Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1998), *12*, 321–331. Printed in the USA.

Web-based configuration assistants

GIUSEPPE ATTARDI, ANTONIO CISTERNINO, AND MARIA SIMI
Dipartimento di Informatica, Università di Pisa, Italy

(RECEIVED October 1, 1997; ACCEPTED February 7, 1998)

Abstract

Configuration assistants are tools for guiding the final user in simple configuration tasks, such as product assembling and customization or study plans generation. For their wide availability, web-based configuration assistants are valuable in fields such as electronic commerce and information services. We describe a general approach for building web-based configuration assistants: from a high-level description of the configuration constraints and of the basic items, given in a declarative language, the hypertext files for user guidance and the Java code for constraint checking are generated. We claim that the general approach of *process-oriented* configuration, where the user is guided through the configuration process by an explanatory hypertext, as opposed to *product-oriented* configuration, where one starts from a high-level description of the product of the configuration, is better suited for many application domains.

Keywords: Configuration Assistants; Process-oriented Configuration; Electronic Commerce

1. INTRODUCTION

In the terminology of expert systems, configuration systems are a subclass of design systems, whose task is to assemble a product out of a set of predefined parts according to problem-specific constraints (Hayes-Roth et al., 1983). Examples of configuration problems are computer equipment configuration (McDermott, 1982; Barker & O'Connor, 1989), software configuration, timetables generation and scheduling, and product configuration in different domains (Tiihonen et al., 1996).

The complexity of a configuration task arises from having to cope with many and interacting design decisions, whose consequences cannot readily be assessed, and with constraints of different nature. Configuration problems are therefore a challenging domain for expert system technology.

For simpler configuration tasks, we can envision interactive configuration assistants which guide the user step by step through design decisions and exploit their knowledge of domain constraints and of constraints deriving from previous choices. A *configuration assistant* (CA) is an interactive tool for configuring systems, which does not make choices on user's behalf but assists the user in assembling a consistent product. Ideal targets for this kind of system are applications where the user wants to retain control on the configuration choices, yet he or she must be made aware of the available alternatives and supplied with necessary information to perform the right choices. A configuration assistant typically addresses sales persons or end-users rather than experts in the configuration domain.

The applicability of this kind of system is restricted by some simplifying assumptions, reflected in the configuration model we adopt. Nevertheless, the range of configuration problems that can be dealt in this way is significant. For example, configuration tasks that are amenable to this simplified vision are study plans compilation or the assembling of a product from a catalogue of components, like a complex piece of modular furniture (kitchen furniture for instance), or a personal computer.

Configuration assistants of this kind running on the Web open opportunities in many fields, including electronic commerce, for the wide availability and the possibility of remote access and use (see http://www.volvocars.com for an example). The system does not need to be installed on the user computer in order to be used and portability problems do not need to be addressed. This is also an application domain where a Java solution (Gosling, 1996) has clear ad-

Reprint requests to: Giuseppe Attardi, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy. E-mail: {attardi, cisterni, simi}@di.unipi.it.

vantages over a server-based solution with constraint checking done on the server side. All the job of the configuration assistant can be done locally, at the client's side, by downloading the necessary Java code. Communication with the server can be reduced to tasks such as user validation, statistics gathering, or archiving.

The major problem with product configuration systems is to maintain product information up-to-date. The cost of long-term management and maintenance of product knowledge, as product models and configuration constraints evolve, may seriously impair the advantages of configuration systems. For this reason *single-use* configuration systems, which do not clearly separate configuration knowledge from the configuration program are doomed to failure. A *general-use* configuration model, which facilitates the development of configuration applications in different domains, and for different product classes, is the only viable alternative (Männistö, et al., 1996; Tiihonen & Soininen, 1997).

In this paper, we describe our general configuration model, characterized as *process-oriented*, in contrast to a *product-oriented* approach. In fact, the aim is guiding the user step by step through the configuration process rather than starting from a high-level description of the product to be configured.

Most configuration models in the literature are *product-oriented*: their task is described as the synthesis, more or less automatic, of a set of objects satisfying a set of predefined constraints (Stumper, 1997). In Klein et al., 1994, the configuration task is described as model construction, the constraints being expressed as a set of logic formulas defining a theory; any model of the theory is a legitimate configuration. An alternative approach is described in Faltings and Weigel (1994), where configuration is described as a constraint satisfaction problem.

Our approaches differ from these because, given the interactive nature of a configuration assistant, and the fact that the user performs all the choices, the configuration process does not do any search in a solution space; the CA is strictly deterministic and thus very efficient.

Our model has been used as the basis for the implementation of a generic tool for developing domain-specific configuration assistants running on the Web. A configuration application is generated starting from a high-level description of the basic components and the constraints expressed in a declarative form. The hypertext files for user guidance and the Java code for constraints checking are automatically generated from this high-level description. The reduced cost of delivering a specific configuration assistant and the efficiency and compactness of the generated code are clear advantages of this approach.

The tool has been used to generate specific configuration assistants in the domain of study plans compilation and submission (the CompAss application). CompAss has been used to generate study plans by students of the Faculty of Letters and Philosophy, and of the Faculty of Sciences at the University of Pisa.

2. THE CONFIGURATION MODEL

2.1. Process-oriented configuration

The standard way to think of a configured product is as an assembly of basic components matching a set of structural and functional requirements. As an alternative approach, we focus on the process of posing a series of relevant questions, with the goal of completely identifying a correctly configured object fulfilling user's requirements.

The metaphor we have in mind is an experienced salesperson posing a series of questions to the customer, according to some standard pattern. According to the answers received, the vendor gains an increasingly accurate idea of the customer's needs, until he or she is able to propose a specific product. If the product is complex and needs to be assembled from a catalogue of components, the vendor uses his or her knowledge of the configuration constraints to guide further choices among the available options.

We can identify two sorts of questions in this dialogue: general questions concerning the strategy in the selling process, functionality of the product, or user's needs, preferences and profile; specific questions whose purpose is to make the user perform selections from a catalogue. Both of them, together with configuration constraints, contribute to the resulting configured object.

