From: AAAI Technical Report FS-96-03. Compilation copyright © 1996, AAAI (www.aaai.org). All rights reserved.

Functional and Structural Reasoning in Configuration Tasks

Staffan Pernler
Ericsson Telecom AB
S-126 25 STOCKHOLM
Sweden
E-mail: etxstap@kk.ericsson.se

Max Leitgeb
Logikkonsult NP AB
S-118 48 STOCKHOLM
Sweden
E-mail: max@lk.se

Abstract

A configuration problem within the technical domain is the problem of putting together different parts into a complex system, given a description of the possible parts and the functionality of the required system. While working with our master thesis (Leitgeb, Pernler 95) at SICS[1] during 93 and 94, we found only a few configuration systems extracting the functional reasoning from the structural.

We herein motivate the need for functional reasoning as a basis to find meaningful control structures separated from the application domain knowledge, and that this, together with a separated structural approach to describe the possible parts of the artefact, might be a neat solution.

We have defined a model for representing configuration problems in two dimensions:

Functional knowledge of the function of the artefact.

Structural knowledge about structure and possible parts of the artefact.

We have named this model the FAST model—the Functional And STructural model, and tested it on a real world application.

Functional and Structural Reasoning over the Domain Knowledge

When we create an object through synthesis[2], what do we strive for? In reality, we are not just looking for an object - no, we search for an object providing a certain number of functions in its environment. The object is characterised both by the functions it provides to its environment, and the internal functions it uses to fulfil the external functionality - without its internal functions, the object can not perform the external ones.

In classical science, when talking about a physical object, we can characterise it both through its structure (its parts) and its functions (Pirsig 74). We claim that the most important characterisation is the functional - in synthesis, we search for parts because of their functions. We are not primarily concerned with aesthetical objects - we do not want any structural part not motivated by a function. Each part realizes one or several necessary functions, or gives the overall system a required quality. Our thesis is that no structural description within the synthesis domain is justified by itself.

Our willingness to use a structural description and a structural reasoning as the basis for synthesis, comes from our view of the object - we can see its structure, touch its parts, divide it into pieces. You can hardly see all the external and internal functions of a motorcycle, for example. To divide its functions into sub-functions requires deep knowledge of the function of the motorcycle. Still, if you want to repair a motorcycle, or customize your own motorcycle, you can no longer use a pure structural reasoning. It is, if not impossible, at least extraordinarily unrealistic to believe that a pure structural reasoning[3] will effectively lead to a solution.

There are other areas where functional reasoning is promising, such as for explanations, creativity and understandability but we will not go deeper into those matters in this paper.

The Functional And STructural problem solving model - FAST

The functional and structural problem solving model is a model for representation and computation in knowledge-based systems for configuration purposes. We believe that a structural decomposition has to be made in a way suitable for efficient use by higher levels of reasoning. For this to be possible and for the description to be reusable we strive for a declarative representation of the functional and the structural knowledge.

Our problem solving model is as follows: when an artefact (object) is to be configured, one start off with the functional description. It has the main function of the object as a top level node. Then, by evaluating

[1] Swedish Institute of Computer Science, PO Box 1263, S–164 28 KISTA, Sweden

[2] Such as configuration, design etc.

[3] Without functional reasoning compiled into it (for example as "control knowledge")

the functional description the configuration will be expanded in a top-down style and the functional parts will be refined successively. When a physical part is needed at the functional level, we reference an object class at the structural level and then the object class will in its local context decide how the current instance shall be configured. The coupling from the functional level down to the structural level is a mapping in order to make the functional reasoning more independent from the structural.

Functional representation & reasoning

In the domain of synthesis, a functional decomposition can be based on the view that the overall function of the system can be decomposed into sub-functions that the system must perform. If we bind the functional decomposition close to the structural decomposition, we risk to disregard functions not directly given by a certain structure, but being based on a dependency not explicit in our structural decomposition. We also risk to loose a broad view on what the system has to fulfil.

The functional decomposition helps to realise the internal and external functions that the system must provide. It uses sub functions as a way to realise a desired function. The functions in the decomposition are of two kinds;

Combined functions A combined function is a function that, to some extent, uses sub-functions to realise its own function. It might reference parts at the structural level.

