Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1998), *12*, 383–397. Printed in the USA.

A classification and constraint-based framework for configuration

DANIEL MAILHARRO
ILOG S.A., 9 rue de Verdun, BP 85, 94253 Gentilly Cedex, France
(Received October 31, 1997; Accepted February 5, 1998)

Abstract

One of the main difficulties with configuration problem solving lies in the representation of the domain knowledge because many different aspects, such as taxonomy, topology, constraints, resource balancing, component generation, etc., have to be captured in a single model. This model must be expressive, declarative, and structured enough to be easy to maintain and to be easily used by many different kind of reasoning algorithms. This paper presents a new framework where a configuration problem is considered both as a classification problem and as a constraint satisfaction problem (CSP). Our approach deeply blends concepts from the CSP and object-oriented paradigms to adopt the strengths of both. We expose how we have integrated taxonomic reasoning in the constraint programming schema. We also introduce new constrained variables with nonfinite domains to deal with the fact that the set of components is previously unknown and is constructed during the search for solution. Our work strongly focuses on the representation and the structuring of the domain knowledge, because the most common drawback of previous works is the difficulty to maintain the knowledge base that is due to a lack of structure and expressiveness of the knowledge representation model. The main contribution of our work is to provide an object-oriented model completely integrated in the CSP schema, with inheritance and classification mechanisms, and with specific arc consistency algorithms.

Keywords: Configuration; Knowledge Representation; Constraint Satisfaction Problem; Classification; Taxonomic Reasoning; Nonfinite Domains; Resource Balancing

1. INTRODUCTION

1.1. Overview

A general definition of what we understand by "Configuration" has been given in Mittal and Frayman, 1989. The configuration of an artifact is a set of interconnected components that are chosen from predefined set of component types, called the *catalog* of component types. Each component type is defined by a set of *attributes* that describe its internal characteristics (price, geometric dimensions, resource production/consumption, etc.), a set of connection *ports* that describe its relations with other components, and a set of constraints on these attributes and ports. The constraints describe compatibility restrictions on component connections (Type A cannot be connected with type B), limitations on resource production/consumption, existential conditions (component of type A needs component of type B), etc.

Reprint requests to: Daniel Mailharro, ILOG S.A., 9 rue de Verdun, BP 85, 94253 Gentilly Cedex, France. E-mail: mailharo@ilog.fr

The problem is completely defined by the catalog of component types. During the search for solution, it is impossible to define a new type or to modify an existing type by adding a constraint, an attribute, or a connection port. This is the difference from the design task where such modifications are allowed (Mittal & Frayman, 1989).

Given the requirements that the desired product must satisfy and any specified optimizing criteria, the configuration tool must produce

- the list of the components that are integrated into the solution;
- the type of each of these components;
- the exact topology of the product, that is, the way the components are connected to each other; and
- the value of the attributes of each component.

The solution must answer the initial requirements (which can be considered as constraints) and must satisfy all the constraints defined on each assembled component. The solution must be optimal according to the optimizing criteria.

Configuration has interested industrial and research communities for 20 yr. Several modeling approaches have been proposed, such as expert system, resource-based paradigm, terminological logic, object-oriented model, and more recently constraint satisfaction problem (CSP). Each approach differs in the knowledge representation model and reasoning algorithms employed, but all agree that one of the main difficulties with configuration tasks is representing the domain knowledge. Configuration needs an expressive and declarative framework to represent different types of knowledge (Klein, 1996): taxonomies, structural, arithmetic and geometric constraints, *part-of* and *uses* relations between components, resource production/consumption, modularity of components, unknown *a priori* set of components, etc.

This knowledge representation aspect of configuration tasks is critical in industrial applications for maintenance and evolution reasons: catalogs of component types have high rates of modification. In the famous example of R1/XCON (McDermott, 1982; Barker & O'Connor, 1989), the configurator created at Digital Equipment Corporation to build computer systems, the catalog contains 17.000 rules and 30.000 component types, and 40% of it is updated each year.

A configuration framework must provide an expressive knowledge representation model that is easy to maintain and must also provide powerful and efficient algorithms to allow optimization in a highly combinatorial context. An important feature is the ability to describe different reasoning methods such as constraint propagation, hierarchical refinement, resource-based balancing, domain-dependent heuristics, etc.

1.2. Previous work

A good overview of what has been used to date to represent and to solve configuration problems is given in Haselbock, 1993.

The rule-based approach is the more commonly used paradigm. However, it seems to be efficient for small problem domains only. The main drawback of this approach is the maintenance, consistency, and evolutability of the knowledge base. The problem is that the knowledge is not well structured:

- Rules can mix domain knowledge with solving strategies.
- Knowledge of a component type is split among several rules.
- Rules are more representative of reasoning steps, that is a transition of state, than a declaration of what is a component or what must be the configured artifact.
- A set of rules can be inconsistent.

The resource-based paradigm (Heinrich & Jungst, 1991) provides an intuitive producer/consumer model but does not account well for the topology of the configured artifact. For example, a rack produces a resource that is its plugging slots. Each card plugged in this rack consumes one or more slots. This kind of knowledge is captured well by resource modeling, but topological constraints as "*cards of type A can only be plugged in slots 1 to 8*" or "*if a card of type A is plugged in slot 1, then two cards of type B must be plugged in slots 2 and 3*" cannot be represented. In real-world configuration problems, there are topological constraints as well as resource balancing constraints. So a framework dedicated to configuration problems must be able to take into account both of the constraint types.

Another interesting paradigm is the object-oriented approach where a configuration problem is considered as a *Classification* problem. Given a generic object and a hierarchy of classes that describes the problem domain, the goal is to refine the generic object until its correct class in the hierarchy is found. This inference process is called *taxonomic reasoning*. For example, when configuring a computer, we know that we need a processor but we do not know exactly which concrete type will be chosen in the final computer. Among the existing applications of this paradigm are the *Concepts* language KL-ONE (Brachman & Schmolze, 1985), the FREEDOM system (Yokoyama, 1990) which defines relations between objects as *consist-of* and *contains* relations, and PLAKON (Cunis et al., 1989) which defines constraints on objects.

In the configuration domain, representing the components catalog as a class hierarchy is quite natural. The object-oriented paradigm strongly improves the maintenance task with its specialization and encapsulation concepts. Subtle concepts as *shared* objects and *part-of* links can easily be represented with special classes of object relations. We believe this paradigm is adequate to represent the domain knowledge required for configuration tasks, and in Section 3 we explain how we have introduced taxonomic reasoning into our constraint-based framework.

The last paradigm we want to talk about is the CSP one. The principle is to "find values for problem variables subject to constraints that restrict which combinations of values are allowed" (Sabin & Freuder, 1996). CSP is an adequate framework to represent configuration problems because it is "highly declarative, domain independent and simple to use". A configuration problem can be considered as a constraint satisfaction problem where the variables are the type, ports, and attributes of each component used in the configured artifact, and the constraints are the initial requirements and the constraints defined in the catalog for each component. Because a configuration task is a *generative process* where the set of the components used in the final solution is not known beforehand and is generated during search, the basic CSP paradigm cannot be applied directly. The port variables' domain is not complete when creating the variable because new possible components are generated during search and the constraint network changes each time a component is generated.

Mittal and Falkenheimer (1990) defines the *Dynamic CSP* (DCSP) as an extension of the basic paradigm that says that only a subset of the initial variables may be part of the final

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

solution. A special kind of constraints called *Activity Constraints* represent deductions that decide if a variable belongs to the solution or not. This model is not adequate because a maximal set of existing variables must be given as the specification of the problem. In configuration problems, providing the maximal set of possible components may be impossible or may produce a huge set that makes the search space impracticable.

