# The Role of Problem Decomposition in Configuration

Diego Magro
Dipartimento di Informatica, Università di Torino
Corso Svizzera 185; 10149 Torino; Italy
magro@di.unito.it

**Abstract.** The paper discusses the role of problem decomposition in the automatic configuration task. Configuration problem decomposition is mainly motivated by the need of reducing computational effort, but it is useful also in supporting interactive configuration. Two kinds of problem decomposition are introduced, both exploiting the implicit decomposition provided by whole-part relation between the components and the subcomponents in a configurable object. The decomposition policies resulting by combining these two kinds of decomposition are discussed and compared experimentally by usinf PC domain as a test bed.

## 1 Introduction

Configuration can be defined as the problem of assembling a set of pre-defined components in order to build an artifact that meets a set of requirements. The components interact in a set of pre-defined ways and during the configuration process no new component type can be introduced, neither a component type can be modified.

In recent years many approaches to automatic configuration have emerged [8]. Some of them make use of CSP framework and its extensions [6, 1, 7, 10] to model and solve configuration problems; other approaches rely on logical frameworks either derived from consistency-based diagnosis [2] or from logic programming [9]. Further frameworks exploit the power of description logics (DL) to represent explicitly the structure of the entities to be configured as well as the complex relations holding among them [5].

The present paper addresses the issue of decomposing a configuration problem into a set of (simpler) subproblems; in particular, it introduces two kinds of configuration problem decomposition, both exploiting the implicit decomposition provided by the partonomic knowledge (i.e. the whole-part relation between the components and the subcomponents): *requirements-based decomposition* and *constraints-splitting decomposition*.

Since configuration can be computationally expensive (it is intractable, in the worst case), one of the main motivations of problem decomposition is controlling complexity, even if *requirements-based decomposition* can be useful also in interactive configuration and in the task of checking the consistency of the requirements that a user imposes on the entity to be configured.

Decomposition mechanisms are introduced in the $\mathcal{FPC}$ framework [4]. $\mathcal{FPC}$ is a KL-One like conceptual language, with a formal DL-based semantics, in which partonomic relations play a major role.

$\mathcal{FPC}$ offers the capability of representing explicitly the structure of the configuration domains and of expressing complex constraints that describe intensionally the set of valid combinations of components and subcomponents in any configurable object.

Preliminary experimental results comparing the decomposition policies (that can be defined by combining the two proposed kinds of decomposition) are also discussed.

## 2 Characterizing Configuration Problems

In the following, a conceptual language for modeling configuration domains is briefly described; moreover, the concept of configuration problem is defined and two different notions of solution to a such a problem are given.

### 2.1 The Conceptual Language

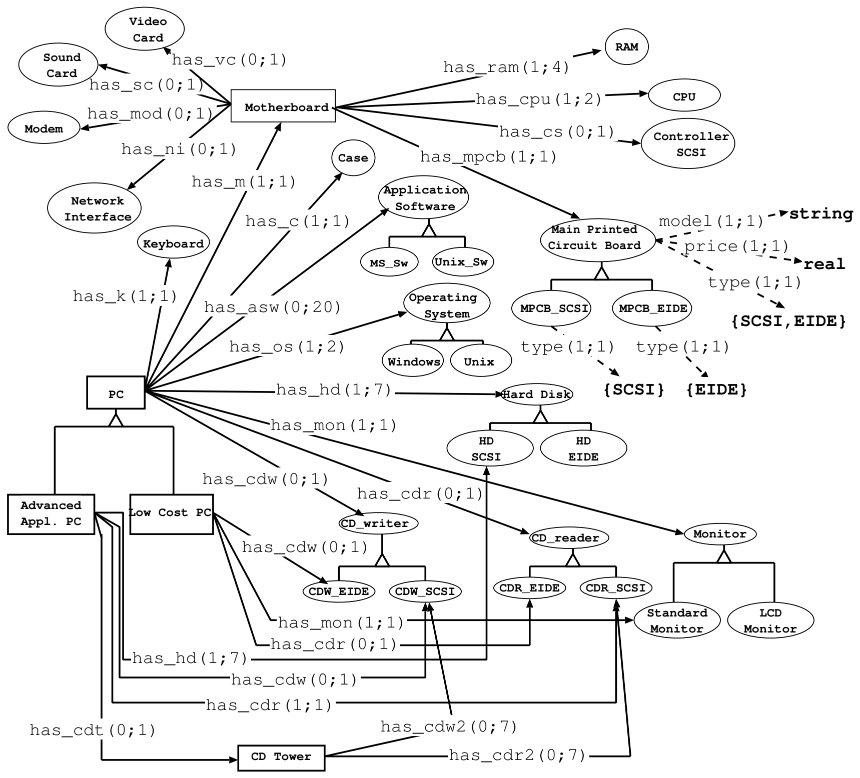
In the present paper the conceptual language $\mathcal{FPC}$ (Frames, Parts and Constraints) is adopted to model the configuration domains. Basically, $\mathcal{FPC}$ is a frame-based KL-One like formalism augmented with a constraint language.