The interaction going on between vendor and customer can be seen as the analogue of the configuration process. The correctness and completeness of the configured product is guaranteed by the fact that an experienced salesperson is able to propose a sufficient number of questions to allow the customer to select from the catalogue all the items that are needed to assemble a product satisfying structural and functional constraints.

A graph, called *choice graph*, is introduced, which can be seen as a representation of the vendor strategy in posing questions to the customer. Each node of the graph represents a possible answer to a general question proposed by the vendor. Configuration constraints are associated to each node and are used to suggest possible selections from the catalogue.

Consider, for example, the task of configuring a personal computer. We can describe the process according to the graph in Figure 1.

The constraints associated at each choice node could be for example as illustrated in Figure 2.

Starting with the root node, a number of items from the catalogue are added to the configuration either because they are required for all PCs (as the motherboard, case, and mouse) or because they are selected by the user (as for CPU, memory, hard disk, and monitor). In fact, the user is prompted to select among available options in order to satisfy composition constraints associated to PCs. More items are added as a consequence of the user performing a choice between the foreseen use of the PC: whether it is intended for game playing, multimedia applications, or office work. Finally,

https://doi.org/10.1017/S0890060498124058 Published online by Cambridge University Press

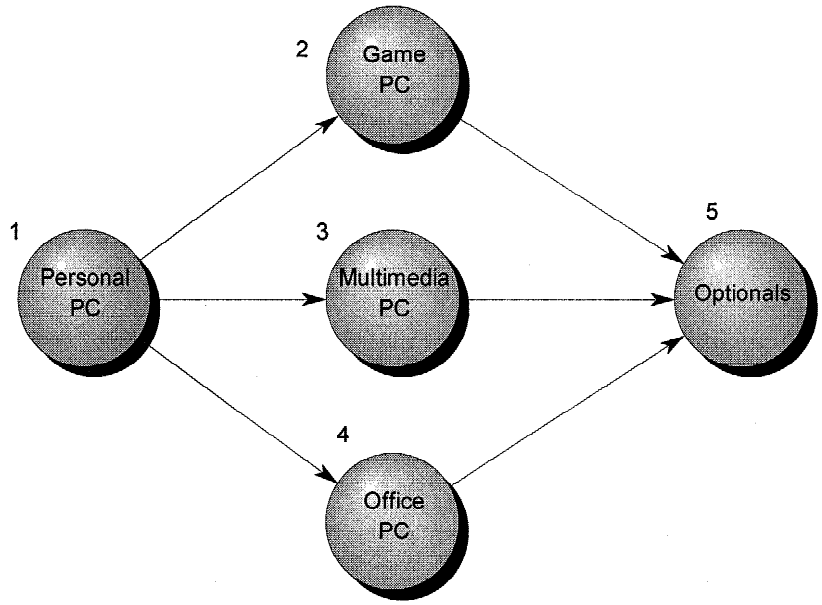

Fig. 1. Configuring a personal computer.

optional components are selected, depending in part on previous choices. The details of the language for expressing constraints will be presented in Section 3.

A *configuration domain* is fully defined by the set of basic components, *the item set*, and *configuration constraints* specific to a configuration application. The basic items we consider are structured objects with attributes.

The *process-oriented* configuration model relies on the *choice graph* (CG), which is a directed acyclic rooted graph.

The *configuration process* starts in the root of the graph, with an empty configuration. Following an edge of the graph corresponds to choosing a possible user alternative in the configuration process, and moving to a successive choice node. Within a choice node, in order to fulfill a configuration constraint the user may be asked to select among a set of possible items. Traversing the graph, the user builds a configuration, accumulating choices and selecting items within choice nodes.

	2. Game PC [Joystick, Joypad](1+)	
1. PC [Motherboard, Case, Mouse] [CPU 200Mz, CPU 333Mz](1), [8Mb](2+) [Hard Disk 2Gb, Hard Disk 4Gb](1) [Monitor 17", Monitor 15"](1) MAX-COST(3000)	**3. Multimedia PC** [DVD]	**5. Optionals** OR(Game PC, Multimedia PC) => [Speakers, Sound card] NOT(DVD) => [CD-ROM]
	4. Office PC [Word Processor, Spreadsheet]	

Fig. 2. Constraints associated with the nodes of the choice graph.

https://doi.org/10.1017/S0890060498124058 Published online by Cambridge University Press

A *valid configuration* is a set of basic items, and a set of user choices, matching a given set of constraints.

The model provides two different notions of valid configuration, *partial* and *final*. A partial configuration can only be checked for correctness but not for completeness; a final configuration must be correct and complete; for example, it must include all the required items. Correspondingly, a system implementing this model can check partial configurations during the configuration process, warning the user as soon as some constraint is violated. At the end of the process the system checks the final configuration.

The choice graph is a representation of the configuration knowledge, which is very convenient for generating a graphic interface to the CA. A possibility is a hypertextual interface where a node of the graph is displayed as a single hypertext unit, with selection buttons for available options which the user is prompted to select, and links to other pages, corresponding to successor nodes.

The possibility of automatic generation of the CA, including its user interface and documentation, has been successfully exploited in the CompAss system. This feature of the model is especially important for dynamic configuration domains, where the catalogue of components or the constraints change frequently.

An important modeling issue, in defining the choice graph, is what should correspond to choice points and what should correspond to constrained selections within a choice node. Figure 3 shows a choice graph for the computer domain, where the configuration expert has decided to offer different paths, meant to match different customer profiles and needs.

Free selection from catalogue is offered for the experienced customer, who knows the rules for putting together a computer, or for the customer who wants to buy a spare component. The constraints associated to this node allow the user to choose what he or she wants, without any guidance.

Nonprofessional PC users may prefer preassembled PC offers, where they need only to choose among a limited number of alternative packages and a few optional components, for example, the monitor. The nodes that correspond to preassembled computer models, Model 1 and Model 2 in the example, define all the items needed for each model. In this solution the user, with appropriate advice, only has to make high-level choices (the model) and the CA adds automatically to the configuration the items that are required.

