# Thoughts on Partitioning Large-Scale Configuration Problems

**Gerhard Fleischanderl**
**Alois Haselböck**

**Siemens Austria, PSE EZE**
**Erdberger Laende 26**
**A-1030 Vienna, Austria**
**E-mail: gerhard.fleischanderl@siemens.at**
**alois.haselboeck@siemens.at**

## Abstract

Most 'interesting' configuration problems tend to be large-scale. Therefore, the configuration networks grow to remarkable sizes and techniques must be considered to keep handling, solving and checking of these networks practically feasible. This paper presents a technique we use in our current configuration project: the partitioning of large problems into pieces of manageable sizes. The very basics of our component-oriented framework for representing configuration domains are presented. Partitioning a configuration problem means identifying subproblems in the configuration graph and solving them separately. Beside the reduction in problem size we provide the user with control mechanisms by displaying a sub-system-graph to guide the user through the configuration process.

## 1 Introduction

Configuring large-scale systems entails problems which are irrelevant when configuring small systems. First of all, that is sheer size of the configuration problem. Due to interrelations, complexity and effort may rise disproportionately along with problem size. In this paper we want to describe another aspect found in large problems, identifying sub-problems. Sub-problems are more easily solved (at least, ought to). Constraint solving techniques (like backtracking) usually are computationally expensive [5]. Often the problem as a whole could not be solved under practical circumstances, thus the definition of sub-problems is the precondition to the feasibility of the solution. The idea of breaking down a large problem into pieces of manageable sizes is also a well-known method in constraint theory (see, e.g., [2] and [6]).

We are dealing with configuration problems and tackle them with a constraint approach. Identifying a sub-problem means finding a sub-set of components and constraints which form a self-contained tasks. The less connections there are to neighboring sub-problems, the better. Our configuration domain is (as we call them) component-oriented systems which are assembled of components from a predefined set, observing the rules laid down as constraints [8], [3]. Application domains are systems constructed from modular components, for instance, telecommunication systems, audio systems or computers.

The paper is organized as follows: In the first part we give a brief introduction into our component-oriented framework of representing configuration networks, where constraints represent the different restrictions and relations between the components. We present a few numbers from our current project, indicating the magnitude of configuration and constraint networks many real-world configuration problems have to deal with. After that we present our approach to partitioning large-scale problems and discuss it. We argue that the main two advantages are reduced problem sizes (computation- and space-expensive methods turn to be applicable) and that the sub-systems will control the configuration process and will therefore lead the user through a usually highly complicated task.

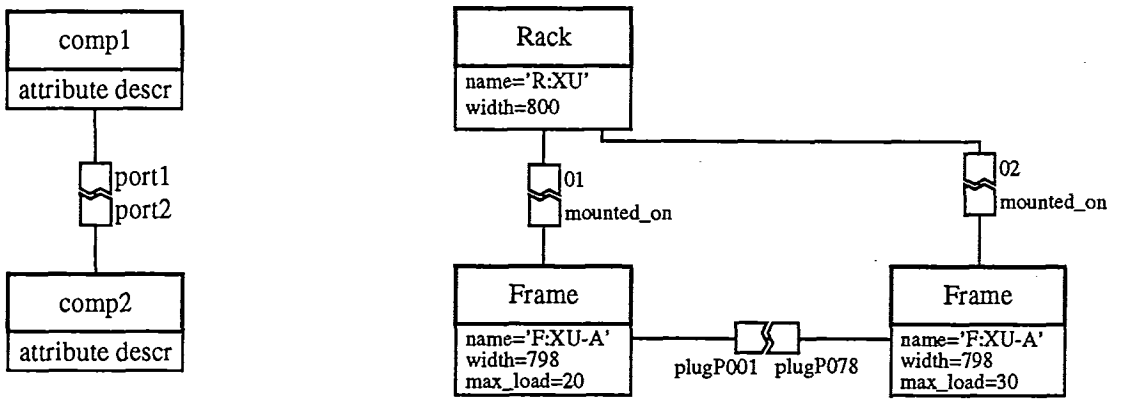
# 2 Problem environment

## 2.1 The configuration framework

There are several theories of representing configuration problems, all of them having in common that components play the central role in the description of the problem domain (see, e.g., [7], [1], [4]). This chapter sketches a simple but general framework for representing technical systems in a component-oriented manner [8], [3]. The first-class objects in this representation framework are components, which are connected via ports to a complex configuration network. Additionally, each component has a type and a set of attributes describing individual properties of that component. A configuration is a network consisting of components and their ports assembled to a complex system.

