A TOOL FOR DEVELOPING INTERACTIVE CONFIGURATION APPLICATIONS

TOMAS AXLING AND SEIF HARIDI

▷ Knowledge based techniques have been proven to be well suited for configuration tasks and several, often successful, systems have been developed. The goal of these knowledge based techniques has in general been to produce "black box" configuration systems, i.e. batch-mode systems that take a set of requirements as input and produce configurations as output. This has resulted in rather complex systems that have been time consuming to develop and in which the choices of components, which are often subjective, are hidden from the user. In this paper we present a less complex and more interactive approach to configuration that we label Interactive Configuration by Selection (ICS). The idea is that a configuration system could be more of a tool, assisting in the task of configuration, than a "black box" automatically producing configurations directly from requirements. To support the approach we have developed OBELICS. OBELICS is a prototype of a tool, implemented in SICStus Objects, that integrates both a problem solving method for ICS and a tool for domain knowledge modeling. With OBELICS we hope to demonstrate that such a tool can make it feasible and profitable to develop configuration applications, even for small scale applications. ◁

1. Introduction

It is common for manufacturing companies to market products that are tailor made according to customers' requirements. Each item to be sold can be regarded as a new product which is assembled from a large predefined set of components to meet the requirements. This type of configuration may be complex and time consuming when performed without any specialized support tool.

Configuration tasks are well suited for knowledge-based support, and several systems have been developed, e.g. XCON [8]. However, one of the lessons learned

Address correspondence to Swedish Institute of Computer Science PO Box 1263, 164 28 Kista, Sweden, Tel: +46 8 752 15 00 E-mail: axling@sics.se, seif@sics.se

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

from these, often successful, systems is that they are time consuming to develop and hard to maintain. It is obvious that maintainability is crucial for applications where the domain is rapidly changing, which usually is the case for configuration tasks: new products are constantly designed and old products are modified.

The cost of developing configuration systems implies that in order to be profitable, only large scale applications can be considered. Therefore there exists a need for methods and tools that speed up the development of configuration systems and make the systems more maintainable.

The goal of existing tools is in general to develop "black box" configuration systems, i.e. batch-mode systems that take either parts or higher level functional descriptions as input and produce configurations as output. These systems may interact with the user while creating configurations by querying the user for design decisions. This is generally done in a program driven fashion, forcing the user to respond to the program. These are rather complex systems that still are time consuming to develop and in which the choices of components, which often are subjective, are hidden from the user.

In this paper we present a less complex and more interactive approach to configuration that we label Interactive Configuration by Selection (ICS). The idea is that a configuration system could be more of a tool, assisting in the task of configuration, than a "black box" automatically producing configurations directly from requirements. As recognized in [10], from the end-user's perspective the goal is not to have a maximally "intelligent" system but rather one that will help him the most in his day-to-day work. To support the approach we have developed OBELICS (OBjEct-oriented Language for ICS). OBELICS is a prototype of a tool, implemented in SICStus Objects [7], which integrates both a problem solving method for ICS and a tool for domain knowledge modeling. The applications built with OBELICS are truly interactive, user centered configurators that react to actions rather than simply prompt for input at well-defined places in the program.

With OBELICS we hope to demonstrate that such a tool can make it feasible and profitable to develop configuration applications, even for small scale applications.

The paper is structured as follows: In section 2 the task of configuration is defined and restrictions that may be imposed on it are discussed. In section 3 the ICS approach to configuration is introduced and argued for. In Section 4 a tool for developing ICS applications is outlined and design rationales are discussed.

2. Defining Configuration

Making very few assumptions about the kinds of knowledge that might be available, [9] defines a configuration task as follows:

Given: (a) a fixed, predefined set of components, where a component is described by a set of properties, ports for connecting it to other components, constraints at each port that describe the components that can be connected at that port, and other structural constraints; (b) some description of the desired configuration; and (c) possibly some criteria for making optimal selections.

Build: One or more configurations that satisfy all the requirements, where a configuration is a set of components and a description of the connections between the components in the set, or, detect inconsistencies in the requirements.

There are three important aspects to this definition; (a) the components that can be used to design some artifact are fixed, i.e., one cannot design new components; (b) the components cannot be modified to get arbitrary connectivity; and (c) a solution not only specifies the actual components but also how to connect them together. [9] also introduces two more restrictions on this definition.

1. The artifacts are configured according to some known functional architectures.
2. The "key component" assumption.

The key component assumption implies that if one has selected a key component then most of the components cannot be selected arbitrarily, because they are dependent on the key component. Most implemented configuration expert systems, such as XCON, Cossack and MICON, rely on these assumptions. We will label configuration tasks that rely on these assumptions "restricted configuration tasks". The rest of this paper is mainly concerned with this restricted class of configuration tasks.

3. Interactive Configuration by Selection

The *Interactive Configuration by Selection* (ICS) approach is mainly derived from the *Distribution Product System* (DPS) [1], a tool for switchgear configuration which is described below.

3.1. DPS

DPS is a tool aiding the work of configuring and selling low voltage switchgear. It was developed by Tomas Axling in 1989 for an ABB subsidiary in Saudi Arabia and still is in active use. The work of configuring and selling low voltage switchgear can be described as follows:

- An inquiry arrives to the company from a customer. The inquiry includes a general specification of the required functionality and constraints.
- The sales engineer then makes an interpretation of the inquiry. He selects the type and number of cubicles, the components needed and so forth.
- In accordance with the selection the sales engineer then creates a quotation which includes:
  - a specification of the parts needed
  - a price calculation
  - single line and front layout drawings
- If the customer approves of the quotation all the parts needed for the order will be ordered or manufactured. Otherwise the order is lost or a new interpretation (and quotation) will have to be made.

