weakly constrained and have many solutions. However, many of those solutions may be discarded by the user as long as more interesting solutions are possible. The user often prefers certain choices to others (e.g., a red color for a car to a blue color) or prefers solutions that minimize or maximize certain criteria such as price and quality. In order to provide satisfactory solutions, a configurator needs to address user preferences and user wishes. Another important problem is to provide high-level features to control different reasoning tasks such as solution search, explanation, consistency checking, and reconfiguration. We address those problems by introducing a preference programming system that provides a new paradigm for expressing user preferences and user wishes and provides search strategies in a declarative and unified way, such that they can be embedded in a constraint and rule language. The preference programming approach is completely open and dynamic. In fact, preferences can be assembled from different sources such as business rules, databases, annotations of the object model, or user input. An advanced topic is to elicit preferences from user interactions, especially from explanations of why a user rejects proposed choices. Our preference programming system has successfully been used in different configuration domains such as loan configuration, service configuration, and other problems.

Keywords: Configuration; Constraint Programming; Preferences; Search

1. INTRODUCTION

Many configuration problems require sophisticated knowledge representation formalisms such as description logic, rules, and constraints, as well as engines that support interactive problem solving including domain reduction, explanation, and repair. This approach has led to the success of web-based configurators and sales configurators where end users can enter their requirements and ask for explanations if the engine cannot meet them.

In spite of these achievements, we argue that certain problems can be encountered when using an uncontrolled interactive configuration approach. Configuration problems often involve large product catalogs, and the given user requests can be met by many different kinds of parts from this catalog. Hence, configuration problems are often weakly constrained and have many solutions. However, many of those solutions may be discarded by the user as long as more interesting solutions are possible. The user often has preferences about the choices and the criteria, for example, preferring pink cars to green cars and luxury cars to economical models. Furthermore, a user may have certain desiderata or wishes that should be satisfied, if possible, for example, the desire to have a CD player in the car if this is possible for the chosen model. Configurations that do not respect these preferences and wishes may immediately be rejected by the user.

Another problem where preferences are helpful is search control. If the component to be configured consists of many parts having many attributes, then the order in which the parts are assembled, classified, and parameterized is important to obtain good search behavior. If decisions are made in an inappropriate order, the resulting problem can become over-constrained, which easily leads to thrashing behavior (i.e., behavior where the same decision is tried again and

Reprint requests to: Ulrich Junker, ILOG SA, 1681 Route des Dolines, 06560 Valbonne, France. E-mail: ujunker@ilog.fr

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

again because backtracking does not remove the reason for the failure). For example, it is a very bad idea to first choose the value of defined criteria such as the price of the resulting configuration. In certain cases, a well chosen order can even lead to backtrack-free search behavior.

In this article, we argue that configurators can provide better problem solving behavior and provide better solutions for the end users if an additional concept, namely preferences, is taken into account. Preferences are complementary to rules and constraints, and they permit us to represent additional knowledge. As observed by Jon Doyle (2002), preferences can play different roles, depending on the concrete task: they can be used to determine preferred solutions respecting given user wishes and preferences, but they can also control reasoning and action. In configuration, we can use preferences to control the different reasoning tasks of the configuration engine such as search, explanation, consistency checking, and reconfiguration. In this case, preferences provide a convenient means to constrain the order between the decisions made by the configurator.

Preferences have been studied in decision theory for a long time, and there are numerous numerical methods for treating them, most using additive utility functions. However, users often specify their preferences in a qualitative form (e.g., in the form of preferences between individual choices) and not in the form of quantitative utility functions. Recent work in artificial intelligence (AI) has brought up new formalisms that allow us to directly represent and reason with qualitative preferences. There are logic-based approaches (Brewka, 1994; Delgrande & Schaub, 2000), graphical approaches (Boutilier et al., 1997; Domshlak et al., 2001), and soft constraints (Bistarelli et al., 1999), to name some good examples.

Based on this research, we introduce a system for preference programming in an existing constraint-based configuration tool (ILOG, 2002). We enrich the constraint language by new primitives for expressing decisions and preferences. These primitives can be combined with other constraints and, for example, used inside rules. Preference programming allows us to treat user preferences, user wishes, and search strategies in a declarative and unified way. A common preference language is used differently by a task-specific problem solver:

- A *preference satisfier* determines preferred solutions respecting the given user preferences and wishes.
- *Preference-based configuration strategies* configure the parts and attributes of a system in an order specified by preferences between the attributes and on the possible attribute values.
- A *preference-based explainer* finds explanations containing the most important constraints according to given preferences between the constraints.

Preference programming is completely open and dynamic. Preferences and decisions can be added at any place and can be assembled from different sources of information. Parts of them can be specified in the form of business rules, together with other configuration rules and constraints. They can also be read from databases or be assembled from annotations of the object model. Certain preferences can immediately be selected by the user. An interactive configurator can even elicit preferences from user interactions, in particular if a user criticizes certain choices of the configurator.

In this way, preference programming improves the ease of use of configuration technology and permits the application developer to write complex configuration strategies without needing any special skills or experience in constraint and search programming. The business expert who is maintaining the configuration knowledge base can adapt the strategies to changing catalogs or sales policies by modifying the business rules and object annotations that state preference information. Thus, preferences provide a high-level form of search programming, which is an alternative to traditional search programming such as PROLOG clauses, search goals as in ILOG SOLVER (Puget, 1992), or nondeterministic procedures as in OPL (van Hentenryck & Puget, 2000). These traditional forms are very powerful, but not very well adapted to configuration specialists.

Finally, the preference programming approach is well adapted to interactive configuration where the limited response time does not always allow us to find the best solution for the user. The preference satisfier performs nested branch and bound searches for treating user preferences. If time is limited, it will not find the best choices but will find good approximations for all of the criteria, not for only some of them.

We explain the features of preference programming by using a vacation adviser example. This example is easy to understand and has a structure that is representative for many configuration problems: it involves a part catalog, a part-of hierarchy, user requirements, compatibility tables, and resource constraints. Furthermore, the vacation domain is ideal to motivate and illustrate different kinds of preferences.

This article is organized as follows: we introduce the vacation example in Section 2 and show which kinds of preferences can be useful for solving it. In Section 3 we introduce the basic concepts of our preference programming approach and show how this basic machinery treats structured, dynamic, and inconsistent preferences. For configuration problems, we need higher level forms of preferences and decisions, which are presented in Section 4. In particular, we discuss how configuration strategies are built for given configuration problems and how user preferences and wishes are treated efficiently. We also describe a mapping of the high-level primitives to the basic concepts. Section 5 discusses how preference programming can be used for different types of problems, ranging from consistency checking to automated problem solving and advanced forms of user interaction. Finally, in Section 6 we summarize ex-

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

periences from real-world applications, such as loan configuration and service configuration.

2. A CASE STUDY: PREFERENCES IN A VACATION EXAMPLE

In this section, we introduce a didactic vacation advisor problem that represents a simplified version of travel configuration problems (SABRE, 2002; Torrens & Faltings, 2002) and that allows us to illustrate preferences in a natural way.

2.1. Description of the vacation example

The problem consists of configuring a vacation package composed of a set of destinations. For each destination, a set of activities involving sports and visits has to be selected. All the selections are constrained by the period when the travel is scheduled, the level of quality expected by the customer, the budget, and the global purpose of the package (relaxation, culture, or recreation). Depending on the season, some destinations are possible or not. Depending on the season and on the destination, some activities are impossible and the price and the quality will be different. A catalog of destinations and activities with their respective compatibilities, prices, and levels of quality is available in the form of an external database that is loaded dynamically when the vacation advisor server application is launched. Table 1 gives an example of a compatibility table, which will be used in Section 3.

In our model, we have defined one subclass for each valid combination of destination and season. Each such combination determines the values for the price and quality attributes. For example, we find Majorca/summer, but not Majorca/winter. Furthermore, Florence/winter is cheaper than Florence/summer.

This example can easily be turned into a real-world application by adding more components. The real-world application should furthermore allow the selection and configuration of an accommodation for each stop and also permit the selection of flights or trains to travel from one stop to the next one. For the sake of simplicity, we omit these issues in this paper.