Atomic functions An atomic function is realised entirely through references to parts in the structural decomposition.

The realisation of a function is a plan of how to satisfy one or several functional demands. A plan is an abstract (reusable) description of how to carry out such a task. It could consist of complex reasoning, as well as a simple description of a demand.

In our functional decomposition, relations can be of two different types;

Has-function The has-function is a way to realise a function through sub-functions. This is a coupling between nodes in the functional hierarchy.

Maps-to The maps-to gives a way to implement a function through communication with parts in the structural decomposition. This means that we use a physical part to realise a function. This is not really a node in the functional decomposition, it is merely a mapping function from the functional to the structural level.

Structural representation & reasoning

The structure of the object to be configured can be described on different levels. A strictly connection oriented description would use direct couplings only. We propose a more abstract (more function oriented) structural description.

Using a motorcycle as an example of a configurable object, consider the following question. Do you believe a motorcycle to have a structural part called a "motor"? If so, considering a decomposition based on "is connected to", where is this so called "motor"? Is the camshaft a part of the motor or the transmission system, and is this obvious from the structural description? If it is not obvious, is the decomposition then natural? Have we gained in understandability and lucidity?

The coupling-based decompositions fail in being "complete" in some sense. We need some kind of abstraction possibility, where we consider structural objects constructed from other objects. Here we will return to the borderline between functional and structural decompositions. A motor is motivated by the need for mechanical power. This mechanical power (which is a function) is then transmitted by the transmission system (transmission is a function) to the wheels etc. So, what we have created, is a semi-functional structure, based on the relation "has-part".

The structural decomposition is made in order to simplify the representation and maintainance of the description of physical objects. It is a way to represent the object classes, couplings and dependencies between them. The decomposition can be viewed as a hierarchy where the abstract nodes (nodes with sub-parts) are complex object classes and the leaf nodes are atomic object classes. There are three different types of part-relations;

Has-part This corresponds to a coupling between a part and its subpart. It is a way to decompose an object class into sub-object classes, each one describing a part of the object.

Is-a This corresponds to possible real world realisations of an object class, that is, a physical object.

Super Corresponds to inheritance from an abstract object which is used for keeping common domain knowledge.

The structural level will represent the structural objects in a way that each object can be referenced. The semantics is that each object in the structural hierarchy can be instantiated and when instantiated, the object and its sub-parts are configured in accordance with the local context at the time of the reference. The local context is defined by functional and structural requirements on the object to be configured. After instantiation the initial configuration can be further refined by the functional reasoning or relations to other structural parts.

Relations and restrictions

In functional and structural reasoning restrictions and relations between functional and structural objects may occur. Implicit restrictions are those expressed by the structural decomposition, static knowledge limiting the set of possible solutions. Explicit restrictions

are dynamic, complementing the structural decomposition with relations not possible to express statically, and while receiving new case specific restrictions dynamically evaluating the static knowledge towards a solution. We have implemented these as constraints.

Constraints are a declarative way of expressing restrictions and relations. Configuration can partly be viewed as a constraint satisfaction problem - we know the possible parts, and we search for a composition fulfilling a number of different restrictions. But we must still be aware of the fact that explicit constraints are just one possible implementation of restriction knowledge.

In our solution constraints can be defined between different objects that are in the sub-hierarchy of an object and its attributes – both from the functional reasoning to the structural realisation, and between structural parts.

We would like to think of constraint solving as limiting a value domain according to other value domains and the restrictions between them. A constraint is merely a formulation of a restriction, which implicitly cuts the value domains of the included attributes in order to create consistency amongst the constraints. The method of using constraints combined with fixes, as done by, for example (Marcus 89) in VT, is just a mid-way solution between rules and pure constraint solving.

The FAST framework

We will here describe the framework which we implemented by extensions to a kernel language in order to get a language with desirable properties. The language we used as kernel was SICStus 2.1 (SICStus 93), which is an effective Prolog implementation with object oriented facilities from SICS (Swedish Institute of Computer Science). The object oriented part was used to achieve a representation with inheritance, dynamic instantiation, and the object oriented view concentrating on objects and their relations. Especially we used the modularisation included by the object oriented paradigm and the sub/super relation needed for structure sharing.