$\mathcal{FPC}$ is described here by means of an example. A formal description of the language and its semantics can be found in [4].

$\mathcal{FPC}$ allows one to describe classes of *atomic* or *complex* components and to organize them in *taxonomies*. *Atomic components* are the basic building blocks of configurations and they are described by a set of features. $\mathcal{FPC}$ represents such features by means of *descriptive slots* associated with the classes of atomic components. *Complex components* are structured entities whose characterization is given mainly in terms of their (sub)components, which can be complex components in their turn or atomic ones. In $\mathcal{FPC}$, *whole-part* relations between complex components and their parts are expressed by means of *partonomic slots* associated with the classes of complex components.

Figure 1 contains a simplified conceptual model for PC configuration. Each rectangle represents a class of complex components, each oval corresponds to a class of atomic components, each dashed arrow represents a descriptive slot, while each solid arrow represents a partonomic slot. In the figure it is stated, for instance, that a PC is a complex object having as *direct parts* one motherboard (that plays a partonomic role named $has\_m$), from one to seven hard disks (see the partonomic slot $has\_hd$), one or two operating systems (playing a partonomic role named $has\_os$) and so on. Hard disks and operating systems are atomic objects (i.e. they are considered as basic components in PC configuration), while a motherboard is a complex component. In fact, its partonomic structure is expressed by means of the partonomic slots $has\_mpcb$, $has\_cs$, $has\_cpu$ and so on. A main printed circuit board is an atomic component whose features are described by the descriptive slots $model$ (whose value is a string), $price$ (whose value is a real number) and $type$ (that takes value in the set of linguistic entities $\{SCSI, EIDE\}$) (the descriptive slots for the other classes are not shown in figure).

Each slot has a *number restriction* and a *value restriction*. For instance, slot $has\_hd$ has number restriction $(1; 7)$ (i.e. it can take from $1$ to $7$ values) and value restriction $Hard\_Disk$ (i.e. its values belong to the class of hard disks).



**DOMAIN CONSTRAINTS**

```
Associated with PC class
[co1] (<has_m,has_mpcb>)(in MPCB_EIDE) ∧ (<has_hd>)(in HD_SCSI (1;7))⇒ (<has_m,has_cs>)(1;1)
[co2] (<has_asw>)(in Unix_sw (1;20))⇒(<has_os>)(in Unix (1;2))
[co3] (<has_asw>)(in MS_sw (1;20))⇒(<has_os>)(in Windows (1;2))

Associated with Advanced Appl. PC class
[co4] (<has_cdw>)(0;0) ⇒(<has_cdt,has_cdw2>)(1;7)

Associated with Motherboard class
[co5] (<has_mpcb>)(in MPCB_SCSI) ⇒(<has_cs>)(0;0)
[co6] (<has_cpu>)(2;2) ⇒(<has_ram>)(4;4)
```

**USER'S REQUIREMENTS FOR an Advanced Appl. PC (*REQS*)**

```
[req1] true ⇒(<has_m,has_mpcb>)(in MPCB_EIDE)
[req2] true ⇒(<has_hd>)(in HD_SCSI (1;1))
[req3] true ⇒(<has_asw>)(in ApplX (1;1))
```

**Fig. 1.** A (simplified) conceptual model $CM_{PC}$ for PC configuration and the user's requirements

Moreover, the figure shows, for example, that the class of PCs is partitioned into two subclasses: the one of advanced application PCs and the one of low cost PCs. Each subclass inherits all the slots of its superclass and it can refine the description of the superclass by specifying additional (w.r.t. those inherited) slots or by restricting the properties expressed by the inherited slots. For example, each advanced application PC can have a CD tower (instead, this is not possible in a low cost PC); moreover, in general, a PC can have an optional CD reader that can be of type either EIDE or SCSI (see the partonomic slot $has\_cdr$ associated with the $PC$ class), while for an advanced application PC a CD reader of type SCSI is mandatory (see the same partonomic slot in the $Advanced\ Appl.\ PC$ class).