Table 1. *Example of compatibility table*

Destination	Visits	Sports
Hawaii		Wind-surfing, hiking
Cote d'Azur	Museums	Hiking, wind-surfing, canyoning
Florida	Attraction park	Wind-surfing
Greece	Museums	Wind-surfing
Bavaria	Museums, Medieval castles	Hiking, canyoning

Figure 1 shows the object model of the problem in the form of a UML diagram. Depending on the purpose of the package, some activities are automatically selected or removed. For example, if the purpose is "relaxation," all sports are removed from the set of activities, and if the purpose is "culture," at least one visit to a museum or medieval castle is selected per stop.

The vacation example has been completely formulated in a constraint-based configuration tool (ILOG, 2002).

2.2. Preferences for the vacation example

A catalog of vacation destinations usually contains many possibilities, and user preferences are important to choose among them. The following kinds of user preferences and wishes are natural:

- preferences based on numerical criteria, such as the total price, the average quality, or the maximum distance to a destination. People may want to minimize price, maximize quality, and/or minimize distance.
- preference about attributes with a discrete domain, such as the activities. For example, the user may prefer visits to attraction parks to visits to museums. Furthermore, a user may prefer canyoning to wind-surfing.
- wishes, such as spending at least one night in a luxury hotel.

Preferences among multiple criteria can easily be in conflict, meaning that it is not possible to choose the best value for all criteria simultaneously. In this case, preferences between criteria and wishes specify how to resolve those conflicts:

- The user prefers maximizing the quality to minimizing the price of the vacation.
- The user wants to spend at least one night in a luxury hotel even if the total price is minimized.

These user preferences and wishes can either be specified individually or be generic and depend on the characteristics of a user or the vacation:

- For a student, minimizing price is more important than maximizing quality. For a manager, the inverse is the case.
- If the purpose of the vacation is cultural, then the user would like to visit a museum in each vacation stop, if this is possible.

Although the class diagram of the vacation example is not very complex, it is still necessary to make choices in an adequate order when configuring a vacation package. The following preferences should be respected by the search procedure:

- The activities of the vacation package should be chosen completely before assignment and configuration of the vacation stops.

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

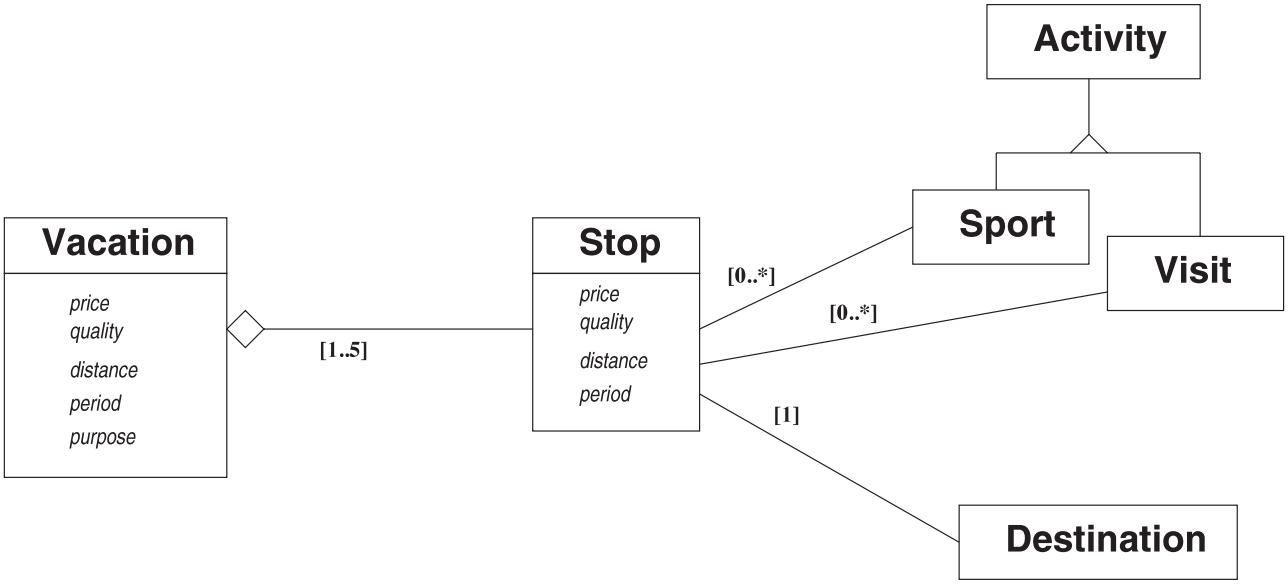


Fig. 1. A vacation adviser class diagram.

- The activities of a vacation stop should be chosen before its type and destination, because a given combination of activities usually permits only certain destinations.

These preferences influence the order in which attributes are assigned. There are also preferences among the different values of an attribute:

- If the total price should be minimized, then cheaper hotels should be considered first.
- The strategy should assign as many activities to a vacation stop as possible.
- There may also be dynamic preferences. For example, the user may prefer Hawaii to Florida if wind-surfing has been selected for a vacation stop.

This list shows that user preferences as well as preferences for search control arise naturally in this domain.

3. PREFERENCE PROGRAMMING: BASIC CONCEPTS

In this section, we introduce decisions and preferences and show how the preferences control the decision-making process that is performed by search procedures. We also discuss structured, dynamic, and inconsistent preferences.

3.1. Decisions

Traditionally, search procedures such as A^* define search as a method for exploring a state space consisting of a set of states and their successor relation. Such a search procedure does not make any assumptions about the structure of the state and is thus quite general. In particular, a search state does not reflect knowledge about the choices or decisions that need to be made to produce the state. A decision corresponds to a choice a search procedure makes when moving from one state to a successor state. Representing states as sets of decisions has several advantages. For example, the order of decisions can easily be controlled, and it becomes possible to determine conflicting decisions if the search procedure runs into a dead end. This alternative way of representing search states has therefore been adopted by approaches based on truth-maintenance systems.

To solve configuration problems, a configurator typically tries out different value assignments to attributes and relations of components. Hence, it adds assignment constraints to the original problem. In order to describe this process, we consider decisions based on constraints. Each decision has an associated constraint and will be explored by a specific search procedure in a task-specific way. For example, a configuration strategy will create two successor states when exploring a decision. In the left successor state, it adds the constraint of the decision, whereas it adds the negation of the constraint in the right successor state. Typically, we will use chronological backtracking to explore the resulting search tree. Hence, the search procedure will first explore the left successor and its subtree. If this subtree does not contain a solution, it backtracks to the right successor state.

Formally, a *decision* is specified by an identifying term t (e.g., a name) and a constraint ϕ. It is written in the form `decision` t:ϕ;. We can also consider decisions as named or labeled 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.

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

The constraint ϕ may not only be a value assignment form $f = v$ where f is an attribute or relation[1] but also an arbitrary constraint formulated in a standard constraint-programming language. One of the key ideas of constraint programming is that arbitrary constraints can be added to the given constraint problem during search. This addition has no critical impact on the complexity of a constraint satisfaction problem because this complexity depends mainly on the number of constraint variables (or attributes and relations, in the case of configuration) and the size of their domains.

In our example, we model the selected visits and sports by two set variables (i.e., constrained variables that are assigned a set of values instead of a single value). In order to select visits to museums, we add the membership constraint `member (Museums, visits)` to the configurator. For each possible visit and sports activity, we introduce a decision consisting of such a membership constraint and a name:

```
decision Museum-d: member(Museums,
  visits);
decision Attraction-d: member(Attrac-
  tions, visits);
decision MedievalCastle-d: member(Medi-
  evalCastles, visits);
decision Hiking-d: member(Hiking,
  activities);
decision Canyoning-d: member(Canyon-
  ing, activities);
decision WindSurfing-d: member(WindSurf-
  ing, activities);
```

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

3.2. Decision-making process

A search procedure explores the given decisions one after the other. We can also say that it determines a total order among the decisions. This order greatly influences the problem-solving behavior and is an important part of a search strategy. The order may be static (i.e., specified initially) or dynamic (i.e., depending on decisions that have been executed successfully). In this section, we make this order explicit, a tactic that allows us to influence it by constraints.

