Towards A Configuration Specification Language Based On Constraints Logic Programming

Mahmoud Rafea

Central Laboratory for Agricultural Expert Systems (CLAES)

El Nor St., Dokki, Giza, 12311, Egypt

mahmoud@esic.claes.sci.eg

From: AAAI Technical Report WS-99-05. Compilation copyright © 1999, AAAI (www.aaai.org). All rights reserved.

Abstract

In this paper, we concentrate on describing a configuration specification language suitable for most of configuration problems. The language and the generated configuration task code are based on the key-component approach. The implementation of the language is based on the CLPFD library of SICStus Prolog. The language consists of a number of Prolog clauses, which are compiled into CLPFD constraints. The problem solver is the engine of the Constraint Logic Programming Finite Domain (CLPFD). A challenging configuration example from the classics is used to demonstrate the efficiency of the language implementation.

Introduction

A component, which is sometimes called a part, is the building block of any system. It may be either configurable (complex) or non-configurable (primitive). Complex component which needs to be configured itself, can be considered as a configuration problem structurally isolated from the original configuration problem, but, functionally part of it. In this way the configuration task becomes scalable. The definitions of the configuration task (Mittal and Frayman, 1989; Najmann and Stein, 1992, Buchheit, 1994; Stefik, 1995) do not differentiate between complex and primitive components.

In this paper, we present a language suitable for most of configuration problems. The language supports scalable configuration systems. In other words, The problem is structured as complex and primitive components. The idea is to help the application developer to rapidly and efficiently encode the system specifications. The language code is then compiled into constraints, which are efficient but difficult to encode manually. The compiled code can be integrated with a suitable interface for user interactions.

The work of this paper is based on a challenging configuration example given by Stefik, 1995. The example is shown in Figure 1. The configuration specification language that describes this example is shown in Figure 2. The corresponding compiled constraints are shown in Figure 3 and Figure 4. In this paper, we concentrate on describing the proposed configuration specification language, section 2, and the corresponding compiled constraints, section 3.

The configuration specification language

In this language, the configuration task is based on the key-component approach. The problem solving of the configuration problem is based on Constraint Logic Programming (CLP), using the Finite Domain type (FD). The implementation of the language is based on the CLPFD library (Carlsson et al, 1997) of SICStus Prolog version 3.7.1.

The language is simple and declarative. The system specifications are encoded as Prolog clauses, which represent the language statements. The encoded system specifications represent the problem knowledge base. The Prolog clauses of the language can be classified into:

- Task specifications clauses
- Functional hierarchy specifications clauses
- Parts sub-model specifications clauses
- Solution Optimization clauses

Notice that the arrangement sub-model cannot be generic, but it can be defined for a particular type of arrangement. In the given example, the cabinet arrangement model is directly coded into constraints. A generic language constructs for cabinet arrangement needs further work.