Haselbock extends the DCSP model in his ConfCS framework (Haselbock, 1993) to handle an unlimited number of variables. He uses a generic definition of the domain. Constraints are defined at a meta level between component types. Meta variables are used to represent any concrete instance of a component type. The constraint is consistent when all possible substitutions of the meta variables with existing instances satisfy the constraint. Because the number of instances of a type is unlimited, the number of variables of the constraint network and the domains of port variables are infinite.

Special constraints called "resource-constraint" are defined to represent accumulative relations on a part of the configuration (all instances of a given type or all components connected to a given component). These constraints have the unique ability to be defined on a set of variables that is not known beforehand and to be able to generate new components to ensure consistency. Because configuration problems are constructive problems (parts of the artifact are generated during search), this kind of constraints on "*unknown world*" is very useful for expressing knowledge about what the desired product should be without knowing exactly the number or type of the components that it will compose. For example, an instrumentation and control hardware architecture is composed of a set of programmable logic controllers (PLC). We do not know how many PLCs are needed but we know that the set of PLCs has to provide a certain amount of computational power and cannot consume more than a certain amount of electrical power.

ConfCS takes into account the dynamic and the constructive aspects of configuration problems well. Its principle lack is that its knowledge representation model is not well structured; there is no taxonomy of the component types and no way to differentiate "part-of" from "uses" relations between components. It does not facilitate maintenance because knowledge about what is a component is split: activity constraints describe the internal structure of a component type and compatibility constraints describe the relations between its attributes and ports.

Another important drawback of the Haselbock model is for representing $1-N$ relations. When a component a may be connected to N components b_i, we need to define N ports in a: *a.p1*, *a.p2*,…, *a.pN*. When the ports are interchangeable (if S is an assignment of the N ports that is a solution, any permutation of the values of the ports remains a solution), this model artificially increases the search space by a factor factorial N ($N!$). Unfortunately, in real-world applications, ports having the same role are often interchangeable.

Furthermore, this model is not versatile enough to represent variable cardinality. It is impossible to represent a cardinality that is computed by an arithmetic formula, or to represent a cardinality that is not limited.

In our approach, we eliminate these two problems by representing an 1-N relation by one constrained set variable with cardinality limited to N instead of N ports. The cardinality of the set is an integer constrained variable that can be computed by a constrained formula. There is no order between the values of the set, so the $N!$ permutations of an assignment correspond to a single solution: The search space is divided by $N!$ compared to the ConfCS approach.

Sabin and Freuder (1996) propose another extension of the basic CSP paradigm, the composite CSP, which focuses on the hierarchical structure of the domain knowledge. The domain of a constrained variable is a set of entire subproblems with their own variables and constraints. When instantiating a variable with one of the possible subproblems, the variables and the constraints of the subproblem are added to the constraint network. This approach is able to handle a nonlimited set of components, and allows an elegant representation of the recursive decomposition of a problem into subproblems. However, it is not clear how consistency can be maintained on such variables. How can the knowledge about the subproblems that belong to the domain of a variable be used to deduce which of them are inconsistent according to the current state of the variables? In the Sabin and Freuder (1996) example, we are configuring a car. The component *engine* has two possible types: *gasoline* and *diesel*. One characteristic of a *gasoline* engine is that it consumes 2 fuel units, against 4 fuel units for the *diesel* engine. If our car has to consume less than 3 fuel units, the diesel type may be removed by an arc consistency propagation.

It is not clear also how constraints on an "unknown world", as the resource-constraints of ConfCS, can be expressed.

1.3. Our purpose

Our general aim is to propose a new framework, dedicated to the solving of configuration problems, that strongly focuses on domain knowledge representation (because most of the complexity lies in its capture) and that allows the description of efficient solving algorithms.

Because we consider a configuration problem both as a classification and as a constraint satisfaction problem, we have developed a framework that deeply blends concepts from the object-oriented and CSP paradigms. The goal is to provide powerful and expressive tools that can capture the different aspects that characterize configuration problems:

- The hierarchical structure of the domain knowledge.
- The critical aspect of maintenance and evolution of the components catalog.
- The generative process (dynamic constraint network).
- The previously unknown set of possible components.

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

- The great variety of constraints (topological, arithmetic, logical, compatibility, resource balancing, etc.).
- The different types of relations between components (has-parts, uses).

Encapsulation and specialization concepts, hierarchical domain CSP (Mackworth et al., 1985), and taxonomic reasoning are well suited to structure the domain knowledge in an inheritance tree of component types. Maintainability is improved by the using concepts of the object paradigm, such as inheritance or data abstraction, by CSP declarativity and expressiveness, and by the clear separation between domain description and solving strategies that the CSP paradigm allows. Class instantiation is quite natural for representing the generative process and the dynamic nature of the constraint network. CSP is well suited for declaring constraints of various types and for describing optimization algorithms and domain-dependent heuristics.

This paper presents the principal features of the knowledge representation model of our framework and provides a simple example extracted from a real-world configuration application on which our approach has been applied. Section 2 presents the example that will illustrate the further sections. Section 3 presents the hierarchical structure of the domain knowledge representation and the taxonomic reasoning implemented as constraint propagations. Section 4 presents ports as constraint set variables with nonfinite domains to capture the "not known beforehand" set of components. Section 5 describes how resource balancing constraints are represented. Section 6 details a solution search on the example of Section 2. Section 7 presents some informal results of a nontrivial application developed with our framework. A conclusion in Section 8 summarizes the paper and outlines future works.

Our framework is based upon the well-known CSP solver library Ilog-Solver (ILOG, 1997). The code examples presented in further sections use the C++ notation.

2. EXAMPLE

The problem is the configuration of an instrumentation and control system of an electrical power plant. Such a system is composed of a set of programmable logic controllers (PLCs). Each PLC is composed of a set of racks into which are plugged processor cards and input/output cards (I/O cards). A processor provides a certain amount of computation power and memory units. An I/O card provides a certain number of connections to sensors and actuators. The specification of the desired system is given by a set of *functions*. Each function specifies the quantities of computation power and memory units and the number of analogic, numeric, and communication connections it needs to be executed. The task of the configurator is to assign the set of functions to a set of processors and I/O cards, and to plug these cards into PLC racks. The goal is to minimize the number of PLCs needed. There are different types of PLCs, racks, processors and I/O cards, and compatibility constraints at each connection level. Each hardware component produces or consumes resources such as electrical intensity, calorific power, plugging slots, computation power, memory, etc. Of course all of these resources have to be balanced.

For the sake of simplicity, we will only focus on an atomic part of the problem: the assignment of functions to processors (see Figure 1). There are three types of functions: FNC, the noncritical functions which survey and command sensors and actuators; FC, the critical ones; and FCOM which are in charge of inter-PLC communication. There are three

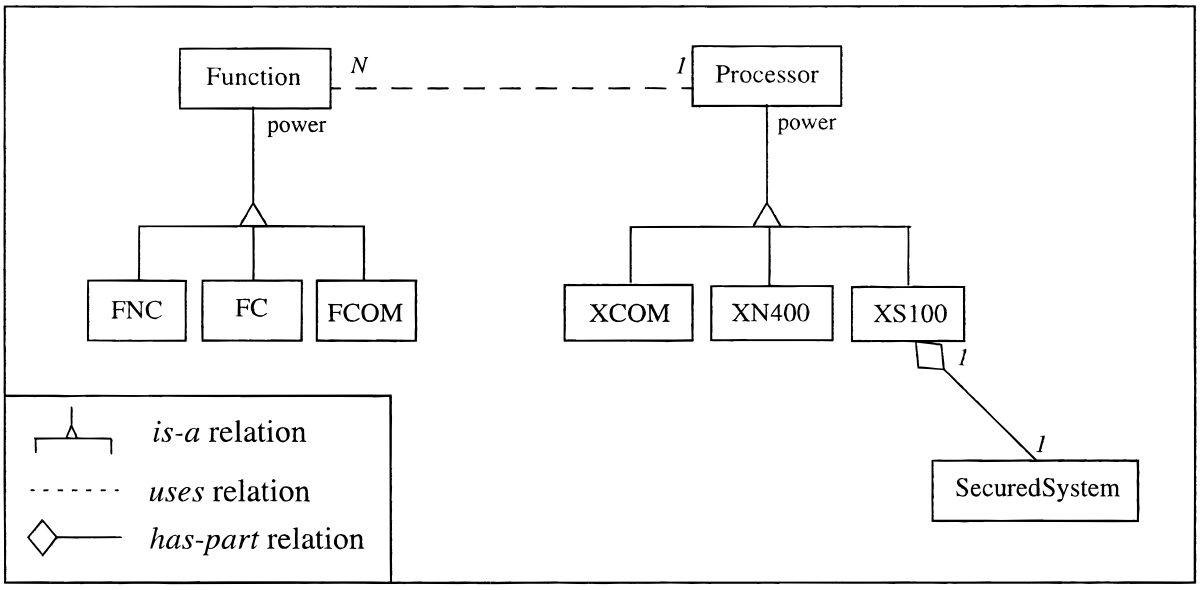

Fig. 1. Example.

