Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2003), *17*, 51–73. Printed in the USA.
Copyright © 2003 Cambridge University Press 0890-0604/03 $16.00
DOI: 10.1017/S0890060403171053

Decomposition strategies for configuration problems

DIEGO MAGRO AND PIETRO TORASSO
Dipartimento di Informatica, Università di Torino, 10149 Torino, Italy
(RECEIVED March 15, 2002; ACCEPTED October 13, 2002)

Abstract

The paper introduces and discusses the notion of decomposition of a configuration problem within the framework of a structured logical approach. The paper describes under which conditions a given configuration problem can be decomposed into a set of noninteracting subproblems and how to exploit such a decomposition, both for improving the performance of the configurator and for supporting interactive configuration. Different kinds of decomposition are considered, but all of them exploit, as much as possible, the explicit representation of the partonomic relations in the $\mathcal{FPC}$ language, a KL-One like representation formalism augmented with constraints for expressing complex interrole relations. The paper introduces a notion of boundness among constraints, which is used for formally specifying different types of decomposition. One decomposition strategy aims at singling out the components and subcomponents that are directly related to the constraints put by the user's requirements; the configurator exploits such decomposition by first configuring that portion of the product and then configuring the parts that are not related to the user's requirements. Another decomposition strategy verifies whether the set of constraints for the product to be configured can be split into a set of noninteracting problems. In such a case the configurator solves the configuration problem by splitting the whole search space into a set of smaller search spaces. Different combinations of these two decomposition techniques are considered, and the impact of the decomposition strategies on the performance of the configurator is evaluated via a set of experiments using the configuration of computer systems as a test bed. The results of the experiments show a significant reduction of the computational effort (both in terms of number of backtrackings and in CPU time) when decomposition strategies are used.

Keywords: Complete and Partial Configurations; Configuration Problem Decomposition; Partonomic Knowledge; Structured Representation

1. INTRODUCTION

In recent years the configuration task has received a significant amount of attention. In particular, in the last decade configuration has been formalized and two main characterizations have been proposed: one describes the configuration in terms of constraint satisfaction problems (CSPs; e.g., Mittal & Falkenhainer, 1990; Stumptner & Haselböck, 1993; Sabin & Freuder, 1996; Fleischanderl et al., 1998; Soininen & Gelle, 1999; Veron & Aldanondo, 2000) and the other one characterizes configuration in terms of a logical approach (e.g., McGuinness & Wright, 1998; Friedrich & Stumptner, 1999; Soininen et al., 2000). Logical approaches to configuration stress the need for an explicit representation of the structural properties of the entities of the domain in terms of specific kinds of relations. There is an increasing consensus on the opportunity of modeling the domain of interest via high-level structured languages in order to ease the elicitation of knowledge about the domain as well as to make easier the interaction with the user who has to specify a specific configuration problem. Independent of the approach chosen to characterize the configuration task, configuration suffers from the same problem affecting many other relevant tasks dealt with via AI techniques: computational complexity. Since the early days of AI, problem decomposition has emerged as one of the most powerful mechanisms for controlling complexity. Obviously, worst case complexity cannot, in general, be reduced, but from a practical point of view, problem decomposition may prove quite effective in providing a solution to a large variety of problems. As concerns configuration, ideally, the solution of a complex problem should be easily assembled by com-

Reprint requests to: Diego Magro, Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy. E-mail: magro@di.unito.it

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

bining the solutions of the subproblems into which the initial problem has been decomposed. Decomposition of the configuration problem should reduce the computational effort as the solution of a number of simpler subproblems is easier (in terms of computational time) than the solution of a single, large configuration problem.

Although speed-up can be an important goal for justifying the development of decomposition strategies for configuration problems, there is at least another good reason for investigating decomposition. With the development of web-based services, there is an increasing number of interesting applications requiring some form of automatic configuration for supporting the (possibly inexperienced) user in specifying and selecting the right product for his or her needs (e.g., Magro & Torasso, 2001*b*). In some cases the configuration can be seen as an interactive process as at the beginning of the interaction, the user is able to provide just some initial requirements; additional requirements as well as choices of specific components are done in the later steps of interaction. In such kind of situations, the configurator should not be too eager to provide a complete configuration of the product because many of the choices made by the configurator may prove too arbitrary because they are not supported (or guided) by explicit choices or requirements of the user (Magro & Torasso, 2001*a*). Therefore, a second reason for some kind of decomposition concerns the ability of the system to produce partial configurations by configuring only those parts of the product that are directly related to the current user requirements and to postpone the configuration of the remaining parts.

The importance of problem decomposition in configuration was discussed in Fleischanderl and Haselböck (1996) in an approach to configuration based on constraints. However, to the best of our knowledge, the issue of automated problem decomposition in configuration has not received a lot of attention. Configurators based on CSP may exploit state of the art techniques of constraint decomposition for improving problem solving efficiency (Gottlob et al., 2000). However, in some configuration domains, modeling the configuration problems as CSPs can be quite difficult. In addition, for some configuration tasks a more structured representation can be more appropriate, for example, in many tasks in which the explanation of a line of reasoning plays an important role (see McGuinness, 2002). The present paper addresses the problem of defining and comparing a set of automated problem decomposition strategies in a structured logical approach to configuration. In particular, we are interested in defining decomposition techniques that can speed up the configuration process and may help in interactive configuration. All decomposition techniques defined in this paper are guided by the compositional structure (i.e., by partonomic knowledge) of products.

Our approach adopts a structured logical representation formalism for capturing taxonomic and partonomic relations among the entities of the domain and expressing constraints among components and subcomponents. Such a representation formalism (the language $\mathcal{FPC}$) is summarized in Section 2, where a sample domain relevant to PC configuration is also introduced. A formal characterization of configuration problems is given in Section 3, and Section 4 describes the two decomposition techniques the paper deals with and the five decomposition policies that we defined by combining them. Section 5 provides a formal characterization of decomposable problems, and Section 6 describes a configuration algorithm adopting one of the five decomposition policies that we take into consideration. The benefits resulting from the adoption of different decomposition policies are discussed in Section 7, where the results of experiments are reported. Section 8 compares our framework to other ones that have been adopted in configuration. A brief discussion and some suggestions for future work are contained in Section 9.

2. THE CONCEPTUAL LANGUAGE $\mathcal{FPC}$

In this section we describe the main characteristics of $\mathcal{FPC}$ (frames, parts, and constraints; Magro & Torasso, 2000), a conceptual language for representing configuration domain knowledge. In $\mathcal{FPC}$ there is a basic distinction between *atomic* components and *complex* ones. The atomic objects are the basic building components of any configuration, and therefore $\mathcal{FPC}$ abstracts from their internal structure (if any). By contrast, complex objects are characterized in terms of subparts, which can be complex or atomic objects. $\mathcal{FPC}$ offers the possibility of organizing classes of atomic and complex objects in a *taxonomy*, as well as the facility of building *partonomies* that (recursively) express the *whole–part relations* between each complex object and each one of its components. The configuration process has to take into account the *constraints* restricting the set of valid combinations of components and subcomponents. These constraints can be either specific to the modeled domain or derived from the user's requirements.

2.1. Frames and parts

In $\mathcal{FPC}$ each *frame* C represents a *class* $\|C\|$ of objects (either atomic or complex) and it has a set of *member slots* associated with it (as usual, if x is a language element, we denote its semantics with $\|x\|$).[1] Any slot p of a frame C is described via a value restriction D (which can be another frame or can denote a nonempty set $\|D\|$ of values of a predefined kind) and a number restriction (i.e., an integer interval $[m, n]$ with $m \leq n$), as usual in the KL-One like representation formalisms. In any frame C, each slot p with value restriction D and number restriction $[m, n]$ captures the fact that the property p for any component of type C (i.e., for any individual of the class $\|C\|$) is expressed by a

[1] To be precise, we should always distinguish between a frame, which is a language element, and the class that it describes. However, in order to simplify the presentation, we will use the words frame and class as synonyms.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

set of elements whose type is D and whose cardinality belongs to the interval $[m, n]$.

Because of the relevance of the whole–part relation between a complex component and a part of it in the configuration task, a special status has been assigned to such relations, which are modeled via partonomic slots, whereas other properties and features of both complex and atomic objects are modeled via *descriptive* slots. In particular, partonomic slots in $\mathcal{FPC}$ capture the *complex–component relation* described in Gerstl and Pribbenow (1995). This relation is assumed to be asymmetric and transitive. Formally, any partonomic slot p of a frame C is interpreted as a relation $\|p\|$ from $\|C\|$ to the class $\|D\|$ denoted by its value restriction D (for partonomic slots, the value restriction is always a frame) such that $(\forall c \in \|C\|)(m \leq \text{card}(\|p\|(c)) \leq n)$, being $[m, n]$, its number restriction, and $\text{card}(X)$ denotes the cardinality of the set X. The meaning is straightforward: any complex component of type C has from a minimum of m up to a maximum of n parts of type D via a whole–part relation named p.

The value restriction D of any descriptive slot q of a frame C always denotes a nonempty set of values of a predefined kind (i.e., integers, reals, strings, etc.). As in the case of partonomic slots, a descriptive slot is interpreted as a relation $\|q\|$ from $\|C\|$ to $\|D\|$ such that $m \leq \text{card}(\|q\|(c)) \leq n$.

In $\mathcal{FPC}$ taxonomies, each class has a single superclass (i.e., taxonomies are trees); it inherits the slots (both partonomic and descriptive) of its superclass and can refine the description of the superclass by specifying additional (w.r.t. those inherited) slots or by restricting the number restriction and/or the value restriction of any inherited slot (according to the rules of strict inheritance typical of KL-One languages). The subclasses of any class are exhaustive (i.e., any instance of a class must be an instance of one of its subclasses) and disjoint (i.e., there is no instance of a class that belongs to two different subclasses of this class).

Figure 1 represents a fragment of the partonomic and taxonomic knowledge of a conceptual model relevant to PCs. Each rectangle represents a class of complex components, each oval represents a class of atomic components, each solid arrow corresponds to a partonomic slot, and each dashed arrow corresponds to a descriptive slot. The conceptual model sketched in the figure states that a PC is a complex object containing, among other things, a keyboard, from one to seven hard disks, an optional CD writer, a motherboard, and so on. The keyboard, the hard disks, and the CD writer are all atomic components whereas the motherboard is a complex component. The motherboard contains a main printed circuit board (described by its model, price, and type), an optional SCSI controller and other components. In the restricted world we are modeling, the PCs can be of two different types only: advanced application PCs or low cost PCs. Any advanced application PC always has a CD writer and a CD reader (i.e., in the advanced application PC class, the number restriction intervals of the inherited slots has_cdw and has_cdr have been restricted from [0,1] to [1,1]). Moreover, in any advanced application PC, the CD writer, the CD reader, and the hard disks must be the SCSI type. (In fact, the value restriction of the inherited slot has_hd has been restricted from the more general type Hard_Disk to the more specific type HD_SCSI). Analogous restrictions hold for the slots has_cdw and has_cdr). All the other partonomic slots of the PC class are inherited as they are. Unlike low cost PCs, an advanced application PC can have a CD tower (which is a complex component).

Given a $\mathcal{FPC}$ conceptual model CM and a class C_1 of complex components in the CM, a *slot chain* starting in C_1 and ending in D is a sequence $\gamma = \langle p_1, p_2, \ldots, p_h \rangle$ $(h \geq 1)$ of member slots such that in the CM there are the classes $C_2, \ldots, C_{h-1}$ of complex components and the class C_h (either of atomic or complex components) such that C_i $(1 \leq i \leq h-1)$ has a partonomic slot named p_i (possibly inherited) whose value restriction is C_{i+1} and C_h has a slot (either descriptive or partonomic and possibly inherited) named p_h whose value restriction is D, which can be either a class of (complex or atomic) objects of the CM (if p_n is a partonomic slot) or a set of $\mathcal{FPC}$ predefined values (if p_n is a descriptive slot). The semantics of such a slot chain is a natural generalization of the semantics of a single slot, because it is interpreted as the relation composition $\|\langle p_1, p_2, \ldots, p_h \rangle\| \equiv \|p_h\| \circ \|p_{h-1}\| \circ \cdots \circ \|p_1\|$ from $\|C_1\|$ to $\|D\|$. That is, if $h = 1$, $\|\langle p_1 \rangle\|$ is a relation from $\|C_1\|$ to $\|D\|$, such that $(\forall c_1 \in \|C_1\|)\|\langle p_1 \rangle\|(c_1) = \|p_1\|(c_1)$; if $i \geq 1$ and $h = i + 1$, $\|\langle p_1, \ldots, p_{i+1} \rangle\|$ is a relation from $\|C_1\|$ to $\|D\|$, such that $(\forall c_1 \in \|C_1\|)(\|\langle p_1, \ldots, p_{i+1} \rangle\|(c_1) \equiv \bigcup_{c_{i+1} \in \|\langle p_1, \ldots, p_i \rangle\|(c_1)} \|p_{i+1}\|(c_{i+1}))$, where the operator $\bigcup$ maintains repeated values of $\mathcal{FPC}$ predefined types, in case p_{i+1} is a descriptive slot.

More intuitively, if p_h is a partonomic slot, the chain represents the subcomponents of a complex component $c_1 \in \|C_1\|$ via the has-part relationships named $p_1, \ldots, p_h$. The subcomponents referred to by the slot chain form a set. (No subcomponent is repeated inside it.)