Suppose our configuration problem has a set $\mathcal{D}$ of n decisions. We can define the order in which the decisions are explored by assigning a position $\pi(\delta)$ to a decision $\delta \in \mathcal{D}$. The ith decision obtains the position i. Hence, π is a mapping from $\mathcal{D}$ to $\{1,\ldots,n\}$. Different decisions obtain different positions:

$$\pi(\delta_1) \neq \pi(\delta_2) \quad \text{if} \quad \delta_i \neq \delta_j. \tag{1}$$

Initially, the $\pi(\delta)$ values are not known, but their possible domain is reduced as the search occurs. After exploring the kth decision, the unexplored decisions can only get the positions in $\{k+1,\ldots,n\}$. Conceptually, we can consider the $\pi(\delta)$ values as constrained variables, a consideration that enables us to formulate constraints on the order in which decisions are explored. As a consequence, the search procedure cannot assign any arbitrary position to any arbitrary decision, but must make assignments in a way that respects these constraints.

In the ith step, the search procedure has already assigned the position 1 to $i-1$ to decisions. It now selects one of the decisions that can have position i and explores it. This exploration can occur in various ways and depends on the concrete task:

1. *Search strategy:* When a decision δ is being explored, a (binary) choice point is set up. In the left branch, the decision δ is added to the current set $\mathcal{C}$ of constraints and we say that the decision has succeeded. In the right branch, the negation of δ is added to $\mathcal{C}$ and we say that the decision has failed. In both branches, constraint propagation is applied after adding a constraint. If the propagation fails, backtracking occurs.
2. *Preference Satisfier:* When exploring a decision δ, we check whether it is consistent w.r.t. $\mathcal{C}$ (i.e., whether there exists a solution). If this is the case, the decision δ is added to $\mathcal{C}$ and we say that it has succeeded. Otherwise, it is dropped. Hence, decisions are tried out in increasing order, and only successful decisions are added to $\mathcal{C}$. In this approach, decisions are made whenever possible. In this respect, they behave exactly as defaults.

The first procedure assigns the positions in a greedy fashion and uses them to define a complete search tree that allows us to enumerate all solutions. Because the values of the positions are chosen dynamically, they can differ in different branches of the search tree.

The second procedure considers alternative values for the positions $\pi(\delta)$ (i.e., alternative decision orders) but executes the decisions in a greedy fashion once these positions are determined. We can enumerate all solutions 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. At first glimpse, it seems that the basic complexity of this approach is much worse than that of the first approach ($O(n!)$ orders instead of $O(2^n)$ leaves in the search tree), although fewer solutions may be obtained if the order is constrained. Indeed, many orders produce the same solu-

[1] Such a value assignment constrains f to have the value v and must not be confused with an assignment statement in an imperative programming language.

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

tions. Junker (2000) showed how irrelevant orders are avoided and how the effort can be reduced to a binary tree of depth n. Furthermore, it is possible to prune certain parts of the search tree that do not contain preferred solutions.

The basic idea of this preference-based search (PBS) is as follows: PBS searches through a binary search tree as a complete search. In each search node, a decision δ is explored. In the left subtree, the constraint ϕ of the decision is added to the current set of constraints and the current position k is assigned to δ. Instead of adding the negation of ϕ in the right subtree, PBS requires that ϕ is refuted by subsequent decisions. If the negation of ϕ can indeed be deduced in a subsequent step $k + l$ (where $l \geq 1$) then δ is explored in this state and obtains the position $k + l$. However, if PBS reaches a state such that no further decision can be selected (i.e., δ is preferred to all remaining decisions in this state) and ϕ is consistent in this state, then PBS cannot extend the current state to a preferred solution and backtracks. We refer the interested reader to Junker (2002) for a detailed description of the PBS algorithm that is used by the preference satisfier.

We give a simple example for determining a preferred solution. Suppose the set $\mathcal{C}$ contains the compatibility constraint between the five vacation destinations and the six vacation activities that was introduced in Section 2. As long as there are no constraints on the order of decisions, we can introduce them in any order. Once the order of decisions is determined, we explore the decisions in a greedy fashion. The first decision `Attraction-d` succeeds and allows Florida as single destination. Florida is compatible only with attractions and wind-surfing. Hence, only those activities succeed:

1	Attraction-d	succeeds
2	Canyoning-d	fails
3	Museum-d	fails
4	MedievalCastle-d	fails
5	Hiking	fails
6	WindSurfing-d	succeeds

3.3. Preferences as constraints on the decision order

Preferences between decisions are a convenient way to specify which decisions should be kept if several decisions are in conflict. More preferred decisions are kept, whereas the less preferred decisions may be removed in case of such a conflict. Following (Brewka, 1989; Junker, 1997), we use preferences of the form `prefer`(t,u) as constraints on the order of decisions. If the term t refers to the decision δ_t and the term u refers to the decision δ_u, then a preference imposes the following constraint on the positions of the decisions:

$$\text{if } \texttt{prefer}\ (\delta_t, \delta_u), \quad \text{then} \quad \pi(\delta_t) < \pi(\delta_u).$$

In our example, we use the following preferences:

```
prefer(MedievalCastle-d, Attraction-d);
prefer(Attraction-d, Museum-d);
prefer(WindSurfing-d, Canyoning-d);
prefer(WindSurfing-d, Hiking-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. When we use the approach of a search strategy, any order is able to produce any solution. For a preference satisfier, preferences that are implied by a solution eliminate certain orders. If we can find an admissible order such that the preference satisfier produces X, then X is called a *preferred solution*. Junker (1997) showed how preferred solutions can be constructed incrementally.

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

Bavaria

1	MedievalCastle-d	succeeded
2	WindSurfing-d	failed
3	Attraction-d	failed
4	Museum-d	succeeded
5	Hiking-d	succeeded
6	Canyoning-d	succeeded

Cote d'Azur

1	WindSurfing-d	succeeded
2	Canyoning-d	succeeded
3	Hiking-d	succeeded
4	MedievalCastle-d	failed
5	Attraction-d	failed
6	Museum-d	succeeded

If a new constraint is added, preferred solutions can become inconsistent and nonpreferred solutions can become preferred ones. For example, if Bavaria is not possible (e.g., overbooked), then our first solution becomes inconsistent and a new preferred solution is obtained. Hence, additional constraints can change the set of solutions in a nonmonotonic way:

Florida

1	MedievalCastle-d	failed
2	WindSurfing-d	succeeded
3	Attraction-d	succeeded
4	Museum-d	failed
5	Hiking-d	failed
6	Canyoning-d	failed

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

Cote d'Azur

1	WindSurfing-d	succeeded
2	Canyoning-d	succeeded
3	Hiking-d	succeeded
4	MedievalCastle-d	failed
5	Attraction-d	failed
6	Museum-d	succeeded

3.4. 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 the idea that all visit decisions are preferred to all sports decisions. In order to express this idea, we introduce two groups of decisions. A group of decisions is specified by providing a unique identifying term (e.g., a name):

```
group(sports-g)
group(visits-g)
```

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

```
contains(visits-g, Museum-d)
contains(visits-g, Attraction-d)
contains(visits-g, MedievalCastle-d)
contains(sports-g, Hiking-d)
contains(sports-g, Canyoning-d)
contains(sports-g, WindSurfing-d)
```

We can now introduce preferences between two decision groups g_1 and g_2, meaning that all elements of g_1 are preferred to all elements of g_2:

```
prefer(visits-g, sports-g);
```

We have thus expressed the following preferences in a more compact form.

```
prefer(Museum-d, Hiking-d);
prefer(Attraction-d, Hiking-d);
prefer(MedievalCastle-d, Hiking-d);
prefer(Museum-d, Canyoning-d);
prefer(Attraction-d, Canyoning-d);
prefer(MedievalCastle-d, Canyoning-d);
prefer(Museum-d, WindSurfing-d);
prefer(Attraction-d, WindSurfing-d);
prefer(MedievalCastle-d, WindSurfing-d);
```

If the first group has n_1 elements, and the second group has n_2 elements, we thus replace $n_1 \cdot n_2$ preference statements by $n_1 + n_2$ containment statements, two groups, and a single preference statement. If the groups are large, this replacement leads to large gains in the size of the preference specifications.

Decision groups can be arbitrarily nested, a fact that allows us to formulate strategies for configuring components. In Section 4.3, we will map each component to a decision group, a mapping that easily permits us to influence the order in which those components are configured.

3.5. Dynamic preferences

Configuration sales strategies can depend on the customer profile, the time period, special offers, and the available stock. Hence, they lead to preferences that are context dependent and that will be activated dynamically during the decision-making process if certain conditions are satisfied. Because preferences are constraints, we can represent these dynamic preferences by logical implications of the form

if *condition* then *preference*

We can distinguish dynamic preferences that depend on data and dynamic preferences that depend on other decisions. Preferences that depend on data can be completely determined in the root state of the search tree, that is, before the configurator makes any decisions. Preferences that depend on other decisions are generated during search and are effective only if they concern decisions that have not been explored so far.

We consider some examples of dynamic preferences. Depending on the purpose of the vacation, the customer may prefer visits to sports activities or vice versa:

- If the purpose is culture, then prefer visits-g to sports-g.
- If the purpose is recreation, then prefer sports-g to visits-g.

Because the purpose is given initially, these preferences depend only on data.

We also give an example of preferences that depend on decisions. For this purpose, we additionally introduce decisions for choosing the destination, a choice which is necessary if the same sets of activities are possible at different destinations. For each possible destination, we suppose that there is a decision such as