In the following, the attention is restricted to partonomic slots only, since in this framework partonomic knowledge is the basis for problem decomposition.

In a $\mathcal{FPC}$ model, a set (possibly empty) of constraints is associated with each class of complex components. These constraints restrict the set of valid combinations of components and subcomponents in complex entities. The conceptual model in figure 1 contains only six of the many constraints present in a real PC configuration domain. Let us consider constraint $co1$. It is associated with the class of PC, thus it holds for each instance of that class. The three *slot chains* $\langle has\_m, has\_mpcb\rangle$, $\langle has\_hd\rangle$ and $\langle has\_m, has\_cs\rangle$, occurring in $co1$, denote the main printed circuit board (subcomponent of any PC), the hard disks (direct components of any PC) and the controller SCSI (optional subcomponent of any PC), respectively. $co1$ means that "if in a PC there is an EIDE main printed circuit board and at least one SCSI hard disk, then a controller SCSI is needed". Constraint $co2$ means that "if in a PC there is at least one Unix application program, then at least one Unix operating system has to be installed"; an analogous constraint for Microsoft application programs and Windows operating systems is expressed by $co3$. Constraint $co5$ is relevant to motherboards and it states that "no controller SCSI has to be inserted into a motherboard with a SCSI main printed circuit board"; $co6$ states that any motherboard with 2 CPUs has to contain four RAM slots. [1]

In a $\mathcal{FPC}$ model each subclass inherits the constraints of its superclass and it can also specify additional constraints w.r.t. those inherited. For example, $co4$ states that in any advanvanced application PC at least one SCSI CD writer must be present either as a direct part (slot $has\_cdw$) or as a subpart in a CD tower (slot chain $\langle has\_cdt, has\_cdw2\rangle$).

## 2.2 Configuration Problems and their Solutions

In this context, a **configuration problem** is a 3-uple $CP = \langle CM, C, REQS\rangle$, where $CM$ is a $\mathcal{FPC}$ conceptual model, $C$ is a class of complex components most specific in the $CM$ taxonomies and $REQS$ is a set of $\mathcal{FPC}$ constraints whose slot chains start in $C$, expressing the user's requirements. $CP$ represents the problem of configuring a complex object of type $C$ (called *target object*) in such a way that both the user's requirements $REQS$ and the conceptual model $CM$ are satisfied.

As usual in configuration task, we assume the **consistency of** $CM$, i.e., we assume that for each class $D$ in $CM$, it is possible to determine an instance $d$ of $D$ satifying both the parto-taxonomic description of $D$ and the constraints occurring in $CM$ for all the classes involved in the specification of $d$.

A **complete configuration** $T$ solving a configuration problem $CP = \langle CM, C, REQS\rangle$ is a description of an instance $c$ of $C$ such that: (1) $T$ explicitly lists all the components occurring in $c$; (2) for each component $d$ occurring in $T$, each slot $p$ associated with $class(d)$ in $CM$ is explicitly instantiated, that is all its values are specified such that

[1] This constraint has been inserted in this simplified sample conceptual model because of its usefulness in the example of section 4.

they respect both the number and the value restrictions of $p$ ($class(d)$ is the taxonomically most specific class in $CM$ to which $d$ belongs); (3) each constraint stated in $CM$ for the classes of the components occurring in $T$ is satisfied; (4) the requirements $REQS$ are satisfied.

As we shall see, in some situations it is useful to be able to introduce into the configuration only those components possibly critical for the satisfaction of the user's requirements. Thus, in the following the concept of *partial configuration* is defined.

