AUTOMATING CREATIVE DESIGN

MIHALY LENART
Fachbereich Architektur
Gesamthochschule Kassel
Germany
and
PETER PADAWITZ
Fachbereich Informatik
Universität Dortmund
Germany
and
ANA PASZTOR
School of Computer Science
Florida International University
Miami Florida USA

1. Logic-Based Design

Design problems are inherently difficult to solve. Reitman (Reitman, 1964) and Simon (Simon, 1973) call them *ill-structured* and *ill-defined*, Rittel and Webber (Rittel and Webber, 1973) *wicked* problems. Due to the difficulties in tackling complex design problems, design automation has been limited to routine and detail design. Routine or detail design means to follow a specified schema with expected results. However, recently interest has shifted toward the automation of complex design tasks, in particular tasks which require creativity. By creative design we understand the development of new, unexpected features or solutions. Design is considered to be a search process with vast and complex search space. Creativity is needed if the search space is not well-defined and only a few heuristics are known for guiding the search.

Unfortunately, very little is known about the human creative process. However, certain creative strategies can be distinguished. Moreover, two approaches, those drawing analogies and those employing probabilistic methods, have been implemented for automating design. An example for drawing analogies is case-based design (Navinchandra, 1991) that uses analogies between the given task and design solutions stored in a database. An example for the probabilistic approach are genetic algorithms (Holland, 1975). Genetic algorithms model the evolution process of living organisms or certain aspects of it. While case-based design was developed specifically for tackling design problems, genetic methods were introduced by Bremermann and others to mesh computer science and evolution in the late 1950's and early 1960's. Genetic methods were rediscovered only recently for solving design problems. As opposed to case-based design, genetic methods can tackle design tasks for which no or little previous knowledge is available.

This paper describes a third, logic-based approach emphasizing the cognitive character of the creative process. This new approach has the following distinctive features:

- Parameters, goals, constraints and solutions are represented as sentences of Horn logic with equality (cf. (Padawitz, 1988)).
- High-level programming is carried out in a functional language so that testing and verification is supported by a corresponding prototyping system like *Expander* (cf. (Padawitz, 1992a)), which is tailored to evaluating the theory of functional and logic programs and their data types (cf. (Padawitz, 1992)).
- Creative solutions emerge not only by new search strategies, but also by redefining the search space and thus the data structures used.
- The specification method supports encapsulation, modular decomposition, hierarchical structuring and reusability.

First-order logic has the advantage of being natural and easily comprehensible. The uniformity of the representation supports prototyping. Negotiable parameters or constraints as well as secondary goals can be changed automatically or by user interaction. If the evaluation of emerging solutions reveals relevant features not considered earlier, then any goal, parameter or constraint can be modified accordingly. Take the design of a computer chip where nothing has been specified about the geometry of the components' arrangement, but certain solutions turn out to have some symmetric features. If these features provide advantages over non-symmetrical solutions, they can be incorporated into the specification. On the other hand, solutions might have deficiencies discovered in the evaluation process. Such deficiencies can be corrected again by changing the specification.

As the mathematical basis of formal specification and prototyping we adopt the syntax and *initial semantics* of Horn logic with equality (see e.g. (Goguen et al., 1978), (Ehrig and Mahr, 1985), (Wirsing, 1990)). The initial model of a Horn clause specification SPEC always exists, and the theory of this model, usually called the *inductive theory* of SPEC, yields a suitable "range of discourse"

for describing as well as checking goals, parameters and constraints. Constructing a specification means at first choosing sorts (types), functions and predicates that the design model should provide, i.e. we must develop the *signature(s)* SIG of SPEC. The set of functions falls into constructors from which the design objects are built up and operations for manipulating the objects and retrieving information about them. Constraints and goals are expressed as first-order conjectures over SIG, which the initial model of SPEC is to satisfy.

Functional programs provide the axioms of SPEC. In logic terms, these programs are Horn clauses: recursive conditional equations defining the operations of SIG and recursive implications defining the predicates. Hence axioms do not express arbitrary properties of the model, such as consistency conditions or functional equivalences. Instead, they define the components of SIG. But we admit partial definitions, which can be completed in subsequent reviews of SPEC. Partiality does not prevent us from evaluating the initial model of SPEC by testing or verifying correctness conditions. Correctness proofs are often based on induction w.r.t. the constructors of SIG. The proof checker of Expander (see above) supports testing as well as verification tasks. Proofs and tests often reveal faults, inconsistencies and inefficiencies, which can be eliminated much easier in the prototyping phase than in subsequent implementation phases.

In general, SPEC undergoes several review and improvement steps. SIG is modified by adding new constructors or auxiliary functions or by changing the generic and module structure. Further correctness conditions come up as lemmas needed for proving actual goals. In particular, conjectures used as induction hypotheses must often be generalized. Within a new iteration of the design process, axioms for auxiliary functions are added, partial definitions are completed and functions are changed on certain arguments. *Program transformation* means adapting a design specification to modified goals.

All this includes decisions on how to modularize SPEC and where to establish genericity with regard to changing demands; which primitive types are to be used; which types or functions are made polymorphic or parameterized. This most important part of the design process is illustrated below at a configuration task of filling shelves with objects and assembling them to shelf systems. Since we aim at a prototype to be tested immediately, we choose *Standard ML* (cf. (Appel and MacQueen, 1991)) as our specification language. While assuming a little familiarity with this language we claim that the differences between Horn clauses and ML programs amount to "syntactic sugar" of the latter.

2. Structured specification of shelf systems

The main part of the specification, which is presented completely in (Padawitz, 1993), falls into five modules, namely two ML-functors and three ML-structures.