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

types of processors: XCOM, processors dedicated to communication tasks; XS100, processors with secured hardware for critical task execution; and XN400, processors with nonsecured hardware. A XS100 processor is composed of one secured system component.

A function is assigned to one processor but one processor can contain several functions. Functions consume the computation power that is produced by processors.

A Function can only consume the computation power that is produced by the processor to which it is assigned:

$$\forall p \in Processor, \sum_{f \in functions_p} power_f \leq power_p.$$

The global computation power production must be greater than or equal to the global consumption:

$$\sum_{f \in Function} power_f \leq \sum_{p \in Processor} power_p$$

XCOM, XN400, and XS100 produce, respectively, 200, 400, and 100 units of computation power. The maximal number of functions that can be assigned to an XS100 processor is limited to 3.

The following compatibility rules are:

- XCOM only support FCOM functions.
- XN400 support FCOM and FNC functions.
- XS100 support FCOM and FC functions.

There are several instances of each function type. Each of these instances specifies its power consumption.

3. CLASSIFICATION AND TAXONOMIC REASONING

3.1. Domain knowledge representation

We have adopted the *component-oriented* model presented in Mittal and Frayman (1989). A component type is described by a set of attributes and connection ports and by a set of constraints. Attributes are implemented with classical constrained variables. Ports are constrained set variables whose domain corresponds to a nonfinite set of components (see Section 4 for more details). Constraints are those available in Ilog-Solver (ILOG, 1997), plus specific expressions defined on ports to represent accumulative relations on incomplete sets of components. These expressions are similar to the "resource-constraints" of ConfCS (Haselbock, 1993) (see Section 5 for more details).

Let us give an example of a component type declaration: A processor has an integer attribute "power" that represents the amount of computation power it provides (from 0 to 1000), a port "functions" that represents the set of functions that are assigned to it, and a local resource balancing constraint on computation power:

```
//Objects that represent the component types
IlcComponentType Processor, Function;

//Declaration of the computation power
attribute of Processor
IlcIntAttr power = Processor.addIntAttr-
("power", 0, 1000);

//Declaration of the functions port of Pro-
cessor
IlcPort functions = Processor.addPort(Ilc-
Uses, Function, "functions");

//Power consumed by the functions connected
to a processor is
//less or equal to the power produced by
this processor
Processor.add(functions.getSum("power") <=
power);
```

The component types taxonomy is organized in a generalization/specialization hierarchy. Each node of the tree describes a type with a set of attributes, ports, and constraints that are inherited by subtypes.

Subtypes can complete their ancestor description with new attributes, ports, and constraints. They can also define more restrictive bounds for the domain of inherited attributes or for the cardinality of inherited ports. For example, XS100 is a processor that produces 100 units of computation power, that can support at most three functions and that is composed of one secured system component:

```
IlcConfigurator cfg;

//Declaration of the type XS100 as a sub-
type of Processor
IlcComponentType XS100 = cfg.addType ("XS100",
"Processor");

// Bounds restrictions on inherited attributes
and ports
XS100.getIntAttr("power").setValue(100);
XS100.getPort("functions").setCardMax(3);

//XS100 is composed of exactly one Secured
system
XS100.addPort(IlcHasPart, SecuredSystem, 1,
"secured_system");
```

3.2. Instantiation and refinement

The goal of the configuration task is to generate the set of components that compose the configured artifact, to put each component at the right place in the types hierarchy, and to inter-connect these components.

Component generation is supported by type instantiation. A component is created with an associated copy of all the attributes, the ports, and the constraints defined in the instantiated type and with a special constrained variable that represents its type. When the instantiated type is not a leaf

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

of the type hierarchy, the component may be *refined* later by a constraint propagation or by an explicit choice point of the search procedure. The refinement of a component corresponds to a more accurate knowledge of what can or what must be the final implementation of the component. Thus, when a component is refined, it becomes an instance of one of the descendant types of its current type, and so all the attributes, ports, and constraints defined for its new type (except those inherited from its current type) are added to the current constraint network.

In our previous example, if we have the instance *aProcessor* that is refined from processor to XS100, restrictions on its "`power`" and "`functions`" variables are performed and a "`secured_system`" port is added to it, as illustrated in Figure 2.

The refinement procedure is powerful because it allows reasoning on a partial description of a component. Inferences can be made on a component without knowing exactly what its concrete implementation in the final artifact will be. Propagations can be performed on the possible types of a component to decide which types remain possible or become required after an event occurs on any constrained variable domain. For example, we know that our configured artifact needs a processor but we do not know which kind of processor. So, we can create an instance of the generic type processor, and begin to work on it with the partial knowledge we have of it, for example, the computation power it provides must be greater than or equal to the sum of the computation power consumed by the functions that are assigned to it. Then, when assigning functions to it, deductions on its possible types are activated. For example, if the quantity of consumed power becomes greater than 100 units, XS100 becomes inconsistent. Or if an FC function is assigned to it, compatibility constraints propagate that only XS100 is consistent and the processor is then refined to XS100.

This mechanism gives great flexibility for writing search algorithms because choosing the type of a component can be separated from the component generation. There are three important operations on a component: its creation, its classification, and its connection with other components. Creation is always the first operation, but there is no mandatory order for the two others. A component can be connected before being classified or can be classified before being connected. It is also possible to interlace the two operations: we can decide to connect some of its ports, then to refine it until a certain level, then to connect some others of its ports, then to refine it again, etc. It is also possible to delay one operation by making choice points on other parts of the configured artifact, hoping that these choices will make interesting propagations on our component.

It is interesting to note that the refinement procedure is perfectly integrated in the CSP schema because it has a monotonic behavior. The set of the possible refinements for a given type is a subset of the possible refinements of its ancestor types. So refining a component consists in reducing the set of its possible classifications, that is, refinement can only reduce the domain of the constrained type variable of a component.

3.3. Type variables

Each component has a constrained variable that represents its type. We call it the *type variable* of the component. This variable has a hierarchical domain as defined in Mackworth et al. (1985). The variable is bound only when it is impossible to refine the component more. In other words, the variable can only be instantiated with leaf types of the hierarchy. This corresponds to the fact that the configured artifact must finally be built with concrete component types available in a constructor catalog. In our example, it is not sufficient to say that the system contains a processor; the configurator must decide exactly if it is a XS100, a XN400, or an XCOM.