As an intermediate possibility, a customer may be guided to configure his or her PC, by selecting the main components, starting with the CPU. Within the CPU node, he or she will be asked to select all required components for such CPU. Next he or she will be offered the choice among different software and later among optional components. Because these are also suitable for the *preassembled PC*s, the node *Optionals* of the graph is shared.

The configuration model itself does not commit to any specific design solution, since no general design rule applies to different configuration domains and user categories. The model itself is flexible enough to accommodate different solutions.

Formalization of the configuration model

In the following, we will introduce formal definitions of choice graph, configuration, and configuration process.

Let I be the set of basic items of the configuration domain. A choice graph can be formally defined as

$$G = \langle S, N, \mathcal{T}, \aleph \rangle$$

with S is the set of nodes in the graph (choice nodes); N is a binary relation among the nodes in the graph denoting the edges of the graph. $\mathcal{T}$ and $\aleph$ correspond to two different classes of *constraint functions* associated to the nodes of the graph: built-in functions of general use (called *composition functions*) and domain-specific functions (called *custom* functions). Constraint functions are predicates used to check that the items selected in a node fulfill configuration constraints. We will denote with $\mathcal{T}_j$ and $\aleph_j$ the functions associated to node s_j.

Composition predicates are functions of type:

$$f\colon J \times \{partial, final\} \rightarrow \{t, f\}.$$

They take as input a set J of items, and return the value *true* or *false*. The last parameter *partial/final* is to make the constraint-checking behavior different in the case of partial or final configurations. These functions check a configuration with respect to constraints local to a choice node; for example, they are used for checking whether a required item has been included.

Custom functions are functions of type:

$$f\colon J \times \{partial, final\} \rightarrow \{t, f, w\}.$$

They take as input a set J of items, and check global properties of the set of items selected so far; for example, the number of items in a configuration or the need to avoid duplicate items. These functions are usually domain dependent. The result, in addition to *true* and *false*, may also be a *warning*.

Constraint functions, invoked with parameter *partial*, provide for incremental constraint checking. On the other hand, some constraints can only be checked on the final configuration; this is done invoking the constraint functions with parameter *final*. The validation component of the CA may thus operate according to two different modes: in incremental mode, it gives immediate feedback on wrong user actions during the configuration process; in final mode, it provides for final validation of all the constraints.

A configuration $K = [C_1, C_2, \ldots C_n]$, is a sequence of choice components. A choice component C_i is a pair $\langle s_i, I_i \rangle$, where $s_i \in S$ is a choice node of the graph and I_i is the set of

https://doi.org/10.1017/S0890060498124058 Published online by Cambridge University Press

items, possibly with repetitions, selected in node s_i during the configuration process.

A configuration process starts in the root node of the graph with an empty configuration, $K_0 = [\ ]$. A configuration is incrementally built as a result of user or system actions of different types: *choose*, *add_item*, *delete_item*. Here is how a configuration K is affected by the actions:

$$choose(s) \rightarrow K \circ \langle s, \{\}\rangle$$

$$add\_item(i,s) \rightarrow K[\langle s, I\rangle / \langle s, I \cup \{i\}\rangle]$$

$$delete\_item(i,s) \rightarrow K[\langle s, I\rangle / \langle s, I - \{i\}\rangle]$$

The "$\circ$" operator is to be interpreted as list concatenation. $K[x/y]$ is configuration K with the x component replaced by y. The $I - \{i\}$ is the deletion of one occurrence of the ith element from I.

Let us formally define *validity* of a partial and final configuration.

DEFINITION 1 (*Validity of partial configuration).* A partial configuration $K = [C_1, C_2, \ldots C_n]$ is valid if the following properties are satisfied:

1. The sequence $[s_1, s_2, \ldots s_n]$ is a path on the choice graph, starting from the root node.
2. Let $\mathcal{T}_j = \{f_1, \ldots, f_k\}$ the set of composition constraint functions associated to node s_j. For each $j = 1, \ldots, n$, there must exist a partition $P_j = \{p_1, \ldots, p_k\}$ of the set of items I_j such that $f_h(p_h, partial) = t$, for each $h = 1, \ldots, k$, i.e., any f_h must be satisfied by a set of items in the partition.
3. Let $H_j = I_j \cup \ldots \cup I_n$, i.e., the set of items in K selected in node s_j and its successor nodes. For each $j = 1, \ldots, n$, and each g in $\aleph_j$ it must be the case that $g(H_j, partial) = t$ or $g(H_j, partial) = w$. ■

DEFINITION 2 (*Validity of final configuration).* A final configuration K is valid if the following properties are satisfied (with $\mathcal{T}_j$, H_j, defined as before):

1. The sequence $K = [C_1, C_2, \ldots C_n]$ forms a complete path on the choice graph, starting from the root node and ending in a leaf node.
2. For each j, there must exist a partition $P_j = \{p_1, \ldots, p_k\}$ of the set of items I_j such that $f_h(p_h, final) = t$, for $h = 1, \ldots, k$.
3. For each j and each g in $\aleph_j$ it must be the case that $g(H_j, final) = t$. ■

Three conditions are thus required to be met: (1) selected choice nodes must lie on a path on the choice graph, originating in the root node; (2) for each node, composition functions are checked on the items selected in the node; and (3) for each node, custom functions are checked on the items selected in the node itself and its successors.

Note that composition predicates in $\mathcal{T}_j$ are not checked independently on the set of items I_j: an item used to satisfy a constraint cannot be used to satisfy another constraint; this is the reason why we look for a partition of the items I_j satisfying all composition functions.

3. THE CONFIGURATION LANGUAGE

To build a configuration application, we need a language for describing the various aspects of the configuration domain: the structure of the items, the item data base, the custom constraint functions, and the *choice graph*, with associated configuration constraints.