The task for DPS is to help with the interpretation and then automatically generate a quotation. If the quotation becomes an order the list of needed parts is produced by the DPS. The system does this by assisting the user in selecting suitable switchgear components (like breakers, bus bars, cubicles) and then puts them together to form a complete switchgear. As output the system produces a

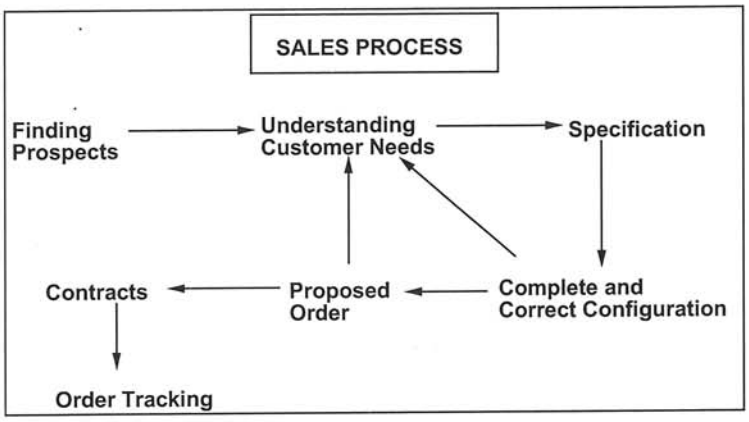

FIGURE 3.1. A generalization of the switchgear sales process

specification, a quotation, front layout and single line diagram drawings, and a price calculation with a complete part list.

The system was very well received and the following gains from using DPS were recognized.

- It cut down the monotonous and laborious work of creating quotations to a minimum (a quotation that could consume weeks of manual labor can be done in a few hours with DPS).
- More consistent appearance of the quotations.
- Easier to follow up on old quotations.
- Even the smallest inquiries can be given a complete quotation.
- The customer receives his quotation earlier.

The major problem with DPS has been to keep it up to date with its domain. This problem can be divided into two parts:

- Price changes and refinements of the text and drawings. This is easily done using the DPS utilities.
- Changes in the product, where products constantly evolve and new solutions are required of the DPS. This is the major problem when it requires modifications in the program.

The switchgear sales process may be generalized to the sales process described in figure 3.1 and generalizing DPS leads us to the definition of the ICS task.

3.2. ICS Tasks

We define an ICS task to be a restricted configuration task that:

1. assists the user in the selection of main components for a product by presenting only valid options;
2. assembles the selected main components and add all other components necessary to make a complete product; and
3. generates the required description of the product.

The task should be performed in an interactive process where the user together with the application successively builds configurations. An ICS system should, at any time during a configuration, be able to present the current state of the configuration and allow the user to alter previously made configuration decisions.

3.3. Applicable Domains

For the ICS approach to be applicable for a domain it must be feasible to represent it as a component hierarchy where:

1. the hierarchy represents all possible goal artifacts.
2. the knowledge of how the component classes can be combined is explicitly represented.
3. the terminals are physical components.

(1) guarantees that all solutions can be found in the hierarchy. (2) guarantees that it can be determined if a partial hierarchy includes conflicting component classes or not. (3) guarantees that it is just a question of assembling components once we have found a partial hierarchy representing a feasible goal artifact, i.e. there cannot be any additional configuration or design to be done for the terminals in the partial hierarchy. Criterion (3) distinguishes the ICS domain from the one of *routine design* (see [4]) in which no such restriction exists.

Examples of domains that fulfill the three criteria for ICS are microcomputer system [5], switchgear and kitchen configuration [6]. An example of domains just outside of ICS' is AIR-CYL's [3](a system for designing air cylinders) which fulfills criteria (1) and (2) but fails criterion (3) when e.g. it is not possible to decompose a piston into existing components.

3.4. An Example

As an example of how ICS systems work we will use an imaginary ICS version of Cossack [5], a configuration system for microcomputer systems.

In the first menu, M1=[PrinterStuff, Processors, Displays, Software], you might choose software. Then from the software menu M2=[CAD, Wordprocessing,...] you choose CAD. In the CAD menu M3=[AutoCAD,...] you choose AutoCAD. The system will now (i) automatically add the extra components that are needed for a microcomputer system running AutoCAD, e.g. in this case an arithmetic processor; (ii) exclude all options in the menus that cannot be used together with AutoCAD, e.g. only graphical printers will remain in the printer menu; and (iii) continue with the M1 menu. You might now continue with selecting Displays, etc. The system will not allow you to exit until you have chosen components enough for a complete microcomputer system. When you have selected all the key components that you felt were needed you exit and let the ICS system produce a specification of the microcomputer system that you have configured.

Essentially the difference between an ICS version of Cossack and Cossack is that the user requirements in the ICS version are basically specified by selecting a number of key components, while in Cossack they are specified in more general terms of constraints. The ICS version works in a interactive fashion, consulting the user for all component choices that are not obvious, while Cossack takes a set of requirements and tries to build a configuration directly from it.

3.5. Motivation

The basic principle behind ICS is that when there are several similar alternatives the system should not try to decide which one is the most suitable, instead it should leave this decision to the user. This may give the impression that ICS systems would only give a fraction of the assistance "black box" systems give. Therefore will we here present a few arguments for the use of ICS systems and try to show that an ICS system can give as much assistance as a "black box" configuration system despite using substantially less domain knowledge.

3.5.1. IMPLICATIONS OF THE "KEY COMPONENT ASSUMPTION" According to the "key component assumption" [9] one can identify some particular component (or a small set) that is crucial for implementing some function in many design domains. Thus, one does not need to consider arbitrary configurations to implement some function, one needs only to start with a "key component" and build suitable configurations on that basis. Systems such as XCON, Cossack and MICON crucially depend on this assumption. XCON actually relies on this assumption to "infer" the functional requirements from the set of components it is given to verify. It has rules which look for certain key components and then ensure that other components needed to have consistent subsystems around those components are included in the configuration. Cossack and MICON use this assumption to build subsystems which are then composed together.

In our experience a configurer usually thinks in terms of key components. He has an abstract sense of what key components are needed to obtain the required functionality. For example while configuring a microcomputer system one would have the intuitive feeling that a display will be needed to meet the desired functionality. One might even know that a color display is wanted. The problem is then to find concrete alternatives for the display and to figure out how the selection of the display restricts the selection of the other components. For example what components are needed to connect the display, can an 8 bit CPU be used (not if the display requires a 16 bit display controller board), etc? This is exactly what an ICS system assists you with.

Examples of applications that depend heavily on the "key component assumption" and the above assumption about thinking in terms of key components are Canard [11] and DPS.