Figure 1(a) shows the basic concept representing a connection between two components. This must be done by connecting two ports, each belonging to a single component: port1 of component comp1 is connected to port2 of component comp2. Of course, it is also possible that a port remains unconnected in a configuration. Figure 1(b) presents a small part of a configuration, where frames are mounted on racks and, furthermore, are connected to other frames via cables (a cable is modelled simply by the connection of two plugs; of course, if further details in representing a cable are necessary, a component representation must be chosen). We see that the different kinds of relations (the hierarchical mounted-on connection, the cable connections between two components at the same level in the hierarchy) can all be described by the same

principle of connecting two ports. In the case of the mounted-on relation, an artificial port called 'mounted_on' of the frame is necessary to act as counterpart of port '01' of the rack.




(a) port-to-port connection

(b) example of a port-to-port connection

**FIGURE 1. Components and Connections**

The basic possibilities and structures of a configuration domain are defined in a component catalog. The component catalog stores

- the component types in a class hierarchy,
- for each component type: the set of attributes, the set of ports, the set of constraints restricting the attribute values and port-to-port connections (see Section 2.2).

A configuration network contains a full description of the system to be configured. It depends on the configuration application which tasks may be performed to handle and manipulate a configuration network, e.g.

- check if all the constraints on the configuration are satisfied,
- change the configuration by interactions via a user interface,
- complete or change (parts of) the configuration network with solving procedures (like backtrack search in a CSP or highly specific, application-dependent configuration routines).

## 2.2 Constraint networks

In general terms, a constraint is a condition which rules out inconsistent or unwanted constellations. The types of constraints typically range from such basic rules like

"Slot '01' of a component of type 'R:XU' may only contain components of type 'F:XU-A'." (C1)

to highly complex conditions reaching over several connection levels like

"All the modules mounted on a frame, which itself is mounted on a rack of type 'R:XU', must have the same value of attribute *service*." (C2)

It is beyond the scope of this paper to describe details of different types and properties of constraints. A constraint is extensionally represented as an object with links to all the components and ports involved into the constraint condition. E.g. constraint (C1) is propagated to the rack component, its port '01' and the connected frame, while constraint (C2) is linked to all modules mounted on a frame, the frame itself, and the rack where the frame is mounted on.

Consequently, when generating and propagating all constraints in a configuration network, a second network - the so-called constraint network - is built *on top of* the configuration network, where the nodes are the individual constraints, connected to the components and ports of the system. Figure 2 shows how a configuration graph is augmented by a constraint network. The port objects and with them the connections between two components are depicted as simple lines here.