3.1. The components data base

The *item structure* consists in an item definition (similar to a `struct` of the language C). The items themselves are described in the *item data base*, according to the defined item structure. At least one field is required in an item description: the *name field*. The name field is a variable-size description of the item to be used by the applet at run time in communicating with the user. Additional fields can be specified, in the item structure, to represent other properties of the items specific of the domain.

For each field, it is possible to specify whether the field is *static* or *dynamic*. The main difference is that the value of static fields is loaded from the item data base, while the value of the dynamic fields is computed from other fields or entered by the user at run time. The name field is necessarily static. It is also possible to specify whether the field must be saved in the data base where the final configuration is stored.

When a user selects an item, during the configuration process, the system adds an instance of the data base item with the same name to the configuration. The values of the static fields are copied from those stored in the data base; dynamic fields are filled according to an action associated to the field. Different instances of the same data base item may appear in a configuration; in this case they agree on the name but they may differ in the values of dynamic fields.

In the domain of the computer hardware, static fields are the name of the component, and for example the price and the code identifying the item within the catalogue. A dynamic field is, for instance, a Boolean flag is indicating whether a customer already owns the item.

Each entry in the *item data base* may contain, in addition to the descriptive fields mentioned above, a documentation field, which is used by the CA for item documentation purposes.

3.2. The constraints language

The choice graph, with associated configuration constraints, is defined in a special declarative language designed for this

https://doi.org/10.1017/S0890060498124058 Published online by Cambridge University Press

purpose. The language supports the definition of two types of blocks: *list blocks* and *choice blocks*.

A list block is simply a way to define a group of items so that it can be referred by name. The operator "#" is the way to refer to all the items belonging to a defined *list block*; for example, `#ref` refers to a block named `ref`, which could be defined as

`ref`[*item* 1, *item* 2, …, *item n*].

An item in a list block can in turn be a reference to another list block.

A choice block corresponds to a node in the choice graph and defines composition constraints for a node. A choice block can have, for example, the following structure:

```
Personal computer {
 [Motherboard, Case],                 /* 1 */
 [CPU 200Mz, CPU 333Mz](1),           /* 2 */
 [8Mb RAM](2+)                        /* 3 */
 OR(Game PC, Multimedia PC)=>         /* 4 */
 [CD-ROM, Sound card, Speakers,
  3D Video Card],
 [Power UPS](1-),                     /* 5 */
 CHOICE(CRT monitor, LCD monitor)     /* 6 */
} Max_cost(3000);                     /* 7 */
```

A block description includes a list of *constraints*, specifying items that are necessarily needed for the block and items that are needed depending on the configuration state, i.e. the presence of other items or previous choices. For example line `1` says that the items `Motherboard` and `Case` are always required for a personal computer. Line `4` says that items on the right of the "=>" operator are to be included in the configuration only if the user has declared to be interested in a PC for playing computer games or in a Multimedia PC. The conditions for inclusion are expressed by means of logical operators: AND (all of them), OR (some of them), NOT (none of them).

Line `2` is an example of a construct that prescribes the selection of a number of items out of a list of items; in particular, the example says that exactly one item out must be selected from the given list of available CPUs. Lines `3` and `5` are similar with different number restrictions: in the first case *two or more* items are required, in the second case *at most one* item is accepted. Line `6` defines the successors of the current node in the choice graph, i.e., the available choices at this level.

Thus, a choice block defines the items required for the node in the configuration. The type of standard controls that are generated concern the admissibility of items (i.e., answers the question "is it correct that this item is in the current configuration"?) or the presence of items (i.e., answers the question "is this required item present in the current configuration"?). These constraints are translated in a set of built-in composition constraint functions, as detailed below.

Other kinds of constraints which are often needed, such as "The cost of the configuration must be at most XXX$", are implemented by *custom constraints functions*, which are typically application dependent: these can be defined by the user or supplied by a library. Custom constraint functions appear at the end of a choice block (as in line `7` above) and, as specified above, apply to all the items in the block and successor blocks.

The first choice block, appearing in the constraint file, corresponds to the root of the choice graph. More formally, each constraint in a choice block has the following structure[1]:

⟨*item list*⟩[**(**⟨*n*⟩[+|−]**)**]

that is an *item list*, optionally followed by a numeric restriction such as (*n*), (*n*+), or (*n*−). A *item list* may be unconditioned or conditioned, and has the following form:

[**AND** | **OR** | **NOT (**⟨*item list*⟩**)** =>][**(**⟨*item list*⟩**)**]

The antecedent part, when present, expresses a condition under which the *item list* in the consequent is returned. If the condition is false the empty list is returned; if the condition is missing the item list is always returned. The items in an item list may include single items, item lists (possibly referred to by name), and also conditional item lists. The item list in the antecedent may also be a list of names of choice nodes.

The evaluation of a Boolean condition applied to an item list proceeds as follows:

1. The item list is evaluated with respect to the current configuration and choice block; this results in a list of items which may be different in different configuration states;
2. The Boolean condition is evaluated on the resulting list:

   AND $(i_1, i_2, \ldots, i_k)$ returns *t* only if all the items $i_1, i_2, \ldots, i_k$ are among the items included in the configuration up to the previous choice node;

   OR $(i_1, i_2, \ldots, i_k)$ if at least one of them is present among them;

   NOT $(i_1, i_2, \ldots, i_k)$ if none of them is present.

The constraint function generated is one of a set of built-in control functions: *Nec* (for simple item lists without number restrictions, meaning that all the items are necessary), *Atleast*, *Atmost*, *Exactly* (for numerically restricted selection from the item list).

3.3. Composition constraint functions

Let *K* be the current configuration, *s* a choice block, and *type* the checking mode (*partial* or *final*). For checking a

[1] The terminal symbols of the language are in bold. Square brackets are used for optional syntactic constructs. The "|" is used for alternatives.

https://doi.org/10.1017/S0890060498124058 Published online by Cambridge University Press

constraint in block *s*, the following built-in functions are used, depending on the constraint type.