3.5.2. EXPLORING THE SPACE OF ALTERNATIVES As pointed out in [11], cognitive scientists have long known that people typically retrieve only a small fraction of available alternatives when generating hypotheses. People tend to anchor on initial guesses, giving insufficient regard to subsequent data. For various other reasons, people may not be able to visualize whole classes of possibilities. An ICS system presents all valid alternatives and also makes it easy to alter previous choices, hence assists designers in exploring the space of alternatives. This gives the designer a better feel for the effects of the constraints on different alternatives and the consequences of assumptions and tradeoffs.

3.5.3. THE NOTION OF OPTIMAL CONFIGURATIONS In many cases configuration is really an optimization task, where one can judge and compare configuration alternatives and ultimately select the best or optimal configuration. The notion of

optimal configuration may involve several optimality criteria which are combined together based on their relative merits. Criteria often work against each other. The combination of different criteria is usually highly subjective. For example while configuring a computer system one might prefer IBM components and a cheap solution. Now say a configuration system is about to choose the display for the computer system and it knows that the IBM display costs $100 more than a comparable alternative, what should it do? To make this decision it needs to know how strong the preference for IBM components is compared to the preference for a cheap solution. This could be done by letting the user specify it. But is the preference for IBM components the same when it comes to selecting a printer, etc? This subjectivity makes it difficult for a system to determine which configuration is optimal. This combined with the fact that criteria may be difficult or even impossible to formulate implies that one should let the user consider several different alternatives and choose the one he finds to be optimal according to his criteria. This is what an ICS system does. It would furthermore support this by reporting what effects the different choices have on the design at each selection.

3.5.4. INCREMENTAL DEVELOPMENT Incremental development with the following stages may be used when developing ICS systems:

1. An ICS system could start by being a system that only assists in finding component alternatives and then (regardless of validity) assembles the selected components and produces the required documentation.
2. Then knowledge could be added to make the system present only valid component alternatives. This is a system in which you cannot include conflicting components in the construction, but the final construction may be incomplete if you forget to select some indispensable parts.
3. A true ICS system, i.e. a system that automatically adds components that must be included to support the selected key components and guarantees a valid construction. The system can then be further refined to reduce the number of manual selections needed to produce a configuration.
4. Other problem solving methods may use ICS domain models if preference knowledge (to enable the system to choose between sets of acceptable alternatives) is added. For example, extra control knowledge may be added to produce an automated alternative generation option. The system may then be used both as a "black box" configuration system and as an ICS system. In this way existing ICS systems may provide a testing ground for configuration problem solving methods which may be interesting from a research point of view.

This means that an ICS can be useful at an early stage of development. For example DPS, which can be described as an ICS system in phase two, was very profitable despite its updating problems. This is in contrast with conventional configuration systems which often are not usable until all knowledge needed by the configuration method is present. The advantages of a system being useful at an early stage are further discussed in [10].

4. A Tool for Developing ICS Applications

In the previous section we argued that ICS systems can be useful. But ICS systems are, like configuration systems in general, expensive to develop and maintain when implemented without some specialized support tool. We believe that the restrictions that ICS imposes on the general configuration task make it appropriate to implement a tool for developing ICS systems.

4.1. Design Rationales

- The problem solving knowledge should be separated from the domain knowledge and be embedded in the tool.
- The domain knowledge should be represented declaratively.
- The tool should support the modeling of the domain. Instead of working with unstructured collections of rules, as in regular rule-based languages, the domain should be structured into chunks of knowledge. The chunks of knowledge should be presented to the user with a graphical user interface, making the domain model easy to understand and work with. The chunks of knowledge should be as independent as possible, letting the user concentrate on one chunk at a time without being concerned about interference with the rest of the domain model.

4.2. What Needs to be Modeled?

The knowledge used in configuration expert systems can be classified into two types: knowledge about components and knowledge about configuration methods. As pointed out in [13], only knowledge about configuration methods has been regarded as important in conventional systems. This is rather surprising considering that almost all knowledge needed for real world configuration is bound to the components.

In many configuration systems, like XCON, there is no clear separation between component knowledge and knowledge about configuration methods since constraints on components are often expressed in the production rules. Moreover it is often not clear, in this kind of system, how a newly added rule will interact with existing rules when there is no explicit problem solving model.

In many cases the component models are mere data structures that represent attributes of components. They need to be interpreted and manipulated in terms of configuration tasks or procedures and the knowledge about the components may be embedded in model manipulation procedures or configuration methods. This embedding of component knowledge in configuration methods makes it difficult to make effective use of the component knowledge.

The central role of components implies that the component model should not be a mere data structure but a model that includes all knowledge about the component. Then the component model can be given an active role in the configuration process. Configuration requirements can be regarded as constraints on components which reduce the configuration problem to a constraint satisfaction problem. These points have recently been, to varying extent, recognized and used [9] [12] [13].

According to these arguments for focusing on component knowledge, most of the domain knowledge should be contained within the component model.

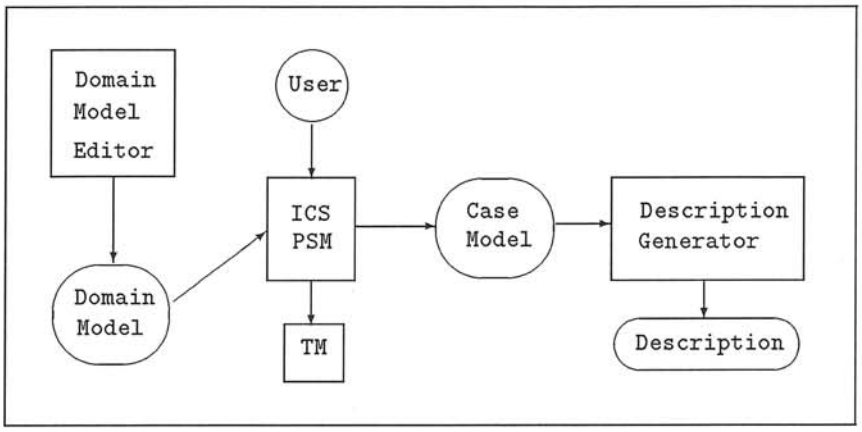

FIGURE 5.1. The main components of OBELICS

For example the knowledge about

- Under what conditions the component can be included in the construction
- Dependency relations on other components
- Attributes of the component

should be modeled in the component model.