If p_h is a descriptive slot, this chain represents the values of the property p_h for the subcomponents of c_1 via the has-part relationships $p_1, \ldots, p_{h-1}$. Because the property p_h may have the same values in different subcomponents, the slot chain denotes a multiset (i.e., there can be repeated property values).

Similarly, a set of slot chains $R = \{\gamma_1, \ldots, \gamma_k\} (k \geq 1)$ [each one starting in C_1 and ending in D_i $(i = 1, \ldots, k)$] is interpreted as the relation union $\|R\| \equiv \bigcup_{i=1}^{k} \|\gamma_i\|$ from C_1 to $\bigcup_{i=1}^{k} D_i$.

For example, in Figure 1, the slot chain ⟨has_m,has_mpcb⟩ starting in PC represents the main printed circuit board of a PC, while the price of that component in a PC is represented by the chain ⟨has_m,has_mpcb,price⟩. The set of slot chains starting in PC {⟨has_hd⟩ ⟨has_cdr⟩⟨has_cdw⟩} represents the set of hard disks, CD reader, and CD writer of a PC.

2.2. Constraints

In modeling the real-world domains the expressive power provided by the frame portion of $\mathcal{FPC}$ is not sufficient; in

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

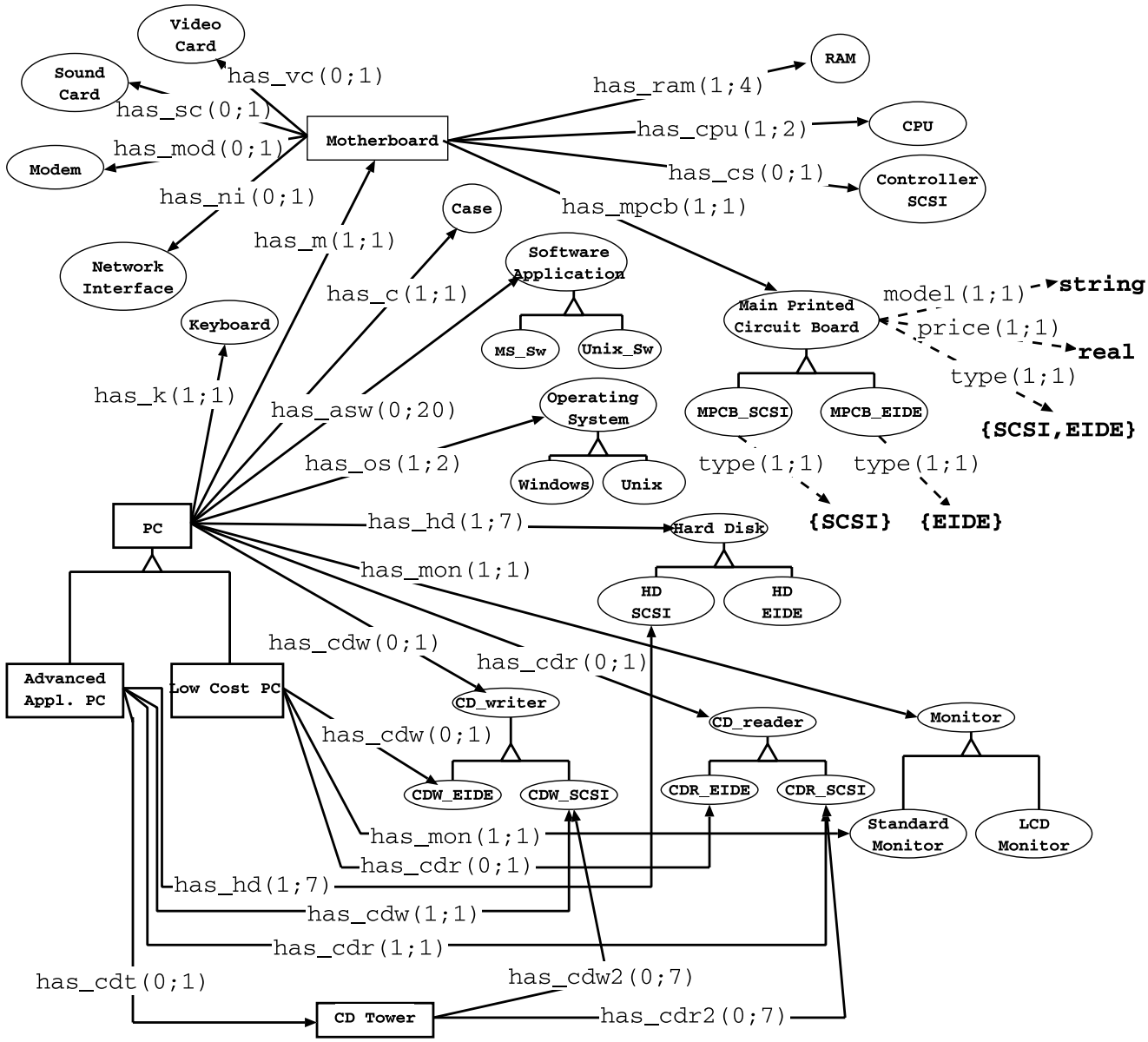


Fig. 1. A sample (portion of a) conceptual model: the partotaxonomic knowledge.

particular, number restrictions and value restrictions cannot capture complex interslot constraints relating the components and the subcomponents of the complex objects.

For this reason, $\mathcal{FPC}$ puts a constraint language at our disposal. However, each constraint is associated with a class of complex components in order to have a uniform structured approach to knowledge representation. In this way, constraints complement partonomic and descriptive slots in capturing the characteristics of a complex object.

Given the class C and a constraint co associated with it, co is of the form $\alpha \Rightarrow \beta$, where α is a conjunction of $\mathcal{FPC}$ predicates or the Boolean constant *true* and β is a predicate or the Boolean constant *false*. The meaning is that for every complex object $c \in \|C\|$, if c satisfies α then it must satisfy β. It should be clear that if α = true, then, for each $c \in \|C\|$, β must always hold, while if β = false, then, for each $c \in \|C\|$, α can never hold. The contradiction true $\Rightarrow$ false is not a valid constraint.[2]

Each class (but the root of the taxonomy) inherits all the constraints of its superclass.

In the following, only a subset of the predicates available in $\mathcal{FPC}$ is described. Let $R = \{\gamma_1, \ldots, \gamma_k\}$ be a set of slot chains starting in a class C. The last slots of the chains occurring in R can either be all descriptive or all parto-

[2] Actually, $\mathcal{FPC}$ is a bit more general, because it allows a constraint to be a clause whose literals are positive or negative $\mathcal{FPC}$ predicates. However, here we restrict the attention to constraints having a Horn-clause form. This restriction does not affect the decomposition mechanisms that we describe.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

nomic. Therefore, in the former case, for each $c \in \|C\|$, $\|R\|(c)$ can generally be a multiset of property values, whereas in the latter it is a set of components.

Four types of predicates are briefly described.

1. $(R)(m;n)$, where the last slots in the chains occurring in R are partonomic and m, n are nonnegative integers with $m \leq n$.
   Here $c \in \|C\|$ satisfies the predicate iff $m \leq \text{card}(\|R\|(c)) \leq n$.
2. $(R)(inI_1 \cup \cdots \cup I_s)$, with $(s \geq 1)$.
   If the last slots in the chains occurring in R are partonomic, each $I_j (j = 1,\ldots,s)$ is a class of components and $c \in \|C\|$ satisfies the predicate iff $\|R\|(c) \subseteq \|I_1\| \cup \cdots \cup \|I_s\|$. Similarly, if the last slots in the chains occurring in R are descriptive, each I_j denotes a set of $\mathcal{FPC}$ predefined values and $c \in \|C\|$ satisfies the predicate iff each $v \in \|R\|(c)$ belongs to $\|I_1\| \cup \cdots \cup \|I_s\|$.
3. $(R)(inI_1 \cup \cdots \cup I_s(m;n))$, where $(s \geq 1)$, the last slots in the chains occurring in R are partonomic, each I_j $(j = 1,\ldots,s)$ is a class of components and m, n are nonnegative integers with $m \leq n$.
   Here $c \in \|C\|$ satisfies the predicate iff $m \leq \text{card}(\|R\|(c) \cap (\|I_1\| \cup \cdots \cup \|I_s\|)) \leq n$.
4. $(f(R))$ RelOp $(g(S))$, where $S = \{\delta_1,\ldots,\delta_t\}$ is a set of slot chains starting in C and different from R, the last slots in the chains occurring in R or S are descriptive, and their value restrictions denote numeric sets.
   Here $c \in \|C\|$ satisfies the predicate iff $f(\|R\|(c))$ RelOp $g(\|S\|(c))$, where RelOp $\in \{=, <, >, \leq, \geq\}$ and f and g are two functions belonging to the set of built-in functions provided by the conceptual language. [These functions include the sum $(\sum)$, the minimum (min), and the maximum (*max*).] For example, $c \in \|C\|$ satisfies $(\max(R)) \leq (\sum(S))$ iff $\max(\|R\|(c)) \leq \sum_{v \in \|S\|(c)} v$.

A (small) subset of the constraints holding in a real computer system domain is reported in Figure 2, where a short explanation in natural language is provided for each constraint.

3. CHARACTERIZING CONFIGURATION PROBLEMS

In this section we formalize both the concept of configuration problem and the notion of solution of a configuration problem in case the goal is the assembly of a complete complex object meeting a set of requirements. As we shall see, sometimes it is not necessary to build the whole complex product and a partial configuration is enough. A definition of solution of a configuration problem capturing the concept of partial configuration will be given in Section 5.

Given a $\mathcal{FPC}$ CM, let MSClasses(CM) be the set of classes in the CM most specific in the taxonomy (i.e., the set of the leaves in the taxonomy); these classes fully describe the types of components that may occur in configurations and they are called *concrete classes*. In particular, MSClasses(CM) = A_MSClasses(CM) ∪ C_MSClasses(CM), where A_MSClasses(CM) and C_MSClasses(CM) are the concrete classes of atomic and complex objects, respectively.

If $C \in$ MSClasses(CM), with p_slots(CM,C) and d_slots(CM,C) we indicate the partonomic slots and the descriptive slots of class C in the CM (possibly inherited), respectively; slots(CM,C) = p_slots(CM,C) ∪ d_slots(CM,C) is the set of all the slots of class C in the CM.

To get rid of some superflous details, in the following we assume that the conceptual descriptions of the concrete classes of atomic components are specific enough that the values of all the properties of any atomic component c are unambigously determined by selecting the concrete type of c. Formally, for each $C \in$ A_MSClasses(CM) and each $d \in d$_slots(CM,C), the cardinality restriction of d is an interval $[m,m]$ and its value restriction D is such that $\text{card}(\|D\|) = m$.

Definition 1. A configuration problem is a tuple CP = ⟨CM,C,REQS⟩, where

1. CM is a $\mathcal{FPC}$ conceptual model,
2. $C \in C$_MSClasses(CM), and
3. REQS is a set of $\mathcal{FPC}$ constraints whose slot chains start in C. ■

In the above CP represents the problem of configuring a complex object of type C in such a way that both the user's requirements REQS and the conceptual model CM are satisfied. In the following the object to be configured is called the target object.

Definition 2 (tentative configuration). Given a configuration problem CP = ⟨CM,C,REQS⟩, a tentative configuration for CP is a pair T = ⟨COMPS,INST_SLOTS⟩, such that:

1. COMPS is a set of components (for each $c \in$ COMPS, class(T,c) ∈ MSClasses(CM) indicates the concrete class to which c belongs);
2. there is exactly one component $c \in$ COMPS such that class(T,c) = C (representing the target object);
3. INST_SLOTS is a set of equalities $\langle p(c) = X \rangle$, where $c \in$ COMPS, $p \in$ slots(CM,class(T,c)), no two different equalities $\langle p(c) = X \rangle$ and $\langle p(c) = Y \rangle$ occur in INST_SLOTS and
   (a) if $p \in p$_slots(CM,class(T,c)), then $X \subset$ COMPS (in this case, the equality $\langle p(c) = X \rangle$ means that c has the set X of direct components through the whole–part relation named p);
   (b) if $p \in d$_slots(CM,class(T,c)), then X is a set of values of $\mathcal{FPC}$ predefined types (in this case, the equality $\langle p(c) = X \rangle$ means that X is the set of values relevant to the property denoted by the slot p for the component c).

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

```
CONSTRAINTS

Associated with PC class:

In any PC, if there is at least one SCSI hard disk and an EIDE main printed circuit board,
then a controller SCSI is needed
[co1] ({<has_hd>})(in HD_SCSI (1;7))∧ ({<has_m,has_mpcb>})(in MPCB_EIDE) ==> ({<has_m,has_cs>})(1;1)

In any PC, if there is a SCSI CD reader and an EIDE main printed circuit board, then a controller
SCSI is needed
[co2]({<has_cdr>})(in CDR_SCSI (1;1))∧ ({<has_m,has_mpcb>})(in MPCB_EIDE) ==> ({<has_m,has_cs>})(1;1)

In any PC, if there is a SCSI CD writer and an EIDE main printed circuit board, then a controller
SCSI is needed
[co3]({<has_cdw>})(in CDW_SCSI (1;1))∧ ({<has_m,has_mpcb>})(in MPCB_EIDE) ==> ({<has_m,has_cs>})(1;1)

In any PC, if there is a Unix application software, then a Unix operating System is needed
[co4] ({<has_asw>})(in Unix_Sw (1;20)) ==> ({<has_os>})(in Unix (1;2))

In any PC, if there is a Microsoft application software, then a Windows operating system is needed
[co5] ({<has_asw>})(in MS_Sw (1;20)) ==> ({<has_os>})(in Windows (1;2))

In any PC, there must be at most 4 EIDE devices
[co6] true ==> ({<has_hd>,<has_cdr>,<has_cdw>})(in HD_EIDE U CDR_EIDE U CDW_EIDE (0;4))


Associated with the Advanced Applications PC class:

[co1]...[co6] inherited from PC class

In any Advanced Appl. PC, there must be two CPUs
[co7] true ==> ({<has_m,has_cpu>})(2;2)


Associated with the Low Cost PC class:

[co1]...[co6] inherited from PC class

Associated with the Motherboard class:

In any motherboard, if there is a SCSI main printed circuit board, then there can't be any
additional controller SCSI
[co8] ({<has_mpcb>})(in MPCB_SCSI) ==> ({<has_cs>})(0;0)


Associated with the CD Tower Class:

In any CD Tower, the number of CD Readers and CD Writers must be greater than zero and less than 8
[co9] true ==> ({<has_cdw2>,<has_cdr2>})(1;7)
```

Fig. 2. A sample (portion of a) conceptual model: the constraints.