Inferences made on the type variable are as follows:

1. When a type is inconsistent, then so are all its descendents (see Figure 3b).

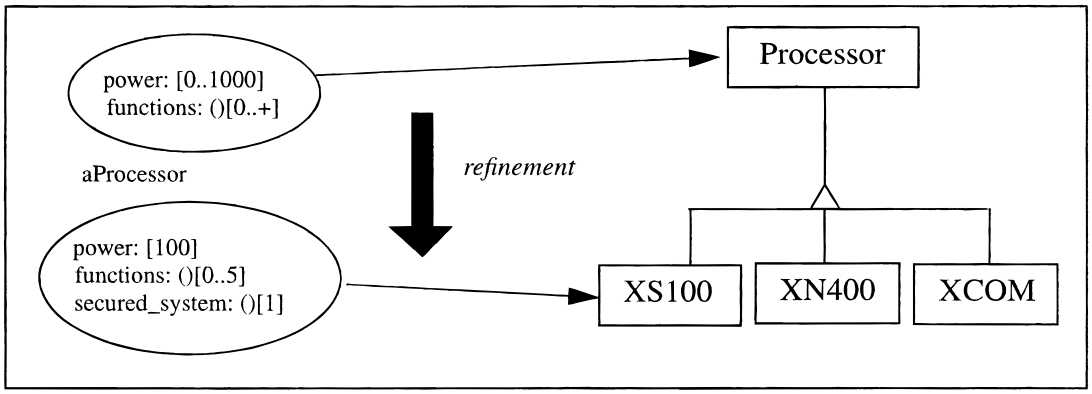

Fig. 2. Refinement procedure: [0..1000] denotes a constrained integer variable whose value is between 0 and 1000. [0..+] denotes a constrained integer variable whose maximal value is the infinity. ( ) [0..5] denotes a constrained set variable whose cardinality is between 0 and 5.

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

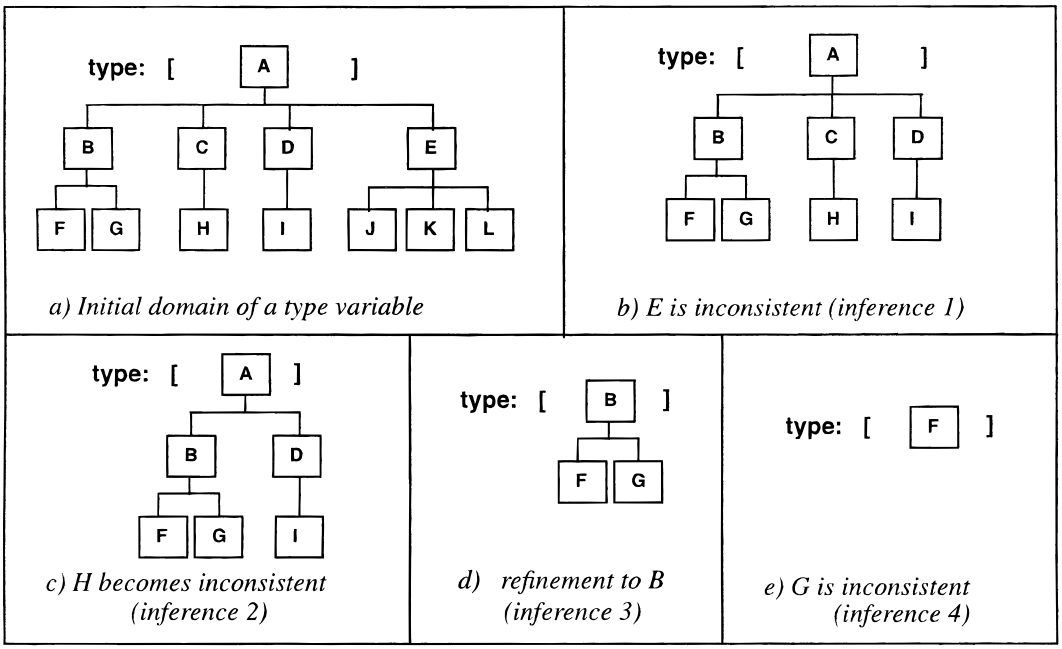

Fig. 3. Inferences on type variables.

2. When a type is inconsistent and when it was the sole descendent of its father, then its father is also inconsistent (see Figure 3c).
3. When the component is refined to a given type, then only the ancestors and the descendents of that type remain consistent (see Figure 3d).
4. When a type becomes inconsistent and when only one possible descendent for the current type of the component remains, then the component is refined to that descendent type (see Figure 3e).

Because of the subsumption semantic that exist between a leaf type and its ancestors, we can easily imagine that the value of a type variable is an implicit set of component types that correspond to the oriented simple path from a root type to a leaf type along the *subsumes* links. For example, if a component is refined to the type XS100, the value of its type variable is {Processor, XS100} since an XS100 instance is also an instance of Processor. We can deduce the two following properties:

- *A possible path contains only possible component types.*
- *A possible type belongs to at least one possible path.*

The soundness of previous inferences can be easily showed with this properties. From the first property, we can say that if a type becomes impossible, all the paths that include it becomes impossible values for the type variable (inference 1). From the second property, we can deduce that when a path becomes impossible, each of its nodes that no more belongs to a possible path, becomes impossible (inference 2). When a component is refined to a component type, only the path that includes this type remain possible (inference 3). If a type T belongs to all the possible paths, the component can be refined to this (inference 4).

a. The initial possible paths are:

({A-B-F} {A-B-G} {A-C-H} {A-D-I} {A-E-J} {A-E-K} {A-E-L})

A belongs to all the possible paths, so the component is refined to *A*.

b. *E* becomes impossible. All the paths that contain *E* are removed:

({A-B-F} {A-B-G} {A-C-H} {A-D-I})

J, *K*, *L* do not belong to a possible path: they become impossible types.

c. *H* becomes impossible. All the paths that contain *H* are removed:

({A-B-F} {A-B-G} {A-D-I})

C does not belong to a possible path: it becomes an impossible type for the component.

d. Only paths containing *B* remain possible:

({A-B-F} {A-B-G})

e. *G* becomes impossible. All the paths that contain *G* are removed:

({A-B-F})

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

The component is refined to F because F belongs to all the possible paths.

4. TOPOLOGY: NONFINITE DOMAIN FOR PORT VARIABLES

4.1. Motivation

The topology of the configured artifact, that is the description of the interconnections of its components, is completely represented by port constrained variables assignment. In our framework, a port is typed, that is, only components of a given type can be connected to it. Connections between components are always constrained by such type restrictions; only cards can be plugged into racks. Because a type defines a subset of the entire existing components set, typing a port reduces its domain size, and thus, reduces the size of the search space.

In configuration problems the set of the components is not known beforehand, and sometimes even the size of the set cannot be estimated. The problem with classical constrained variables is that the set of the possible values must be known before creating the variable, so it is impossible to define the domain of port variables.

However, we wanted to have port variables that would be able to represent partial connections and to post constraints that could generate new components to insure consistency. Even if the components do not exist yet, we wanted to express constraints such as "The sum of the size of the cards that are plugged into a rack cannot exceed the capacity of the rack". We wanted to make *generative inferences* that create a new component each time there is no existing component that can satisfy the constraint. For example, we know that a processor produces at most 1000 units of computation power, and we know that 4000 units are required to execute a set of functions. We can easily deduce that at least four processors are required, and so we can create them.

To represent such knowledge, we have defined a new type of variable whose domain contains the components that already exist and a representation in intension of the ones that will be created later. Ports are such variables.

4.2. Constrained set variables

A port is implemented as a constrained set variable (Puget, 1992). It is a multivalued constrained variable whose value is a set of components and whose domain is the set of the possible component sets that can be assigned to the variable. In practice, the domain of a constrained set variable is represented by two sets: the *possible set* that contains all the elements that *can* belong to the value of the variable and the *required set* that contains all the elements that *do* belong to the value of the variable. When the variable is instantiated, the required set is equal to the possible set.

A constrained set variable is systematically associated with an integer constrained variable that represents its cardinality. Consistency between these two variables is maintained by the following relation:

$$|Required(V)| \leq Min(Card(V)) \leq Max(Car(V)) \leq |Possible(V)|,$$

where $|S|$ denotes the number of elements of the set S; V, the constrained set variable; $Required(V)$, its required set; $Possible(V)$, its possible set; and $Card(V)$, its associated cardinality variable.