```
decision Florida-d: destination =
  Florida;
```

Furthermore, we suppose that all these additional decisions have been added to a decision group called

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

destination-g which is considered after the visit group and the sports group. This arrangement is made sure by the following preferences:

```
prefer(visit-g, destination-g);
prefer(sports-g, destination-g);
```

We are now able to state that Hawaii is preferred to Florida if wind-surfing has been selected:

```
if member(wind-surfing, sports) then
  prefer(Hawaii-d, Florida-d)
```

Because sports activities are chosen before the destination, this dynamic preference is generated before the destinations are chosen. In fact, the preferences between the sports group and the destination group make sure that this dynamic preference is only applied to decisions that are not yet explored when the dynamic preference is generated.

3.6. Inconsistent and metapreferences

The preferences that are obtained from the various statements given above risk being 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(sports-g, visits-g);
prefer(visits-g, destination-g);
prefer(destination-g, sports-g);
```

If the number of decisions is finite, then inconsistent preferences are obtained only if there are cyclic preferences. In all other cases, it is possible to assign a position number to the unexplored decisions that respects the preferences. In contrast to preferences, decisions with an inconsistent constraint ϕ do not provide any problem since they immediately fail when being explored by the search procedure. As a consequence, the search procedure backtracks to the previous search state and thus removes the inconsistent constraint.

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

```
decision sports-preference:
  if the customer likes sports
  then prefer(sports-g, visits-g);
decision culture-preference:
  if the purpose of vacation is culture
  then prefer(visits-g, sports-g);
```

Then metapreferences can be used to resolve potential conflicts between those preference rules:

```
prefer(sports-preference,
  culture-preference);
```

It is important that a decision d that consists of a preference between two decisions d_1 and d_2 has a higher priority than the decisions d_1 and d_2. A simple way to achieve this condition is to create two decision groups g^p and g^d, where the first one contains the preference decisions and the second one contains the decisions to which the preferences are applied. Furthermore, we specify that g^p is preferred to g^d by a hard preference.

Thus, metapreferences can control which elements in a set of cyclic preferences can be retracted and which should be kept. Future work is needed to investigates which cases can be adequately treated by this approach and which cases may require alternative approaches such as removing all the preferences that are in conflict.

4. PREFERENCES PROGRAMMING FOR CONFIGURATION PROBLEMS

In the last section, we introduced the basic preference programming features and now we investigate how to treat user preferences and provide declarative search programming for configuration problems. For treating those problems efficiently, we will enhance our preference-programming language by higher level primitives and exploit the structure of configuration problems when formulating preferences.

A configuration problem consists of selecting and arranging parts from a given catalog such that given user requirements as well as generic constraints on the parts are satisfied (Mittal & Frayman, 1989). Configuration problems have several particularities that have an impact on decisions and preferences:

- *Knowledge maintenance* requires high-level, declarative, and object-oriented knowledge-representation languages because configuration knowledge (e.g., part catalog) changes frequently. A language for preference representation needs to be high level and declarative as well such that a domain expert can adapt it to changing sales and configuration strategies.
- *Knowledge representation* requires that preferences and decisions expressed for different classes are inherited via a taxonomy by the objects to be configured. Furthermore, decisions need to be able to explore a part-of hierarchy recursively.
- *Dynamic object generation and classification* require dynamic generation of decisions and preferences to configure newly inherited attributes and newly generated objects.
- *Multiple configuration tasks* such as consistency checking, configuring, explaining, optimizing several criteria,

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

and repairing all need to be supported by a single engine and ask for a common framework for decisions and preferences.

4.1. Configuration language

In order to state configuration problems, we need a formalism for knowledge representation that is sufficiently rich to support composed components as instances of classes and concepts as well as rules and constraints (Soininen et al., 2000). In the literature, different approaches for solving configuration problems have been developed. Most approaches are based on or evolved from a single knowledge representation paradigm (e.g., description logic, rules, logic programming, constraint satisfaction, or constraint programming), but have been extended in order to cover the full range of configuration knowledge. In this paper, we focus our discussion on a constraint-based configuration approach that has been extended by an object language and logical rules formulated on constraints, following the line of Haselböck and Stumptner (1993) and Mailharro (1998). However, parts of our discussion may also be relevant for other approaches that support composed components, rules, and constraints.

We briefly outline our configuration language (ILOG, 2002) before discussing which form of decisions and preferences are needed for it. The language supports classes, objects, attributes, relations, expressions, constraints, and rules. Given concrete classes can be organized as subclasses of mutually exclusive abstract classes, an organization that leads to a forestlike class taxonomy. This class hierarchy is used in two ways. On the one hand, it defines the domain for unknown component types. On the other hand, it defines the inheritance scheme for the instances of classes.

Each class can have a set of field declarations consisting of a field name and a domain. The domains can be refined by subclasses. Fields can represent attributes (e.g., in the case of a string, integer, or floating-point domain) or relations between the given class and a target class. Relations include assignment as well as partonomic relations. We distinguish fields containing individual parts and fields containing anonymous instances of a class. In the first case, the field value is a single instance of the target class or a set of such instances. In the second case, the field value is a concrete subclass of the target class or a set or bag of those subclasses.

Components are represented by objects that are instances of classes and that inherit attributes and relations from them. Instances of abstract classes can be classified dynamically to a concrete class. This process will complete the instances by the additional fields of the subclass and impose the domain restrictions of the subclasses. In order to state constraints on arbitrary instances of a class, we introduce logical variables for objects.

There is a rich layer to formulate expressions, constraints, and rules. Expressions are of different types, namely integer, floating-point, object, object-set, class, class-set, or class-bag. There are constant expressions (e.g., integer number, floating-point number, objects, classes); field expressions of the form $f(e)$ or $e.f$, where f is a field and e is an object expression; arithmetic expressions, such as $e_1 + e_2$; set expressions; and sophisticated expressions over sets, such as sum-over-set. Constraints cover comparison expressions, set operators, compatibility, exclusion, requirement constraints, and Boolean combinations of constraints (including negation and implication). Logical rules can be formulated as universally quantified constraints of the form `forall` x `in` C:$\phi(x)$ where x is a logical variable, C is a class, and $\phi(x)$ is a constraint containing x as a free variable.

4.2. Which preferences are needed for configuration?

We now investigate how to apply decisions and preferences to configuration problems. As mentioned in the introduction, we want to use preferences for different aspects of configuration problems, such as user preferences and search control. User preferences about choices, such as the destination of a vacation, or about defined criteria, such as the total price of the vacation, are an adequate way to pick out interesting solutions among an often very large solution set. Similarly, user wishes or desiderata represent conditions that should be satisfied whenever possible.

A second application of decisions and preferences consists of controlling the search process in a user-friendly way. In order to find a configuration, a search procedure needs to determine the types and field values of all relevant components. It usually starts to configure a root component by choosing its type and its field values. This process is repeated recursively for each subpart. The resulting search behavior rather depends on the form of the search decisions that are explored and on the order in which they are made. If chronological backtracking is used in a systematic tree search, then a bad search order can easily lead to thrashing. In our vacation example, it is important to choose the type of a vacation stop before making any decision about its price. Furthermore, a bad search order can produce very bad solutions first. For example, if we want to minimize price, the search strategy should choose cheaper hotels first instead of considering first-class hotels.

In addition to this consideration, a configuration strategy needs decisions that follow the partonomic structure of a configuration and that take into account parts that are generated dynamically. Hence, strategies need to be applied to an unknown number of instances of a given class. Moreover, strategies need to be open and to cope with fields that are dynamically added to a component when it is classified.