Let PartOf(c) = $\{d \in \text{COMPS}/(\exists\langle p(d) = X\rangle \in \text{INST_SLOTS})(p \in p_\text{slots}(\text{CM},\text{class}(T,d)) \wedge c \in X)\}$ be the set of components in the tentative configuration T of which $c \in$ COMPS is a direct part:

4. for the component $c \in$ COMPS such that class$(T,c) = C$, card(PartOf(c)) = 0;
5. for each component $c \in$ COMPS such that class$(T,c) \neq C$, card(PartOf(c)) = 1;
6. for each component $c \in$ COMPS and for each $p_1, p_2 \in p_\text{slots}(\text{CM},\text{class}(T,c))$ (with $p_1 \not\equiv p_2$), if two equalities $\langle p_1(c) = X\rangle$ and $\langle p_2(c) = Y\rangle$ occur in INST_SLOTS, then $X \cap Y = \emptyset$;
7. T respects the partotaxonomic restrictions stated in CM;
8. no constraint stated in CM is violated in T;
9. no requirement occurring in REQS is violated in T. ■

A tentative configuration can be seen as a snapshot taken during the configuration process and it represents a configuration "under construction."

COMPS is the set of components occurring in the tentative configuration; condition 2 assures that exactly one instance of the target object is being configured. Because each slot p is interpreted as a relation $\|p\|$ (see Section 2),

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

each equality $\langle p(c) = X \rangle \in$ INST_SLOTS expresses the value of this relation, computed on the component c (i.e., it states that $\|p\|(c) = X$).

Condition 4 assures that no component containing the target object has been introduced into the tentative configuration. Conditions 5 and 6 express the following assumption:

ASSUMPTION 1 (exclusiveness assumption on parts). A component cannot be a direct part of two different components (condition 5), neither a direct part of a same component through two different partonomic relations (condition 6). ■

Condition 5 also states that each component different from the target object must be a direct part of another complex component (possibly the target object).

Condition 7 states that the tentative configuration respects the number and the value restrictions of each slot occurring in it. A tentative configuration, according to Definition 2 reported above, may contain no information for some slot p present in the conceptual model. In this case, computing the truth value of any constraint mentioning p is not always possible in T (i.e., this value could be unknown in T). Conditions 8 and 9 state that in a tentative configuration the truth value of each constraint (either occurring in the conceptual model or imposed as a requirement) can be either true or unknown, but not false.

Given Definition 2 above, a tentative configuration T can be represented as a tree whose nodes are the components in COMPS or the values (possibly repeated) of $\mathcal{FPC}$ predefined types that occur in the equalities contained in INST_SLOTS, whose root is the component c for which $class(T, c) = C$ and such that for each equality $\langle p(c) = \{c_1, \ldots c_n\} \rangle \in$ INST_SLOTS with $n \geq 1$, there are n edges labeled p, each one connecting the node c with its child $c_i (i = 1, \ldots, n)$ and for each equality $\langle p(c) = \emptyset \rangle \in$ INST_SLOTS there is an edge connecting c with a node denoting the empty set (no edge other than these is present in the tree).

DEFINITION 3 (complete configuration). A solution to a configuration problem CP = ⟨CM, C,REQS⟩ is a tentative configuration T = ⟨COMPS,INST_SLOTS⟩ for CP, such that:

1. for each component $c \in$ COMPS and each slot $p \in$ slots(CM,class(T, c)), there is an equality $\langle p(c) = X \rangle \in$ INST_SLOTS;
2. each constraint stated in CM for the classes of the components in COMPS is satisfied in T;
3. the requirements REQS are satisfied in T. ■

Any T respecting Definition 3 is called a complete configuration for the configuration problem CP.

Condition 1 assures the *completeness* of the configuration, because a value is specified for each slot of any component occurring in it.

The last two conditions state that in a complete configuration the truth value of each constraint (either occurring in the conceptual model or imposed as a requirement) is *true*. (It is worth noting that these two conditions are stricter than conditions 8 and 9 in Definition 2.)

As usual in configuration tasks, we make the following assumption.

ASSUMPTION 2 (consistency of the conceptual model). During problem solving, the conceptual model is consistent, where an $\mathcal{FPC}$ CM is consistent iff for each concrete class C of complex objects occurring in CM, the configuration problem ⟨CM, C,Ø⟩ admits at least one solution that is a complete configuration. ■

It is worth noting that the assumption above is general enough even if it explicitly refers only to concrete classes of complex objects. In fact, each class of atomic objects is always consistently defined, provided that for each slot occurring in it, both the cardinality and the value restrictions are nonempty sets and that the restriction rules for inherited slots stated in Section 2.1 are followed. If the conceptual model did not contain any constraint, these observations would apply also to classes of complex objects. The assumption above assures that when the conceptual model also contains $\mathcal{FPC}$ constraints, the concrete classes of complex objects have meaningful descriptions (i.e., for each concrete class C of complex objects occurring in the conceptual model, at least one individual of that class can always be fully specified. In other words, the problem of finding a complete configuration for a complex object of type C, without any requirement but the satisfaction of its conceptual description, always has a solution). The concrete classes are the leaves of $\mathcal{FPC}$ taxonomies; thus, their consistency implies the consistency of all classes occurring in the conceptual model.

4. REQUIREMENTS-BASED DECOMPOSITION AND CONSTRAINTS-SPLITTING DECOMPOSITION

4.1. A motivating example

As stated in the Introduction, the automatic configuration task can be, in general, computationally expensive. In fact, the components and the subcomponents of a configurable object usually have to respect a set of constraints, both endogenous (i.e. specific to the modeled domain, such as technical or physical constraints) and exogenous (imposed as additional requirements by any agent requiring a customized product). Usually, these constraints link together many components and subcomponents. Thus, rarely can a configuration be computed by a process making only a set of local choices. In most cases the configuration task is performed by a search process. When a choice made for a component during the configuration activity prevents another (or even the same) component from being configured, the process

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

```
CONFIGURATION PROBLEM:
Configuring an Advanced Applications PC satisfying the following requirements:

Containing a SCSI Main printed circuit board
[req1] true ==> ({<has_m,has_mpcb>})(in MPCB_SCSI)

With the applX Unix application software
[req2] true ==> ({<has_asw>})(in applX (1;1))

With a CD Tower
[req3] true ==> ({<has_cdt>})(1;1)
```

Fig. 3. The user's requirements for an advanced application PC.

(if possible) backtracks, revises a past choice, and explores a different path in the search space.

If we consider the conceptual model in Figures 1 and 2,[3] we see, for example, that the (endogenous) constraints co4 and co5 link together the operating systems and the software applications components. The choice of the operating systems to be installed in any PC cannot be a *local* choice; in fact, the satisfaction of the number and value restriction of the has_os slot is not enough (i.e., installing at least one and at most two Unix or Windows type operating systems). The operating systems that should be inserted in the configuration of a PC depends on the kind of the software applications that have been included in it, and vice versa. Similar interdependencies hold, for instance, between the main printed circuit board and the SCSI controller (see the constraints from co1 to co3 and co8).

In many cases, the search space is quite huge and many paths in it do not lead to any solution. Moreover, there is no guarantee that a solution always exists for any configuration problem.

Because we are interested in problem decomposition as a way of controlling complexity and for supporting interactive configuration, we explore two different decomposition strategies.

4.1.1. Requirements-based decomposition

A first kind of decomposition (requirements-based decomposition) requires the ability of recognizing which components and subcomponents might be critical for the satisfaction of the user's requirements. In this way, the problem of configuring a complex object can be decomposed into two subproblems: the subproblem of configuring those parts of the target object that are related in some way to the requirements (i.e., to the exogenous constraints) and the subproblem of completing such a partial configuration by considering the remaining components.

The consistency assumption on the conceptual model (Assumption 2) gives a good reason for considering first the components and subcomponents that might be critical for the satisfaction of the user's requirements and then the other ones. In fact, given this assumption, if a configuration problem has no solution, this is due to the exogenous constraints (which can be inconsistent either by themselves or w.r.t. the conceptual model). By first considering the components possibly involved in these constraints, we expect that the inconsistency of the requirements will be detected early (in average) in the configuration process (saving useless work).

Let us suppose, for instance, that a user states the set of inconsistent requirements consisting in those listed in Figure 3 plus the requirement that no Unix operating system should be installed ($([\text{req4}]\text{true} \Rightarrow (\{< \text{has_os} >\})(\text{in-Unix}(0;0))$). It should be clear that the constraints req2, req4, and co4 cannot be satisfied in the same configuration. The components that can be critical for meeting the requirements from req1 to req4 are the main printed circuit board, the software applications, the operating systems, and the CD tower, because they are directly mentioned in the requirements. Besides these components, the SCSI controller must also be considered, because it is related to the main printed circuit board by the constraints co1, co2, co3, and co8; furthermore, all the other components referred to by these constraints have to be taken into consideration (hard disks, CD readers, and CD writers).[4]

The other components (such as the keyboard, video cards, RAM memory slots, and so on) do not have any influence on the requirements. In this case, introducing these last components into the configuration would be a waste of time, because no solution can be found, whatever choices are made for them.

Moreover, requirements-based decomposition allows the configurator to perform a more focused search (even when the requirements can be met). In fact, whenever the config-

[3] The conceptual model represented in these figures expresses only a fragment of the knowledge actually needed to configure a PC. However, in the examples, we assume that it is a complete model of the domain.

[4] Actually, the decomposition mechanism that we shall present in the following would consider also the CPUs as components *possibly* critical for the requirements satisfaction.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

uration process enters a failure path in the search space in which a requirement is violated, a revision of any choice made for a component that had been recognized *a priori* to be independent from the requirements would be useless and it can be easily avoided by considering such components in a second phase.

The ability to distinguish which parts of the configurable object might be critical for the requirements satisfaction also has other motivations. In fact, if the task is that of checking the consistency of the requirements, the configuration of the whole product would be a useless effort in many cases. In this case, the computation of a partial configuration, involving only the components that may prove critical for the requirements, is enough. Moreover, this ability is important in any system supporting an interactive configuration. In this task, we cannot assume that all the requirements are always known from the beginning because they can be incrementally added by the user. In this scenario, the system should not be too eager to complete the work (i.e., to configure the entire target object); it is important for it to recognize which components can be critical for the satisfaction of the current set of requirements and to limit its choices to them by leaving the user the possibility of adding some requirements for components not yet introduced into the configuration.

In the (consistent) set of requirements in Figure 3, req3 specifies that a CD tower must be present in the configuration. However, nothing else is stated about the CD tower. If, for example, we want to allow the user to express some requirements for it in a second phase, the configurator should only insert into a configuration a (generic) CD tower without configuring it. The user can specify at a later time how many and which kinds of CD readers and CD writers she/he wants. Furthermore, when a product needs to be reconfigured in order to meet a new set of requirements, it can be helpful to single out the subset of components that will be involved in the reconfiguration process.

4.1.2. Constraints-splitting decomposition

A second kind of decomposition (constraints-splitting decomposition) is based on a different observation. As we said, the constraints (both endogenous and exogenous) usually link together the components and the subcomponents. Moreover, the constraints themselves can be *bound* [through the (sub)components to which they are relevant] in a way that a choice made in order to satisfy one of them may restrict the choices available for satisfying another one.

We saw that in the considered conceptual model the software applications and the operating systems are linked together by the constraints co4 and co5. In other words, we can say that these two constraints are bound together. In fact, in order to satisfy co4, we could install in a PC two (different) Unix operating systems. This choice has an impact on the choices available for satisfying co5, since, given the previous choice, the only valid possibility is not to install any Microsoft application. (Note that the choice of adding both some Microsoft applications and a Windows operating system is not available any more, because the conceptual model states that at most two operating systems can be installed in a PC[5] and two Unix operating systems have already been installed.)

However, in many cases not every constraint interacts with each other and the ability of recognizing the sets of (possibly) interacting constraints offers the basis for a problem decomposition that considers each set of bound constraints separately.

Let us again consider the requirements in Figure 3. The constraints co4, co5, and req2 refer to the same components (i.e., software applications and operating systems), thus they interact among themselves. However, req3 interacts neither with the other requirements nor with the endogenous constraints. This means that if, for instance, the configuration process reaches a region in the search space where one or more constraints among co4, co5, and req2 are violated, revising a previous choice for the CD tower would not move the search away from the failure region. This suggests that the requirement req3 and the set of constraints co4, co5, and req2 could be considered separately (either in sequence or in parallel) in order to have a more focused search.