In section 2 a configuration is defined as a set of components and a description of the connections between the components in the set. This implies that we must be able to model the connections in an effective way. For this reason, there should be a separation between the component model and the connection model. The component and connection models constitute the domain model of a product family.

5. OBELICS

In this section we will outline OBELICS (OBjEct-oriented Language for ICS), a tool for creating ICS applications, and then describe its components in some detail.

As shown in figure 5.1, OBELICS consists of a domain model editor, a generic problem solver for configuration tasks including a truth maintenance component, and a description generator. For each configuration application, the domain model editor assists in creating a domain model that describes the components, connections and constraints on them. The problem solver takes such a domain model and additional information and constraints entered by the user to create interactively a specific configuration (case model). The case model is available to the user during a configuration session allowing the user to view the current state and to alter previous decisions. The truth maintenance component is responsible for keeping the case model consistent upon changes. When the case model is finalized, it is used as an input to the description generator to create the necessary textual and drawing information.

OBELICS has been constructed mainly using SICStus Prolog Objects system, an extension of Prolog for object oriented programming [7]. Since SICStus Objects has been crucial in particular for the knowledge representation in the domain model,

we will give a brief description of the object system. For details the reader may consult [7]. We assume familiarity of Prolog and the notions of logic programming, as well as the notions of object oriented programming.

5.1. SICStus Prolog Objects System

SICStus Prolog Objects System (the short name is Prolog Objects), is an efficient extension of SICStus Prolog with object oriented programming. In Prolog Objects, an *object* is a named collection of predicate definitions. In this sense an object is similar to a Prolog module. The object system can be seen as an extension of SICStus Prolog's module system. In addition an object may have attributes that are modifiable. Predicate definitions belonging to an object are called *methods*. So, an object is conceptually a named collection of methods and attributes. Some of the methods defined for an object need not be stored explicitly within the object, but are rather shared with other objects through the inheritance mechanism. The Object system is based on the notion of *prototypes*, which basically allows "classes" to be first-class objects, and provides a mechanism in addition to inheritance known as *method delegation*.

Objects may be defined in a file, or dynamically created during the execution of a program. When an object is created, during load-time or run-time, it inherits the methods and attributes of its prototypical object(s). Objects defined in a file can be either *static* or *dynamic*. Also methods can be either dynamic or static. These properties are inherited by subobjects. Objects created during execution are dynamic. Multiple inheritance is also allowed as well as light weight objects called *instances*. Such an object may only inherits from one object, and have only attributes derived from of its super object.

An object `object-identifier` is declared by writing it in the following form:

```
object-identifier :: {
        sentence-1 &
        sentence-2 &
        :
        sentence-n
        }.
```

where `object-identifier` is a Prolog term that is either an atom or a compound term of the form `functor(V1,...,Vn)`, where `V1,...,Vn` are distinct variables. The object body consists of a number of *sentences*, possibly none, surrounded by braces, where each sentence is a *method-clause*. A *method* is a number of method-clauses with the same principal functor. A method-clause has a clausal syntax similar to that of Prolog, but instead of the usual predicate calls in the body of a clause there are *method-calls*. Ordinary Prolog goals are also allowed in a prefixed form, using ':' as a prefix. Goals in the body of a non-unit clause have the normal control structures of Prolog. Atomic goals in the body of a method-clause may be in a number of forms, besides being Prolog goals. The most relevant ones for this paper are:

- `goal` to send the message `goal` to the object `self`.
- `object::goal` to send the message `goal` to object `object`.
- `object<:goal` to delegate the message `goal` to object `object`.

A few examples will clarify the concepts needed for this paper. The following is an object called `listObject`. Different from ordinary Prolog modules, the methods included there can be specialized by other subobjects.

```
listObject :: {

        append([], L, L) &
        append([X|L1], L2, [X|L3]) :-
                append(L1, L2, L3) &

        member(X, L) :-
                member(X,L) &

        length([], 0) &
        length([_|L], N) :-
                length(L, N1),
                :(N is N1+1)
        }.
```

The following object `apt1` could be part of a larger database about free apartments at a real-estate agency. `apt1` is defined by using attributes. These can be retrieved and modified efficiently by the methods `get/1` and `set/1` respectively. The attributes are defined with their default values. Immediate super objects are declared using the method `super/1`. Several declaration of super/1 means multiple inheritance.

```
apt1 :: {
        super(apartment) &

        attributes([
                streetName('York'),
                streetNumber(100),
                wallColour(white),
                floorSurface(wood)])
        }.
```

Objects can be created dynamically using the method `new/2` inherited from the initial object `object`. The arguments of `new/2` are the new object, and the list of its immediate super objects respectively. This method can specialized. In OBELICS, we have specialized `new/2` so that if a method exists which has the same name as an attribute in any of the super objects, the method will be used to compute the value of the corresponding attribute.

The ability to create dynamic objects during domain modeling as well as the ability to define complex relations between objects, attributes, etc. is important for modeling configuration domains. The expressions used in various relations may also contain constraint expressions. These are provided by the underlying Prolog system and are not specific to the object system. In the following section we allow expressions containing attributes of objects. These attributes will be prefixed with `@`. For example, `apt1@streetNumber` means the value of the attribute `streetNumber` in the object `apt1`, whereas `@streetNumber` is the value of the attribute in the

`self` object. Such an attribute will be subject to the truth maintenance mechanism, which belongs specifically to OBELICS and not to the object system. The self-object may be retrieved in any method using the method `self/1`.

5.2. Domain Modeling

The OBELICS domain model is a form of component hierarchy that describes is-a and part-of relations. The component hierarchy should represent all the possible artifacts in the domain. The domain model editor presents hierarchies in an understandable way and makes it easy to add and modify components. Figure 5.2 illustrates the editor and also exemplifies OBELICS' component hierarchies. In the component hierarchies each component class is described as follows.

COMPONENT CLASS name

Description: A textual description of the component class.
If the component class does not have any subcomponents or subclasses we label it a primitive component class. If it does have subcomponents we call it a composite component class. For a primitive component class the description describes the component represented by the component class.