Although the basic preference programming layer of Section 3 could, in principle, be used to formulate user preferences and search strategies, it would not be efficient in time and space. For example, the strategy would need $n := f \cdot d$ decisions for f fields and d values in the field domains, a

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

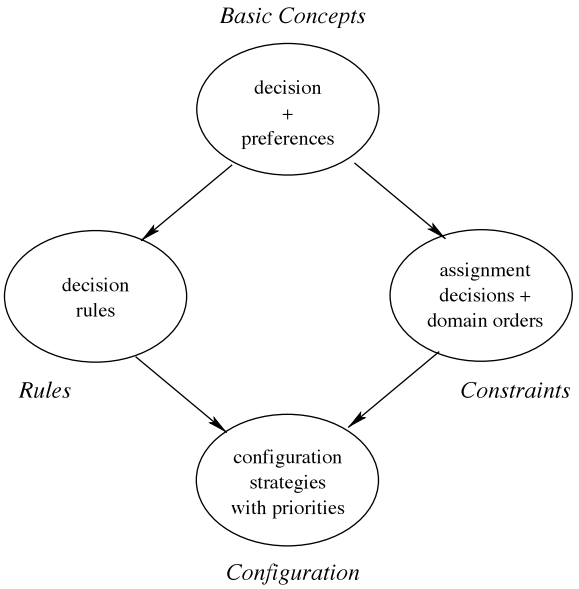


Fig. 2. The preferences for configuration.

need that is not affordable for realistic problems. To handle this situation, the basic idea is to introduce higher level primitives that behave as an aggregate of basic decisions and preferences but have internally more efficient data structures and efficient algorithms. We thus follow the example of constraint programming, where global constraints, such as `allDiff`, combine a clean modeling facility with an efficient algorithm from graph theory.

In order to produce these high-level primitives of preference, we exploit the different forms of configuration knowledge, such as objects, constraints, and rules. Figure 2 shows this plan. We first apply the concepts of decisions and preferences to constraint programming. Each field has a domain, and it is natural to represent an order for the domain values in a compact form. Similarly, we represent all search decisions for assigning a value to this field in the form of a compact decision set. For example, we generate the set of assignment decisions for the destination field of a stop-object by the statement `assign(Stop.destination)`. A decision set is similar to a decision group, although its elements are not added explicitly but are generated implicitly when needed. For example, a decision

```
decision d(Stop.destination, Hawaii):
  Stop.destination = Hawaii
```

is generated if Hawaii is a best value w.r.t. a given domain order. Domain orders can be specified by priorities of values or by preferences between values, as illustrated in the following examples for the domains of visit and sports activities[2]:

```
priorityOrder(visitsOrder, VisitDomain);
priority(visitsOrder, MedievalCastle, 1);
priority(visitsOrder, Attraction, 2);
priority(visitsOrder, Museum, 3);
partialOrder(sportsOrder, SportDomain);
prefer(sportsOrder, WindSurfing,
  Canyoning);
prefer(sportsOrder, WindSurfing, Hiking);
```

There are also predefined domain orders, namely, the decreasing and increasing orders of integers. We directly use the field as the identifier for the decision set. Thus, we can easily reference the decision set in `preference` and `contains` statements. We simply write `prefer(Stop.destination, Stop.price)` to indicate that the destination of the stop needs to be chosen before its price.

The next step is to assemble these elements in order to get a complete configuration strategy for composed components. In the next section, we discuss in detail how the structure of components is exploited to set up configuration strategies. We also introduce primitives for configuring components as soon as they are assigned to the current configuration.

Finally, we also apply the concepts of decisions and preferences to rule programming, as indicated in the left branch of Figure 2. Logical rules of the form `forall` x `in` C:$\phi(x)$ are true for all instances of C. However, rules can also be used to express user wishes that can be satisfied for some instances, but not for all. For example, the application developer may write a rule stating that each vacation stop should include a visit to a museum. This rule may succeed for some destinations, but not for those that have no museums. We call those rules decision rules.

4.3. Configuration strategies

In this section, we discuss how a configuration strategy can be built upon decisions and preferences. For this purpose, we exploit the structure of the given configuration problem.

In order to determine a configuration, a search strategy typically starts to configure a root component and then follows a part-of hierarchy or other relations between components. When configuring a component, it tries out different value assignments to the fields of the component, including an implicit classification field. We can model such a strategy as follows:

1. For each object i, we introduce a decision group `group`(i). All decisions needed for configuring this object are contained in its decision group. As the identifier for the decision group, we directly use the object i, which allows us to refer to the group in containment and preference statements. For example,

[2] More sophisticated forms of domain orders (e.g., mixing priorities and preferences) will be examined in the future.

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

the order for configuring objects can be influenced easily in this way.

2. For each field f of the object i, we add a decision set of the form `assign`(f) or `assignElements`(f) to the group i.
3. In order to influence the order of the decisions, we apply a domain order o to a field f by using a statement of the form `preferValues`(f,o). Furthermore, we can influence the order in which fields are configured by preference statements between fields.

Typically, such a strategy will not be formulated for individual objects, but for arbitrary instances of a class. This formulation can easily be achieved by rules, such as the following strategy for instances of vacation stops:

```
forall X in Stop:
  contains(X,assignElements(X.visits))
    and
  contains(X, assignElements(X.sports))
    and
  contains(X,assign(X.destination)) and
  contains(X, assign(X.classification))
    and
  prefer(X.visits, X.destination) and
  prefer(X.sports, X.destination) and
  prefer(X.destination, X.classifi-
    cation);
```

Preferences about field values may be parameterized independently of the basic strategy by formulating separate rules:

```
forall X in Stop:
  preferValues(X.visits, visitOrder) and
  preferValues(X.sports, sportsOrder);
```

We thus obtain a convenient way for formulating strategies for configuring an object. However, one element is missing. When assigning objects to selected relations, the strategy needs to configure the assigned objects as well. We achieve this behavior by using modified versions of the decision sets `assign`(f) and `assignElements`(F). For example, when we assign vacation stops to a vacation object, these stops need to be configured as well. In the following strategy, the configureElements statement will add the decision groups of assigned vacation stops, an addition that means that they will be configured as well:

```
forall X in Vacation:
  contains(X, assignElements(X.visits))
    and
  contains(X, assignElements(X.sports))
    and
  contains(X, configureElements(X.stops))
    and
  prefer(X.visits, X.stops) and
  prefer(X.sports, X.stops);
```

We are now able to formulate complete configuration strategies sophisticated enough to allow us to configure complex components. There is also a user-friendly way to specify those strategies. The user only needs to indicate how the strategy should traverse a given class diagram by specifying field priorities and by selecting object and object-set fields for which the elements should be configured as well. The field priorities will be used to influence the order of the fields inside an object group. The other information, namely, decision sets and domain orders, is completed by default choices or can be specified differently.

This approach for treating configuration strategies has several advantages for configuration. First of all, the approach is *user friendly*. It is very easy to specify or parameterize such a strategy in terms of field priorities. Hence, the application developer can program complex search strategies without needing special skills in constraint and search programming. Second, the approach is *open*: new decisions can be inserted at any place in the object-oriented decision structure. New preferences between fields can be added easily. Hence, the configuration strategies are well adapted to configure fields and objects that do not exist initially but are generated during search. Third, it is possible to specify the search strategy in the form of rules, as shown above. The application developer can write rules that complete information depending on the given context. For example, the domain order for choosing the destination of a vacation can depend on the chosen activity. We offer such a *rule-based search-programming* facility in a more high-level business-rule language. Finally, complex multiphase strategies can be obtained easily. We introduce multiple strategies and encapsulate each of them inside a decision group.

4.4. Wishes and user preferences

Many configuration problems, such as product and service configurations, involve large catalogs and usually have many solutions. In order to find configurations that are interesting for the user, a configurator has to take into account user preferences and wishes (or desiderata). For example, the user might prefer canyoning to wind-surfing and hiking. The user may also prefer cheaper vacations and may have desiderata or wishes such as visiting an attraction park in each stop. Multiple preference criteria, such as minimizing price and maximizing quality, may be in conflict. In such a case, the user may specify which criterion is more important by specifying a preference between them.