As we shall see, in the $\mathcal{FPC}$ framework, the partonomic knowledge is the basis for performing both kinds of decomposition.

Table 1. *Decomposition policies*

		Requirements Based		
		No_dist	After	Never
Constraints splitting	No	1	2	4
	Yes	—	3	5

4.2. The decomposition policies

Table 1 summarizes the decomposition policies that we are considering. Because the two kinds of decomposition (i.e., requirements-based decomposition and constraints-splitting decomposition) are two orthogonal dimensions, a decomposition policy is defined by selecting a value for each dimension.

In requirements-based decomposition, there are three distinct choices:

1. *no_dist*, if no distinction is performed between the (sub)components possibly critical for the requirements satisfaction and those that have no influence on

[5]This is a somewhat arbitrary restriction that we introduced in our simplified conceptual model.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

it (thus, no requirements-based decomposition is actually performed);
2. *after*, if the (sub)components possibly critical for the requirements satisfaction are kept distinct from the other ones; both set of (sub)components are taken into consideration, but the noncritical (sub)components are considered after all the critical ones;
3. *never*, as in the previous case, but only the set of possibly critical (sub)components is considered; the other ones are never taken into consideration.

For the constraints-splitting decomposition there are only two possible values: *no*, meaning that no constraints-splitting decomposition is performed, and *yes* meaning the opposite. The latter type of decomposition works only on the set of constraints *directly* associated with the *target object* (i.e. the one that should be configured).[6]

In Table 1, each policy that we take into consideration is identified by a number. The policies for which Requirements_based ≠ never (i.e., policies 1, 2, and 3) configure the whole product. The other ones (i.e., 4 and 5) leave out of the configuration those components that are recognized to have no influence on the possibility of meeting the requirements.

The entry with the dash corresponds to a combination that we do not consider. In fact, whenever we do not make the "coarser" distinction between the components (possibly) related to the requirements and the other ones, it also seems reasonable to not perform the "more detailed" analysis on the interaction among constraints.

5. FORMAL CHARACTERIZATION OF DECOMPOSABLE PROBLEMS

The decomposition mechanisms are based on a *bound* relation among constraints that makes use of the partonomic knowledge in order to recognize the possibility of interaction among constraints. Intuitively, two constraints are bound if and only if the choices made during the configuration process in order to satisfy one of them *could* influence those actually available for the satisfaction of the second one.

For the sake of simplicity and without loss of generality, in the following we assume that in the conceptual model the classes of complex components have only partonomic slots.

DEFINITION 4. Given a configuration problem CP = ⟨CM, *C*,REQS⟩ and a tentative configuration T = ⟨COMPS, INST_SLOTS⟩ for CP, the set CONSTRS(c) of constraints relevant to a complex component $c \in$ COMPS is defined as follows:

- if c is the target object, CONSTRS(c) is the set of constraints in REQS plus those associated with class(T, c) in the conceptual model CM;
- if c is a (sub)component of the target object, and $c' \in$ COMPS is the component containing c as a direct part, CONSTRS(c) is the set of constraints in CONSTRS(c') mentioning some slot relevant to component c plus those associated with class(T, c) in CM. ■

The bound relation $\mathcal{B}_c$ is defined on any subset CON $\subseteq$ CONSTRS(c) as follows.

DEFINITION 5. Let $u, v, w \in$ CON; if u and v contain a partonomic slot p of class(T, c) then $u\mathcal{B}_c v$ (i.e., if u and v refer to the same direct part of c, they are bound); if $u\mathcal{B}_c v$ and $v\mathcal{B}_c w$ then $u\mathcal{B}_c w$ (transitivity); the constraints cannot be bound in any other way. ■

It is easy to see that $\mathcal{B}_c$ is an equivalence relation.

PROPERTY 1. If U is an equivalence class in the quotient set CONSTRS(c)/$\mathcal{B}_c$, every constraint in U can interact with any other constraint in U in the configuration process of c. Whereas, if V $\in$ CONSTRS(c)/$\mathcal{B}_c$ is different from U, because of the exclusiveness assumption on parts (Assumption 1), constraints in V do not interact in any way with those in U in the configuration process of component c. ■

Let us suppose, for example, that we want to configure an advanced application PC (Figs. 1 and 2) aapc1 meeting the requirements in Figure 3. The set of constraints relevant to aapc1 is

CONSTRS(aapc1)
= {co1,co2,co3,co4,co5,co6,co7,req1,req2,req3};

the quotient set induced by the bound relation $\mathcal{B}_{\text{aapc1}}$ is

CONSTRS(aapc1)/$\mathcal{B}_{\text{aapc1}}$
= {{co1,co2,co3,co6,co7,req1},{co4,co5,req2},{req3}}.

The main problem of configuring aapc1 can be split into three noninteracting subproblems, each one corresponding to a different class of constraints in the quotient set.

DEFINITION 6. Given a configuration problem CP = ⟨CM, *C*,REQS⟩, a tentative configuration T = ⟨COMPS, INST_SLOTS⟩ for CP, a component $c \in$ COMPS and the the set CONSTRS(c) defined in Definition 4, the subset CONSTRS$_{\text{REQS}}$(c) $\subseteq$ CONSTRS(c) of the constraints relevant to component c that are bound to the requirements is defined as follows:

- if c is the target object, CONSTRS$_{\text{REQS}}$(c) is the set of constraints $v \in$ CONSTRS(c) such that there exists a requirement req $\in$ REQS such that $v\mathcal{B}_c$req;
- if c is a (sub)component of the target object, and $c' \in$ COMPS is the component containing c as a direct part, CONSTRS$_{\text{REQS}}$(c) is the the set of constraints $u \in$ CONSTRS$_{\text{REQS}}$(c') that mention some

[6] A recursive constraints-splitting decomposition could be defined as one that also works on the constraints associated with the components and subcomponents of the target object. However, it is worth investigating the benefits of the nonrecursive constraints-splitting decomposition, because its implementation is simpler than that of its recursive version.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

slot relevant to component c (let us call this set S), plus all the constraints v associated with class(T,c) in CM such that there exists a constraint $u \in S$ such that $u\mathcal{B}_c v$. ■

Section 6.2 reports two examples of sets of constraints bound to the requirements computed both for the target object and for a direct component of it.

As mentioned in Section 4.1, in some cases, the configurator should compute partial configurations by taking into consideration a subset of the (sub)components of the target object. The following definition exploits the bound relation to formally characterize when a partial configuration is a solution of a configuration problem.

DEFINITION 7 (partial configuration). A solution to a configuration problem CP = ⟨CM, C,REQS⟩ is a tentative configuration T = ⟨COMPS,INST_SLOTS⟩, such that:

1. for each component $c \in$ COMPS and each partonomic slot $p \in p_\text{slots}(\text{class}(T,c))$, if no equality $\langle p(c) = X\rangle$ occurs in INST_SLOTS, then p does not occur in any constraint $v \in \text{CONSTRS}_{\text{REQS}}(c)$ or the truth value of each constraint $v \in \text{CONSTRS}_{\text{REQS}}(c)$ in which p does occur can be determined in T without knowing the value of $\|p\|(c)$;
2. no constraint stated in CM is violated in T and for each component $c \in$ COMPS the constraints associated with class(T,c) belonging to $\text{CONSTRS}_{\text{REQS}}(c)$ are satisfied in T.
3. the requirements REQS are satisfied in T;
4. for each component $c \in$ COMPS and each partonomic slot $p \in p_\text{slots}(\text{CM},\text{class}(T,c))$, if p does not occur in any constraint of the set $\text{CONSTRS}_{\text{REQS}}(c)$, no equality $\langle p(c) = X\rangle$ occurs in INST_SLOTS;
5. for each atomic component $c \in$ COMPS and each descriptive slot $d \in d_\text{slots}(\text{CM},\text{class}(T,c))$, there is an equality $\langle d(c) = X\rangle \in$ INST_SLOTS. ■

Any T respecting Definition 7 is called a partial configuration.

Conditions 1 and 2 in the definition above are a weak form of the same conditions in Definition 3. In fact, condition 1 in Definition 3 requires that the set of direct parts of each component occurring in the configuration is always specified for each partonomic relation of that component. On the contrary, each (partial) configuration satisfying Definition 7 can leave unspecified the set of direct parts of a component $c \in$ COMPS through a whole–part relation named *p, provided that p is not essential for satisfying some constraint bound to the requirements* in which it occurs (i.e. it must be possible to compute the truth value of such a constraint in T and that value must be *true*).

We assumed that the conceptual descriptions of concrete atomic classes contain the values of each property of their individuals (Section 3); therefore, for each atomic component $c \in$ COMPS, the values of its properties are specified in class(T,c) and they are inserted into the configuration (condition 5).

Similarly, condition 2 in Definition 3 requires that *all* the constraints occurring in CM for the classes of the components are satisfied in T (i.e., their truth value is true in T). In contrast, condition 2 in Definition 7 requires only that all those constraints in *CM that are bound to the requirements* are satisfied in T. The other ones need only to be not violated (i.e., their truth value in T can be either true or unknown).

Condition 3 is the same as in Definition 3, while condition 4 assures that only the components and the subcomponents related to the requirements occur in T.

PROPERTY 2 (completion). Each partial configuration T = ⟨COMPS,INST_SLOTS⟩ solving the configuration problem CP = ⟨CM, C,REQS⟩, according to Definition 7, can always be extended to a complete configuration solving CP according to Definition 3. ■

Sketch of the proof. Let $c \in$ COMPS and $\{p_1,\ldots,p_n\}$ be the set of partonomic slots of class(T,c) that have been left unspecified in T. ■

The assumption of consistency of the CM (Assumption 2) assures that it is always possible to compute a complete configuration $\overline{T}$ = ⟨$\overline{\text{COMPS}}$,$\overline{\text{INST_SLOTS}}$⟩ for an instance $\bar{c}$ of class(T,c). For condition 1 of the complete configuration definition (Definition 3), for each $p_i \in \{p_1,\ldots,p_n\}$, there is an equality $\langle p_i(\bar{c}) = X_i\rangle \in \overline{\text{INST_SLOTS}}$.

If each X_i is a set of complex components, it is easy to see that

$$\text{EXT_T} = \left\langle \text{COMPS} \cup \left(\bigcup_i^n X_i\right), \text{INST_SLOTS} \cup \left(\bigcup_i^n \{\langle p_i(c) = X_i\rangle\}\right)\right\rangle$$

is a tentative configuration for CP that extends T and still satisfies all the conditions listed in Definition 7 of the partial configuration except for condition 4. If any X_i is a set of atomic components, for each $x_i \in X_i$, the equalities $\langle d(x_i) = Y\rangle$ contained in $\overline{\text{INST_SLOTS}}$ have to be added to INST_SLOTS, too.

The concept of EXT_T can be further extended by considering each complex component $x \in \bigcup_i^n X_i$ and by (recursively) specifying it in the same way as specified in the complete configuration $\overline{T}$. Again, it is easy to see that the new tentative configuration EXT_T′ built in such a way still satisfies all the conditions listed in Definition 7 except for condition 4.

By repeating this kind of extension for each component $c \in$ COMPS occurring in the original partial configuration T, a complete configuration solving CP w.r.t. Definition 3 can be built. ■

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

```
PROCEDURE mainProc(CM,C,REQS){
   T = ⟨{c},∅⟩;/*c represents the target object*/
   class(T,c) = C;
   CONSTRSREQS(c) = REQS ∪ {con ∈ constrs(CM,C)/(∃ req ∈ REQS)(con Bc req)} ;
   /* constrs(CM,C) is the set of constraints
      associated with C in CM */
   CLASSES_CON = CONSTSREQS(c)/Bc ;
   WHILE(CLASSES_CON ≠ ∅ ){
         c_constrs(T,c) = extract(CLASSES_CON);
         Extended_T = configure(T,CM,c);
         IF(Extended_T == FAILURE){
            RETURN FAILURE;}
         T = Extended_T;
       }//while
   RETURN T;
}//mainProc
```

Fig. 4. A configuration algorithm: mainProc procedure.

6. THE CONFIGURATION MECHANISMS

In this section we report the general structure of a configuration algorithm that adopts policy 5 (see Section 4.2) for computing partial configurations according to Definition 7. Policy 5 makes use of both requirements-based decomposition and constraints-splitting decomposition, therefore we use it to illustrate both these decomposition mechanisms. The modifications to the algorithm in order to implement the other decomposition policies are straightforward.