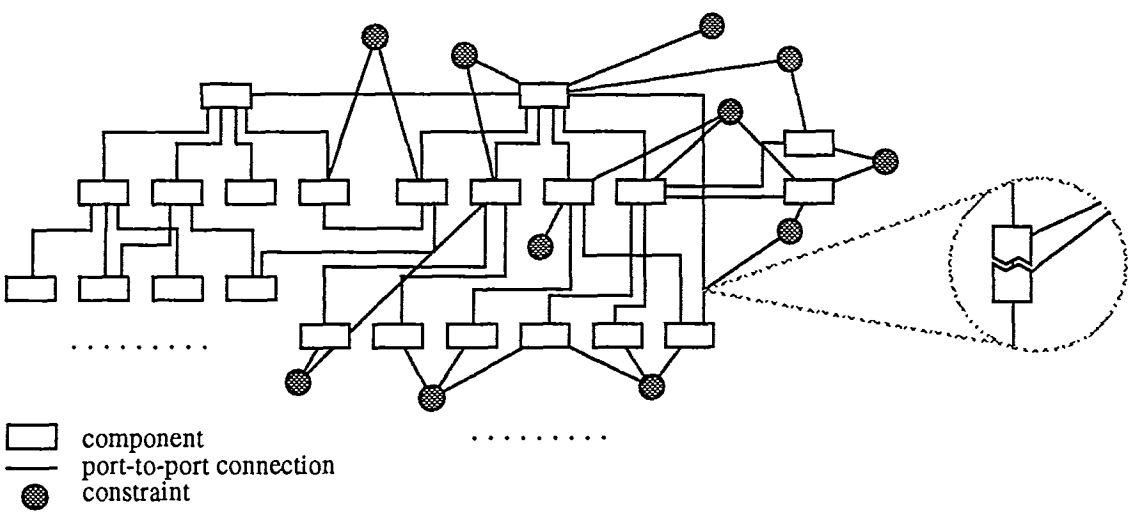



**FIGURE 2. Configuration Network and Constraint Network**

We distinguish in our framework between configuration network and constraint network. By handling these networks separately, we are able to apply specialized procedures to the constraint network.

A constraint network can be used in various ways. The main tasks a user may carry out on a constraint network are:

- Consistency check: check if all the constraints of the configuration or part of it are fulfilled.
- Solving: automatically extend a (possibly empty) configuration to achieve additional requirements, which are formulated by special constraints.
- Explanation: explain why a given configuration is inconsistent by showing all the constraints that have failed.

## 2.3 Exemplary application domain

Our configurator has to deal with system sizes which let "naive" representation and solving methods run into serious performance problems (both with time and memory). We use a telecommunication system to outline how many elements are there in a large configuration problem. (We are presenting approximate numbers which were slightly simplified for the sake of clarity.)

There are 20 types of racks, 50 types of frames, 200 types of modules, 50 types of cables, and 50 types of other 'organizational' units. Each component has 5 attributes. Each rack has on average 10 ports, a frame 50 ports, a module 1 port, a cable 2 ports, and each other unit 5 ports.

Each port has a port domain which is a constraint on its own. There are 5 constraints at a typical rack, 10 constraints at a frame, 1 constraint at a module, 1 constraint at a cable, and 5 constraints at an other unit. On average, half of the constraints cover 5 components or ports, the other constraints covering typically 10 components or ports (thus, many constraints establish quite complex relationships).

A large configuration comprises 200 racks, 1000 frames, 30,000 modules, 10,000 cables and 2000 other units. Summing it up, the configuration network comprises about 43,000 components with 215,000 attributes and 112,000 ports. At run-time, all the constraints associated with a component in the configuration would be instantiated, thus yielding appr. 61,000 constraint instances having appr. 450,000 references to components or ports. (In addition, there are lots of internal data structures used in the constraint solver.)

## 2.4 Difficulties found in large-scale configuration problems

Building the full networks (for components as well constraints) to solve the problem may consume too much memory, thus preventing the application of configurators on smaller computers like typical PCs. Another way to handle this problem is the usage of a database system. But we have learned in our project that it is not easily feasible to keep and maintain the constraint network as data in the database. Changes made manually via a user interface or tasks like copying parts of an other configuration into the project make it impossible to maintain the

constraint network over the time. On the other hand, to build the *whole* constraint network in working memory for a large configuration is practically impossible.

When the computation of the solution to a large-scale problem lasts too long, users are left in the dark about where the bulk of computation effort arises.

In theory there are optimal solutions to configuration problems. In practice the task of writing and testing the component catalogue (knowledge base) often becomes too complex to be carried out with the whole system in mind. For the sake of feasibility, the knowledge engineer has to break down the tasks into smaller portions.

Summing it up, most 'interesting' configuration problems tend to be large-scale. Therefore, the configuration networks grow to remarkable sizes and techniques must be considered to keep handling, solving and checking of these networks practically feasible.

# 3 Ideas from our approach

## 3.1 Partitioning a configuration network into packages

To tackle large-scale configuration problems, we resort to 'divide and conquer'. We propose dividing the configuration network into smaller pieces and solving these pieces one by one. We call each piece a *package*, reflecting the assumption that solving a package is an own, isolated sub-process.

The definition of the partitioning of a configuration network and a constraint network into packages is simple: A package consists of a set of components and ports; each component and each port belongs to at least one package. Let P be the set of components and ports a constraint is propagated to. The constraint simultaneously belongs to all packages of all objects of P. Figure 3 shows the partitioning of our example configuration network (Figure 2) into two packages. Package 2 in this example includes the configuration plus a constraint network, while for package 1 the constraint network is not generated in this situation.