It is interesting that we can already use our abstract preference programming language to write down preferences about and between criteria. We use the same syntax as for

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

search strategies, namely, decision sets and domain orders, but change their search behavior. For example, we formulate user preferences such as minimizing the price and maximizing the quality of `myVacation` as follows:

```
assign(myVacation.price);
assign(myVacation.quality);
preferValues(myVacation.price,
  minFirst(priceDomain));
preferValues(myVacation.quality,
  maxFirst(qualityDomain));
```

These statements are now created with the help of a preference satisfier instead of a configuration strategy. Instead of trying out one value assignment after the other (e.g., to compute the price), the preference satisfier determines the best value assignment by solving an optimization subproblem. This subproblem can, for example, be solved by a constraint-based branch and bound procedure that uses the corresponding criterion as an optimization objective. Once the best value v is found for the price, we create a decision assigning v to the price. This decision is either executed (meaning that the price is constrained to v) or needs to be refuted by first optimizing other criteria, such as the quality. We thus obtain extreme solutions, where either the price is minimized first and then quality is maximized or the quality is maximized first and the price is minimized as a second criterion. Preferences between criteria constrain those orders and state which criteria are more important. These preferences may depend on given characteristics of the user, such as the profession:

```
for x in Vacation:
  if x.profession = student then
    prefer(x.price, x.quality);
for x in Vacation:
  if x.profession = manager then
    prefer(x.quality, x.price);
```

For a student, the price will always be minimized as the first criterion in this case. For a manager, only quality can be maximized first. Following the approach in Ehrgott (1997) and Junker (2002), it is also possible to find compromises with such a lexicographical approach, assuming that the criteria are expressed on the same scale. This kind of compromise is achieved by minimizing the maximum of two criteria, fixing it to the best value, and then minimizing the minimum of the two criteria.

Individual wishes or desiderata can be directly formulated in our existing preference programming language. For example, the user wants to visit an attraction park if this is permitted by the other criteria. We add a decision for this purpose:

```
decision visit-park: member(Attrac-
  tions, myVacations.visits);
```

In contrast to the configuration strategy, a preference satisfier tries to satisfy this wish whenever possible. Hence, if there is a solution permitting the visit of an attraction park, then it will only produce those solutions. However, a satisfiable wish may be in conflict with other wishes or other criteria. If the preference satisfier treats these other decisions first, then it will not satisfy the given wish. Preferences between wishes and criteria again permit the user to state which of them is more important.

Some wishes apply to arbitrary instances of a class and will be satisfied for some of them, but not for others. In order to express them in our preference programming language, we introduce a decision rule, which has syntax similar to the default rules in Poole (1988). An example is to visit an attraction park at each vacation stop:

```
decision-rule attraction-rule(x) on Stop:
  member(Attraction, visits(x));
```

Instances of logical rules must always be satisfied. In contrast, instances of decision rules are handled as basic decisions. Some of them may be satisfied, whereas others can fail. For example, consider three vacation stops with already chosen destinations, such as Paris, Venice, and Vienna. If Venice has no attraction park but the other two destinations do offer attraction parks, then the second decision will not be satisfied but it will not prevent the satisfaction of the other decisions.

```
decision attraction-rule(Paris):
  member(Attraction, visits(Paris));
decision attraction-rule(Venice):
  member(Attraction, visits(Venice));
decision attraction-rule(Vienna):
  member(Attraction, visits(Venice));
```

The decision rule is a kind of decision group, the elements of which are not explicitly added. In fact, the elements of the decision rule are the decisions created for its instances. A decision rule has a name as identifier (such as `attraction-rule` in the rule above). Instances of decision rules are identified by terms such as `attraction-rule(Paris)`. It is thus possible to refer to decision rules and their instances in containment and preference statements.

4.5. Semantics of high-level configuration preferences

In the previous sections, we introduced high-level primitives for stating decisions and preferences in a compact form and thus obtained an abstract preference programming language. The same language can be used for differ-

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

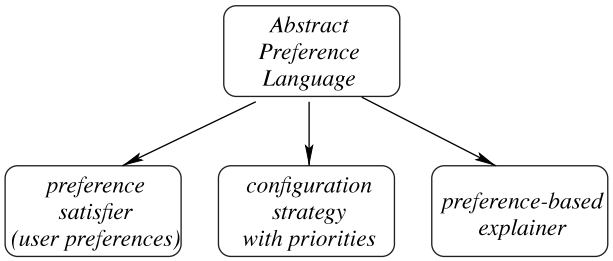


Fig. 3. Preference-based problem solving.

ent tasks, such as satisfying user preferences or controlling search. Figure 3 shows this principle. The tasks themselves differ in the way the basic decisions and preferences interact with the constraint solver. Each task can associate a different search algorithm with the abstract decisions and preferences. Having a common abstract language for preference programming has several advantages: first, a uniform syntax is used; second, the mapping of high-level decisions and preferences is independent of the given tasks; third, new possibilities are obtained (e.g., we can use decision rules as search directives); and fourth, new kinds of tasks can easily be added without requiring the development of a new preference language. Of course, task-specific short-cuts or synonyms can always be added.

In this section, we show how these primitives can be mapped to basic preferences and decisions that are treated as specified in Section 3.2. This translation itself is valid for all tasks. However, tasks determine how decisions are executed, that is, which choice points are generated. The semantics of the basic decisions and preferences is thus task dependent. We now complement the semantics for high-level primitives by describing their mapping to basic decisions and preferences.

We present semantics that interpret the syntactical language constructs by an adequate mathematical structure. This structure consists of several components. First of all, it contains

- a set $\mathcal{D}$ of decisions and
- a binary preference relation $\prec \subseteq \mathcal{D} \times \mathcal{D}$ on decisions

that correspond to the basic decisions and preferences we want to obtain in the end. Decision rules, decision groups, and decision sets all regroup several basic decisions. Therefore, we interpret such a decision expression e by a subset D_e of $\mathcal{D}$. For reasons of uniformity, we also treat basic decisions in this way. A domain order o for domain Dom is interpreted by a preference relation $\prec_o \subseteq \text{Dom} \times \text{Dom}$. We do not require that this relation be transitive, but we expect that its transitive closure is irreflexive, that is, it represents a strict partial order.[3] For prioritized domain orders, we additionally need a priority function p_o from the domain values to integers.

[3] If this is not the case, no solution be obtained.

Table 2. *Decisions*

Decision Expression e	Elements of D_e
`decision` e:ϕ	e
`decision rule` $r(x)$ `on` C:$\phi(x)$	`decision` $r(i)$:$\phi(i)$
`assign`(f)	`decision a`(f,v):$f = v$
`assignElements`(F)	`decision a`(F,v):`member`(v,F)
`group`(g)	

Hence, we consider a mathematical structure consisting of a set $\mathcal{D}$, a relation $\prec$, a set D_e for each decision expression, a relation $\prec_o$ for each domain order, and a function p_o for each prioritized domain order. Such a structure is an interpretation of a preference program if it satisfies several rules. First of all, a decision d is interpreted by a singleton containing d. A decision rule r on class C is interpreted by the set of all its instances $r(i)$. An assignment decision set for field f is interpreted by a set containing an assignment decision $a(f,v)$ for each domain value v. Table 2 summarizes these rules and also indicates the constraint of the chosen decisions.

Domain orders have to respect the rules in Table 3. The predefined domain orders `minFirst` and `maxFirst` have a fixed interpretation, namely, the increasing or decreasing order on the supplied integer domain. The preference relation for prioritized domain orders is obtained by comparing the priorities of the domain values.

Preference and containment statements impose additional conditions that are listed in Table 4. A containment statement between a decision group g and a decision expression e means that all elements of D_e are also in D_g. A preference between two decision expressions e_1 and e_2 means that all elements of D_{e_1} are preferred to all elements of D_{e_2}. When we apply a domain order o to a field f, we lift the preference relation of o to the assignment decisions of f. Because there is a single assignment decision for each domain value, this lifting is straightforward. Finally, statements specifying priorities of domain values or preferences between domain values constrain the priority function or preference relation of a domain order in an obvious way.