Functional problem solving objects

By separating the functional objects in two parts; a mapping and a reasoning part, they can be specified in a way that gives full usability and re-usability.

The mapping part deals with mapping attributes, objects etc. to structural level objects and by using abstract names given in the mapping, make the functional reasoning generic. When applying the functional object to another domain, the only thing that has to be changed is the mapping.

The reasoning part is, as described above, a way of defining a generic functional reasoning. How that is implemented is today much left to the user. He can freely combine the mechanisms of SICStus Prolog and our functional framework including the constraint solver. A functional object can be of three kinds;

1. An abstract node, referring to other nodes. This could be viewed as decomposing a function into sub-functions.

```
function_elevation::{
      decompose(Instance):-
            function_suspension(Instance),
            function_safety(Instance),
            function_transport(Instance) }.
```

   A reference to the decomposition is made by referencing the method decompose, as in `function_elevation::decompose(Instance)`, where `Instance` is the instance of the structural object handled.

2. An atomic function realises a function through some functional reasoning.

```
function_balance::{
  set_equal_weight(Obj_1_weight, Obj_2_weight):-
     constrain([Obj_1_weight,=,Obj_2_weight])
  mapping(Inst):-
     Object_1 = Inst<>Car_assembly::weight,
     Object_2 = Inst<>Counterweight::weight,
     set_equal_weight(Object_1, Object_2) }.
```

   This example shows a limited functional reasoning. It introduces a constraint that the weight of two balanced objects must be equal. It is referred to by a call to the method `balancing::mapping(I)` with I set to the instance name of the structural object.

3. Or a mix of the above. It is possible to have decompositions and mappings in the same functional object.

Structural problem solving objects

Complex objects

An object class that has one or several has-part relation(s) is defined as a complex object. When instantiated, it will automatically create instances of its necessary sub-part(s) and during evaluation it might increase or decrease the number of sub-parts.

Generally:

```
<name> ::  {
  super(complex_object) &
  has_part(<object class>, <nr>) &
  <other_attributes> }.
```

Example:

```
motor ::  {
  super(complex_object) &
  has_part(generator,1)}.
generator ::  {
  weight(10) }.
```

The above declaration will introduce the following;

- automatic instantiation of a declared part (or parts - as many as declared or calculated, <nr> argument above).
- automatic access to a part through a new operator - the with_part (reference) operator, written as <>, and the access to its instance is done through the abstract name of the part.

  Example of an execution in SICStus:

```
:- motor::new(motor_One).
    A new motor instance is created.
yes
:- motor_One<>generator::weight(A).
A=10 ?
    It has a generator instance, with a method
    weight which is 10.
yes
```

Constrained objects

We constructed constrained objects with access to a constraint solver and a mechanism to propagate changes. For complex objects this propagation will also call its parts to ask them to perform propagation of a change. Constraints can be formulated in three ways:

- between the internal attributes of an object.
- between attributes either internal or belonging to one of its parts (or its parts part, etc.).
- from the funtional reasoning concerning two possibly structurally independent objects with a functional relation.

The domain of an attribute can be an interval, a finite domain of numbers or atoms, or a restricted form of table.

Generally:

```
<name> ::  {
  super(constrained_complex) &
  initial_constraints(<constr_list>) &
  <other_attributes> }.
```

Example:

```
motor ::  {
  super(constrained_complex) &
  has_part(generator,1) &
  weight([[0],[+]]) &
  own_weight([10]) &
  initial_constraints([
    [weight,=,own_weight,+,generator::weight]
  ]) }.
```

The above declaration will introduce the following;

- The set of `initial_constraints` will be a part of the constraint set for this part. It will be evaluated for consistency initially and when new constraints are introduced. Using the `generator` in the example of complex objects above, a `motor` instance will after initial constraint solving have a value domain of [20] for the attribute `weight`[4].
- The super object `constrained_complex`[5] will allow the functional reasoning to act on the instances of the object and its parts during constraint solving.