A **partial configuration** $T$ solving a configuration problem $CP = \langle CM, C, REQS\rangle$ is a description of an instance $c$ of $C$ such that: (1) $T$ explicitly lists only the components of $c$ possibly critical for the satisfaction of the requirements $REQS$ (i.e. those components involved in the constraints potentially interfering with the requirements $REQS$); (2) for each component $d$ occurring in $T$, a slot $p$ associated with $class(d)$ in $CM$ is explicitly instantiated (such that both its number and value restrictions are respected) only if it occurs in a constraint potentially interfering with the requirements $REQS$; (3) each constraint stated in $CM$ for the classes of the components occurring in $T$ and such that it potentially interferes with the requirements $REQS$ is satisfied; (5) the requirements $REQS$ are satisfied.

It is worth noting that the *partial configuration* requires that the description of the target object $c$ is complete enough to assure that both the requirements $REQS$ and the constraints stated in $CM$ and potentially interfering with $REQS$ are satisfied. This is enough to have the guarantee that each partial configuration $T$ can be extended to a complete configuration $T'$ still representing a solution for $CP$. In fact, the set of valid combinations of the components and subcomponents of $c$ that are not explicitly listed in $T$ is not influenced by the user's requirements: the *consistency of* $CM$ (see above) assures that this set is not empty.

As we shall see, the *bound relation* defined in section 2.3 provides a formal characterization of the concept of *potentially interfering constraints*.

### 2.3 Recognizing Potentially Interfering Constraints

Partonomic knowledge plays a major role in defining the decomposition mechanisms since it can be straightforwardly used in recognizing possibly interacting constraints.

We assume that the following **exclusiveness assumption on parts** holds: in any configuration $T$ (either complete or partial), a component can not be a direct part of two different (complex) components, neither a direct part of a same (complex) component through two different whole-part relations.

Potential interference among constraints is captured by the **bound relation**. Intuitively, two constraints are *bound* iff the choices made during the configuration process in order to satisfy one of them *may* interact with those actually available for the satisfaction of the second one. If $c$ is a complex component in a configuration, the *bound* relation $\mathcal{B}_c$ is defined in the set $CONSTRS(c)$ of the $\mathcal{FPC}$ constraints that $c$ must satisfy, as follows: let $u, v, w \in CONSTRS(c)$. **If** $u$ and $v$ contain both a *same* partonomic slot $p$ of $class(c)$ **then** $u\mathcal{B}_c v$ (i.e. if $u$ and $v$ refer to a same 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).

It is easy to see that $\mathcal{B}_c$ is an equivalence relation. If $U$ is an equivalence class in the quotient set $CONSTRS(c)/\mathcal{B}_c$, every constraint in $U$ may interact with any other constraint in the same class during the configuration process of $c$. Instead, if $V \in CONSTRS(c)/\mathcal{B}_c$ is different from $U$, because of *exclusiveness assumption on parts*, the constraints in $V$ and those in $U$ refer to different (sub)parts of $c$, and thus these two sets of constraints do not interact each other.

## 3 The Decomposition Policies

The automatic configuration task can be, in general, computationally expensive. In fact, the set of constraints specific to the domain and those imposed as additional requirements usually link together many components and subcomponents in a configurable object. Therefore, a configuration can rarely be computed by a process making only a set of local choices. In most cases, a configuration is built by means of a search process: when a choice is made for a component during the configuration activity that prevents another (or even the same) component to be configured, the process backtracks (if possible), revises a past choice and explores a different path in the search space. In many cases, this space is quite huge and many paths in it do not lead to any solution. Moreover, even if the domain model is consistent, there is no guarantee that a solution always exists for each configuration problem, since the user's requirements can be inconsistent either by themselves or w.r.t. the domain model.

Controlling complexity is one of the main motivations for decomposing configuration problems even if it is not the only one.