6.1. The configuration algorithm

The configuration mechanisms consist of the *mainProc* procedure (Fig. 4), the (recursive) *configure* procedure (Fig. 5), and the *insertDirectComponents* procedure (Fig. 6).

```
PROCEDURE configure(T,CM,c){
   dirComplComps(T,c) = ∅ ;
   PART_SLOTS = {p ∈ p_slots(CM,class(T,c)) /
    p occurs insome constraint of c_constrs(T,c)} ;
   Extended_T = insertDirectComponents(T,CM,c,PART_SLOTS);
   IF(Extended_T == FAILURE){//IF-1
     IF(there is no open choice){
       RETURN FAILURE;
     }ELSE{BACKTRACK;}
   }ELSE{/*Extended_T =/= FAILURE*/
      T = Extended_T;
      WHILE(dirComplComps(T,c)≠ ∅ ){
        c' = extract(dirComplComps(T,c));
        c_constrs(T,c') = I_constrs(T,CM,c') ∪
                          L_constrs(T,CM,c');
        Extended_T = configure(T,CM,c');
        IF(Extended_T == FAILURE){//IF-2
          IF(there is no open choice){
             RETURN FAILURE;
          }ELSE{BACKTRACK;}
        }ELSE{/*Extended_T =/= FAILURE*/
          T = Extended_T;
         }//IF-2
      }//WHILE
    }//IF-1
    RETURN T;
}//configure
```

Fig. 5. A configuration algorithm: configure procedure.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

```
PROCEDURE insertDirectComponents(T,CM,c,PART_SLOTS){
  WHILE(PART_SLOTS≠∅ ){
    p = extract(PART_SLOTS);
    IF(a cardinality for p cannot be chosen without
       violating any constraint in c_constrs(T,c)){
      RETURN FAILURE;}
    - choose a cardinality n for the slot p in such a way that
      no constraint in c_constrs(T,c) is violated;
    - create a set X={x1,...,xn} of new components;
    COMPS_T = COMPS_T ∪ X;
    INST_SLOTS_T = INST_SLOTS_T ∪ {⟨p(c) = X⟩};
    IF(the types for the components in X cannot be chosen without
       violating any constraint in c_constrs(T,c)){
        RETURN FAILURE;
    }
    - FOR EACH xi∈X choose a concrete class Ci in such a way that
        no constraint in c_constrs(T,c) is violated and set
        class(T,xi)= Ci;
    - remove any satisfied constraint from c_constrs(T,c) and
      from each c_constrs(T,d), such that c is a component
      (either direct or indirect) of d;
     - remove from PART_SLOTS those slots that do not occur in any
        constraint of c_constrs(T,c);
    IF(the components in X are complex components){
      insert them in dirComplComps(T,c);
    }ELSE{
      FOR EACH xi∈X and FOR EACH q ∈ d_slots(CM,class(T,xi)),
      set INST_SLOTS_T = INST_SLOTS_T ∪ {⟨q(xi) = V⟩} ,
      where V is the set of values specified for q in CM;
     }
  }//WHILE
  RETURN T
}//insertDirectComponents
```

Fig. 6. A configuration algorithm: insertDirectComponents procedure.

The search for a configuration progressively extends and modifies a *tentative* configuration. The tentative configuration that the configurator is working on is denoted by $T = \langle \text{COMPS}_T, \text{INST_SLOTS}_T \rangle$.

The mainProc procedure (Fig. 4) accepts as input a configuration problem $\text{CP} = \langle \text{CM}, C, \text{REQS} \rangle$ (Definition 1) and returns either a partial configuration T solving CP according to Definition 7 or the FAILURE message (in case the configuration problem does not admit any solution).

At the beginning, the tentative configuration T contains only a component c such that $\text{class}(T, c) = C$, representing the target object.

The set $\text{CONSTRS}_{\text{REQS}}(c)$ of the constraints that are relevant to the target object and bound to the requirements is then computed. According to Definition 6, this set contains all the requirements plus those constrains stated in the conceptual model for the class C of the target object that may interact with the requirements during the configuration process. Because the requirements-based decomposition is adopted, all the other constraints (if any) associated with class C do not interact with the requirements and are therefore not taken into consideration.

The constraints-splitting decomposition is adopted, too; therefore, $\text{CONSTRS}_{\text{REQS}}(c)$ is partitioned into a set of classes of bound constraints by means of the bound relation $\mathcal{B}_c$ (Definition 5). $\text{CONSTRS}_{\text{REQS}}(c)/\mathcal{B}_c$ is the quotient set (i.e., the set of the equivalence classes) induced in the set $\text{CONSTRS}_{\text{REQS}}(c)$ by the relation $\mathcal{B}_c$; thus, it contains all the above-mentioned classes of bound constraints. CLASSES_CON is initialized to this quotient set.

Every class of constraints in CLASSES_CON corresponds to a subproblem, and each subproblem can be solved independently of each other.

In the WHILE loop, the subproblems are solved sequentially. At each execution of the WHILE body, one class of constraints $c_\text{constrs}(T, c) \in \text{CLASSES_CON}$ is activated for the target object c and the configure procedure is invoked to extend T by taking into consideration only the current active constraints in $c_\text{constrs}(T, c)$ and the components and subcomponents of c possibly critical for the satisfaction of

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

these constraints. [The other (sub)components do not play any role in the subproblem relevant to the current class of constraints c_constrs(T, c).] If any one of these subproblems fails to admit a solution, the whole problem has no solution and a global FAILURE message is returned. Otherwise, the configuration is progressively extended until all the subproblems are solved. The solution is then returned.

If CLASSES_CON contains more than one class, it means that the main problem has actually been split into a set of independent subproblems (in this case, the main problem is said to be constraints-splitting decomposable). In this version of the algorithm, the subproblems are sequentially solved. However, a parallel version of the algorithm could be implemented that sends these subproblems to a pool of configurators working in parallel. The experiments discussed in Section 7 were performed with the sequential version of the algorithm. Nevertheless, the constraints-splitting decomposition strategy proved to also be effective within such an algorithm.

Each call to configure$(T$,CM,$c)$ (Fig. 5) corresponds to a request to extend the configuration of the complex component $c \in \text{COMPS}_T$ by taking into consideration the tentative configuration T, the CM, and the the current active constraints in c_constrs(T, c). In response, configure$(T$, CM,$c)$ returns either a tentative configuration in which T has been successfully extended or the FAILURE message, indicating that no configuration exists for component c that satisfies the constraints in c_constrs(T, c).

This procedure first extends T by inserting into the tentative configuration a set of new *direct* components of c. Each time a direct complex component of c is inserted into the configuration, it is added to the set dirComplComps(T, c). In a second phase, configure is recursively invoked one time for each component in this set.

At the beginning, no direct component of c exists yet, thus dirComplComps(T, c) is empty. No other components of c but those possibly critical for the satisfaction of the set c_constrs(T, c) of current active constraints have to be considered (in fact, the other components of c are not involved in the subproblem that is being solved). Therefore, the set PART_SLOTS of the partonomic slots that are going to be considered for (partially) configuring the component c contains only those partonomic slots of component c that are mentioned in the active constraints c_constrs(T, c).

The direct components of c are inserted into the configuration by insertDirectComponents procedure (Fig. 6).

Whenever insertDirectComponents procedure fails, a backtracking occurs that revises a previous choice. If no previous choice can be revised, configure fails.[7] Each time a complex direct component of c is inserted into the tentative configuration, insertDirectComponents procedure put it into dirComplComps(T, c).

[7] The backtracking mechanism is not described here, because it is out of the scope of this paper.

The WHILE loop in configure procedure extracts one of these components c' at a time; the set of active constraints c_constrs(T, c') for c' is computed and the process is repeated to configure that portion of c' possibly critical for the satisfaction of this new set of active constraints.

This set is computed as follows:

$$c\text{_constrs}(T, c') = I\text{_Constrs}(T,\text{CM}, c') \cup L\text{_Constrs}(T,\text{CM}, c'),$$

where I_Constrs$(T,$CM,$c')$ is the set of constraints that c' "inherits" from the component c, which it is direct part of, namely,

$$I\text{_Constrs}(T,\text{CM}, c') = \{\text{con} \in c\text{_constrs}(T, c)/ \\ (\exists p \in p\text{_slots}(\text{CM},\text{class}(T, c'))) \quad (p \text{ occurs in con})\}$$

(i.e., it is the set of constraints in c_constrs(T, c) containing some partonomic slot relevant to the component c'). L_Constrs$(T,$CM,$c')$ is the set of constraints associated with class(T, c') that are bound in c' to some constraint belonging to I_Constrs$(T,$CM$)$, namely,

$$L\text{_Constrs}(T,\text{CM}, c') = \{\text{con} \in \text{constrs}(\text{CM}, \text{class}(T, c'))/ \\ (\exists\ \text{con}_{\text{in}} \in I\text{_Constrs}(T,\text{CM}, c')) \quad (\text{con}_{\text{in}}\ \mathcal{B}_{c'}\ \text{con})\}.$$

The other constraints possibly associated with *class* (c_comp(T)) are not involved in the subproblem that is being solved; thus, they are not considered.

When all the direct complex components of c contained in dirComplComps are successfully (partially) configured, configure returns the extended configuration. If any one of such complex components does not admit any configuration that satisfy the set of active constraints, the FAILURE message is returned.

In regard to the insertDirectComponents procedure (Fig. 6), it accepts as input a tentative configuration T, the CM, a complex component $c \in \text{COMPS}_T$, and a set PART_SLOTS of partonomic slots relevant to class(T, c). It returns either a tentative configuration in which the configuration of the component c has been extended by considering only partonomic slots in PART_SLOTS or the FAILURE message.

This procedure performs the two basic kinds of choices of the configuration process:

1. given a partonomic slot $p \in$ PART_SLOTS, it chooses its cardinality, that is, it chooses the number n of direct components $x_1, \ldots, x_n$ of the component c, through the whole–part relation represented by p;
2. it chooses the type of each such component x_i (i.e., the value of class(T, x_i) $(i = 1, \ldots, n)$).

In performing these choices, it must respect both the partotaxonomic description (i.e., the number and the value re-

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

strictions of p) and the active constraints in $c_constrs(T,c)$. If either the cardinality of p or the type for any x_i cannot be chosen without violating any constraint in $c_constrs(T,c)$, a FAILURE message is returned to configure procedure, which manages the backtracking. Otherwise, the tentative configuration T is extended by both adding the set $X = \{x_1,\ldots,x_n\}$ of new components to COMPS_T and adding the equality $\langle p(c) = X\rangle$ to $\mathrm{INST_SLOTS}_T$. If $x_1,\ldots,x_n$ are complex components, they are inserted into $\mathrm{dirCompl\text{-}Comps}(T,c)$. After the two above-mentioned choices have been successfully made for p, all the satisfied constraints are removed from the tentative configuration T. (In fact, any subsequent extension of T cannot change their truth value.) Whenever a constraint is removed from $c_constrs(T,c)$, it can be that some slots in PART_SLOTS will no longer occur in any active constraint for the component c. These slots need to be removed from PART_SLOTS, because they are no longer critical for the satisfaction of the constraints in $c_constrs(T,c)$.

It is worth noting that each component in $X = \{x_1,\ldots,x_n\}$ is a *new* one, that is, it is different from the other components in COMPS_T: this assures that the exclusiveness assumption on parts (Assumption 1) actually holds. Moreover, whenever an atomic component is inserted into a configuration, the values for its properties are also stated in the configuration.

6.2. A configuration example

In this section we briefly describe the behavior of the configuration algorithm by means of an example in which the configurator builds a partial configuration (according to Definition 7) by adopting decomposition policy 5 (Table 1). Because the backtracking mechanism is out of the scope of this paper, we discuss a simple example in which the configurator is able to go straight to the solution without entering any failure path in the search space.

Let us suppose that we want to configure an advanced application PC (described in the PC conceptual model PC_CM shown in Figs. 1 and 2) meeting the requirements REQS listed in Figure 3. The procedure mainProc(PC_CM,Advanced Appl. PC,REQS) is invoked. The process starts with a tentative configuration T_1 containing only the component aapc1 representing the target object (thus class(T,aapc1) = Advanced Appl. PC). The set of constraints relevant to aapc1 is CONSTRS(aapc1) = {co1,co2,co3,co4,co5,co6,co7,req1,req2,req3} (see Definition 4). All these constraints are bound to the requirements; thus, $\mathrm{CONSTRS}_{\mathrm{REQS}}$(aapc1) = {co1,co2,co3, co4,co5,co6,co7,req1,req2,req3} (see Definition 6). Because policy 5 is adopted, only the components and the subcomponents of aapc1 possibly critical for the satisfaction of these constraints will be considered during the configuration process (requirements-based decomposition). Moreover, constraints-splitting decomposition takes place and thus according to the bound relation (Definition 5), $\mathrm{CONSTRS}_{\mathrm{REQS}}$(aapc1) is partitioned into the set of equivalence classes

$$\begin{aligned}\mathrm{CLASSES_CON} &= \mathrm{CONSTRS}_{\mathrm{REQS}}(\mathrm{aapc1})/\mathcal{B}_{\mathrm{aapc1}}\\ &= \{\{\mathrm{req1,co1,co2,co3,co6,co7}\},\\ &\quad \{\mathrm{req2,co4,co5}\},\{\mathrm{req3}\}\}.\end{aligned}$$

It is easy to see, for instance, that req1 is directly bound in aapc1 to co1, co2, co3, and co7, because all these constraints contain the partonomic slot has_m of class(T,aapc1) = Advanced Appl. PC; moreover, it is bound in aapc1 by transitivity to co6, in fact req1 is directly bound in aapc1 to co1, which is directly bound in aapc1 to co6 (because they both contain the partonomic slot has_hd of Advanced Appl. PC class). Moreover, req3 contains only the partonomic slot has_cdt, which does not occur in any other constraint; therefore, req3 is not bound to any constraint (except itself).

CLASSES_CON entails a set of independent subproblems (one subproblem for each equivalence class of constraints). The configurator solves them sequentially. Let us suppose that it starts with the subproblem corresponding to the set {req1,co1,co2,co3,co6,co7}.[8] Figure 7 contains five snapshots of the configuration process relevant to this first subproblem; the tentative configuration is represented as a tree (see Section 3) in which the thick arrow indicates the current component (i.e., the one that is being configured).