Finally, we consider only interpretations that are minimal w.r.t. the sets $\mathcal{D}$, D_e, $\prec$, and $\prec_o$ in order to avoid the choice of irrelevant decisions and preferences. Further-

Table 3. *Domain orders*

Domain Order	Preferences
`minFirst`(o, Dom)	$\prec_o := <$
`maxFirst`(o, Dom)	$\prec_o := >$
`priorityOrder`(o, Dom)	$v_1 \prec_o v_2$ iff $p_o(v_1) < p_o(v_2)$
`partialOrder`(o, Dom)	

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

Table 4. *Preference and containment constraints*

Constraint	Condition
`contains`(g,e)	$D_e \subseteq D_g$
`prefer`(e_1,e_2)	$d_1 \prec d_2\ \forall\ d_1 \in D_{e_1}, d_2 \in D_{e_2}$
`preferValues`$(f,\prec_o)$	$a(f,v_1) \prec a(f,v_2)$ iff $v_1 \prec_o v_2$
`priority`(o,v,w)	$p_o(v) = w$
`prefer`(o,v_1,v_2)	$v_1 \prec_o v_2$

more, we introduce some pragmatic restrictions to make sure that preference statements do not change the solutions of the underlying constraint-satisfaction problem (CSP). For this purpose, we assume that the constraints are formulated in conjunctive normal form (i.e., each constraint is a disjunction of basic constraints or of their negations). We require that each of those constraints contains at most one containment-, preference-, or preferValue-constraint, and we also forbid negated occurrences of those constraints. Hence, we forbid disjunctive preferences of the form `prefer(a,c) or prefer(b,c)`. Thus, we can guarantee that preferences do not introduce additional solutions and that their interpretation uniquely depends on the chosen solution of the underlying CSP (i.e., the problem that is obtained by removing all preference statements). However, inconsistent preferences can lead to a loss of solutions.

5. FROM USER-DRIVEN TO PREFERENCE-DRIVEN PROBLEM SOLVING

In this section, we describe how preferences and decisions can be used to solve different kinds of configuration problems. In particular, web-based configurators differ in the amount of problem-solving effort made by the engine. On the one hand, certain configurators simply check whether user choices are consistent, meaning that the user has to do all the difficult work. On the other hand, advanced configurators optimize given user preferences. We try to order different configuration problems according to the amount of work that need to be done by the configurator. The result is a spectrum of configuration problems ranging from problems requiring a high level of problem-solving effort from the user to problems requiring a high level of problem-solving effort from the configurator:

1. *Constraint checking:* The user chooses values for all attributes and relations that need to be configured and submits this tentative configuration to the configurator, which determines the constraints and rules that it violates.
2. *Solution checking:* The user submits a tentative configuration to the configurator, which determines parts of the solution that violate given constraints and rules and thus need to be relaxed.
3. *Domain reduction:* The user makes the choices to find a configuration by an iterative process. Each time, a new choice is made, it is submitted to the configurator, which removes values that are no longer possible due to the user choices made so far. Hence, the configurator performs domain reduction (either by local-consistency techniques or by maintaining global consistency).
4. *Automated completion:* The user submits some choices to the configurator, which completes them by solving the resulting CSP. If the user is not satisfied by the solution, he or she can modify some of the choices of the configurator.
5. *Automated completion with explanations:* As in the previous level, the configurator completes partial user choices by finding a solution. The user can ask for explanations about why certain choices have been rejected. The explanations allow the user to identify user choices that are incompatible with rejected choices. Thus, the user can decide whether she or he sticks to the former choices or whether to modify some of them in the next iteration.
6. *Preferred solution:* The user submits initial requirements but no longer submits choices. Instead, she or he indicates preferences about the possible values. The configurator determines a preferred solution that satisfies these preferences as much as possible and justifies the preferred solution by a search profile that lists the decisions in the order in which they have been explored. If the user wants certain decisions to be made in a different order, he or she can provide additional preferences to change the order of the decisions in the next iteration.
7. *Preferred solution with explanations:* As in the previous level, the configurator determines a preferred solution that follows the user preferences. The user can ask for explanations about why certain choices have been rejected. This explanation consists of the choices of the configurator. The user can then select one of the choices in the explanation and state that it is less important than the rejected decision. Internally, an additional preference is added, leading to another preferred solution in the next iteration.

The last two points can also cover (multicriteria) optimization because preferences can be formulated on defined criteria such as the total price of a configuration.

Thus, the spectrum of configuration problems ranges from user-driven problem solving to preference-driven problem solving. In all these cases, the purpose of configuration is to determine a solution that satisfies the user, that is, that respects his or her preferences. If these preferences are implicit, the user has to make all the choices and uses the configurator just for checking and propagating the constraints. If these preferences are made explicit, the config-

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

urator will be able to propose solutions that better satisfy the user. As we indicated above, these preferences can be acquired from explanations for rejected choices. Hence, the user needs only to criticize solutions. At the beginning of the spectrum, the user is making all choices according to his or her preferences and the configurator just checks the constraints. At the other end of the spectrum, the configurator is making all the choices and the user just checks whether his or her preferences are respected.

6. EXPERIENCES FROM REAL-WORLD APPLICATIONS

Preference-based configuration strategies have been used in several industrial applications in domains such as finance and insurance. The preference programming features are used as follows in those domains:

- Loan configuration involves different kinds of financial products, each of which is characterized by special attributes and relations that are created when the financial product is classified. The preference-based configuration strategy takes these inherited fields into account and configures them as well. Furthermore, the financial expert attached priorities to a small number of characteristic attributes, such as loan duration and monthly rate, in order to control the order in which choices are made.
- Health plan configuration requires that health plans and their services be assigned to a subscriber while respecting eligibility constraints. Depending on the characteristics of the subscriber, certain health plans are desirable choices that should be made if possible. These desiderata can be coded by decision rules. Preferences between the rules help to choose the best rule if several of them apply to the given characteristics.
- In portfolio management, preferences permit the configurator to choose among the available stocks and shares.

These examples demonstrate that the high-level preference programming features provide significant help in business domains where experience with classical product configuration is not very frequent. A business expert can dramatically change the behavior of the configuration engine by changing some priorities or by adding business rules influencing user preferences and wishes. An application developer can program configuration strategies for a constraint-based configuration engine even if she or he is not a constraint programming specialist. This shows that our approach improves ease of use of configurators and allows them to do complex tasks with a few declarative statements.

7. PREVIOUS AND RELATED WORK

In this section, we briefly discuss related work on preference handling in AI. Configuration is becoming a standard application for different preference-handling approaches based on nonmonotonic reasoning, decision theory, and soft constraints, as the recent research literature shows.

Preference programming has evolved from previous work in nonmonotonic reasoning about prioritized default logic (Brewka, 1989, 1994) and nonmonotonic rule-based systems, such as EXCEPT II (Junker, 1993). In those approaches, preferences have been applied to default rules and have been expressed in a logical theory or in a logical rule language. In this article, we adopted similar syntax but embedded the preference statements in a constraint language instead of a logical language. Furthermore, we developed high-level primitives for decisions and preferences and attached task-specific behavior to those constructs. In fact, supernormal defaults correspond to wishes in our system. However, we have not incorporated default rules having a prerequisite, which leads to additional complexity in the definitions of preferred solutions (Brewka, 1994; Delgrande & Schaub, 2000).

Our approach has many features in common with the conditional-preference networks in Boutilier et al. (1997) and Domshlak et al. (2001). We can treat preferences about criteria as Boutilier et al. (1997), but we basically determine a preferred solution with regard to a lexicographical criteria instead of a *ceteris paribus* approach. This difference makes our treatment of dynamic preferences much simpler and does not have any negative impact on the complexity of our approach. The advantage of the CP-networks is that they give clear conditions for the consistency of dynamic preferences.

Preference programming 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 from approaches such as valued CSPs or semiring-based CSPs (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 CSPs and semiring CSPs 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 between solutions (either lattices or semi-rings). Future work is needed to understand which kind of preferred solutions can be characterized in this framework.

8. CONCLUSION

In this article, we developed a preference programming approach that provides advanced problem-solving behavior for configurators. Preferences can be formulated inside an

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

expressive constraint and rule language. The same preference statements can be used for different tasks, such as satisfying user preferences and wishes, and controlling search strategies, explanation detection, and global consistency checking of the available options.

This is particularly helpful for web-based configuration and E-business applications, where customers directly access a product configurator through the web without having a sales representative able to filter, modify, and orientate customer demands. Customers usually have no detailed knowledge about the product they are buying. Without knowledge about the product catalog and the existing possibilities, it is not straightforward to formulate requests that constrain configuration to the preferred choices of the customers. Even if the customer succeeds in doing this, the requests for preferred choices may be in conflict. A preference-based configurator avoids those difficulties by formulating requests as wishes that are satisfied if possible and by directly taking into account the user preferences. Because preference programming is provided in the form of a constraint and rule language, preferences can be assembled from different sources such as business rules deriving preferences from customer profiles, user input, and user interactions, such as rejections of proposed choices.