Multivalued ports are very useful in configuration problem description because most of the intercomponent relations are $1-N$ or $N-M$ relations and cardinalities of these relations are often not fixed. For example, a set of cards is plugged into a rack. The size of this set is not fixed, but it can be limited by the physical size of the rack. An example of the $N-M$ relation is that functions are dispatched on a set of I/O cards and these cards are shared between several functions.

When N components of type A have to be connected to one among M components of type B, we use M set variables that have N possible values; the space complexity to represent the possible assignments is $N \times M$. Set variable does not induce a bigger space complexity than other type of constraint variable. Whatever is the type of constrained variable used, this memory complexity is unavoidable since the domain of a constraint variable has to be represented explicitly. If we use Boolean variables to represent the fact that a component A is connected to a component B, we need $N \times M$ such Boolean variables. If we use simple enumerated variables, we need N variables which contain M possible values. Fortunately, Ilog-Solver provides efficient internal data structures based upon bit-sets that allow low memory consumption and fast data access. Ilog-Solver set variables have been successfully used to model large-scale applications. For instance, an airline crew scheduling application at Air France had to form 3000 crews with a pool of 2000 people.

Set variables have the ability to reduce the search space by cutting symmetries. For instance, suppose we have to connect three cards a, b, c into a rack. If we use a model with three variables $p1$, $p2$, $p3$ that represent the three connections, and if $(p1 = a, p2 = b, p3 = c)$ is a solution, then any permutation of a, b, c leads to the same solution. The second model consists of using one set variable with a cardinality limited to 3. Because there is no order into a set variable, there is only one possible assignment of the variable that is $(a\,b\,c)$. Using constrained set variables instead of single variables induces a drastic reduction of execution time by dividing the search space size by a factor $N!$, with N representing the number of components to connect.

4.3. Port variables

The possible set of a port variable contains all the components that already exist for the type of the port, and it also

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

contains a particular value, called *wildcard*, that represents the set of all the components that have not been generated yet. The required set, that represents the effective connections to the port, contains only existing components, that is, it never contains *wildcard*.

While *wildcard* is in the domain, new components can be added to it. When *wildcard* is removed from the domain by a constraint propagation or by a choice point, the port is *closed* and new components are ignored.

While the port is not closed, the upper bound of the cardinality variable is $+\infty$ because an unlimited number of components can be generated. Of course, it is possible to define another maximal value by posting an explicit constraint on the cardinality. The lower bound of the cardinality corresponds to the size of the required set as for a classical constrained set variable.

Each time a type is instantiated, all the ports dedicated to this type, except the closed ports, have their domain increased with the new component but the maximal value of their cardinality remains unchanged. When all the instances of a type are known, the type is closed and so are all the ports dedicated to it.

Figure 4 shows an example where three instances of the type `Function`, `f1`, `f2`, and `f3` have a port "`process`" on the type `Processor`. There are two existing instances of `Processor`, `p1` and `p2`. When a new processor `p3` is created, `f2` and `f3` have the domain of their port `process` increased with `p3`. `f1` is not updated because its port is closed.

Domain extension of a port variable is well integrated in the CSP paradigm because it has a monotonic behavior. Inserting a new component into the domain of a port does not change the domain at all. The new component was already taken into account implicitly *via wildcard* before it became concrete. It was also integrated *via* the cardinality $+\infty$ upper bound. A component that is removed from a domain never comes back (except when back-tracking) because *wildcard* expansion never generates the same component instance twice.

4.4. On demand components generation

A port can activate type instantiations when there are not enough existing components in its domain to insure consistency with the minimum value of its cardinality variable. This is what we call the *on-demand inference*. For example, if we add to our previous example a constraint on the minimal cardinality of the port of `f2`, which says it is greater than or equal to 4, two new processors `p4` and `p5` are generated because there are only two existing processors in the domain of `f2` port. Figure 5 illustrates this.

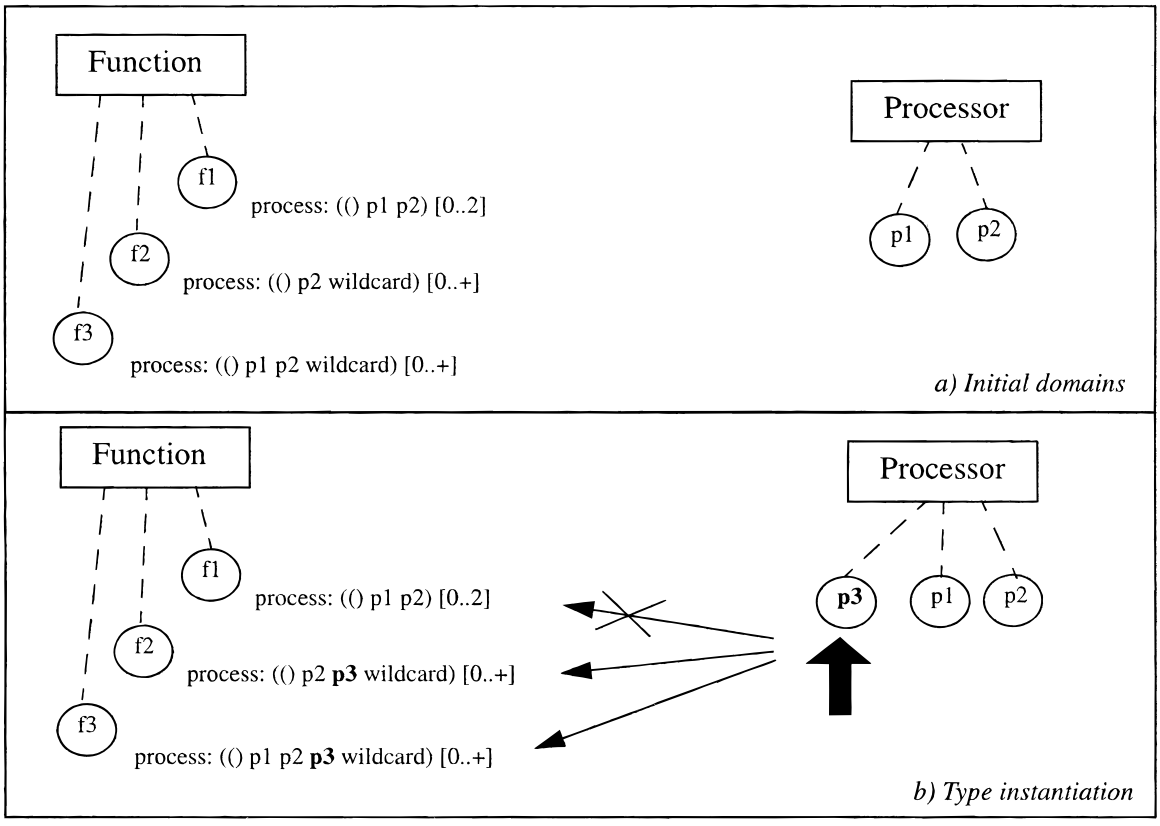

Fig. 4. Increasing domain of port variables (see Figure 2 for notation conventions).