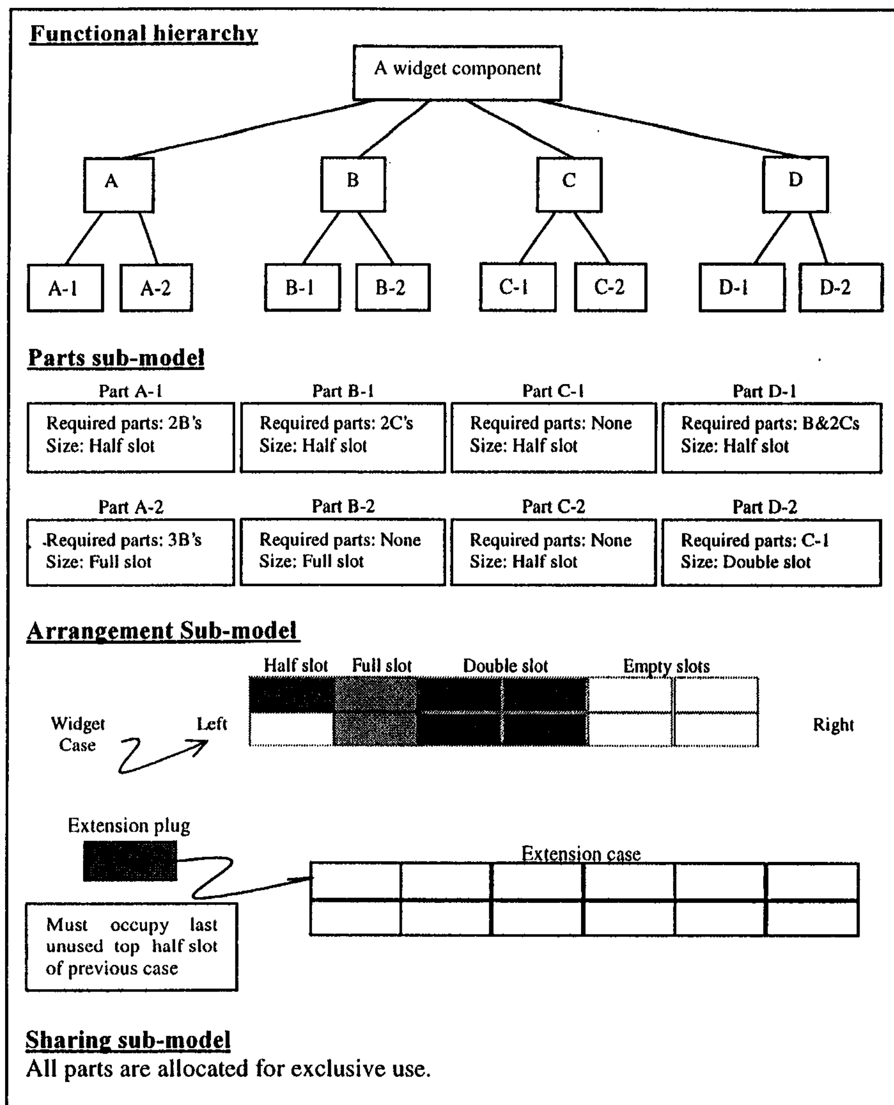

Figure 1: Configuration example showing the widget model which is written by Stefik, 1995

Task specifications

The main purpose of the task specification is to describe the system in terms of its complex components. The complex component is termed 'sub-system'. Consequently, the configuration task is divided into smaller configuration tasks or sub-tasks. This helps in:

- Encoding of the system in a structural way.
- Facilitates the compilation process by compiling a subsystem at a time.
- Enhances the system performance especially for large systems.

To define a system, the following construct is used:
system <system> : [<subsystem>, ...].

Where the non-terminal <system> is a Prolog atom denoting the name of a system and the non-terminal <subsystem> is a Prolog atom denoting the name of a subsystem. The list: "[<subsystem>, ...]" should contain all the subsystems that constitute the system <system>. Notice that if there are different alternatives for declaring the system <system>, the system/1 declaration can be repeated.

For each subsystem two constructs need to be defined:
1. **subsystem** <subsystem>: [<key-component>, ...].

2. <subsystem> **init_components** [<number> - <key-component>, ...].

Where the non-terminal <key-component> is a Prolog atom denoting the name of a key-component. The list: "[<key-component>, ...]" is a Prolog list containing the names of all key-component that constitute the subsystem <subsystem>.