Furthermore, preference programming provides significant help in establishing and maintaining adequate configuration strategies. It is well known that configuration catalogs and compatibility constraints need to be changed frequently. The same is true for configuration strategies that evolve in parallel to the catalog and that define sales policies concerning, for example, promotions and bundles. When a new component is introduced into the catalog, it may also be necessary to change the configuration strategy in order to propose it in priority at a low price, for example. Preference programming permits the configuration expert to control the existing strategy in the form of high-level business rules or by changing priorities of attributes and relations declaratively. Often only a few preferences and priorities need to be specified to adapt a given default strategy that is formulated in the preference programming language.

Preferences are also important for detecting explanations and for controlling a consistency checker.

The preference programming system has evolved from earlier work in nonmonotonic reasoning where preferences between default rules have been very successful. Similar to this work, preference programming supports complex preference representations, including dynamic (or context-dependent) preferences, structured preferences, soft preferences, and metapreferences for resolving conflicts between soft preferences. Preferences are represented explicitly and have simple semantics: during decision making, the preferences impose constraints on the order of decisions and thus control the search strategy. The preference satisfier additionally uses preferences to prune solutions that do not satisfy given user preferences and wishes. It implements the PBS method from Junker (2002). This method is very useful to produce multiple configurations that are sufficiently different (e.g., one that minimizes the price and another one that maximizes the quality of the configuration).

Future work will be devoted to improving the pruning behavior of the PBS algorithm and to elaborating further applications of preferences in the context of configuration. Examples are preference-based explanations, consistency checking, and symmetry breaking based on substitutability (Freuder, 1991).

ACKNOWLEDGMENTS

The contributions of this paper profited from many discussions with our colleagues, in particular Olivier Lhomme, Xavier Ceugniet, Jean-François Puget, and Didier Vidal. We would also like to thank the anonymous reviewers for their helpful comments and Kathleen Callaway for polishing our English. This article is a significantly extended version of an article presented at the IJCAI-01 workshop on configuration.

REFERENCES

Bistarelli, S., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G., & Fargier, H. (1999). Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. *Constraints 4(3)*, 199–240.

Boutilier, C., Brafman, R., Geib, C., & Poole, D. (1997). A constraint-based approach to preference elicitation and decision making. In *Working Papers of the AAAI Spring Symposium on Qualitative Preferences in Deliberation and Practical Reasoning* (Doyle, J., & Thomason, R.H., Eds.), pp. 19–28. Menlo Park, CA: AAAI Press.

Brewka, G. (1989). Preferred subtheories: An extended logical theory for default reasoning. *Proc. Eleventh Int. Joint Conf. Artificial Intelligence* (Sridharan, N.S., Ed.), pp. 1043–1048. San Mateo, CA: Morgan Kaufmann.

Brewka, G. (1994). Reasoning about priorities in default logic. *Proc. Twelfth National Conf. Artificial Intelligence (AAAI)*, pp. 940–945. Menlo Park, CA: AAAI Press.

Delgrande, J.P., & Schaub, T. (2000). Expressing preferences in default logic. *Artificial Intelligence 123(1–2)*, 41–87.

Domshlak, C., Brafman, R.I., & Shimony, S.E. (2001). Preference-based configuration of web page content. *Proc. Seventeenth Int. Joint Conf. Artificial Intelligence*, pp. 1451–1456. San Francisco, CA: Morgan Kaufmann.

Doyle, J. (2002). Preferences: Some problems and prospects. In *AAAI-02 Workshop on Preferences in AI and CP: Symbolic Approaches*. Menlo Park: AAAI Press.

Ehrgott, M. (1997). A characterization of lexicographic max-ordering solutions. *Methods of Multicriteria Decision Theory: Proceedings of the 6th Workshop of the DGOR Working-Group Multicriteria Optimization and Decision Theory*, pp. 193–202. Egelsbach, Germany: Häsel-Hohenhausen.

Freuder, E.C. (1991). Eliminating interchangeable values in constraint satisfaction problems. *Proc. Ninth National Conf. Artificial Intelligence (AAAI)* (Dean, K., & McKeown, T.L., Eds.), pp. 227–233. Cambridge, MA: MIT Press.

Haselböck, A., & Stumptner, M. (1993). An integrated approach for modelling complex configuration domains. *13th Int. Conf. Expert Systems, AI, and Natural Language*, Avignon, France, pp. 625–634.

ILOG. (2002). ILOG JConfigurator V2.0: Product information. Available on-line at www.ilog.com/products/jconfigurator/.

Junker, U. (1993). Dynamic generation of assumptions and preferences. In *Actes des Vièmes Journées du Laboratoire d'Informatique de Paris Nord* (Bidoit, N., Ed.), pp. 1–21. Paris: Villetaneuse.

Junker, U. (1997). A cumulative-model semantics for dynamic preferences on assumptions. *Proc. Fifteenth Int. Joint Conf. Artificial Intelligence*, pp. 162–167. San Francisco, CA: Morgan Kaufmann.

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press

Junker, U. (2000). Preference-based search for scheduling. *Proc. Seventeenth National Conf. Artificial Intelligence (AAAI)*, pp. 904–909. Menlo Park, CA: AAAI Press.
Junker, U. (2002). Preference-based search and multi-criteria optimization. *Proc. Eighteenth National Conf. Artificial Intelligence (AAAI)*, pp. 34–40. Menlo Park, CA: AAAI Press.
Mailharro, D. (1998). A classification and constraint based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12(4)*, 383–397.
Mittal, S., & Frayman, F. (1989). Towards a generic model of configuration tasks. *Proc. Eleventh Int. Joint Conf. Artificial Intelligence* (Sridharan, N.S., Ed.), pp. 1395–1401. San Mateo, CA: Morgan Kaufmann.
Poole, D. (1988). A logical framework for default reasoning. *Artificial Intelligence 36(1)*, 27–47.
Puget, J.-F. (1992). Object-oriented constraint programming. *Artificial Intelligence, Expert Systems, Natural Language: Twelfth Int. Conf.*, pp. 129–138, Avignon, France.
SABRE. (2002). Sabre trip shopping: Product information. Available online at www.sabretravelnetwork.com/products_and_services/travel_agencies/s1_000999.htm.
Soininen, T., Niemelä, I., Tiihonen, J., & Sulonen, R. (2000). Unified configuration knowledge representation using weight constraint rules. *ECAI-2000 Workshop on Configuration*, Berlin, pp. 79–84.
Torrens, M., & Faltings, B. (2002). Using soft CSPs for approximating Pareto-optimal solutions sets. *AAAI-02 Workshop on Preferences in AI and CP: Symbolic Approaches*. Menlo Park, CA: AAAI Press.
van Hentenryck, P., & Puget, J.-F. (2000). Search and strategies in OPL. *ACMTCL: ACM Transactions on Computational Logic 1*.

Daniel Mailharro works in the Research and Development Department of ILOG SA in Paris, France, where he initiated and led the configuration activity of ILOG for 7 years. He created and developed the configurator product ILOG Configurator, which was commercialized in 1999. Before joining ILOG, Daniel worked for 7 years as a consultant specializing in constraint programming for Telecom and for energy and transport companies. Mr. Mailharro received his MS in artificial intelligence from Paris University. He has served as a member of several configuration workshop program committees including AAAI, ECAI, and IJCAI.

Ulrich Junker is a Distinguished Scientist in ILOG's Research and Development Department team, where he innovates preference-based methods for configuration, optimization, and explanation and incorporates them into industrial tools. His 15 years of experience in combinatorial problem solving covers topics such as constraint programming, truth maintenance, nonmonotonic reasoning, configuration, and methods for preference handling. He provided key contributions to several research projects conducted at GMD, IFP, and ILOG. Past experience at ILOG also includes consulting for large-scale optimization projects of customers in domains such as air traffic management, crew scheduling, supply chain, and configuration. Dr. Junker recently chaired the CPAIOR'00 workshop on the integration of constraint programming, AI, and operations research methods and the AAAI'02 workshop on preferences in AI and CP. He received a PhD and an MS from the University of Kaiserslautern.

https://doi.org/10.1017/S089006040317103X Published online by Cambridge University Press