Attributes: Attributes like price, size and auxiliary variables. They are described in a declarative form. Some attribute values are entered by the user, some are calculated and some are constants. The subclasses inherit the attributes. Modes for calculating attribute values are:

- Attribute whose value is constant.
- Attribute whose value is to be entered by the user.
- Attribute whose value is to be calculated with a specified method.
- Attribute which is defined here, but the method for calculating its value is defined in the subclasses.

Preconditions: The constraints that must be satisfied if the component class is to be considered as a valid option, i.e. the constraints that state if it is correct to include the component class in the construction. The preconditions are defined as an OBELICS constraint expression.
An OBELICS constraint expression (OCE) is a conjunction or a disjunction of OCEs or a simple constraint expression. A simple constraint expression is an equality/inequality of attributes accessible in the component class, a linear rational constraint, or a call to an access method. A linear rational constraint is an linear equation of attributes accessible in the component class enclosed in curly brackets, e.g. `{@price<@k+100}`. An access method is one of OBELICS' predefined methods for accessing other instances, for example:

instance(CC,Expr,I) which means "There is an instance I of `CC` that satisfies `Expr`", `Expr` being of the same type as preconditions. This method enables one to access other instances and their attributes using `Expr` as criteria.

no_instance(CC,Expr) which means "There does not exist an instance of `CC` that satisfies `Expr`", `Expr` being of the same type as preconditions.

apply(F,I,Res) which means "`Res` is the result of applying `F/2` on the part-of hierarchy rooted at instance `I`.

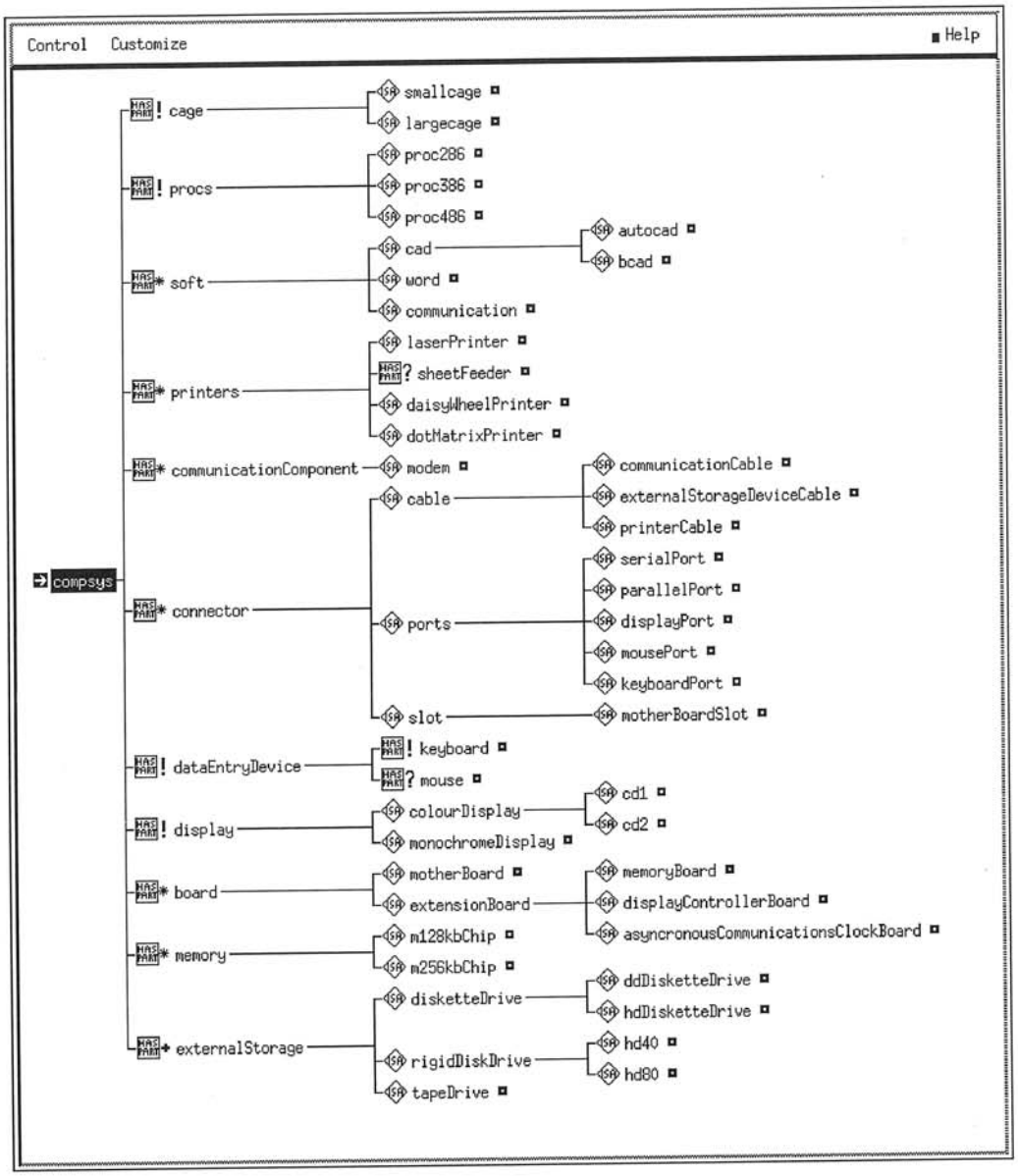

FIGURE 5.2. The Domain Model Editor

Dependencies: Constraints stating what components and connections must exist if an instance of this component class is be included in the construction. The selection of this component class is not confirmed until the dependencies are satisfied. For example a software package might require a selection of an arithmetic processor, then a selection of a software package is not confirmed until an arithmetic processor is selected. Dependencies are expressed in two ways:

dependency(CC,I:OCE) which states "there must exist an instance I of component class `CC` that satisfies `OCE`",

connection(ConClass,CompClasses) which states "there must exist an instance of the connection class `ConClass` connecting component class instances as specified in `CompClasses`". The `CompClasses` is a list of pairs, `(CC,I:OCE)`, each describing what class(`CC`) an instance I belongs to and its associated constraints (`OCE`) that must be satisfied.