To be effective, the partitioning into subproblems as indicated above requires that the dependencies between two packages are low. Thus, the number of port-to-port relations between two packages and the number of cross-package constraints should be near zero. The standard constraint solving procedure has to be modified to join together the results from packages and deal with cross-package relationships. (Of course, these cross-package constraints may lead to heavy re-configurations of early packages, when invalid decisions are made there but considered not until the second package has been done. We argue that it depends on the quality of the partitioning and on good solving heuristics to make such inefficiencies unlikely.)

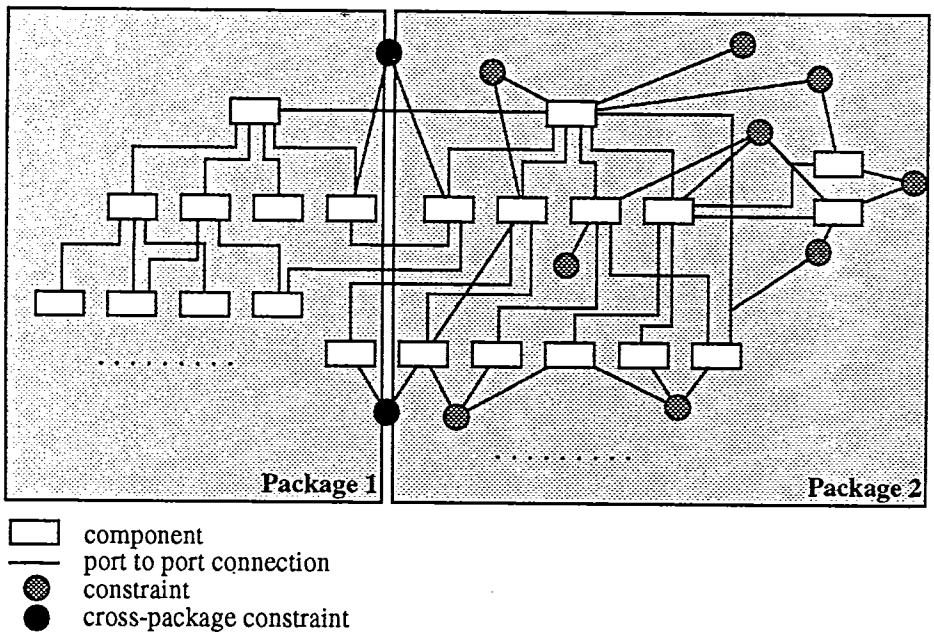



**FIGURE 3. Packages**

Usually, the partitioning of a configuration problem into packages comes naturally from the structure of the system to be configured. A human expert who configures a system by hand is able to identify packages suited for configuring step by step. Moreover, complex systems must be designed to be *configurable*. And one such criterion is modular structure, which entails the partitioning into packages. If a system is not designed in a modular manner, configuration is a nightmare anyway. One of the key tasks during the development of a configurator is to find packages suitable for automatic check and solve routines accepted by the user.

It is also an interesting problem to find methods for automatically partitioning a configuration network (described in a generic manner by the component catalogue) using graph theory. Yet, computationally optimal packages (if they can be found at all) need not coincide with packages expected by the user, thus leading to problems of understanding. On the other hand, the packages ought not to repeat the inefficiencies of existing guidelines carried out by hand.

## 3.2 Packages control the configuration process

After packages have been identified, they should form a graph which leads the user through the configuration process step by step. A package graph is a network where the nodes are the package identifiers. There is an arc between two packages, if there is at least one cross-package constraint connecting the packages. The package graph serves as action-plan. Each package represents an action to be done to carry out the full task. An example is presented in Figure 4.