If the constraint is a simple item list, the constraint function which is used is

$$Nec(S(K, s, item\_list), J, type).$$

If the constraint is an item list with numeric restrictions:

$$item\_list(n) : Exactly(n, S(K, s, item\_list), J, type)$$

$$item\_list\ (n+) : AtLeast(n, S(K, s, item\_list), J, type)$$

$$item\_list\ (n-) : AtMost(n, S(K, s, item\_list), J, type)$$

Checking a configuration means to devise, for each choice block *s*, a partition of the items selected in *s*, satisfying all the constraint functions associated to *s*. These functions take as input a set *J* of selected items, corresponding to an element of a partition, and check the corresponding constraint against this set. An error is generated if no partition of the items selected in *s* can be found, satisfying all the constraint functions associated to *s*. In what follows, the items in *J* are only identified by their names. Moreover, #*J* means the number of items in *J*. The behavior of the constraint functions can be described as follows:

$$Nec(L, J, partial/final) = t \text{ if } L = \emptyset \text{ or } L = J; f \text{ otherwise.}$$

$$AtLeast(n, L, J, partial) = t \text{ if } L = \emptyset \text{ or } J \subseteq L; f \text{ otherwise.}$$

$$AtLeast(n, L, J, final) = t \text{ if } L = \emptyset \text{ or } \#J \geq n \text{ and } J \subseteq L; f \text{ otherwise.}$$

$$AtMost(n, L, J, partial/final) = t \text{ if } L = \emptyset \text{ or } \#J \leq n \text{ and } J \subseteq L; f \text{ otherwise.}$$

$$Exactly(n, L, J, partial) = t \text{ if } L = \emptyset \text{ or } \#J \leq n \text{ and } J \subseteq L;\ f \text{ otherwise.}$$

$$Exactly(n, L, J, final) = t \text{ if } L = \emptyset \text{ or } \#J = n \text{ and } J \subseteq L; f \text{ otherwise.}$$

3.4. Custom functions

For the definition and enforcement of domain-specific constraints, the user can define special custom functions, in addition to the standard composition constraints resulting from the constraint file.

Custom constraint functions are meant for checking properties of the configuration which depend on the properties of the single items. They are used for example to impose conditions on aggregate values or to set lower or upper bound to values in the configuration.

For example, in the computer hardware domain (Figure 4), the choice nodes labelled "Price limited x" could be introduced to constrain the maximum price allowed for a configuration. To check a constraint of the configuration such as this one, we use a custom constraint function that computes the sum of the prices of a set of items, as given in their price field, and compares the result to the limit price, given as argument to the custom function. To allow for more flexibility and locality, custom functions are applied to the items selected in the node where the function is introduced and to those selected in its successor nodes, rather than to all the items in the configuration. In Figure 4, we show the choice graph of Figure 3 after the introduction of two additional nodes, `Price limited 1` and `Price limited 2`; with this extension an additional constraint about the price is imposed to all the configured PCs.

The custom functions are defined in a scripting language, which is a simplified version of Java. The library supplied by the scripting language offers a set of aggregate functions to easily define standard conditions like the upper bound, the uniqueness of items, etc.

A custom function returns one of three values $\{t, f, w\}$, where the *w* value is introduced to signal a warning condition. In our example of computer hardware, we can use the warning to alert the user that the cost of the items selected so far is close to the limit price.

4. GENERATION OF A CONFIGURATION APPLICATION

A configuration application is generated by using a compiler, which takes as input the following data files: the *item structure* file, the *item data* file, the *custom functions* file, and the *constraint file*, which defines the choice graph. All these specifications are given in input, as separate text files in human readable form, to the CA generator.

The compiler is divided in two modules: C1 and C2 (Figure 5). The first module is needed for generating the Java code for the applet and the items data base. The second module of the compiler generates a binary representation of the constraints and the HTML files for documentation and user guidance.

More specifically, the module C1 of the compiler takes as input three of the data files described above (the *item structure*, the *items data*, and the *custom constraints functions*) and produces three files which are used in the second step of the compilation process: a Java program, information about the items in HTML form and the binary version of the items data base. The generated Java applet depends on the configuration domain only for the item structure and the custom constraint functions; these elements are Java classes generated by the compiler and later combined with the rest of the applet. The applet will also use the items data base and a binary representation of the configuration constraints. The Java code includes the *item Java class* and the *custom constraint functions Java classes*. The item class is the class that describes the format of the item data base and offers to the configuration applet a set of methods for read-

https://doi.org/10.1017/S0890060498124058 Published online by Cambridge University Press

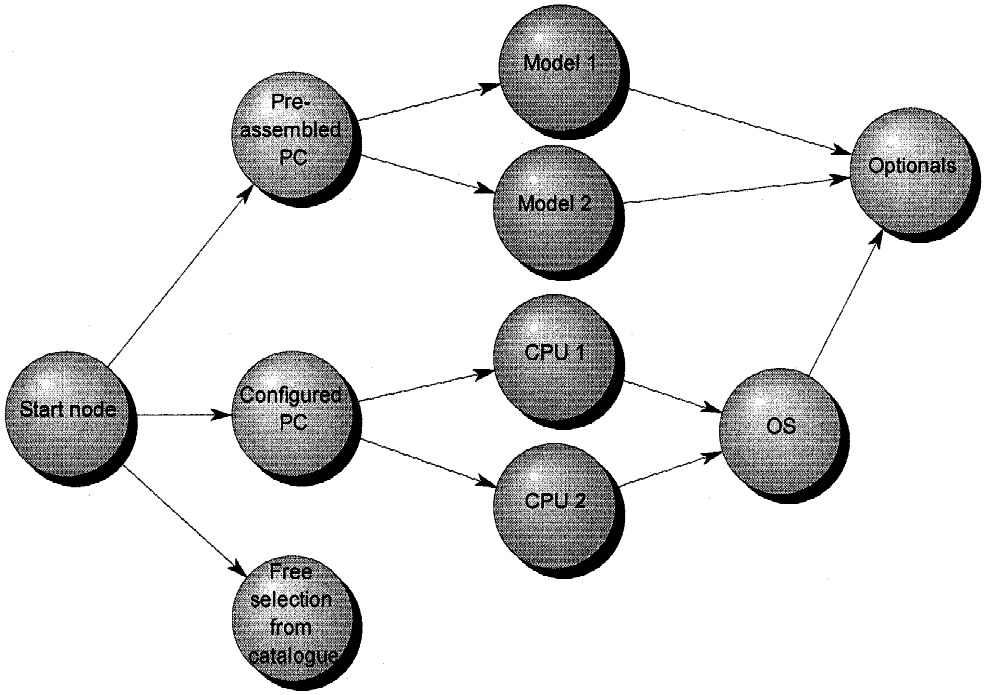

Fig. 3. Alternative configuration strategies.