This section introduces two kinds of decomposition of a configuration problem $CP = \langle CM, C, REQS \rangle$:

**1. Requirements-based decomposition**: this decomposition technique decomposes $CP$ into two subproblems: the subproblem of computing a *partial configuration* for $CP$ and the subproblem of extending this partial configuration in order to compute a *complete* one (actually, it is worth noting that in those tasks in which only a partial configuration is required the latter subproblem do not need to be solved). The ability of individuating these two subproblems has (at least) four main motivations: (1) given the consistency assumption of the conceptual model (section 2.2), if $CP$ has no solution, this is due to the requirements $REQS$. By first considering the components potentially involved in the satisfaction of $REQS$ (i.e. by first building a partial configuration for $CP$), it is expected that the inconsistency of the requirements can be detected early in the configuration process (saving useless work); (2) even if $CP$ is consistent, whenever the configuration process enters a failure state in the search space in which a requirement is violated, a revision of any choice made for a component that can be recognized a priori not to be critical for the satisfaction of $REQS$ would be useless and it can be easily avoided by considering such components in a second phase (if needed); (3) if the task is the consistency testing of $REQS$, the configuration of the whole object would be, in many cases, a useless effort and the computation of a partial configuration is enough (in this case, only the first of the two subproblems needs to be solved); (4) when supporting an interactive configuration, the configurator can not assume that all the requirements are always known from beginning, since they can be incrementally added by the user. In this scenario, the configurator should not be too eager to configure the entire complex object, but it should build only a partial configuration by leaving the user the possibility

| | | Requirements-based | | |
|---|---|---|---|---|
| | | $c\_no\_reqs$ | $c\_reqs$ | $p\_reqs$ |
| **Costraints-splitting** | $no\_co$ | 1 | 2 | 4 |
| | $yes\_co$ | – | 3 | 5 |

**Table 1.** Decomposition policies

of adding some requirements for the components not yet introduced into the (partial) configuration [3].
**2. Constraints-splitting decomposition**: in general, many constraints in $CP$ (both those occurring in $CM$ and those in $REQS$) can be *bound* (and thus a choice made in order to satisfy one of them may restrict the choices available for satisfying another one). However, in many cases it does not happen that every constraint interacts with each other and the ability of recognizing the sets of interacting constraints is the basis for decomposing the whole configuration problem into a set of smaller and independent subproblems each one relevant to a set of *bound* constraints.

If there is more than one set of bound constraints, $CP$ is said to be *constraints-splitting decomposable*.

By combining these two decomposition techniques, it is possible to define a set of decomposition policies. Table 1 summarizes those considered in the present paper (each number identifies a policy). In this table, $c\_no\_reqs$ and $c\_reqs$ mean that a *complete configuration* is searched for without performing a requirements-based decomposition ($c\_no\_reqs$) or by performing such a decomposition ($c\_reqs$); $p\_reqs$ means that this kind of decomposition is used to search for a *partial configuration* only; $yes\_co$ means that a constraints-splitting decomposition is attempted and $no\_co$ means the opposite. The ”finer” constraints-splitting decomposition is not attempted when the ”coarser” requirements-based one is not performed, thus the combination $c\_no\_reqs$ - $yes\_co$ is not considered.

## 4 An Example

In order to illustrate how the decomposition techniques are used in a configuration process, let’s consider the configuration problem
$CP = \langle CM_{PC}, Advanced\ Appl.\ PC, REQS\rangle$, i.e. the problem of configuring an advanced application PC, given the PC domain $CM_{PC}$ shown in figure 1 and the user’s requirements $REQS$ listed in the same figure (the requirements specify that the PC has to contain an EIDE main printed circuit board, exactly one SCSI hard disk and the Unix application $ApplX$). Moreover, let’s suppose that policy $5$ (table 1) is adopted. [2]

At the beginning of the process only the component $aapc$ (representing the target object) is inserted into the configuration. The set of constraints that the target advanced application PC $aapc$ must satisfy is

[2] The backtracking mechanism is out of the scope of this paper, thus we describe a particular case in which the configuration process goes ”straight” to the solution without any backtracking.