Model objects - classifiers

If an abstract object class is to be realised through one of several models, it is called a classifier object. Those models (realisations of the classifier object) that does not fulfil the constraints on the classifier object are removed. Consider the example above, with an object class motor. There might be several different motors, sharing the same general construction with a generator, weight etc. We choose an abstract motor, which will keep track of its possible instances, and remove instances according to the constraints on the abstract motor. To make this possible, we introduced the general object classifier, and a relation `is_a`, making an object a model (realisation) of a classifier object.

Generally:

```
<name1> ::  {
  super(classifier) &
  <common_attribute> &
  <attribute1> &
  ... }.
<name2> ::  {
  is_a(<name1>) &
  <attribute1> &
  ... }.
```

Example:

```
motor ::  {
  super(classifier) &
  super(complex_object) &
  has_part(generator,1) &
  weight([10]) &
  horsepower([]) }.
m18 ::  {
  is_a(motor) &
  horsepower([18]) }.
m25 ::  {
  is_a(motor) &
  horsepower([25]) }.
```

The above declaration will introduce the following;

- Common attribute declarations, saying what every `motor` has such as a `generator` and a `weight`.
- Abstract attribute declarations, like `horsepower` in motor above, that every motor must have a value domain for.

[4]The attribute weight is initially declared to range from zero to plus infinity.

[5]An object which is of the class `constrained_complex` is both complex (ie. has sub-parts) and may be constrained.

- The is_a relation/declaration, saying that an object is a concrete object of a certain object class, such as m18 and m25 above.
- During the execution, a classifier object will have an attribute models, keeping all possible concrete objects which are not yet out-constrained, and when an attribute of the classifier is updated, all models will automatically be checked to see if they still are part of the set of possible solutions. If all concrete objects are out-constrained, we will fail.
- If a new instruction is made on the motor object above, the instance would after this new have an attribute models, containing a list of m18 and m25, and its attribute horsepower would be a finite domain with 18 and 25. If the horsepower was constrained to be more than 18, 18 would disappear from the finite domain of possible horsepower for the classifier motor and m18 would disappear from models.

Evaluation

Our current implementation can express the VT problem, an elevator configuration problem described by (Yost 93). VT has been used as a benchmark problem for configuration systems. Due to combinatorial explosion when all constraints are included, we can only solve it excluding a few (less than three, out of more than 140) constraints. This is of course not satisfying, but a more intelligent constraint solver can probably solve the complete set of constraints, using different constraint solving techniques and more efficient representations of internal data structures etc.

Some mathematical problems remain to be solved, such as the sine function of an interval, though today a superset of the correct intervals can be found. We can express and evaluate finite domains, intervals, and tables, for example. This expressiveness was needed to effectively allow the user to represent constraints as specified in our model. Also, our current implementation only creates functional and structural consistency. It does not select one specific solution, thus a choice-mechanism is needed.

Conclusions

One conclusion is that the complex and cryptic interactions between structural objects are mostly due to functional demands and with a functional decomposition we 'decode' these interactions and relate them to their functional origin, thus clarifying their existence and meaning. We also avoid 'dirty' objects, filled with problem specific functional 'control knowledge', or pseudo-structured 'rule-bases with meta-rules' which often is the case when different types of knowledge are mixed instead of separated.

We have developed a problem solving model, based on a functional and structural domain knowledge specification, with constraint reasoning for dynamic case-specific evaluation. Through an implementation of a framework of general, re-usable problem solving mechanisms and a representation of the VT problem in this framework, we have taken the first steps towards validating the general usefulness and applicability of our idea and its model.

Generally, we felt that the need for domain specific control knowledge was greatly reduced using declarative representation forms such as constraints[6], objects and logic.

Future work

The functional reasoning is presently limited to hierarchical decomposition, abstraction through mapping, with refinement through the usage of plans. We would like to see a more flexible and powerful functional reasoning. For the structural decomposition, we believe an extension of the dynamic creation of complex parts to be a useful extension of our current implementation.

As functional reasoning is quite a new concept, especially separated functional reasoning, it might take time to gain acceptance in the community of configuration system engineers. A new paradigm (Kuhn 62) introduces new view points for knowledge engineers as well as users of knowledge based systems.

References

Chandrasekaran. *Generic Tasks in Knowledge-Based Reasoning: High Level Building Blocks for Expert System Design.* IEEE Expert 1(3), 1986.