At the beginning the set of current active constraints for the target object is $c_constrs(T_1,\mathrm{aapc1})$ = {req1, co1,co2,co3,co6,co7}. The set of partonomic slots that have to be considered are those ones in Advanced Appl. PC class that occur in some active constraints, namely, PART_SLOTS = {has_hd,has_m,has_cdr,has$_c$dw}. [See the tentative configuration T_1 in Fig. 7(a)].

After the direct components of aapc1 relevant to these slots have been inserted into the configuration, the situation is as depicted in Figure 7(b): no more direct parts of aapc1 need to be inserted for solving the current subproblem and the constraint co6 is satisfied in T_2, thus it is removed from the set of active constraints.

The motherboard mb1 is a direct complex component of aapc1 (i.e., $\mathrm{dirComplComps}(T_2,\mathrm{aapc1}) = \{\mathrm{mb1}\}$), therefore it must be (partially) configured. mb1 inherits from aapc1 the set $I_constrs(T_3,\mathrm{PC_CM,mb1}) = \{\mathrm{req1,co1,co2,co3,co7}\}$ of constraints (because all these constraints contain partonomic slots relevant to mb1). These constraints activate the set of local constraints $L_constrs(T_3,\mathrm{PC_CM,mb1}) = \{\mathrm{co8}\}$ (in fact, req1 $\mathcal{B}_{\mathrm{mb1}}$ co8), thus $c_constrs(T_3,\mathrm{mb1}) = \{\mathrm{req1,co1,co2,co3,co7,co8}\}$. The set of partonomic slots of Motherboard class occurring in these constraints is {has_mpcb, has_cs,has_cpu} [see Fig. 7(c)].

[8] Solving this subproblem actually means finding a partial configuration for CP_1 = ⟨PC_CM,Advanced Appl. PC,{req1}⟩, according to Definition 7.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

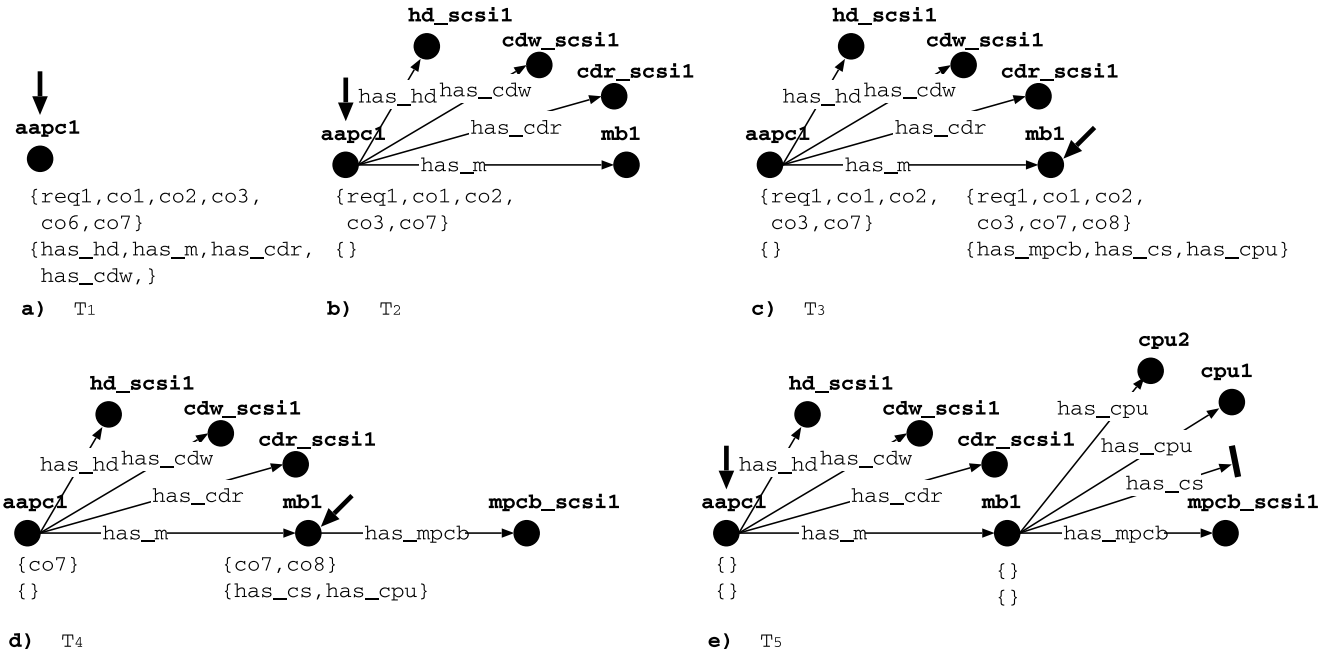


Fig. 7. A configuration example: the first subproblem.

It is worth noting that the initial set c_constrs(T_3,mb1) of active constraints corresponds to the set CONSTRS$_{\text{REQS}}$(mb1) (see Definition 6). This is true in general when requirements-based decomposition is adopted. In fact, in such a case, for the target object only the constraints bound to the requirements are activated, therefore, the set I_constrs(T,CM,c) for each direct component c of the target object contains only constraints bound to the requirements; each constraint in L_constrs(T,CM,c) is bound to a constraint in I_constrs(T,CM,c) by definition, thus it is bound to the requirements, too. This reasoning can be recursively repeated for each subcomponent.

After the main printed circuit board of type SCSI mpcb _scsi1 is inserted [Fig. 7(d)], the constraints req1, co1, co2, and co3 are satisfied in T_4; thus, they are deleted from the tentative configuration. The configuration process then considers has_cs and has_cpu slots: two CPUs (cpu1 and cpu2) and no SCSI controller are inserted into the configuration [Fig. 7(e)]. Configuration T_5 contains no active constraints (i.e., both the set {req1,co1,co2,co3,co6,co7} of the constraints initially active and constraint co8, activated by the first ones in the mb1 component, are satisfied in T_5); therefore, T_5 is a partial configuration solving the first subproblem.

The second subproblem corresponds to the set of constraints {req2,co4,co5}. These become the initial active constraints for the target object, as shown in Figure 8(a). The configurator solves the second subproblem by taking into consideration only the slots has_asw and has_os. Configuration T_7 is a partial configuration solving both the first and the second subproblems. Because there is no interaction between these two subproblems, the portion of the tentative configuration relevant to the first one (the one with dashed arcs and grey labels and nodes in T_6 and T_7) is not involved in any way in the resolution of the second one.

The third subproblem, corresponding to constraint req3 (Fig. 9), is solved by introducing a CD tower (the component cdt1 in T_9) into the configuration, where T_9 is a partial configuration solving the main problem according to Defi-

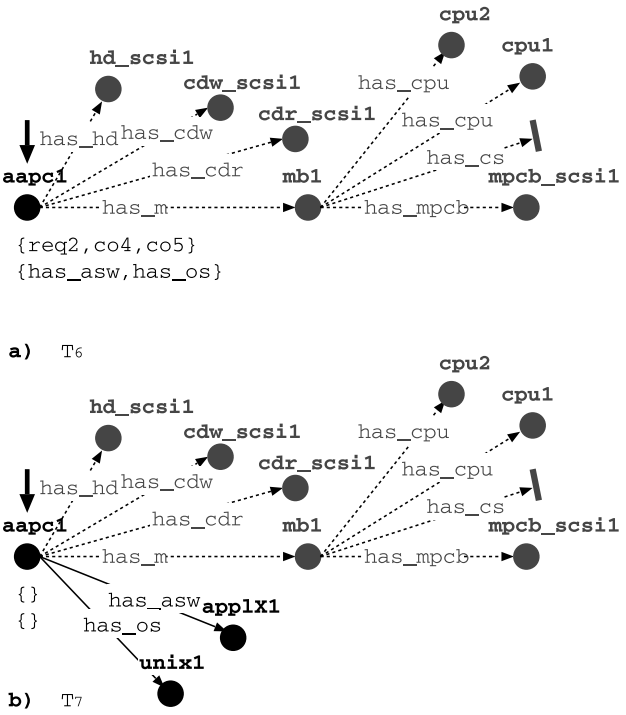


Fig. 8. A configuration example: the second subproblem.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

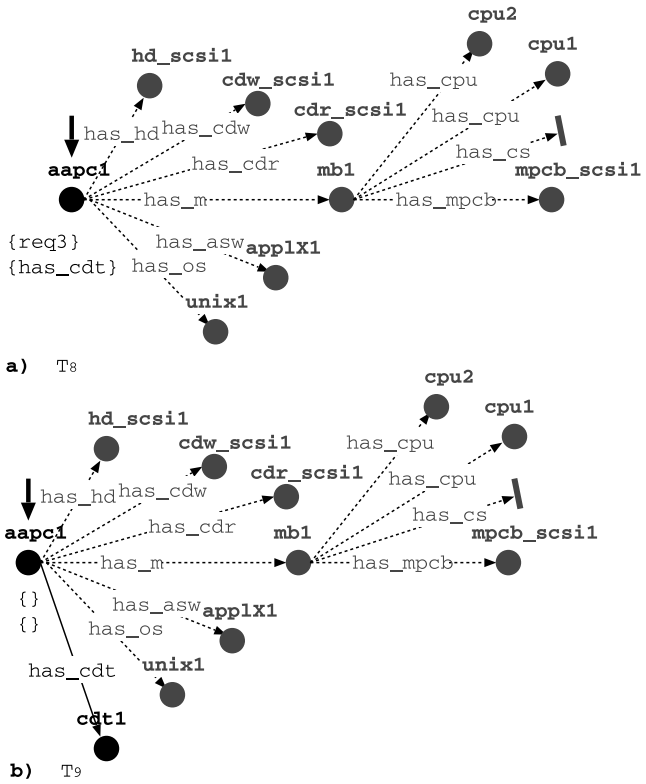


Fig. 9. A configuration example: the third problem.

nition 7. Because decomposition policy 5 is adopted, the configuration process stops at this point. It is worth noting that some components and subcomponents of an advanced application PC have not been considered during the configuration process, because they are not critical for the satisfaction of the requirements (i.e., keyboard, case, video card, modem, etc.).

Figure 10 shows an example where the user incrementally specifies her/his requirements. In fact, after stating the requirements listed in Figure 3, she/he might put some additional requirements on the components and subcomponents relevant to partonomic slots not specified in the partial configuration T_9. In particular, requirements REQS′ in Figure 10 require that the CD tower cdt1 contains two CD readers and two CD writers. In order to satisfy the user's request, the configure procedure has to be invoked to configure component cdt1 given the set of constraints relevant to that component. The invocation of configure has the effect of extending T_9 by means of a solution to the configuration problem $CP_{cdt_1} = \langle$PC_CM,CD Tower,REQS′$\rangle$.

The partial configuration T_{10} reported in Figure 10 is a solution because it meets all the requirements REQS ∪ REQS′.

7. EXPERIMENTAL RESULTS

In this section we briefly discuss a first set of experiments showing the advantages (in terms of speed-up) of using decomposition strategies for configuration. The domain chosen for this experiments is the configuration of computer systems. The conceptual model used in the experiments is a larger version of the one reported in Figures 1 and 2, so a larger variety of entities is considered, the partonomies are deeper, the taxonomy includes additional subclasses, and the constraints capture more restrictions on the combinations of components and subcomponents. Table 2 summarizes the main characteristics of the domain (complex/

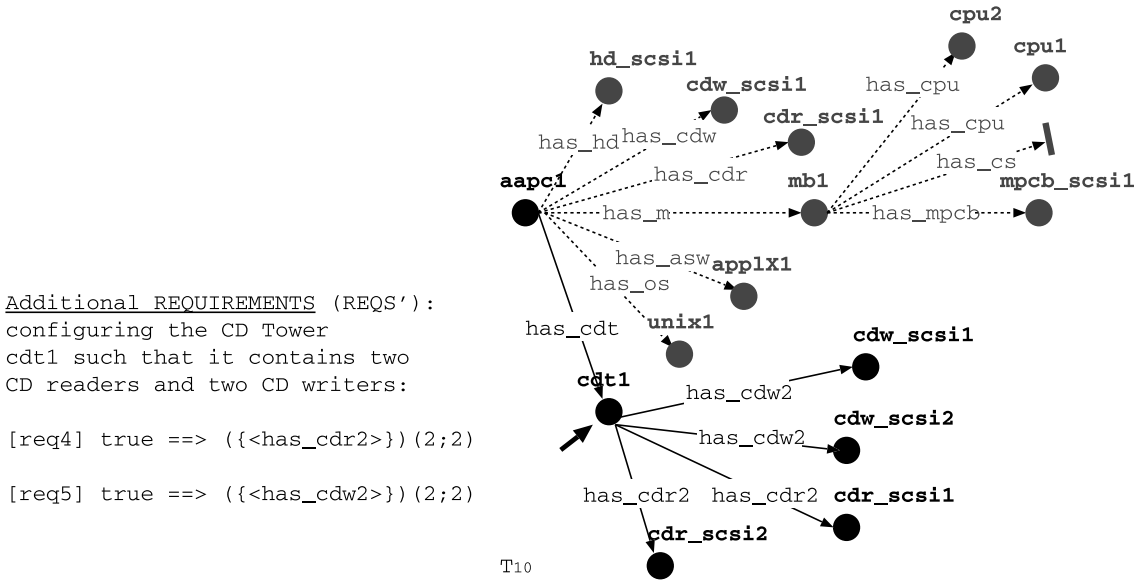


Fig. 10. A configuration example: configuring the CD tower component.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

Table 2. *Computer systems domain: Main characteristics*

Max. no. levels in partonomies	4
No. classes total/concrete	
Complex	21/17
Atomic	72/56
Ave. no. slots in each class	
Complex (partonomic slots)	8.3
Atomic (descriptive slots)	4
$\mathcal{FPC}$ constraints	
Total no.	56
Ave. no. in each concrete complex class	12.3
Ave. no. different slots in each constraint	2.3
Ave. no. predicates in each constraint	1.9
Ave. no. different slot chains in each constraint	1.9

concrete classes means classes of complex/concrete objects).