The clause init_components/2 defines the list of key components that are not required by any other components, but are part of the subsystem. The non-terminal <number> is an integer value denoting the number of instances needed from the associated key-component in configuring the subsystem. Notice that there is no restriction on the length of the list parameter for both substsem/2 and init_components/2 clauses.

```
% Task Specification
% ------------------------
system widget:[widget_sys].
subsystem widget-sys:[a,b,c,d].
widget_sys init-components [1-a,1-d].

% Functional Hierarchy
% ---------------------------
a consist_of [a1,a2].
b consist_of [b1,b2].
c consist_of [c1,c2].
d consist_of [d1,d2].

% Parts Submodel
% -------------------
a1 require [keyComp(b,2)].
a2 require [keyComp(b,3)].
b1 require [keyComp(c,2)].
b2 require [].
c1 require [].
c2 require [].
d1 require [keyComp(b,1),keyComp(c,2)].
d2 require [comp(c1,1)].

a1 consume [resource(case,slot,1)]. % Half slot
a2 consume [resource(case,slot,2)]. % Full slot
b1 consume [resource(case,slot,1)].
b2 consume [resource(case,slot,2)].
c1 consume [resource(case,slot,1)].
c2 consume [resource(case,slot,1)].
d1 consume [resource(case,slot,1)].
d2 consume [resource(case,slot,4)]. % Double slot

% Solution Optimization
% ---------------------------
a max 10.   b max 10.   c max 10.   d max 10.

price(a1, 10).   price(a2, 15).   price(b1, 10).
price(b2, 15).   price(c1, 10).   price(c2, 15).
price(d1, 10).   price(d2, 15).

maxPrice(widget_sys, 100).
maxResources(widget_sys, case, slot, 12).
```

Figure 2: The widget model written by the proposed configuration specification language

It should be remarked that the subsystem could not have more than one definition. To express the different subsystem alternatives, the subsystem name can be suffixed or prefixed by a suitable word and a new instance of system/1 is defined with the new subsystem.

Functional Hierarchy

The purpose of the functional hierarchy is to define the key components that constitute the subsystem. In fact, a key-component represents a class that can be specialized to a particular component. The corresponding generated constraints ensure that the configured sub-system contains all the needed components.

For each key component the following construct must be defined:

<key-component> **consist_of** [<component>, ...].

Where the non-terminal <component> is a Prolog atom denoting the name of a component.

Parts sub-model

This model ensures that the configured system contains all the components that make a functioning system. Also, It ensures that the number of components neither exceeds nor less than the defined requirements. Two constructs are currently used. The first construct, require/2, defines the dependencies between components. Each primitive component must be represented even if it does not require any other component for its functioning. It can be defined using the following syntax:

<component> **require** [keyComp(<key-component>, <number>), ...]. or
<component> **require** [comp(<component>, <number>), ...]. or
<component> **require** [keyComp(<key-component>,<number>), ..., comp(<component>, <number>), ...]. or
<component> **require** [].

The second construct, consume/2, defines needed resources. For each primitive component the following construct must be defined:
<component> **consume** [resource(<resource-provider>, <resource>, <number>), ...].