ing, writing, and accessing item components. The custom constraints functions classes are a Java version of the custom constraint functions. These classes are managed by a Java class that maps the function calls to the proper functions.

The module C2 of the compiler takes as input the constraints file, the items information, and the items data base. It generates HTML files and the binary version of the constraints.

The HTML files generated are to be used in user interface of the configuration assistant. In particular, a set of HTML skeleton files are generated out of the choice graph: for each node in the graph a file is generated with selection icons for the items in the node and hypertextual links to the items descriptions. The file also contains a few lines of text, which synthetically describe the node constraints (for example "Choose at least three items out of the following:"), which can be enriched with additional text deemed useful to guide the user during the configuration process. In addition, the file contains choice icons and hyperlinks to other HTML files in correspondence of available choices.

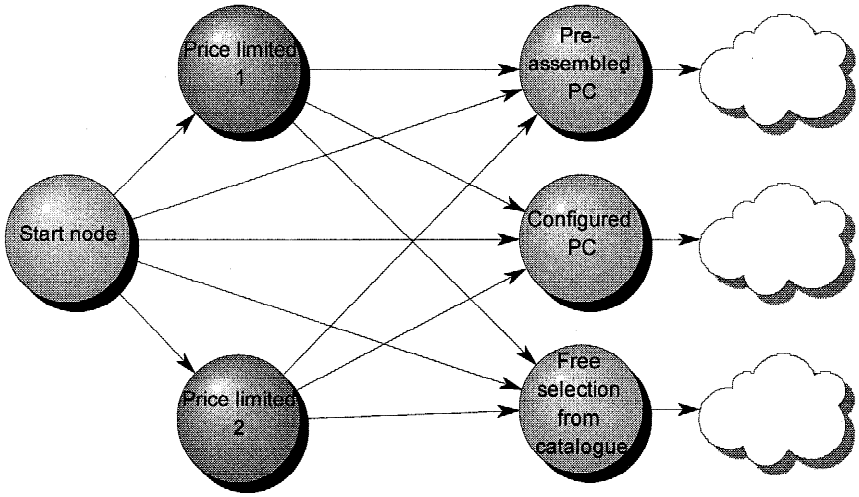

Fig. 4. Price limited configurations.

https://doi.org/10.1017/S0890060498124058 Published online by Cambridge University Press

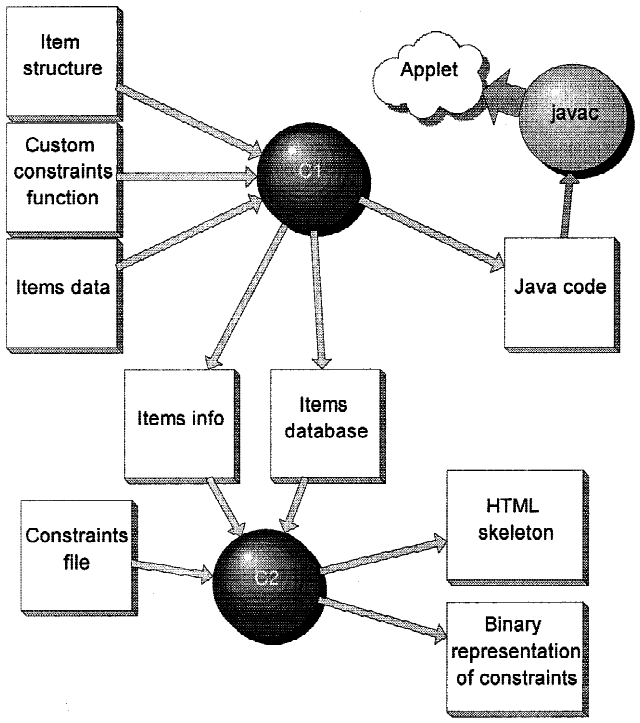

Fig. 5. Generation of a configuration application.

A compact binary representation of the choice graph is also generated by C2 and it is the primary data structure used by the applet for checking the validity of a configuration.

5. COMMUNICATION WITH THE SERVER

The system uses a custom server to store configurations in a data base. The server, written in Java, uses the JDBC interface to interact with the data base.

The configuration assistant runs on the client machine after the downloading of the CA applet from the http server. When the user has completed the configuration process and the final configuration has been validated, the CA applet submits the configuration to the server; the server accepts the connection, generates a unique identifier for the configuration, and generates a HTML page that contains the configuration and its unique identifier. The client receives back this HTML page as result of the configuration process and can decide to print it locally or simply take note of the identifier for future use.

At the moment, we provide this simple mechanism of authentication of the configurations generated by our CA. To check that a configuration is really generated by our system it is sufficient to find a configuration stored in the server data base with a given identifier. We plan to support other authentication mechanisms to allow a more sophisticated authentication that can be suitable for electronic commerce. This is not a simple task because all authentication mechanisms adopt a model based on physical tokens or on electronic certificates. Both these approaches require that the user has a browser and an authentication token.

6. CompAss: A CONFIGURATION ASSISTANT FOR PLANS OF STUDY COMPILATION

CompAss (COMPilazione ASSistita di piani di studio) is a system to assist students in the task of producing a study plan. CompAss and its associated support tools have been developed in the context of a pilot project for the Faculty of Letters and Philosophy of the University of Pisa and have been further developed and extended for use in other courses of study within the university.