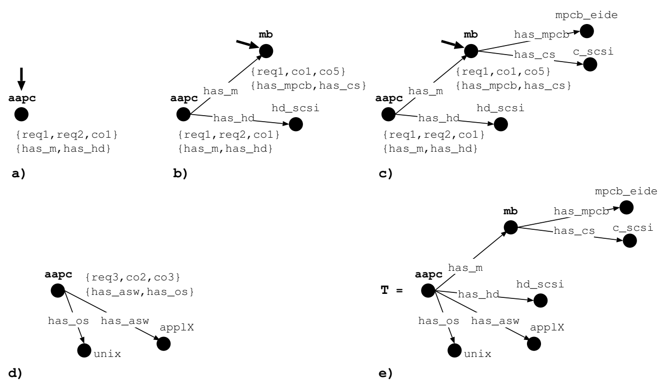


**Fig. 2.** A configuration example

$CONSTRS(aapc) = \{req1, req2, req3, co1, co2, co3, co4\}$ (i.e. it must satisfy all the domain constraints associated with the $Advanced\ Appl.\ PC$ class plus the user's requirements $REQS$). Policy 5 makes use of *requirements-based decomposition*, thus the set of constraints *bound* to the requirements is computed: $CONSTRS_{REQS}(aapc) = \{coi \in CONSTRS(aapc)/\ (\exists reqj \in REQS)(coi\ \mathcal{B}_{aapc}\ reqj\} =$ $\{req1, req2, req3, co1, co2, co3\}$; constraint $co4$ is not critical for the satisfaction of $REQS$, therefore it is not included in $CONSTRS_{REQS}(aapc)$. Moreover, policy 5 requires only a *partial configuration*, thus $co4$ will never be considered (i.e. the subproblem of completing the partial configuration does not need to be solved). The *constraints-splitting* decomposition is attempted and $CONSTRS_{REQS}(aapc)$ is partitioned into a set of classes of bound constraints w.r.t. the relation $\mathcal{B}_{aapc}$: $CL\_CONSTRS(aapc) = CONSTRS_{REQS}(aapc)/\mathcal{B}_{aapc} = \{V_1, V_2\}$, where $V_1 = \{req1, re2, co1\}$ and $V_2 = \{req3, co2, co3\}$. $CP$ is *constraints-splitting decomposable*, in fact $|CL\_CONSTRS(aapc)| > 1$. Each class of bound constraints induces a subproblem. Figure 2.a reports the subproblem associated with $V_1$. Only the (sub)components possibly critical for the satisfaction of $V_1$ are considered while solving this first subproblem; therefore only the set $P\_SLOTS(aapc) = \{has\_m, has\_hd\}$ of the partonomic slots of $Advanced\ Appl.\ PC$ class occurring in the constraints belonging to $V_1$ are taken into consideration. The direct components of $aapc$ relevant to these slots are then inserted into the configuration (i.e. the motherboard $mb$ and the SCSI hard disk $hd\_scsi$: see figure 2.b). $mb$ is a complex component, hence it has to be configured. $mb$ must satisfy all the constraints that it inherits from $aapc$ (i.e. all the constraints in $V_1$ involving some components of $mb$), plus the local ones (i.e. those associated with $Motherboard$ class in $CM_{PC}$): $CONSTRS(mb) = \{req1, co1, co5, co6\}$. All the inherited constraints are *bound* to the requirements, thus all (and only) the local constraints *bound* to the inherited ones w.r.t. $\mathcal{B}_{mb}$ have to be considered: $CONSTRS_{REQS}(mb1) = \{req1, co1, co5\}$ ($co6$ is discarded, in fact it is not critical for the satisfaction of $REQS$). Again, only the components of $mb$ relevant to the partonomic slots $P\_SLOTS(mb) = \{has\_mpcb, has\_cs\}$ are inserted into the configuration (i.e. the EIDE main printed circuit board $mpcb\_eide$ and the controller SCSI $c\_scsi$). The partial configuration depicted in figure 2.c represents a solution to the subproblem induced by $V_1$. A solution to the subproblem induced by $V_2$ is computed in a similar way (figure 2.d). The two solutions are then merged to produce a partial configuration solving the whole problem $CP$ (figure 2.e).

The final partial configuration $T$ describes only a portion of the advanced application PC; it is worth noting that in an interactive scenario, the user could specify an additional set of requirements involving some of those components not explicitly mentioned in the partial configuration (e.g. she could state that she wants two CPUs and no internal CD writer). In this case, the configurator searches for an expansion of the partial configuration satisfying these new constraints. In this process, no choice relevant to the partial configuration $T$ needs to be revised.

## 5 Experimental Results

A test set of $153$ configuration problems (containing also some inconsistent problems) has been designed to test the performance of the different policies w.r.t. the computational effort. All the problems are relevant to a same real domain of PC configuration. The configuration system has been implemented in Java using JDK 1.3 and the experiments have been performed on a Duron 700/Windows 2000 PC with 128 Mbytes of memory. A configuration problem $CP$ is considered solved iff the configurator either provides a solution for $CP$ or detects its inconsistency, within the timeout of 360 seconds.