Subcomponents: The component class' subcomponents and the number of possible occurrences of them. The number of possible occurrences are:

! : Exactly one is required to make the construction complete. We label these subcomponents *required component classes.*

? : One may be included. We label these subcomponents *optional component classes.*

* : Zero or more may be included. We label these subcomponents *optional iteration component classes.*

\+ : One or more must be included. We label these subcomponents *required iteration component classes.*

Subclasses: The component class' subclasses.

The subclass and subcomponent relations also describe the inheritance scheme. Subclasses and subcomponents will inherit methods through an *inheritance by overriding* mechanism.

5.2.1. Representing Connections To describe possible connections between components we use connection classes. In a connection class we specify which component classes are connected, and for each component class in the connection class, we specify whether a component instance may occur in other connection instances. If it may occur in other connection instances we specify which connection classes they should belong to (e.g. the current connection class). For each of those connection classes, we specify whether the connection possibility is definite or optional.

The following is an informal description of how connection classes are represented in OBELICS.

CONNECTION CLASS name

Description: A textual description of the connection class.

Between: A set of triples (CC,SC,OSC) that describe the components the connection class can connect. The triples are defined as follows:

CC is a component class.

SC is a set of connection classes.

OSC is a set of connection classes.

Each triple states that a connection of the class must include a component

C of class CC that is not included in any other connections except for connections of classes that occur in SC or in OSC if granted by the user.
Hence SC states shareability, i.e. indicates if a component may be simultaneously used in more than one connection.
OSC states optional shareability. An example of this is a printer and a modem that are using the same port. This is possible if the printer and the modem are not to be used simultaneously and the inconvenience of switching between the two is acceptable. The question of shareability is hence subjective and should therefore be dealt with in an interactive fashion, as opposed to the cases where a component is clearly sharable.

5.3. An Example Domain

This section describes how we made a domain model for the example in section 3.4 using OBELICS.

The example is based on Cossack [5], a configuration system for microcomputer systems. The component hierarchy we created is shown in figure 5.2. This hierarchy could have been designed in many other ways, some a lot more suited to OBELICS, but to corroborate the flexibility of modeling in OBELICS we followed the Cossack model as much as possible.

The example was implemented in the following steps.

1. The ICS Cossack component hierarchy was built by defining the component classes needed for the domain. At this stage we did not include all dependencies and preconditions needed. The component classes were defined using the domain model editor which represents them as objects in SICStus Objects. For example the component classes `compsys` and `cad` were represented as:

```
compsys::{
   super(component)&
   description('computer system')&
   attributes([minmem(0),memory(0),total_price(0),name('PC')])&
   description(total_price,'Price of the computer system')&
   description(name,'Label for the computer system')&
   description(memory,'Amount of selected memory')&
   precondition(true)&
% Methods for calculating attribute values
   total_price(P):- self(S), sum(S,price,P)&
   memory(X):-sum(memory,no_kb,X)&

   displayable([total_price,memory])&
   user_specified([name,minmem])&
   subclasses([])&
   subcomponents([(connector,*),
             (procs,!),
             (soft,*),
             (printers,*),
             (cage,!),
             (dataEntryDevice,!),
```

```
                (display,!),
                (board,*),
                (memory,*),
                (motherBoard,!),
                (externalStorage,+),
                (communicationComponent,*)])
}.

cad::{
   super(software)&
   description('CAD software package')&
   attributes([dimensions(2)])&
   preconditions({@dimensions>1,@dimensions=<3})&
   user_specified([dimensions])&
   subclasses([autoCAD,3dCAD])&
}.
```

`super(software)` defines the inheritance relation for `cad`.
`attributes([dimensions])` introduces a new attribute, `dimensions`. The values of `dimensions` will be stored in `cad` instances, but are specified by `cad`'s subcomponent/subclass instances.
`preconditions({@dimensions>1,@dimensions=<3})` is a constraint that `cad` imposes on `dimensions`.
`user_specified([dimensions])` states that the value of `dimensions` may be set by the user during configuration.
`subclasses([autoCAD,3dCAD])` states that `cad` has subclasses `autoCAD` and `3dCAD`.

2. The connection classes were defined. For example:

```
printerToPort::{
   super(connection)&
   description('connection between printer and port')&
   between([(printer,[],[]),
            (cable,[],[]),
            (port,[portBoard],[printerToPort])])
}.
```

`printerToPort` describes connections that consist of a printer and a cable, neither included in any other connection, and a port instance that may also be included in a `portBoard` connection and, if the user permits it, in another `printerToPort` connection.

3. The remaining preconditions and dependencies were added to the component classes. For example:

```
laserPrinter::{
   super(printer)&
   description('Laser printer')&
   precondition((@graphic=true,{@price=8}))&
   connection(printerToPort,[(laserPrinter,L:self(L)),
                             (cable,C:true),
                             (port,P:P@type=serial)])
```

```
}.
```

The precondition imposes constraints on the attributes `graphic` and `price`. These are attributes of `printer` which `laserPrinter` is a subclass of. This means that for a `printer` instance to select `laserPrinter` as subclass it must be possible to set the value of `graphic` to `true` and the value of `price` to `8` in the `printer` instance.
The connection statement implies that for a `laserPrinter` instance there must be an instance of the connection class `printerToPort` that connects the `laserPrinter` instance, a `cable` instance and `port` instance that satisfies `type=serial`.

```
autocad::{
   super(cad)&
   description('AutoCAD')&
   precondition(true)&
   displayable([])&
   user_specified([])&
   attributes([price(7)])&
   subcomponents([])&
   dependency(procs,P:P@speed=high))&
   dependency(printers,Pr:Pr@graphic=true)&
   dependency(display,D:{D@no_pixels>1500})
}.
```

The first dependency states that if an instance of `autocad` is included in the hierarchy instance, an instance of `processor` where `@speed=high` must also be included before the hierarchy instance is considered valid.

5.4. The Problem Solving Method

The purpose of the problem solving method is to produce configurations from user inputs and domain models. In this case domain models are component hierarchies including connection classes. In OBELICS we represent a configuration as a hierarchy of component class instances following the structure of the component hierarchy and a set of connection class instances. In a sense, a configuration is an instance of a component hierarchy. If all constraints, preconditions and dependencies, are satisfied in a component hierarchy instance, it represents a valid configuration. Usually a set of connection instances are needed to satisfy the dependencies.