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

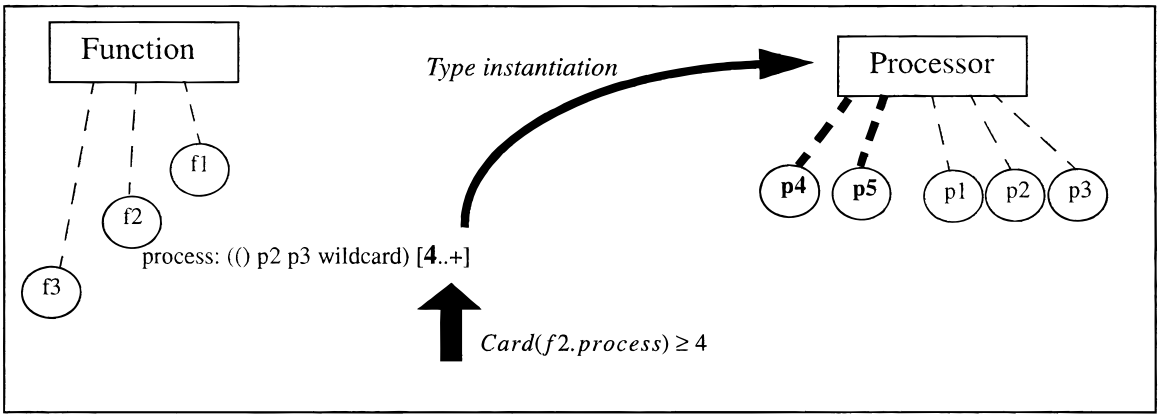

Fig. 5. On-demand inference (see Figure 2 for notation conventions).

It is possible to make a choice point on a port variable that will try to assign the *wildcard* value. In that event, a new component is created and tried in *wildcard*'s place. If the choice leads to an inconsistency, *wildcard* is removed from the possible set of the port. Because all components of a same type are considered equivalent at creation time, another component generation would just lead to a similar inconsistency.

4.5. Has-part relation

Ports represent relations between components. Specific relations as *has-part* relations are represented by specific ports. *Has-part* relations describe an ownership relation between two components where one (the part) exists only inside of the other (the owner). Parts of a component are not shared; they belong to their owner component. When a component is created, its parts are also created. This behavior is easily implemented with a specific on-demand inference that directly assigns the components it creates to the port, instead of just making the possible set grow. We call this *Has-part inference*. In the example of Section 2, secured system components are part of XS100 processors. So each time an XS100 processor is created, or each time a processor is refined to XS100, one secured system is created and assigned to the processor.

Part components are not shared by several composite objects inside the scope of a particular *Has-part* relation, but they can be shared through several different *Part-of* relations. For instance, when configuring a computer, if we say that a processor is a part-of the computer, the processor instance *p1* cannot be shared by several computers; *p1* is a part of one and only one computer. Now if we say that the processor is part-of the computer and is also part-of the mother-board, then the instance *p1* is shared by an instance of computer and an instance of mother-board.

Figure 6 illustrates a sequence of a component generation due to a choice-point (Figure 6a), a component refinement due to a compatibility constraint propagation (Figure 6b) and another component generation due to a Has-part inference (Figure 6c).

5. RESOURCE CONSTRAINTS

Heinrich and Jungst (1991) describe a component as a producer and/or consumer of resources. In a configuration solution, resources have to be globally balanced, that is, the total amount of produced resource must be greater or equal to the total amount of consumed resource.

However, a configured artifact is a structured set of components where each component produces resources to answer the demand of the components that are directly connected to it. For example, the computer power resource produced by a processor is directly consumed by the functions that are executed on it. Resources have to be balanced locally for each component to insure that the component produces enough resources to satisfy its connected components needs.

In practice, the resource production/consumption of a component *C* is represented by one of its attributes, which we denote `res` in the following. This value depends on the amount of resource consumed or produced by the components that are connected to *C*. For example, the electrical intensity consumed by a programmable logic controller is the sum of its rack consumption plus its own consumption. The rack consumption is the sum of the consumption of the cards plugged in it, etc.

To represent such accumulative functions on the connected components of a given component, we have defined accumulative expressions on port variables. As an example, we will detail the most useful expression that is the sum of `res` on the components connected to a port. The computation power consumed by a processor corresponds to the sum of its assigned functions consumption:

```
IlcIntVar sum = Processor.getPort("functions").
getSum("power");
```

The minimum value of `sum` is the sum of `res` on the components that belong to the required set of the port. The

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

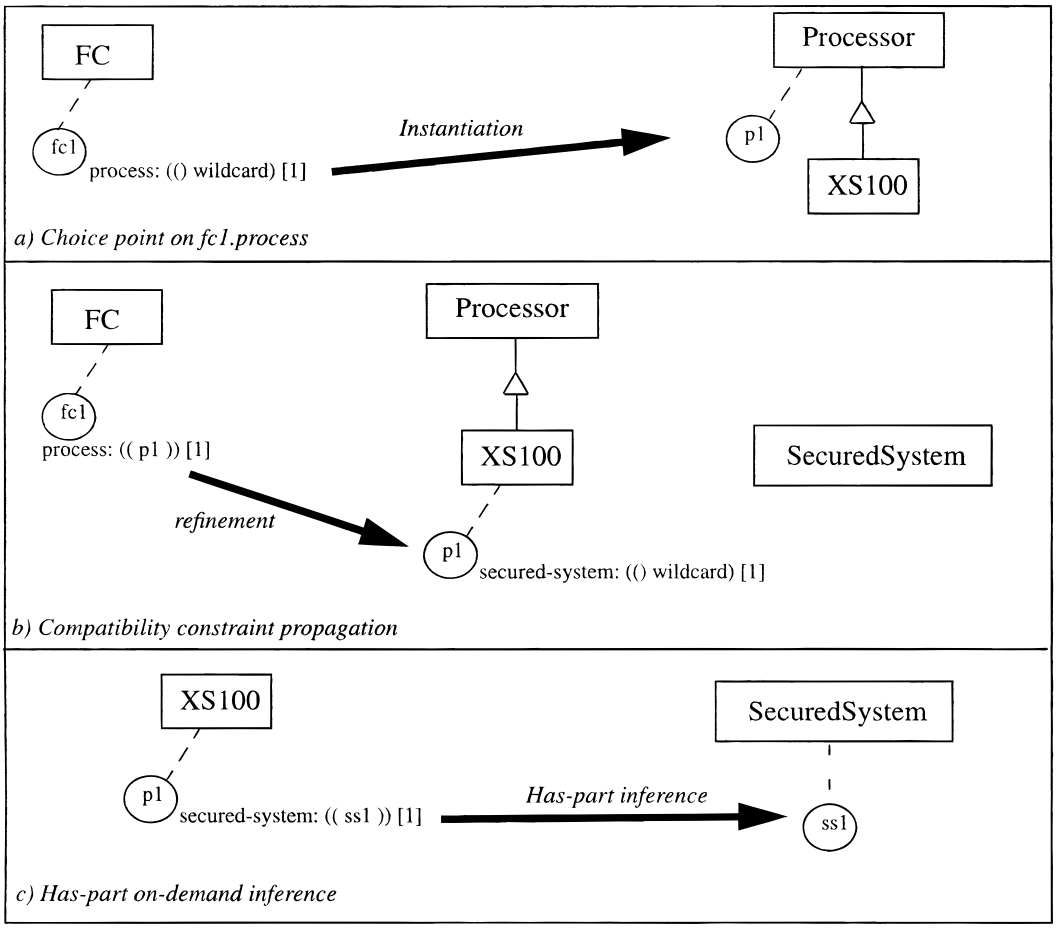

Fig. 6. Choice point, refinement and Has-part inference.

maximum value is the sum of `res` on the components that belong to the possible set:

$$\begin{cases} Min(sum) = \sum_{e \in Required(Port)} e.\text{res} \\ Max(sum) = \sum_{e \in Possible(Port)} e.\text{res} \end{cases}$$

Consistency between the port, its cardinality, the `sum` and the `res` variables, is insured by a set of propagations. We give several of them as an example:

1. Each time a component C is connected to the port, the minimum value of the sum is increased with the minimum value of the C attribute `res`.
2. Each time a component C is removed from the possible set of the port, if the port is closed, the maximum value of the sum is decreased by the maximum value of the C attribute `res`.
3. When bounds of the `res` variable of a component C are changed, minimum or maximum value of `sum` can be recomputed depending on whether C belongs to the required or to the possible set.
4. When the maximal value of the sum is decreased, the possible elements (including wildcard) that have a too big value for their attribute `res` are removed from the domain of the port.
5. When the minimal value of the sum is increased, new components are generated if the sum of the possible element's attribute `res` is not sufficient.
6. When the bounds of the cardinality variable are changed, bounds of the sum can be computed using the definition bounds of the attribute `res` (denoted $Min_I(res)$ and $Max_I(res)$):

$$Min(Card(Port)) \times Min_I(res) \leq sum \leq Max(Card(Port)) \times Max_I(res).$$

Let us illustrate the previous propagations by a simple example. Suppose we have a rack that has a port E in which are connected a set of cards. Suppose that each card produces from 10 to 100 (denoted *[10..100]*) units of a given resource R. The value of the resource (denoted r_i) produced by the ith card depends on the functions that are assigned to it. This value is computed during the search by constraint

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

propagations. Suppose now that we want to compute the sum S of R produced by the cards of the rack. Suppose that at the beginning, there are two existing cards *c1* and *c2* that produce, respectively, *[30..50]* and *[90..100]* units of R. Table 1 shows how the consistency is maintained between the constrained variables E, *Card(E)*, S, and r_i.

Local resource balancing is represented by a simple constraint on a component:

```
// computation power production of a processor
IlcIntAttr prodPower = Processor.getIntAttr("power");

// Processor port where power consumers are connected
IlcPort func = Processor.getPort("functions");

// Local resource balancing constraint
Processor.add(func.getSum("power") <= prodPower);
```

Each component type has a special port that contains all its instances. Global resource balancing can be expressed on these port variables:

```
IlcConfigurator cfg;
IlcComponentType Processor, Function;

// F represents the set of the instances of Function
IlcPort F = Function.getInstancesPort();
```

Table 1. *Resource constraint example*

Event	Propogations	S	$Card(E)$	$Required(E)$	$Possible(E)$
Initial state	There is no required card so the minimum value of S is 0. There is no limitation on the cardinality of E, so the maximum value of S is $+\infty$.	*[0..+]*	*[0..+]*		*c1:[30..50]* *c2:[90..100]* Wildcard
$r_1 \in E$	*c1* produces at least 30, so the minimum value of S is increased with 30 (Propagation 1).	*[30..+]*	*[1..+]*	*c1:[30..50]*	*c2:[90..100]* Wildcard
$Card(E) \leq 4$	There is at most four required cards that produce 100 each. S is at most 400 (Propagation 6).	*[30..400]*	*[1..4]*	*c1:[30..50]*	*c2:[90..100]* Wildcard
$S \leq 100$	*c2* is removed because it produces at least 90, that added to the 30 of the already required card *c1*, makes the minimum value of S greater than 100. (Propagation 4).	*[30.100]*	*[1..4]*	*c1:[30..50]*	Wildcard
$r_1 \geq 40$	Because *c1* is required, the minimum value of S is increased. (Propagation 3).	*[40..100]*	*[1..4]*	*c1:[40..50]*	Wildcard
$S \geq 70$	The maximum amount of resource produced by the existing card is 50. A new card is needed to satisfy the demand; *c3* is generated (Propagation 5).	*[70..100]*	*[1..4]*	*c1:[40..50]*	*c3:[10..100]* Wildcard
$r_3 = 80$	*c3* is removed because it produces 80, that added to the 30 of the already required card *c1*, makes the minimum value of S greater than 100. (Propagation 4). The maximum amount of resource produced by the existing card is 50. A new card is needed to satisfy the demand; *c4* is generated (Propagation 5).	*[70..100]*	*[1..4]*	*c1:[40..50]*	*c4:[10..100]* Wildcard
$r_4 = 10$	The maximum amount of resource produced by the existing cards is 60 (50 by *c1* and 10 by *c4*). A new card is needed to satisfy the demand; *c5* is generated (Propagation 5).	*[70..100]*	*[1..4]*	*c1:[40..50]*	*c4:[10]* *c5:[10..100]* Wildcard
Close E	*c5* is required because it is the sole possible card that can produce the difference between the required card production (50) and the minimum demand (70). r_5 is limited to 60 because the sum cannot be greater than 100, and *c1* already produces 40.	*[70..100]*	*[2..3]*	*c1:[40..50]* *c5:[10..60]*	*c4:[10]*
$r_5 \leq 30$	Because all the producers are known (E is closed), the maximum value of S can be recomputed. (Propagation 3).	*[70..90]*	*[2..3]*	*c1:[40..50]* *c5:[10..30]*	*c4:[10]*
$r_3 \notin E$	Because all the producers are known (E is closed), the maximum value of S can be recomputed. (Propagation 2). The minimum value of *c5* is increased to satisfy the demand.	*[70..80]*	*[2]*	*c1:[40..50]* *qc5:[20..30]*	

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

```
// P represents the set of the instances of
Processor
IlcPort P = Processor.getInstancesPort();

// Global resource balancing constraint
cfg.add(F.getSum("power") <= P.getSum("pow-
er"));
```

6. A SAMPLE OF SOLVING

This section details the execution of the solution search for the example presented in Section 2, to show how taxonomic reasoning, port extensions, and resource constraints interact. The goal is to find the configuration that minimizes the number of processors needed to support the execution of the functions listed in Table 2.

The algorithm used is a classical Branch and Bound procedure that alternates choice points on variable assignment and constraint propagations. When an inconsistency is detected, back-track is performed until the last choice point, to try another value. The decision variables are the port `process` of the functions. These variables are instantiated in the order of the functions creation, that is, the order given in Table 2. We use a heuristic that chooses existing components before trying to generate a new one.

At the beginning, there is no processor. So all the `process` ports have the following form `(() wildcard) [1]`, that denotes a cardinality constrained to be equal to 1 and no existing processor.

Table 3 describes each choice point and its consequent propagations until the first solution is reached. $a \rightarrow b$ denotes a choice point that chooses the value a and assigns it to b.

Optimization is done by posting a more restrictive constraint on the cost function each time a solution is found, such that the cost of the nth solution is strictly better (greater or less) than the cost of the $(n-1)$th solution. In our example, the cost function is the number of processors. The first solution contains six processors. The second must contain at most five processors, and so on. Table 4 describes the steps until the second solution is reached.

Table 2. *Data*

Instance	Type	Computation power consumption
fcom1	FCOM	180
fcom2	FCOM	30
fcom3	FCOM	200
fnc1	FNC	220
fc1	FC	20
fc2	FC	30
fc3	FC	10
fc4	FC	40
fc5	FC	70

The optimal solution contains four processors. A possible distribution of the functions is: *fcom1* and *fcom3* in *p1*; *fcom2* and *fnc1* in *p2*; *fc1*, *fc2*, and *fc4* in *p3*; *fc3* and *fc5* in *p4*.

We do not detail the steps to reach this solution and to prove optimality because they are similar to those presented in Tables 2 and 3.

7. APPLICATION EXPERIENCE

We have experimented our framework on a real-world problem described in Mailharro and LeQuenven (1995). The application has been developed at Electricite De France (EDF). It consisted in configuring the instrumentation and control hardware and software architecture of nuclear power plants. The configurator had to assign a predefined set of functions to a hardware architecture (cubicles, racks, processors, I/O cards, physical connection to sensors and actuators), and to a software architecture (programs, cyclic tasks, logical connections). Each part of the configuration had to satisfy a various and complex set of constraints: safety, capacity, compatibility, etc.

The application has been used to configure a real nuclear power plant. It had to generate and interconnect several thousands of components. The execution time was about one hour on a Sun Sparc 20. Such a configuration required several weeks when it was done manually.

8. CONCLUSION

We have presented a new configuration dedicated framework whose objective is to provide an expressive and structured knowledge representation model and a set of powerful and efficient algorithms that allows construction and optimization of solutions. The knowledge of a configuration problem domain is concentrated in the catalog of component types that define generic local models of the different modules that can be combined to form the configured artifact. Because this catalog is constantly modified, the maintainability of the knowledge representation model is critical.