Policy $1$ was able to solve only $98$ problems out of $153$ (i.e. its *competence* was of $64.1\%$). For the $98$ solved cases the average computation time was 15016 msec. (with 3494 backtrackings in average). Policy $2$ obtained much better results, since it solved $140$ problems (i.e. its competence was of $91.5\%$). For the $140$ solved cases the average computation time was $13007.4 msec.$ (with $1400.6$ backtrackings in average). That is, policy $2$ was able to solve most of the difficult problems that policy $1$ was unable to solve, without penalizing the average cost.

As concerns the comparison between policies $2$ and $3$, it came out that their competence was the same (i.e. they both solved the same $140$ problems), while policy $3$ was slightly better since in average it saved $5.7\%$ of CPU time and $6.7\%$ of backtrackings on the $140$ solved cases. However, the benefits of policy $3$ become much more evident if we compare them on the set of the $45$ solved problems that were actually constraints-splitting decomposable. For these problems, policy $3$ was able to save in average $28.3\%$ of CPU time and a corresponding $28.1\%$ of backtrackings.

The comparison between policies $4$ and $5$ gave similar results.

## 6 Discussion and Future Work

In the present paper, we have described two kinds of decomposition techniques for attacking the problem of automatically decomposing a configuration problem into a set of independent subproblems. The first one (*requirements-based decomposition*) is based on the capability of singling out the (sub)components of the configurable object that are related to the user's requirements and by first configuring that portion of the target object involving these components. The other decomposition technique (*constraints-splitting decomposition*) is based on the possibility of partitioning the constraints holding for the target object into sets of (potentially) interacting constraints. Such constraints partitioning induces a decomposition of the main configuration problem into a set of smaller independent subproblems.

Different decomposition policies obtained by combining these two decomposition methods have been compared experimentally among them in a real PC configuration domain. The preliminary experimental results showed that the first kind of decomposition allows the configurator to solve (within a time threshold of 360 sec.) many difficult configuration problems that the approach without decomposition was not able to solve (within the same time threshold). Moreover, the experimental results demonstrate that further improvements can be obtained by complementing *requirements-based decomposition* with *constraints-splitting decomposition*.

In the experiments, a centralized approach was adopted in which the subproblems resulting from constraints-splitting decomposition were sequentially considered by a configurator; further experiments are needed in order to measure the advantages of distributing these subproblems among a set of configurators running in parallel.

Moreover, many problems in the test set used in the experiments were not constraints-splitting decomposable. This fact suggests the opportunity of investigating a *recursive* constraints-splitting decomposition technique that attempts to partition the constraints not only in the target object, but also in its components and subcomponents.

## References

1. G. Fleischanderl, G. E. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, (July/August 1998):59–68, 1998.
2. G. Friedrich and M. Stumptner. Consistency-based configuration. In *AAAI-99, Workshop on Configuration*, 1999.
3. D. Magro and P. Torasso. Interactive configuration capability in a sale support system: Laziness and focusing mechanisms. In *Proc. IJCAI-01 Configuration WS*, pages 57–63, 2001.
4. D. Magro and P. Torasso. Supporting product configuration in a virtual store. *LNAI*, 2175:176–188, 2001.
5. D. L. McGuinness and J. R. Wright. An industrial-strength description logic-based configurator platform. *IEEE Intelligent Systems*, (July/August 1998):69–77, 1998.
6. S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proc. of the AAAI 90*, pages 25–32, 1990.
7. D. Sabin and E.C. Freuder. Configuration as composite constraint satisfaction. In *Proc. Artificial Intelligence and Manufacturing. Research Planning Workshop*, pages 153–161, 1996.
8. D. Sabin and R. Weigel. Product configuration frameworks - a survey. *IEEE Intelligent Systems*, (July/August 1998):42–49, 1998.
9. T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Unified configuration knowledge representation using weight constraint rules. In *Proc. ECAI 2000 Configuration WS*, pages 79–84, 2000.
10. M. Veron and M. Aldanondo. Yet another approach to ccsp for configuration problem. In *Proc. ECAI 2000 Configuration WS*, pages 59–62, 2000.