In order to test the performance of the different policies, we designed a test set of 153 randomly generated configuration problems. The mean number of constraints expressing user's requirements in each problem is 4.4. In 69 cases (45.1%), the requirements are inconsistent w.r.t. the conceptual model. These problems do not admit any solution. In addition, 51 problems (i.e., one-third) are constraints-splitting decomposable.

The configuration system was implemented in Java (i.e., the configuration algorithm was implemented as a set of Java procedures working on conceptual models and user requirements in which classes, slots, and constraints are all Java objects) using JDK1.3. The experiments were performed on a Duron 700/Windows 2000 with 128 MB of RAM.

Because the configuration of a complex entity may be quite expensive from a computational point of view, we have imposed a threshold of 360,000 ms of CPU time (i.e., 6 min) for solving a single problem. This means that a configuration problem is considered solved if and only if the configurator either finds a solution for it or it detects the inconsistency of the requirements w.r.t. the conceptual model, within this timeout.

We first tried the simplest policy 1 in Table 1. Because this policy does not make any attempt to subdivide the configuration problem, as expected, the results were not good because it was able to solve only 98 out of 153 configuration problems (competence = 64.1%). It is worth noting that in the 98 solved problems, the average computation time was 15,015.7 ms (standard deviation = 40,171.2 ms) and the average number of backtrackings was 3494.1 (standard deviation = 8818.6). Among the 69 inconsistent problems, only 17 were solved (i.e., 24.6%). For these problems, the average CPU time was 31,913 ms (standard deviation = 64,598.7 ms).

Much better results can be obtained when policy 2 is adopted. This policy tries to solve the problem by first configuring the portion directly related to input requirements and then completing the configuration with the other (sub)-components not involved in the input requirements. This policy is quite effective because it was able to solve (within the time limit) 140 out of 153 cases, so that its competence was 91.5%. (Thus, the competence increased with respect to the 64.1% obtained under policy 1).

Table 3. *Policy 2 versus policy 3 (global)*

	Policy 2	Policy 3
No. backtrackings		
Average	1400.6	1306.2
Standard deviation	3803.7	3600.7
CPU time (ms)		
Average	13007.4	12261.0
Standard deviation	39621.3	38636

As reported in Table 3, the average computation time for solving the 140 problems is 13,007.4 ms. This is a good result because the configuration system is able to solve most of the difficult problems that policy 1 was unable to solve without penalizing the average cost. It is interesting that policy 2 solved 65 of the 69 inconsistent problems (i.e., 94.2%), confirming the intuition that the inconsistency can be detected earlier in the configuration process if the components possibly involved in the satisfaction of the requirements are considered first. The time spent in solving these problems was an average of 24,626.3 ms (standard deviation = 55,268.3 ms).

Table 3 compares the performance of policy 2 with respect to policy 3. This latter extends policy 2 by considering the possibility of splitting constraints into separate classes and configuring the product by taking advantage of this subdivision. The competence of policy 3 was the same as that of policy 2, that is, policy 3 solved exactly the same set of problems that policy 2 did. However, policy 3 was (slightly) better in both the average number of backtrackings (−6.7%) and the average computation time (−5.7%). In comparing the results reported in Table 3 it is worth noting that only some of the 153 problems in the test set admit constraints splitting. In fact, for only 45 of 140 configuration problems that were solved by policies 2 and 3 was constraint splitting possible (i.e., 32.14%). This means that the overhead for attempting the splitting was small and there are some benefits in performing the splitting, even when we compare the entire test set. The benefits become much more apparent when we focus our attention only on the subset of cases where constraint splitting was possible.

Table 4 reports the comparison between policies 2 and 3 on the 45 problems where policy 3 could actually exploit the splitting of the constraints in guiding the configuration process. We see a significant reduction in terms of backtrackings (−28.1%) and average computation time (−28.3%). By inspecting Tables 3 and 4 we can see a strong correla-

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

Table 4. *Policy 2 versus policy 3 (constraints-splitting decomposable problems)*

	Policy 2	Policy 3
No. backtrackings		
Average	1046.3	752.6
Standard deviation	3404	2562.6
CPU time (ms)		
Average	6491.9	4653.2
Standard deviation	17825.5	13073.2

tion between the reduction of the number of backtrackings and the saving in computational effort.

It is also worth noting that the decomposable problems seem to be easier to solve than typical configuration problems occurring in the test set. In fact, if we compare the average CPU time of the 140 problems with the average CPU times of the 45 decomposable problems we see a reduction from 13,007.4 ms to 6,491.9 ms (with policy 2).

Regarding policies 4 and 5, we have to take into consideration that policy 4 does not try to produce a complete configuration, but it stops as soon as the portion related to input requirements has been configured. Policy 5 is like 4 but it has been extended by exploiting constraints splitting (when it is possible). Both policies solved the same number of problems, 142 (i.e., 92.8% of the whole test set).

If we compare these two policies, we get results similar to those obtained for policies 2 and 3: on the 142 problems that they solved, there is some reduction, on average, both in the number of backtrackings (−6.7% for policy 5 w.r.t. policy 4) and in CPU time (−3.2% for policy 5 w.r.t. policy 4). Again, the advantages are much more significant if we compare the behavior of policies 4 and 5 on the subset of 47 problems (33.1% of the test set) where constraint splitting was possible: on average, −28.1% of backtrackings and −26.8% of CPU time for policy 5 w.r.t. policy 4.

8. COMPARISON WITH PREVIOUS WORK

8.1. Configuration ontologies

The relevant concepts for the various configuration tasks have been described in several works (Mittal and Frayman, 1989; Mailharro, 1998; Soininen et al., 1998; Felfernig et al., 2000; Soininen, 2000). We have defined the decomposition mechanisms in the $\mathcal{FPC}$ framework in which many of these concepts can be expressed. Besides those concepts explicitly mentioned in Section 2, it is worth discussing four other concepts occurring in configuration ontologies.

8.1.1. Resources

In the $\mathcal{FPC}$ framework, both resource satisfaction and resource balancing can be expressed and actually have been exploited to model the computer system configuration domain that we used in experiments. For instance, the RAM memory slots produce a certain amount of memory capacity, and each software application needs a certain amount of memory. These properties are represented with the descriptive slots RAM_mem and Sw_mem associated with the RAM and Software_Application classes, respectively, and taking values in a set of positive numeric values. The constraint true $\rightarrow (\max(\{\langle \text{has_asw,Sw_mem}\rangle\})) \leq (\Sigma(\{\langle \text{has_m,has_ram,RAM_mem}\rangle\}))$, associated with the PC class, states that the memory capacity resource must be satisfied.[9] For the sake of simplicity and in order to keep the example in Section 6.2 small enough, this constraint and the descriptive slots occurring in it are not represented in Figures 1 and 2.

In the $\mathcal{FPC}$ framework every constraint is always associated with a class of complex objects; moreover, in each constraint, the slot chains explicitly refer to a set of components of individuals of the class with which the constraint is associated. This allows the representation of *contexts* (i.e., sets of components) in which resources must be satisfied or balanced.

8.1.2. Shared components

Another issue that it is worth discussing is that of shared components. As stated in Section 3, in the present work we assume that partonomic relations are exclusive (Assumption 1). This assumption is quite natural in several configuration domains. However, in some cases shared components must be represented.

We use an example to briefly discuss some issues relevant to what can be retained and what should be modified in the framework in order to handle shared components.

Let us consider, for example, the conceptual model depicted in Figure 11, where a hydraulic system is composed of two subsystems having two tanks as their part. We suppose that the tanks can be shared by the two subsystems. If we require the first subsystem to have only TypeA tanks

$$([\text{req1}]\ \text{true} \rightarrow (\{\langle \text{has_sub1,has_tank1}\rangle\})(\text{in TypeA}))$$

and the second at least one one TypeB tank,

$$([\text{req2}]\ \text{true} \rightarrow (\{\langle \text{has_sub2,has_tank2}\rangle\})(\text{in TypeB}(1;2))),$$

we may wonder if the decomposition strategies can be applied to this situation.

The conceptual model states only that the tanks *can* be shared by the two subsystems, but not that they *must* be shared. If a valid configuration exists in which (at least) one of the two tanks is shared, then a valid configuration in which each subsystem has its own tanks also exists and it can be constructed by separately configuring the two sub-

[9] For brevity, here we do not consider the memory consumption relevant to the operating system.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

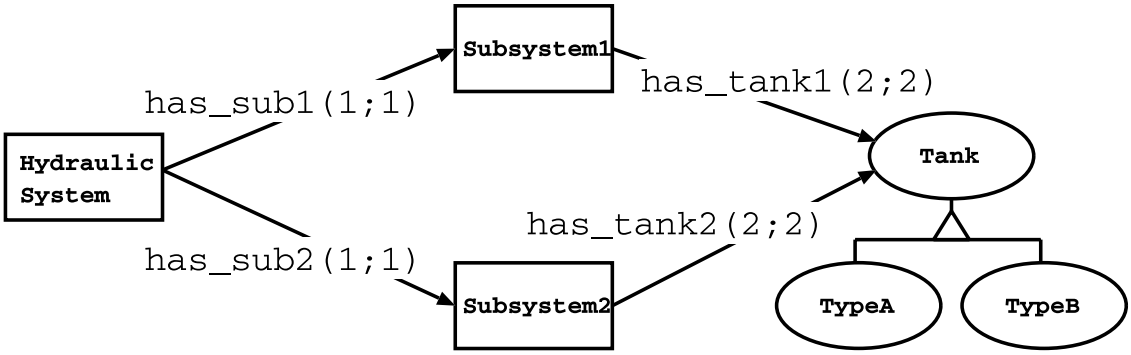


Fig. 11. A portion of a hydraulic system conceptual model.

systems; conversely, if both the subsystems can be separately configured, a valid configuration of the whole system does exist. Therefore, the subproblems of configuring the two subsystems are independent to a certain extent, because the main problem admits a solution if and only if the two subproblems are solvable separately.

According to the bound relation (Definition 5), req1 and req2 (associated with the HydraulicSystem class) are not bound; therefore, this relation is still able to recognize such an independence.

However, if some components can be shared, it is often convenient to produce configurations in which they are actually shared. For example, a hydraulic system with three tanks (two TypeA and one TypeB tanks, in which one of the TypeA tanks is shared by the two subsytems) could be preferred to a system in which four tanks are present. To produce such configurations, the procedure in Figure 6 has to be slightly modified. This procedure always inserts a set of new components each time a cardinality greater than zero is chosen for a partonomic slot. To handle shared components, this procedure should be given the possibility of using components already present in the tentative configuration it is working on each time a shared partonomic relation is considered.

To express the situations in which the components *must* be shared, $\mathcal{FPC}$ has to be extended with a set of new predicates. For instance, we could model the fact that the tanks must be shared by the two subsystems by means of the constraint [shared]true $\rightarrow$ shared($\langle$has_sub1,has_tank1$\rangle$,$\langle$has_sub2,has_tank2$\rangle$). In this case, the two requirements req1 and req2 cannot be considered separately. The bound relation still works, because it places req1, req2, and shared in a same class of bound constraints.

8.1.3. Functions

Mittal and Frayman (1989) present a model of configuration tasks in which functional architectures play a basic role. The configuration algorithms that we presented here does not rely on functional architectures. However, $\mathcal{FPC}$ provides the means of describing a set of functions in order to express some typical classes of requirements. The $\mathcal{FPC}$ models contain, for each function, a mapping into a set of $\mathcal{FPC}$ constraints. These mappings are used to automatically produce the requirements in a $\mathcal{FPC}$ constraint-based form. The user can further specify her requirements by directly adding a set of constraints to those derived by the functions that she selected (see Magro & Torasso, 2001*b*).

8.1.4. Connections

In some configuration tasks also the connections (either physical or logical) between components represent an important piece of knowledge. These connections are usually represented by means of ports. In our framework this knowledge cannot be explicitly expressed. To model the computer system domain we used in experiments, a set of $\mathcal{FPC}$ constraints has be derived from this knowledge. For instance, if, in each complex object of class *C*, the components of, say, type TypeA need to be connected to components of type TypeB, a set of $\mathcal{FPC}$ constraints is added to the conceptual description of *C*. These constraints state that in any individual of *C*, the number of TypeB components must be greater than or equal to the number of TypeA components.

To model and reason in those domains in which more complex constraints relevant to the connections have to be represented, $\mathcal{FPC}$ framework should be extended.

8.2. The open world semantics and the justification of configuration elements issues

As pointed out by some researchers, the classical open world semantics of description logics and of classical propositional or predicate logic is not suitable for configuration task (see, for example, Weida, 1996; Soininen, 2000).