OBELICS problem solving method is rather simple. It works in a top-down fashion, i.e. it takes the top component in a component hierarchy and creates an instance of it, and then evaluates the instance. The evaluation of the top component instance will in turn cause the creation and evaluation of the other instances needed for a configuration.

The following describes what OBELICS' PSM does when evaluating an instance:

1. **Set user-specifiable attribute values** The user is prompted to set the instance's user-specifiable attribute values.
2. **Try to satisfy the instance's preconditions** If any of the preconditions fails then the instance cannot be included in the construction, so the evaluation of the instance fails.

3. **Find subgoals** From the dependencies and the set of required subcomponents a set of goals is extracted. These goals all have the form "there must exist an instance of component class/connection class *CC* with attribute values that satisfy *Expr*". These are required goals, i.e. for the evaluation of the instance to succeed, the whole set of goals must be satisfied.
   From the set of optional subcomponents the user may add optional goals. These goals have the same form and are treated in the same way as the required goals but they do not need to be achieved for the evaluation to succeed. Only subcomponents with satisfiable preconditions may be selected as optional goals.
   From the set of subclasses a specialization is chosen, if the set is not empty. The choice may only be from valid subclasses, i.e. subclasses with satisfiable preconditions. If the set is not empty but none of the subclasses is valid, the evaluation of the instance fails. If there is only one valid subclass, it is trivially chosen. If there is more than one, the user is asked to make the choice. From the chosen subclass a goal is extracted and added to the set of required goals.
4. **Evaluate subgoals** When the PSM has tried to achieve all the optional goals and has achieved all the required goals the instance is successfully evaluated.

If the evaluation of an instance fails, it is removed and the truth maintainer is notified. If the instance has subinstances, these are also removed. The case of instances that are created to satisfy the removed instance's dependencies is not so trivial. The user may still want to include these components in the configuration whether the instance is removed or not. Therefore the user is prompted to make this decision.

Figure 5.3 illustrates the relation between a component hierarchy and an instance of it. In other words, it illustrates the relation between a domain model and a configuration.

5.5. The Truth Maintainer

A major problem for OBELICS is the problem of keeping constraints in the instance hierarchy satisfied. We have separated this part from the rest of OBELICS PSM, into a separate truth maintainer (TM).

The TM's activities can be divided into two main areas: maintaining consistency between attribute values and maintaining dependency relations. The TM maintains a constraint network for each of these areas: one for attributes and the relations between them (ACN), and one for the instances and the relations between them (ICN). When creating an instance the TM analyses its constraints and augments the constraint networks accordingly. The networks are then used by the TM to maintain consistency when the hierarchy instance changes.

The events that affect the ACN are, besides adding new instances, user changes of attribute values and context changes of the methods used to calculate attribute values. For example the attribute `tot_memory`, which is calculated with a method which sums all the `memory` instances value of the attribute size, changes if a memory instance is deleted even though no explicit updates of attribute values have occurred. These events trigger recalculation of the affected part of the ACN. The

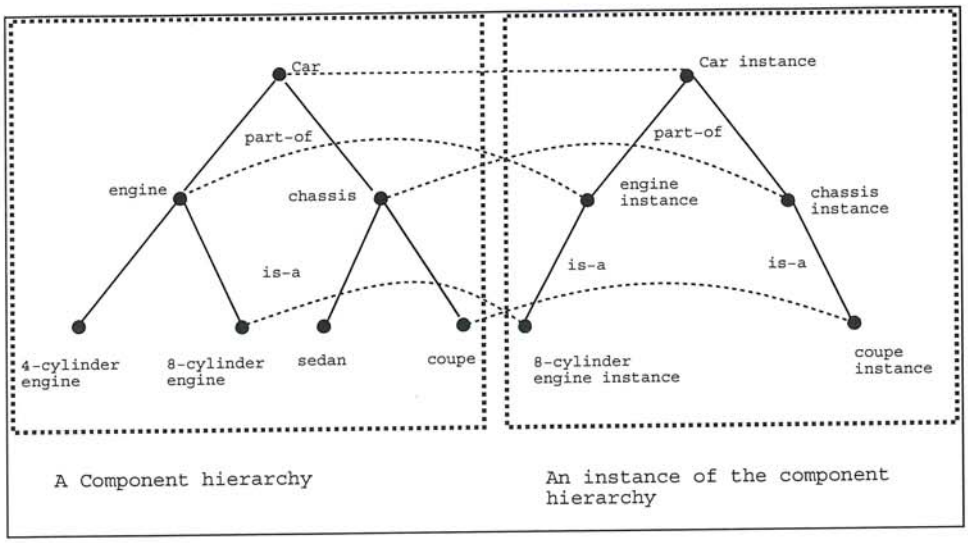

FIGURE 5.3. Relations between a component hierarchy and an instance of it

current values of the attributes are stored in the network.

When an instance is deleted or modified TM uses ICN to find instances that are dependent on the instance. The TM will try to re-satisfy these instances dependencies by e.g. creating new instances.

When constraints imposed by previously made choices prevent the instantiation of a required subcomponent, it is necessary to backtrack and modify some of the previously made choices. This is presently done by using fixes[1] that are explicitly represented in the domain model. This is an area for improving the truth maintainer. At present this kind of situation should be avoided as much as possible, by structuring the domain model with this in mind.

The truth maintainer can only guarantee consistency if it can analyze all the constraints in the domain model. This means that OBELICS cannot allow arbitrary Prolog expressions as constraints. They must be in the format described in section 5.2.

5.6. The User Interface

OBELICS PSM interacts with the user through a GUI. The GUI has two main parts:

- During the configuration there are choices of design alternatives that have to be made and instances' user-specifiable attribute values have to be set. This part of the GUI is responsible for presenting the alternatives in an understandable way to the user and for reporting the choices to the PSM. This part is also responsible for presenting explanations of design decisions made by the PSM.

[1] A fix is knowledge of what to do in case of failure

- A part that makes it possible for the user, at all times during a configuration session, to view and edit the current state of the configuration, e.g. what component class instances have been included so far.