Where the non-terminal <resource-provider> is a Prolog atom denoting the name of a component which provides the resource <resource>. The non-terminal <resource> is a Prolog atom denoting the name of a resource.

```
widget_sys :: {
attributes([components([]), resources([]), total_res([]), total_price([])]) &

key_comp_const(a,A) :- A = 1 &
key_comp_const(d,A) :- A = 1 &

comp_const(widget_sys,A,B,C,D) :-
  :(A#>0#/\(B#>0#/\(C#>0#/\D#>0))) &
comp_const(a,A,B-C,D-E) :-
  :((B#=1#/\D#=0#\/B#=0#/\D#=1)#/\(C#=A*B#/\E#=A*D)) &
comp_const(b,A,B-C,D-E) :-
  :((B#=1#/\D#=0#\/B#=0#/\D#=1)#/\(C#=A*B#/\E#=A*D)) &
comp_const(c,A,B-C,D-E) :-
  :((B#=1#/\D#=0#\/B#=0#/\D#=1)#/\(C#=A*B#/\E#=A*D)) &
comp_const(d,A,B-C,D-E) :-
  :((B#=1#/\D#=0#\/B#=0#/\D#=1)#/\(C#=A*B#/\E#=A*D)) &

req_const(a1,A,B,C) :-
   :((C#>0#<=>A)#/\C#=B*2*A) &
req_const(a2,A,B,C) :-
   :((C#>0#<=>A)#/\C#=B*3*A) &
req_const(b1,A,B,C) :-
   :((C#>0#<=>A)#/\C#=B*2*A) &
req_const(d1,A,B,C,D) :-
   :((C#>0#<=>A)#/\C#=B*1*A#/\((D#>0#<=>A)#/\D#=B*2*A)) &
req_const(d2,A,B,C) :-
   :((C#>0#<=>A)#/\C#=B*1*A) &

res_const(a1,A,B,C) :-
   :(C#=B*1#/\(C#>0#<=>A)) &
res_const(a2,A,B,C) :-
   :(C#=B*2#/\(C#>0#<=>A)) &
res_const(b1,A,B,C) :-
   :(C#=B*1#/\(C#>0#<=>A)) &
res_const(b2,A,B,C) :-
   :(C#=B*2#/\(C#>0#<=>A)) &
res_const(c1,A,B,C) :-
   :(C#=B*1#/\(C#>0#<=>A)) &
res_const(c2,A,B,C) :-
   :(C#=B*1#/\(C#>0#<=>A)) &
res_const(d1,A,B,C) :-
   :(C#=B*1#/\(C#>0#<=>A)) &
res_const(d2,A,B,C) :-
   :(C#=B*4#/\(C#>0#<=>A)) &

start :-
  :domain([A,B,C,D,E,F,G,H],0,1),
  I=[J,K,L,M,N,O,P,Q],
  R=[S,T,U,V,W,X,Y,Z],
  key_comp_const(a,A1),
  key_comp_const(d,B1),
  comp_const(widget_sys,A1,C1,D1,B1),
  comp_const(a,A1,A-E1,B-F1),
  comp_const(b,C1,C-G1,D-H1),
  comp_const(c,D1,E-I1,F-J1),
  comp_const(d,B1,G-K1,H-L1),
  req_const(a1,A,E1,M1),
  req_const(a2,B,F1,N1),
  req_const(b1,C,G1,O1),
  req_const(d1,G,K1,P1,Q1),
  req_const(d2,H,L1,R1),
  res_const(a1,A,J,S),
  res_const(a2,B,K,T),
  res_const(b1,C,L,U),
  res_const(b2,D,M,V),
  res_const(c1,E,N,W),
  res_const(c2,F,O,X),
  res_const(d1,G,P,Y),
  res_const(d2,H,Q,Z),
  :clp_sum_list([M1,N1,P1],C1),
  :clp_sum_list([O1,Q1],D1),
  :(J #= E1),  :(K #= F1),  :(L #= G1),  :(M #= H1),
  :clp_sum_list([I1,R1],N),
  :(O #= J1),  :(P #= K1),  :(Q #= L1),
  :(S1 #= S+(T+(U+(V+(W+(X+(Y+Z))))))),
  :(S1 #> 0),
  :labeling([],I),
  :labeling([],R),
  :labeling([min],[S1]),
  :(T1 is J * 10 + (K * 15 + (L * 10 + (M *15 + (N *
10 + (O *15 + (P * 10 + Q * 15))))))),
  set(components(I)),
  set(resources(R)),
  set(total_res(S1)),
  set(total_price(T1)) &

super(widget)
}.
```

Figure 3: The compiler generated constraints for the specifications of subsystem 'widget_sys'.

```
widget :: {
main :-
   widget_sys :: start,
   widget_sys :: get(total_res(A)),   :(A=<12),
   widget_sys :: get(total_price(B)),:(B=<100),
   :true &

super(utility)
}.
```

Figure 4: The compiler generated constraints for the specifications of system 'widget'.

Solution optimization

An optimal configuration solution is the main objective for a configuration task. Optimization is usually related to system price and resources utilization for a particular specification. In the current implementation three optimization parameters can be defined. The first parameter determines the maximum number of a component in a subsystem. It can be defined using the following syntax:

<component> **max** <maximum number>

The second optimization parameter determines the maximum number of a resource in a subsystem. It can be defined using the following syntax:

maxResources(<subsystem>, <resource-provider>, <resource>, <maximum number>).