We have explained that our approach consists in considering configuration problems both as classification problems and as constraint satisfaction problems. We have combined object-oriented concepts with CSP concepts to adopt the strengths of both. The object-oriented approach allows the structuring of the domain knowledge as a generalization/specialization tree of component types with inheritance mechanism that improves its modularity and maintainability. The CSP approach provides a declarative and expressive formalism to represent the constraints that must be satisfied and the solving strategies.

We have shown how taxonomic reasoning can be implemented with constraint propagations and hierarchical domain CSP. We have shown how dynamic CSP can be supported by type instantiation and object refinement mechanism.

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

Table 3. *First solution search*

Step	Action	Propagations
1	*Wildcard* → *fcom1*	• Generation of processor *p1* and connection to *fcom1*; • *p1* is not XS100 because *fcom1* needs 180 power units whereas XS100 only produces 100 units; • *p1* becomes impossible for *fc1*, *fc2*, *fc3*, *fc4*, *fc5* functions because of compatibility constraints.
2	*p1* → *fcom2*	• XCOM becomes an impossible type for *p1* because 210 power units are needed (*fcom1* + *fcom2*) and XCOM only produces 200 units; • *p1* is refined to XN400 because it is the sole possible refinement for *p1*; • *fcom3* and *fnc1* become impossible for *p1* because they need more power than available in *p1* (190 free against 200 and 220 asked).
3	*Wildcard* → *fcom3*	• Generation of processor *p2* and connection to *fcom3*; • *p2* is not XS100 because *fcom3* needs 200 power units whereas XS100 only produces 100 units; • *p2* becomes impossible for *fc1*, *fc2*, *fc3*, *fc4*, *fc5* functions because of compatibility constraints; • *fnc1* becomes impossible for *p2* because it needs more power than available in *p2* (maximum 200 free against 220 asked).
4	*Wildcard* → *fnc1*	• Generation of processor *p3* and connection to *fnc1*; • *p3* is refined to XN400 because of compatibility constraints; • *p3* becomes impossible for *fc1*, *fc2*, *fc3*, *fc4*, *fc5* functions because of compatibility constraints.
5	*Wildcard* → *fc1*	• Generation of processor *p4* and connection to *fc1*; • *p4* is refined to XS100 because of compatibility constraints; • A secured-system is generated and connected to *p4* (Has-part inference).
6	*p4* → *fc2*	• *fc5* becomes impossible for *p4* because it consumes more power than available in *p4* (50 free against 70 asked).
7	*p4* → *fc3*	• *fc4* becomes impossible for *p4* because XS100 support at most three functions.
8	*Wildcard* → *fc4*	• Generation of processor *p5* and connection to *fc4*; • *p5* is refined to XS100 because of compatibility constraints; • A secured-system is generated and connected to *p5* (Has-part inference). • *fc5* becomes impossible for *p5* because it consumes more power than available in *p5* (60 free against 70 asked).
9	*Wildcard* → *fc5*	• Generation of processor *p6* and connection with *fc5*; • *p6* is refined to XS100 because of compatibility constraints; • A secured-system is generated and connected to *p6* (Has-part inference).

Table 4. *Second solution search*

Step	Action	Propagations
10	$\lvert Processor \rvert < 6$	• Back-track to step 9: wildcard becomes impossible for *fc5*. Inconsistency is detected because cardinality is at least 1 and there is no possible processor; • Back-track to step 8: wildcard becomes impossible for *fc4*. Inconsistency is detected because cardinality is at least 1 and there is no possible processor; • Back-track to step 7.
11	*Wildcard* → *fc3*	• Generation of processor *p5* and connection to *fc3*; • *p5* is refined to XS100 because of compatibility constraints; • A secured-system is generated and connected to *p5* (Has-part inference). • Type processor is closed because the maximal number of instances is reached; • All ports `process` are closed; • *fc5* is assigned to *p5* because it is the sole possibility; • *fc4* becomes impossible for *p5* because it consumes more power than available in *p5* (20 free against 40 asked); • *fc4* is assigned to *p4* because it is the sole possibility.

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press

Because the set of the components that compose the configured artifact is not known beforehand, we have proposed implementing connection ports as constrained set variables with nonfinite domain: This allows reasoning on the partial knowledge that is available when constructing the artifact. We have explained the semantic and the mechanisms associated with such variables.

We have also explained how port variables generate components on demand considering their cardinality and how has-part relations are implemented by a specialization of this on-demand mechanism. Finally, we have shown how accumulative functions defined on port variables can be used to represent resource balancing constraints that are able to generate components to ensure consistency.

Our future work will take into account the concept of component interchangeability defined in Freuder, (1991) and Haselbock (1993) to reduce the search space by eliminating equivalent assignment of port variables. Interchangeability must be context dependent because two components are interchangeable only in comparison to the role they have just as they are being considered. For example, when two cards of two different types are plugged into a rack, they are interchangeable if they have the same physical size. But if we consider them as sensor connections producer for function assignment, they may not be interchangeable if they do not provide the same number or the same type of physical connections.

REFERENCES

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

Brachman, R.J., & Schmolze, J.G. (1985). An overview of the KL-ONE Knowledge Representation System. *Cognitive Science 9(2)*, 171–216.

Cunis, R., Gunter, A., Syska, I., Bode, H., & Peters, H. (1989). PLAKON, an approach to domain independent construction. *Proc. Conf. IEA/AIE*.

Freuder, E.C. (1991). Eliminating interchangeable values in constraint satisfaction problems. *Proc. AAAI Conf.*, 227–233.

Haselbock, A. (1993). Knowledge-based configuration and advanced constraint technologies. Ph.D. Dissertation, Institut fur Informationssysteme, Technische Universitat Wien.

Heinrich, M., & Jungst, E.W. (1991). A resource-based paradigm for the configuring of technical systems from modular components. *Proc. 7th Conf. on Artificial Intelligence Applications*, 257–264.

ILOG (1997). *Reference Manual and User's Manual, ILOG Solver*, 4.0 Edition. ILOG, Paris, France.

Klein, R. (1996). A logic-based description of configuration: The Constructive Problem Solving approach. In *Workshop Notes of AAAI Fall Symposium on Configuration*, 1–10b. AAAI Press, Menlo Park, CA.

Mailharro, D., & LeQuenven T. (1995). A constraint based tool for automatic sizing of an instrumentation and control architecture. *Proc. First Ilog Solver Users Conf.*

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

Mackworth, A.K., Mulder, J., & Havens, W. (1985). Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence 1*, 118–126.

Mittal, S., & Frayman, F. (1989). Towards a generic model of configuration tasks. *Proc. Eleventh Int. Joint Conf. of Artificial Intelligence*, 1395–1401.

Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. *Proc. AAAI Conf.*, 25–32.

Puget, J.F. (1992). Programmation par contrainte orientee objet. *Proc. Int. Conf. on Expert Systems of Avignon*, 129–138.

Sabin, D., & Freuder, F. (1996). Configuration as Composite Constraint Satisfaction. In *Workshop Notes of AAAI Fall Symposium on Configuration*, 28–36. AAAI Press, Menlo Park, CA.

Yokoyama, T. (1990). An object-oriented and constraint-based knowledge representation system for design object modeling. *Proc. Sixth Conf. on Artificial Intelligence Applications*, 146–152.

Daniel Mailharro is an enginer that works in the Research and Development Department of ILOG, S.A. in Paris, France. After achieving a Master of Science at Paris University, he worked for 7 years as a consultant specializing in constraint programming for TéléCom, and energy and transport companies. Currently, he is focusing his efforts on constraint problems at ILOG.

https://doi.org/10.1017/S0890060498124101 Published online by Cambridge University Press