5.7. Generating Descriptions of the Configurations

As mentioned earlier, a hierarchy instance represents a configuration, i.e. an artifact. A hierarchy instance is not a very presentable description of an artifact. The required description is usually a textual specification and a set of drawings. Other forms of texts may be component lists and scripts to be used by other applications etc. To support the generation of this type of output it is possible in OBELICS to define mappings between hierarchy instances and the descriptions of the configurations that they represent.

This mapping is implemented using two additional object classes; text and drawing objects.

TEXT name
Describes: An expression stating what components, if any, the object can describe.
Subtexts: Texts that may be included.
Pretext: A text fragment to be included ahead of the subtexts. The fragment may include variables whose values are to be found in the component instances the object describes.
Posttext: A text fragment to be included after the subtexts.

DRAWING name
Type: A classification of the object.
Describes: An expression stating what components, if any, the object can describe.
Constraints: The constraints that must be satisfied if the drawing is to be included.
Drawing: Knowledge of how the object is to be drawn.
Composition: Knowledge of how subdrawings are to be inserted.
Subdrawings: The set of drawing types that can be inserted.
Specializations: The set of possible specializations of the drawing.

For example:

```
software_text::{
   super(compsys_text)&
   describes(all,(compsys,soft),true)&
   pretext(["
            \section{Software}
            \begin{itemize}"])&
   subtexts([cad_text,word_text,com_text])&
   posttext(["
            \end{itemize}"])
}.
```

The `describes(all,(compsys,soft),true)` statement means that the text class describes all `soft` instances descending from the current `compsys` instance. The

`pretext` is to be inserted ahead of the subtext instances, e.g. `cad_text` instances, and the `posttext` after.

To generate a drawing describing a hierarchy instance from a drawing hierarchy the drawing generator starts with the top object of the drawing hierarchy and:

1. **Create Instances** Create an instance of the drawing object for each component instance that the drawing object describes. Record in each drawing instance the knowledge of which component instance it describes.
2. **Evaluate Instances** For each instance do
   (a) Generate the frame of the drawing using the instance's drawing knowledge.
   (b) If there are any subdrawings, create and evaluate instances of them and insert the instances in the frame using the instance's composition knowledge.
   (c) If there are any specializations, find one with valid preconditions. Create and evaluate an instance of it.

A similar method is used to generate texts describing a hierarchy instance from a text hierarchy.

6. Conclusions

Although OBELICS is only a prototype its run-time efficiency and the relative ease with which it was implemented demonstrate that Prolog, when extended with tools such as Objects, is very suitable for implementing this type of applications. There are no reasons to use some other, "more efficient", language for developing a fielded version of OBELICS. We believe that the algorithm rather than the runtime performance determines the speed of this kind of application, and in Prolog it is usually easier to find the algorithm, refine it, or replace it with a better algorithm than in e.g. C++.

We have used OBELICS to re-implement DPS, a switchgear configuration tool. The re-implementation was done with only a fraction of the effort required to implement the original DPS. During the re-implementation many of the claimed benefits of OBELICS were corroborated. The domain model was easy to understand and work with. The application could be developed incrementally etc.

In collaboration with Ericsson Telecom AB we have also produced an example of how OBELICS can be used for configuration of telecommunication products. This example shows how OBELICS can work with parameterized functional requirement descriptions, in this case a traffic matrix describing the communication between users. The application takes such a matrix as input and uses it to plan how the users should be connected in an optimal way according to a communication cost function. This plan is then passed to OBELICS' regular problem solving method which does the actual configuring. In this way we have an application that accepts parameterized functional requirement descriptions and produces an optimal configuration from that while we still have the opportunity to freely (within the limits of the constraints in the domain model) modify the result. The point here was to show the ease with which other problem solving methods can communicate with OBELICS.

Another interesting OBELICS application, developed by Hesham A. Hassan (SICS), is a support tool for assisting knowledge engineers in configuring problem solving models from a library of tasks and their corresponding problem solving methods similar to the CommonKADS library [2].

At this time the claimed ease of maintenance has not yet been properly tested in practice, but we still believe that the ease of maintenance is perhaps the greatest advantage of OBELICS.

Acknowledgment

Thanks are due to Klas Orsvärn, SICS, who has contributed to the work, and the ideas behind OBELICS.

REFERENCES

1. Tomas Axling. A tool for developing interactive configuration applications. Master's thesis, Department of Computer and Systems Science, KTH, 1993.
2. J. Breuker and W. Van de Velde, editors. *The CommonKADS Library: reusable components for artificial problem solving.* IOS-Press, Amsterdam,Tokyo, August 1994.
3. D. Brown and B. Chandrasekaran. Knowledge and control for a mechanical design expert system. *IEEE Computer*, 19:92–100, July 1986.
4. David Christopher Brown. *Expert systems for design problem-solving using design refinement with plan selection and redesign.* PhD thesis, The Ohio State University, Columbus, OH 43210, 1984.
5. F. Frayman and S. Mittal. Cossack: A constraint-based expert system for configuration tasks. In *Proceedings of the 2nd International Conference an Applications of AI to Engineering*, 1987.
6. Sabine Geldof. Kitchen design: Knowledge level analysis. Technical Report D/WP2/2, Free University of Brussels, 1991.
7. S. Haridi. Prolog objects in SICStus Prolog library manual. Technical Report T93:02, SICS, 1993.
8. John McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.
9. S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *Proceedings of the 11th IJCAI*, 1989.
10. R. Pfeifer and P. Rademakers. Situated adaptive design: Toward a new methodology for knowledge system development. Technical Report 3, Free University of Brussels, 1991.
11. D.B. Shema, J.M. Bradshaw, S.P. Covington, and J.H. Boose. Design knowledge capture and alternative generation using possibility tables in Canard. In *Proceedings 4th AAAI Knowledge Acquisition Tools for Expert Systems Workshop*, 1989.
12. Michael Vitins. A prototype expert system for configuring technical systems. In *Knowledge-based Systems in Industries*, Baden-Dättwil, Switzerland, 1986. Brown Boveri Research Center.
13. T. Yokoyama. An object-oriented and constraint-based knowledge representation system for design object modeling. In *Proceedings of the Sixth IEEE Conference on AI Applications*, 1990.