Plans of study approval is a time consuming job for all the courses of study in the university, due to the high number of submissions each year (around 4000 for the Faculty of Letters and Philosophy) and the high rate of incorrect submissions. One of the requirements was that students could use any computer located in the various departments of the faculty to compile plans of study; data had to be collected in one single place for archival. The Java solution was the obvious choice and offers additional advantages such as the possibility of using the system from home.

The Web page of the CompAss configuration assistant is vertically divided in two parts (Figure 6). The right part contains the *navigation frame*. The left part contains the *configuration frame* and an *application specific tool bar*.

The navigation frame is a HTML frame displaying a normal hypertextual document; it displays the available choices together with any informative text deemed useful to guide the user to do the right choices during the configuration process. Documentation on the items can be obtained by clicking on the "i" round icon.

Special icons associated to choice points and to items are used to perform configuration actions: intermediate choices or item selections; when these icons are selected they send messages to the configuration program, the *validator*, which is a Java Applet associated to the configuration frame.

The configuration frame on the left contains the Java applet managing the configuration. The applet receives input by direct interaction in its client area (handled through events in the AWT) or by selection of special icons in other frames (the navigation frame and the tool bar frame). Whenever a configuration action is performed the applet reacts by checking the current partial configuration, accepting the change or prompting the user if any configuration constraints is violated.

Tool icons in the toolbar denote general utility or application specific actions available to operate on the partial configuration displayed in the configuration frame (i.e., item deletion, final configuration validation, abortion of the configuration process, printing, or submission of the final configuration).

Interaction between HTML pages and the Java applet is implemented by using JavaScript to post the events of icon selection to the configuration applet. With this solution, the

https://doi.org/10.1017/S0890060498124058 Published online by Cambridge University Press

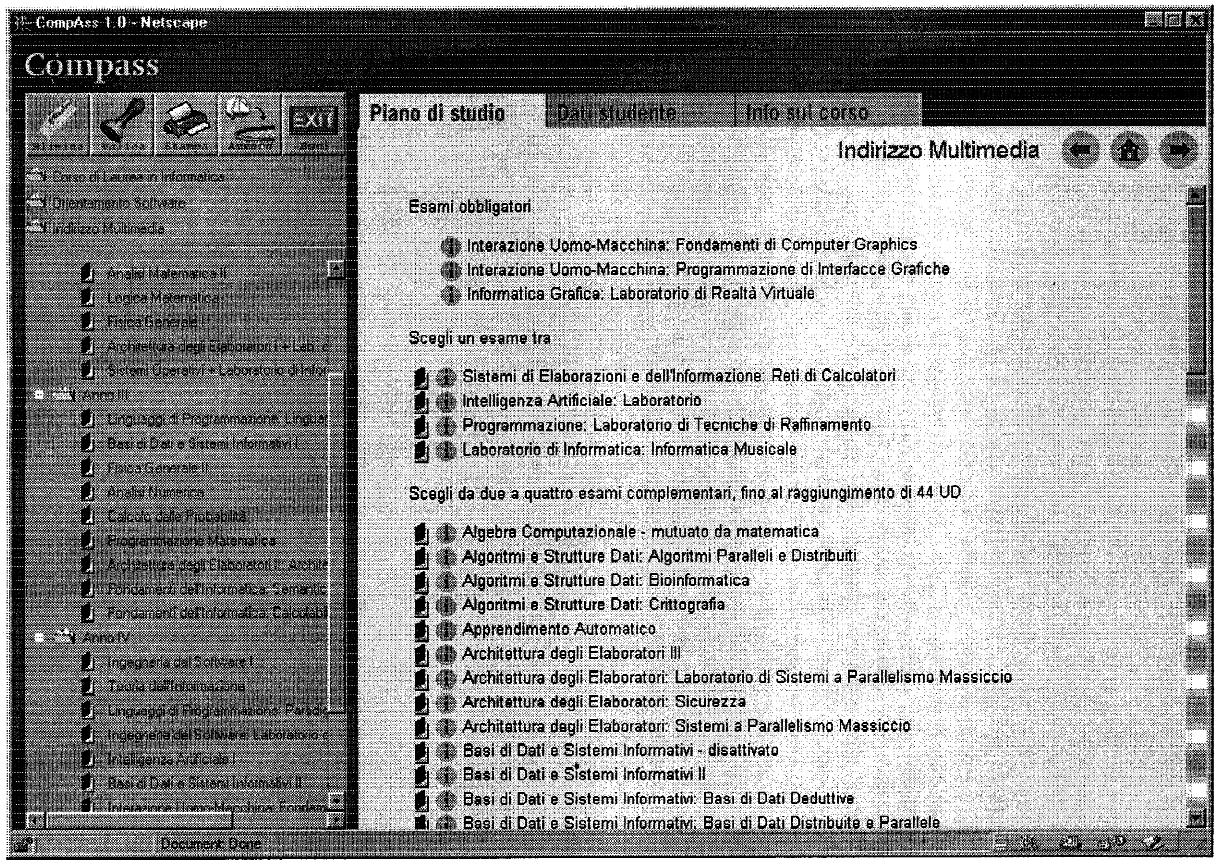

Fig. 6. The user interface of CompAss.

interface can exploit all the power of the HTML language and standard browsing capabilities, while still allowing user interaction with the Java program.

In this configuration application, the basic items are the courses offered by the faculty; the constraints file implements the rules for plan of study formation; it includes a choice graph where nodes correspond to choices such as the course of study, the orientation, the field of specialization and so on, together with the necessary constraints. A configuration is a legitimate plan of study, i.e., a list of courses which a student plans to take, fulfilling all the requirements imposed by the faculty.