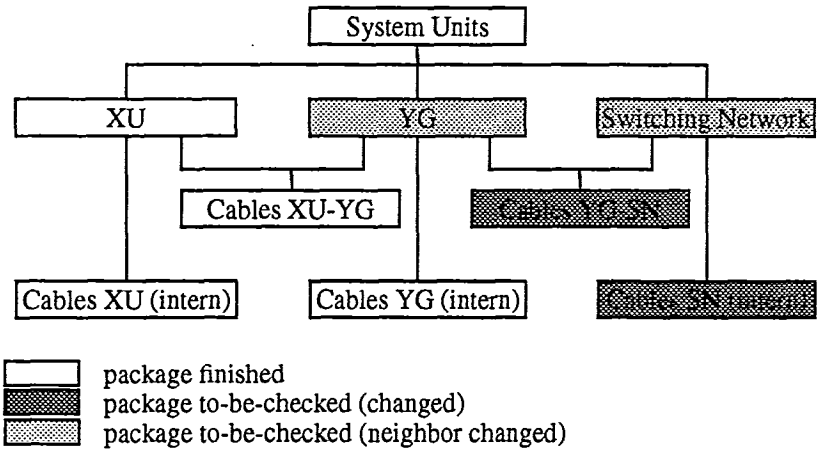



**FIGURE 4. Package Graph**

A given package is in one of the following states: finished, to be checked due to changes in the package itself, to be checked due to changes in a neighbor package. The state is presented in the user interface.

Packages need not be configured in a predefined sequence, yet this is often the most convenient and most efficient path through the package network.

Changes in one package can affect lots of other packages, thus requiring their re-check. Wide-range effects of changes are propagating relatively slowly through the package network.

# 4 Outcomes and conclusions

## 4.1 Advantages

If packages are well designed, they break down the problem into manageable subproblems.

Packages lead the user through the configuration process and present intermediate states of the solving process in a clear manner. If the packages match the user's concepts of the solving process, the user will have more trust in the results produced by the configurator. Users who have expert knowledge in the domain often want to be more involved in the solving process. Packages are one step forward to involve the user more closely in that.

The important task of maintaining a configuration is supported by packages. Local changes, i.e. changes in a single package, do not require the consideration of the whole knowledge base any more.

In case of restricted computer resources, e.g. small main memory, the constraint network may be generated just for the package currently dealt with by the constraint solver. After having finished and unloaded one package, the next one is loaded.

Many filtering, solving, and optimization methods are computationally expensive, so their applicability is restricted to problems of reasonable size. To break down the sizes of the portions opens the door for a wide range of constraint satisfaction techniques which would otherwise be awfully overcharged on the full, huge networks.

## 4.2 Further work

As packages are closely related to the design concepts of the system, criteria for a *design to be configurable* promise advantages for a simpler configuration process.

Methods for finding a good partitioning automatically, e.g. based on graph theory, could support the development of configurators.

Bringing together the users' needs (what suits them best for solving their tasks) while designing a working configuration system will be the greatest challenge in developing configurators.

# References


[1] R.Cunis, A.Günter, I.Syska, H.Bode, and H.Peters. *PLAKON - an approach to domain-independent construction.* In Proc. Intl. Conf. on Industrial and Engineering Applications of AI and Expert Systems (IEA/AIE-89), Tullahoma, Tenn., June 1989.

[2] Rina Dechter and Judea Pearl. *Tree clustering for constraint networks.* Artificial Intelligence, 38:353-366, 1989.

[3] Gerhard Fleischanderl, Gerhard Friedrich, Alois Haselböck, and Markus Stumptner. *Knowledge-based Configuration of Switching Systems.* In Proc. 15th Intl. Switching Symposium (ISS-95), pp.158-162, Berlin, April 1995.

[4] M.Heinrich and E.W.Jüngst. *A resource-based paradigm for the configuration of technical systems from modular components.* In Proc. IEEE Conf. on AI Applications (CAIA-91), pp.257-264, 1991.

[5] Alan K. Mackworth and Eugene C. Freuder. *The complexity of some polynomial network consistency algorithms for constraint satisfaction problems.* Artificial Intelligence, 25:65-74, 1985.

[6] Itay Meiri, Rina Dechter, and Judea Pearl. *Tree decomposition with applications to constraints processing.* In Proc. 8th National Conf. on AI (AAAI-90), pp.10-16, Boston, Mass., Jul./Aug. 1990.

[7] Sanjay Mittal and Felix Frayman. *Towards a generic model of configuration tasks.* In Proc. 11th Intl. Joint Conf. on AI (IJCAI-89), pp.1395-1401, Detroit, Mich., Aug. 1989.

[8] Markus Stumptner, Alois Haselböck, and Gerhard Friedrich. *COCOS - A Tool for Constraint-Based, Dynamic Configuration.* In Proc. IEEE Conf. on AI Applications (CAIA-94), pp.373-380, San Antonio, Tex., Mar. 1994.