The third optimization parameter determines the subsystem with the best price. Two types of clauses need to be defined. The first is the prices of components and the second is the maximum price of the subsystem. They can be defined using the following syntax:

price(<component>, <price>).
maxPrice(< subsystem>, < maximum price>).

The generated configuration constraints

The system specification knowledge base is compiled into CLPFD constraints. Those constraints are encapsulated inside objects. If the configuration task includes more than one system defined by system/1 clause, the generated objects will be represented as a forest. Each system top object is named after the system being configured and saved in a separate file. Also, each subsystem is represented in a separate object, which inherits the system top object and saved in a separate file. The constraints of each subsystem are encapsulated in the subsystem object. The object and the file are named after the name of the subsystem.

Each clause in the specification knowledge base is compiled into a CLPFD constraint. The constraints generated from the specification given in Figure 2 are illustrated in Figure 3 and Figure 4. Some clauses are mapped to a particular constraint while others are combined together in one constraint. The mapping between Prolog clauses and the generated constraints are summarized in table 1.

It is important that calling the constraint main/0 of the object 'widget' (Figure 4) will give the best possible solutions. The solution is extracted from the attributes defined in each subsystem object. Each subsystem solution consists of:

- The total price is extracted from the attribute total_price/1.
- The number of total resources needs to allocate is extracted from the attribute total_res/1.
- The components are extracted from the attribute components/1.
- The resources needed by each component are extracted from the attribute resources/1.

No.	Prolog clause	CLPFD constraint predicates
1	init_component/2	key_comp_const/2
2	subsystem/1 and consist_of/2	comp_const/X
3	require/2	req_const/X
4	consume/2	res_const/4
5	1, 2, 3, 4 and max/2	start
6	system/1, price/2, maxPrice/2, and maxResources/4	main

Table 1: Summarizes the mapping between Prolog clauses and the compiler generated constraints

Conclusion

Compiling the specification, given in Figure 2, produced the constraints, given in Figures 3 and 4. Running, the generated code, it outputs three solutions. Figure 5 depicts those three solutions. One of those solutions, solution I, is the solution given in Stefil, 1995. Interestingly, the two other solutions are not documented and are better than the documented one. This points to that the use of CLPFD in configuration will help in generating the best solutions that may not be recognizable by human expert professionals.

A-1	B-1	C-1	C-1	D-2
B-1	C-1	C-1	C-1	

Solution I

A-1	B-2	B-2	B-2	C-1	D-1
				C-1	

Solution II

A-1	B-2	B-2	B-2	C-2	D-1
				C-2	

Solution III

Figure 3: Configuration solutions allocated to a widget cabinet

The language is missing some important features that will be considered in future work. These features are:

- Specialized libraries and language statements for the arrangement sub-model specifications.
- Considering the connections sub-model specifications.
- Considering the sharing sub-model

References

Buchheit, M., Klein, R., and Nutt, W. 1994 Configuration as model construction: The constructive problem solving approach. In Proceedings of the 3rd International Conference on Artificial Intelligence in Design, AID'94.

Carlsson, M., Ottosson, G., and Carlson, B. 1997 An Open-Ended Finite Domain Constraint Solver, in book: Programming Languages: Implementations, Logics, and Programming, Editors: H. Glaser and P. Hartel and H. Kucken, Lecture Notes in Computer Science, Volume: 1292, pages: 191--206, Springer-Verlag, Southampton.

Mittal, S., and Frayman, F. 1989 Towards a generic model of configuration tasks, IJCAI, Vol. 2, pp.1395-1401.

Najmann, O. and Stein, B. 1992 A theoretical framework for configuration. In Proceedings of the 5th IEAAIE.

Stefik, M. 1995 Configuration, in book: Introduction to knowledge systems, Morgan Kaufmanns Publishers.