For instance, given a consistent DL knowledge base $\Sigma = \langle$TBox,ABox$\rangle$, under open world semantics, a particular individual description *d* may be consistent with Σ (i.e. $\Sigma' = \langle$TBox,ABox $\cup \{d\}\rangle$ may be satisfiable), even if *d* contains some concept names that do not occur in Σ. However, in a configuration task, where Σ is a conceptual model and individuals are intended to represent configuration elements, such individual descriptions should be forbidden. In fact, it is assumed that all basic concepts needed to express con-

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

figurations are explicitly mentioned in the conceptual model. Moreover, open world semantics would allow fully specified individuals to restrict more roles than any primitive or defined concept of Σ that they instantiate. This means that no basic concept in Σ is able to account for all the properties of such individuals. Again, in a configuration task, these situations should not occur. In fact, it is assumed that the conceptual model does explicitly contain the description of all the types of elements that can occur in configurations. In order to cope with these issues, Weida (1996) formalized the concept of the closed terminology assumption (CTA).

Similar problems also occur when formalizing a configuration conceptual model as a logic theory. In fact, let T be a first-order theory describing a configuration domain, in which all the objects that can appear in a configuration are explicitly represented (by means of constants). If we think of a configuration as a model for T,[10] under the classic semantics of first-order languages, a configuration might contain objects not represented in T. To rule out these models, in Soininen (2000) only Herbrand models were allowed. However, as pointed out by the author, this is not enough. In fact, Herbrand models could contain unintended relations among elements. This is not appropriate for a configuration task, because any relation occurring in a configuration must be "justified," that is, there must be some "reason" in theory T for the relation to hold. Soininen's proposal is to further restrict the set of allowed models to *stable Herbrand models* (see Soininen, 2000, for details).

Our framework is close to description logic based approaches to configuration and basically follows the CTA stated in Weida (1996). In fact, the application does not add any new concept to the conceptual model during problem solving. Moreover, we distinguish between abstract classes (nonleaf classes in $\mathcal{FPC}$ taxonomies) and concrete classes (the leaves in $\mathcal{FPC}$ taxonomies). In each tentative configuration (thus, in both complete and partial configurations), each component (either complex or atomic) instantiates one (and only one) concrete class (Definition 2, item 1). Besides this, every component occurring in any complete configuration is "finished" in the sense specified in Weida (1996). In fact, if a configuration contains the equality $\langle p(c) = X \rangle$, p is a slot (possibly inherited) of the (concrete) class $\mathrm{class}(T, c)$ instantiated by the component c (Definition 2, item 3) and such an equality expresses a property respecting the restrictions for slot p stated in the conceptual description of $\mathrm{class}(c)$ (i.e., the equality is justified by the conceptual description of $\mathrm{class}(c)$, Definition 2, item 3). Moreover, each complete configuration specifies every slot $p \in \mathrm{class}(c)$, for each component c occurring in the configuration (Definition 3, item 1) and all the constraints (possibly inherited) associated with $\mathrm{class}(c)$ in the conceptual model are satisfied by c (Definition 3, item 2).

[10]For the sake of simplicity, we can assume here that T also contains the requirements.

The case of partial configurations (Definition 7) is slightly different, because for them it is not requested that every component c specify all the slots relevant to the class that it instantiates (which is still a concrete class). However, each partial configuration can be extended to a set of complete configurations (Property 2). Thus, each partial configuration represents a common subconfiguration of a set of finished configurations.

Differently from Weida's approach, the framework discussed in this paper does not exploit CTA to progressively specify the type of components. In fact, configuration problem definition (Definition 1) assumes that the concrete class instantiated by the target object is known *a priori*. Furthermore, as soon as a component is inserted into a tentative configuration during problem solving, the application chooses the concrete class that such component instantiates. To make such choices the application makes use of a set of focusing mechanisms able to rule out a set of impossible classes (Magro & Torasso, 2001*a*). These choices may be retracted (by means of a backtracking mechanism) as soon as the configurator reaches a failure state in the search space.

Our framework does not require that the components that can be part of a configuration be explicitly listed in the conceptual model, because the application creates them dynamically. Thus, the approach does not commit to Herbrand models. However, each element occurring in a configuration is justified. Indeed, in each configuration there is exaclty one component representing the target object (Definition 2, item 2) and, as said above, each relation occurring in a complete configuration is justified by the description of the basic concepts in the conceptual model. A partonomic relation needs an additional justification for being stated in a partial configuration, because only the properties possibly critical for satisfying the requirements can be specified in such configurations (Definition 7). (It is worth noting that the requirements are expressed in the vocabulary of the conceptual model: see Definition 1, item 3.)

As in Soininen (2000) but different than Mittal and Falkenhainer (1990), no subset minimality condition is imposed on configurations.

9. DISCUSSION AND FUTURE WORK

Problem decomposition has been recognized as an important issue in configuration. Fleischanderl and Haselböck (1996), for instance, stress the need for designing configurable large-scale systems (in which connections among components by means of ports play an important role) in a modular manner. If the interaction among modules (called packages) is kept low, the configurator can benefit from the package structure by focusing on one package at a time. In that approach, the responsibility of defining packages is left to the product designer. Instead, in the present paper we have argued that in some configuration domains the com-

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

positional structure of the product can be exploited to automatically decompose configuration problems.

The experimental results showed that major improvements in the performance of the configuration system can be obtained by singling out the components and subcomponents that are directly related to the constraints established by the user requirements and by first configuring that portion of the product.

We also showed that further improvement can be obtained by analyzing the set of constraints for the target object to be configured and decomposing them into independent sets. Splitting the search space into a set of smaller search spaces allows us to strongly reduce the number of useless backtrackings (as made apparent in the section devoted to experimental results) and therefore to improve performance.

The results reported in the paper were obtained in a set of experiments made in a real-world domain relevant to computer systems configuration. We obtained similar results by applying these techniques to a nontechnical domain relevant to configuration of students' curricula according to Italian law.

Actually, the improvements reported in the paper are an underestimate of what could be obtained if a parallel execution of the configuration subproblems were performed instead of a sequential one. In fact, it is possible to replace the loop in procedure mainProc (Fig. 4), which implements a sequential resolution of the various subproblems, with a parallel invocation of different configurators.

In the sequential approach, the total amount of time needed for solving the whole configuration problem is the sum of the times needed for solving the different subproblems, whereas in the parallel approach the total amount of time is reduced (in the best case) to the time needed for solving the most difficult subproblem plus the time spent in decomposing the problem and assembling the solutions to subproblems. However, the way we have decomposed the configuration problem assures that the global configuration can be obtained by simply summing up in the target object the results of the configurations relevant to the subproblems. Moreover, the basic criterion for the decomposition strategies is the *bound* relation (Definition 5): it is worth pointing out that this relation, in practice, is inexpensive to be computed, so that there is almost no overhead when we try to split the constraints, even in those problems that are not decomposable by constraints splitting.

The constraints-splitting mechanism that we have defined and used for partitioning the constraints associated with the target object can be extended in order to perform constraints-splitting decomposition in the components and subcomponents as well. Such a form of recursive constraints-splitting decomposition is described in Magro et al. (2002), and some preliminary results are reported that show its effectiveness.

A discussion of all the relevant aspects concerning interactive configuration was beyond the scope of this paper. Nevertheless, we argued that decomposition (in particular, requirements-based decomposition) is an important capability in such a task.

As discussed in Section 8, many concepts occurring in the configuration ontologies can be represented in $\mathcal{FPC}$; thus, the decomposition techniques that we have defined in this framework can be applied to several configuration domains. We have seen that the extension of the $\mathcal{FPC}$ framework in order to represent and use shared components is straightforward. Instead, some work has to be done in order to extend the $\mathcal{FPC}$ concepts, the configuration, and the decomposition mechanisms to connections between components and to complex constraints on these connections.

The differences between a structured logical approach to configuration and a CSP-based one make sometimes difficult to understand how one side can benefit from the results obtained by the other. Nevertheless, we have started investigating the possibility of adapting some decomposition techniques defined for large SAT problems (Park & Gelder, 2000) in order to further decompose a configuration problem into a set of loosely interacting subproblems. For instance, consider a configuration problem P_1 that can be decomposed into two subproblems P_{11} and P_{12} by means of constraints-splitting decomposition. If we add to the constraints relevant to P_1 a new constraint v that is bound with a constraint relevant to P_{11} and with a constraint relevant to P_{12}, we get a new configuration problem Q that is no longer decomposable by constraints splitting. However, Q is "nearly decomposable." Indeed, as soon as the constraint v is satisfied in a tentative configuration (if possible), the problem Q becomes constraints-splitting decomposable (in the context of such a tentative configuration). This suggests that the control strategy of the configurator should be guided by the constraint v and the configurator should try to first satisfy this constraint. Magro et al. (2002) contains a preliminary discussion about this issue.

ACKNOWLEDGMENTS

This research was partially supported by grants from the University of Torino and the Italian Space Agency (ASI). The authors are indebted to Luca Anselma for his support in implementing the decomposition mechanisms and in running the experiments.

REFERENCES

Felfernig, A., Friedrich, G.E., & Jannach, D. (2000). UML as domain specific language for the construction of knowledge-based configuration systems. *International Journal of Software Engineering and Knowledge Engineering 10(4)*, 449–469.

Fleischanderl, G., Friedrich, G.E., Haselböck, A., Schreiner, H., & Stumptner, M. (1998). Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems 13*, 59–68.

Fleischanderl, G., & Haselböck, A. (1996). Thoughts on partitioning large-scale configuration problems. *AAAI 1996 Fall Symposium Series*, pp. 1–10.

Friedrich, G., & Stumptner, M. (1999). Consistency-based configuration. *AAAI-99, Workshop on Configuration.*

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press

Gerstl, P., & Pribbenow, S. (1995). Midwinters, end games, and body parts: A classification of part–whole relations. *International Journal of Human–Computer Studies 43*, 865–889.
Gottlob, G., Leone, N., & Scarcello, F. (2000). A comparison of structural CSP decomposition methods. *Artificial Intelligence 124*, 243–282.
Magro, D., & Torasso, P. (2000). Description and configuration of complex technical products in a virtual store. *Proc. ECAI 2000 Configuration Workshop*, pp. 50–55.
Magro, D., & Torasso, P. (2001*a*). Interactive configuration capability in a sale support system: Laziness and focusing mechanisms. *Proc. IJCAI-01 Configuration Workshop*, pp. 57–63.
Magro, D., & Torasso, P. (2001*b*). Supporting product configuration in a virtual store. *LNAI 2175*, 176–188.
Magro, D., Torasso, P., & Anselma, L. (2002). Problem decomposition in configuration. *Proc. ECAI 2002 Configuration Workshop*, pp. 50–55.
Mailharro, D. (1998). A classification and constraint-based framework for configuration. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing 12(4)*, 383–397.
McGuinness, D. (2002). Configuration. In *The Description Logic Handbook. Theory, Implementation, and Applications* (Baader, F., McGuinness, D., Nardi, D., & Patel–Schneider, P., Eds.). Cambridge: Cambridge University Press.
McGuinness, D.L., & Wright, J.R. (1998). An industrial-strength description logic–based configurator platform. *IEEE Intelligent Systems 13*, 69–77.
Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. *Proc. AAAI 90*, pp. 25–32.
Mittal, S., & Frayman, F. (1989). Towards a generic model of configuration tasks. *Proc. IJCAI-89*, pp. 1395–1401.
Park, T.J., & Gelder, A.V. (2000). Partitioning methods for satisfiability testing on large formulas. *Information and Computation 162*, 179–184.
Sabin, D., & Freuder, E. (1996). Configuration as composite constraint satisfaction. *Proc. Artificial Intelligence and Manufacturing Research Planning Workshop*, pp. 153–161.
Soininen, T. (2000). *An approach to knowledge representation and reasoning for product configuration tasks*. PhD thesis. Espoo, Finland: Helsinki University of Technology. Available on-line at www.soberit.hut.fi/pdmg/papers/soin00App.pdf.
Soininen, T., & Gelle, E. (1999). Dynamic constraint satisfaction in configuration. *Configuration Papers from the AAAI Workshop*, pp. 95–100. AAAI Technical Report WS-99-05.
Soininen, T., Niemelä, I., Tiihonen, J., & Sulonen, R. (2000). Unified configuration knowledge representation using weight constraint rules. *Proc. ECAI 2000 Configuration Workshop*, pp. 79–84.
Soininen, T., Tiihonen, J., Männistö, T., & Sulonen, R. (1998). Towards a general ontology of configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12(4)*, 383–397.
Stumptner, M., & Haselböck, A. (1993). A generative constraint formalism for configuration problems. *LNAI 728*, 302–313.
Veron, M., & Aldanondo, M. (2000). Yet another approach to CCSP for configuration problems. *Proc. ECAI 2000 Configuration Workshop*, pp. 59–62.
Weida, R. (1996). Closed terminology in description logics. *Proc. AAAI-96*, Vol. 1, pp. 592–599.

Pietro Torasso received the Laurea degree in computer science from the University of Torino in 1974. Since November 1995 he has been a Professor of computer science at the University of Torino and he was the head of the Dipartimento di Informatica until 2001. Previously he was Associate Professor at the University of Torino, a Professor at the University of Udine, and a Professor at the Campus of Alessandria. He contributes to several areas of artificial intelligence (knowledge-based systems, knowledge representation, case-based reasoning, model-based diagnosis, configuration) and to human–machine interaction. He was the winner of the ECCAI award in 1990 and since 2000 has been is an ECCAI fellow.

Diego Magro received the Laurea degree in computer science from the University of Torino in 1997. Since 1999 he has been a Research Assistant at the University of Torino. In 1998 he worked as a Research Assistant at CSELT (the research lab of Telecom Italia, an Italian telecommunications company). His research interests are configuration, model-based diagnosis, and case-based reasoning.

https://doi.org/10.1017/S0890060403171053 Published online by Cambridge University Press