Frayman, S Mittal. *COSSACK: A Constraint-Based Expert System for Configuration Tasks.* 2nd International Conference on Applications of AI to Engineering, Boston, August -87.

A Gupta, D Siewiorek. *M1: A Small Computer System Synthesis Tool.* Conference on AI Applications, -90.

Kuhn. *The structure of scientific revolutions.* University of Chicago Press, ISBN 0-226-45803-2.

M. Leitgeb, S. Pernler. *Functional and structural reasoning in configuration tasks.* Master Thesis in Computing Science #66, ISSN 1100 - 1836, Computing Science Dept, Uppsala University, Box 311, S–751 05 UPPSALA, Sweden.

S. Marcus. *SALT: A Knowledge Acquisition language for Propose-and-Revise Systems.* Artificial Intelligence 39 (1989), p. 1 - 37.

S. Marcus, J Stout, J. McDermott. *VT: An Expert Elevator Designer that uses Knowledge-based Backtracking.* AI Magazine, Spring -88.

J McDermott. *R1: A Rule-Based Configurer of Computer Systems.* AI, 19, 1982.

[6] although this, for the constraint part, partly might be an effect of our implementational choice with overall functional constraints being mapped down onto the structural decomposition.

H Nwana, R Paton, T Bench-Capon, M Shave. *Facilitating the development of knowledge based systems A critical review of acquisition tools and techniques.* AI communications, v4, June/Sept 1991.

M. Pegah, J. Stichlen, W. Bond. *Functional representation and Reasoning About the F/A-18 Aircraft Fuel System.* IEEE EXPERT, April 1993.

Robert M. Pirsig. *Zen and the Art of Motorcycle Maintenance.* ISBN 91-7642-353-0, 1974.

D Searls, L Norton. *Logic-based configuration with a semantic network.* Journal of logic programming 1990:8.

J Andersson et al. *SICStus Prolog User's Manual Version 2.1 #8.* SICS Technical Report, T93:01, ISSN 1100-3154.

G. Yost. *Configuring elevator systems.* AI Research Group, Digital Equipment Corporation, 111 Locke Drive, Marlboro, MA 02172.

Terminology

The Configuration Problem

A1 We have a finite domain of object classes and a finite domain of direct[7] relations between these object classes. An instance of an object class is called an object. An instance of a direct relation is called a coupling.

A2 Each object is restricted by the properties of its corresponding class, and each coupling is restricted by the properties of its corresponding relation.

A3 A coupling is a direct structural relation. We define a dependency to be an indirect or functional relation between objects.

A4 In a configuration problem the connectivity (the number of possible couplings) between objects is finite.

A5 The properties of objects, couplings, and dependencies realizes structural and functional requirements on the combination of objects.

A6 External[8] functional requirements restrict the possible couplings and objects.

Remarks:

A1 tells us that all possible object classes and relations are known beforehand and that they constitute finite sets.

A2 claims that the objects and couplings have static semantics - it is not possible to adjust them. If adjustable, we have a (sub) design problem. Note that design is more general than configuration, i.e. configuration is a sub problem of design, even if some design problems in part are configuration problems. For example, it is usual within the design domain to configure an abstract artefact to be designed, and then to design (specify) its parts.

A3 In the configuration domain the direct relations have a structural nature. This separates the configuration domain from related domains (cf. (Leitgeb, Pernler 95)).

A4 settles the fuzzy borderline versus problems in the design area, for example VLSI-design.

A5 describes the consequences of internal functional requirements - the structural and functional properties of objects and their couplings.

Problem knowledge

Knowledge about a specific problem application domain, such as configuration of computer systems. The problem knowledge is the knowledge of Definitions **A1** to **A4** within a specific application domain.

Problem solving knowledge

Can be divided into application domain problem solving knowledge, and general problem solving knowledge. The former is beforehand knowledge of **A5** to **A6**, for example as functional knowledge about objects and their combination(s).

[7]Direct in the sense that it is a relation between two objects, not dependent on any other object to act as a bridge (cf. **A3**)

[8]Internal are the restrictions described in **A5**, external are those given by a user in a specific case (for a specific solution)