The official submission of the plan must be done on paper forms, since it requires a signature by the student. Our current solution is that the plan is printed locally, after completion and verification by CompAss, and automatically sent to the server and registered in a temporary area. When the student submits the plan to the secretary's office, the plan is retrieved and transferred to the archives of submitted plans. CompAss saves a lot of work for secretaries who previously had to type in the plans from the paper forms submitted by students and eliminates the routine work of the faculty committees which had to verify and approve the plans.

The plan of study manager running on the server accepts communications from several CompAss clients, receives data from plans of study, generates HTML pages, stores data in a data base, and gathers statistics on the number of users and on the pattern of use of the system. An instance of CompAss can be seen at the Web address http://compass.di.unipi.it.

7. CONCLUSIONS AND FUTURE WORK

We have described a model for a process-oriented configuration and a general tool for generating configuration assistants for the Web; the strategy has been successfully exploited in the specific configuration domain of study plans compilation.

We believe that other configuration applications are amenable to this simple paradigm. Future work will be to experiment in different domains in order to define exactly the range of applications that are worth tackling with this approach and to come out with a general enough configuration language.

We also plan to enhance the configuration language by introducing specializations of the basic item data structure and thus the possibility to define different kinds of items through inheritance. We also plan to introduce a support for three-dimensional visualization of configured products, using as a basis the current tree structured display of the config-

https://doi.org/10.1017/S0890060498124058 Published online by Cambridge University Press

uration. We are developing a graphic editor for the choice graph, to give to the configuration expert a higher level tool to define configuration constraints.

Finally, we want to add provisions for security, thus develop new models for submission of the configurations and user authentication, to make the system suitable for electronic commerce.

ACKNOWLEDGMENTS

Many people have contributed in various ways to the development of the CompAss application for plans of study and to the CompAss project in general. We wish to thank, in particular, the project manager, Vincenzo Macrì; and the many people who contributed the expert knowledge about study plans: Cesare Letta, Dipartimento di Scienze Storiche del Mondo Antico; Mirko Tavoni, Dipartimento di Lingue e Letterature Romanze; Paolo Rossi, Dipartimento di Fisica; Theo van Boxel for the graphics design; and Andrea Riboldi for the icons of the graphical interface.

REFERENCES

Barker, V.E., & O'Connor, D.E. (1989). Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM 32(3), 298–318.*

Faltings, B., & Weigel, R. (1994). Constraint-based knowledge representation for configuration systems. Technical Report No. TR-94./59, Département d'Informatique, Laboratoire d'Intelligence Artificielle, EPFL, Lausanne, Switzerland.

Gosling, A.J. (1996). *The Java Programming Language*. Addison Wesley Publishing Co., Reading, Massachusetts.

Hayes-Roth, F., Waterman, D.A., & Lenat, D.B. (Eds.), (1983). *Building Expert Systems*. Addison Wesley Publishing Company, Reading, Massachusetts.

Klein, R., Buchheit, M., & Nutt, W. (1994). Configuration as model construction: The constructive problem solving approach. In *Proc. Third Int. Conf. on Artificial Intelligence in Design*, AID '94, pp. 201–218. Kluwer, The Netherlands.

Männistö, T., Peltonen, H., & Sulonen, R. (1996). View to product configuration knowledge modelling and evolution. In *Configuration Papers from the AAAI Fall Symposium*, (Faltings, B. & E.C. Freuder, Eds.), pp. 111–118. AAAI Press, Boston, MA.

McDermott J. (1982). R1: A rule-based configurer of computer systems. *Artificial Intelligence 19(1),* 39–88.

Stumper, M. (1997). An overview of knowledge-based configuration. *AI Communications 10,* 111–125.

Tiihonen, J., Soininen, T., Männistö, T., & Sulonen, R. (1996). State of the practice in product configuration—a survey of 10 cases in the Finnish industry. In *Knowledge Intensive CAD*, First Edition (T. Tomiyama, M. Mäntylä, S. Finger, Eds.) pp. 95–114. Chapman & Hall, London.

Tiihonen, J., & Soininen, T. (1997). *Product configurators—information system support for configurable products*. Technical Report TKO-B137, Helsinki University of Technology, Laboratory of Information Processing Science. Also published in *Increasing Sales Productivity through the Use of Information Technology during the Sales Visit, A Survey of the European Market*, Hewson Consulting Group.

Giuseppe Attardi is associate professor at the Dipartimento de Informatica of the University of Pisa, where he teaches Computer Graphics and Java programming. He has worked at the MIT Artificial Intelligence Laboratory, at the International Computer Science Institute in Berkeley and at the Sony Research Laboratory in Paris. He has been project leader of several European ESPRIT projects. He participated to the development of Omega, a calculus of descriptions for knowledge representation and reasoning, and of the first MIT window system. He has worked on actor languages and concurrency, and developed ECoLisp, and Embeddable Common Lisp. He also developed CMM (Customisable Memory Manager) a garbage collector for C++. He is currently working on the technique of categorisation by context, for automatically classifying Web documents. Prof. Attardi is an editor of Computational Intelligence and has served as member of several program committees, including IJCAI, ECAI, ECOOP and KR.

Antonio Cisternino received a Diploma degree in computer science from the University of Pisa in 1997. He is currently a fifth year student towards a Laurea degree in computer science. He participated in the development of several projects including CompAss and Sentinel (a security mechanism for Windows95). His main research interests are Web programming and agent programming. He is also active in the RoboCup domain.

Maria Simi is associate professor of Artificial Intelligence at the Dipartimento di Informatica of the University of Pisa. She was visiting scientist at the MIT Artificial Intelligence Laboratory (1978–1981) and at the ICSI, Berkeley (1993). She has been working and published scientific results in the following areas: memory management in systems with contexts, programming methodologies based on abstract data types, programming by demonstration, description logics and taxonomic reasoning, context based knowledge representation and reasoning, and expert systems. Prof. Simi is a founding member of the Italian Association for Artificial Intelligence (AI*IA) and a member of the steering committee from 1988 to 1991. She is a member of the advisory board of the journal "ESRA/Expert Systems research and Application" and of the Editorial Board of the journal "Archivi & Computers".