The structure *Aux* contains auxiliary types and functions used by the other modules. *Aux* provides a datatype of *streams*, which allows us to enumerate the elements of very long list only up to an element, which satisfies a given condition. The example used here is the list of all permutations of a list of objects. Streams are implemented in ML by alternatingly defining and calling a function $s : unit \rightarrow stream$ up to the list element searched for. *Aux* also provides I/O primitives used by actualizations of the functor *Display* (see below).

The functor *Container* takes a structure parameter *Obj* of a signature *Object*. This determines that objects to be put into a container must have a type (*obj*), a length (*le*) and a height (*he*). They must be comparable via an equality relation *eq* and each object must be equipped with a top constraint such that it may be put on top of another object only if the top constraint holds true. Finally, a Boolean value *Aligned* indicates whether or not objects must be aligned when put on top of each other.

```
signature Object = sig type obj
                       val le : obj→int
                       val he : obj→int
                       val eq : obj*obj→bool
                       val topConstraint : obj→bool
                       val Aligned : bool end
functor Container(structure Obj : Object)
      = struct open Obj ...end
```

The structure returned by *Container* provides a datatype *objE*, which embeds *obj* into a "supersort" by adding the constant constructor *undef* to the elements of type *obj*.

```
datatype objE = def of obj|undef
fun Eq(def(a),def(b)) = eq(a,b)
|   Eq(undef,undef) = true
|   Eq _ _ = false
```

The type definition for containers has two levels. The abstract datatype *contAbs* provides the constructors *new* and *add*. *new* denotes an empty container. *add(c,a,x,y)* represents the container constructed from the container c by putting a into c such that *(x,y)* is the position of the leftmost-lowest corner of a in c. *cont* extends *contAbs* by the static attributes of a container c.

```
abstype contAbs = new|add of contAbs*obj*int*int
with type cont = contAbs*(int*int*int)
    fun base(_,(b,_,_)) = b
    fun len(_,(_,l,_)) = l
    fun hei(_,(_,_,h)) = h
    fun New(attrs) = (new,attrs)
    fun Add((a,x,y),(c,attrs)) = (add(c,a,x,y),attrs)
    fun get((new,_),i,j) = undef
    |   get((add(c,a,x,y),attrs),i,j)
        = if inside(a,x,y)(i,j) then def(a)
          else get((c,attrs),i,j)
    and inside(a,x:int,y:int)(i,j)
```

```
        = x≤i andalso i<x+le(a)
          andalso y≤j andalso j<y+he(a)
    fun coord((add(c,a,x,y),attrs),b)
        = if eq(a,b) then (x,y)
          else coord((c,attrs),b)
end (*contAbs*)
```

base(c) returns the least y-axis of the position of an object to be put into c. *len(c)* and *hei(c)* denote the length and height of c, respectively. Further basic operations for building containers and accessing their contents are encapsulated into the abstract datatype definition. As an abstract datatype, *contAbs* hides the constructors *new* and *add* from all functions except from those encapsulated between *with* and *end*. Hence *contAbs* yields the interface for refinements of *Container* insofar as only the functions in the capsule need to be redefined when containers are actually implemented. All functions defined in the sequel do not use *new* and *add* and thus are independent of the actual implementation. Such a refinement interface considerably reduces the verification task put forth by the implementation. In fact, the entire implementation is correct if and only if each function of the implementing datatype is equivalent to its counterpart of the implemented datatype. Since initial semantics (see above) provides the model- and proof-theoretic basis for most ML programs (cf. (Padawitz, 1992)), one may employ a suitable proof checker like Expander (see above) for proving such equivalences.

The following predicates are checked before a new object is put into a container. Note that they do *not* use the constructors of *contAbs*.

```
fun free(c,x,y) = Eq(get(c,x,y),undef)
val occupied = not o free
fun onTop(c,x,y)
    = y = base(c) orelse occupied(c,x,y-1)
fun aligned(c,a,x,y)
    = y = base(c) orelse
      (occupied(c,x,y-1) andalso
       let val def(b) = get(c,x,y-1)
           val (i,_) = coord(c,b)
       in x = i andalso le(a) = le(b) end)
```

For the actual design part of the program we must select a concrete strategy for filling containers. Defining a strategy typically involves setting exit points where the execution of substrategies is stopped and the control is passed over onto a higher level of execution. In ML, an exit is implemented by raising an exception. Resuming the control at a higher level corresponds to handling the exception. The filling process starts by calling the function *fill*, which raises and handles four exceptions via seven auxiliary functions, which altogether make up the filling strategy.

```
exception noFilling and nextPerm and
          Full and Restart of int
fun fill(objs)(bounds)(sizes)
    = let val full = nil
          val first = true
      in fillRest(objs)(full)(sizes)(bounds)(first) end
```

fill tries to enter the objects listed in *objs* into a bounded space of several containers whose lengths and heights are listed in *sizes*. *fillRest* has two further parameters: *full* is the actual list of containers already filled up (while *sizes* represents the current list of empty containers). The Boolean value *first* denotes whether the filling process is started the first time or restarted. A restart occurs if the objects do not fit into the containers when they are entered in the order given by *objs*. Then we compute a new permutation of the list of objects, which have been put into the last container or could not be entered at all, and try to enter these objects in the new order. If necessary, a further container is unpacked, and so on, until we either find a packable order or had to unpack all containers and to try all permutations of the entire object list. In the latter case, the exception *noFilling* is raised and passed over to the caller of *fill*.

Including *fillRest*, the entire procedure consists of seven mutually recursive functions. Most of them access the current filling state, given by the triple *(objs)(full)(sizes)*. Hence they could easily be equipped with constraint parameters, which change the filling strategy locally by causing switches to particular state transformations. For instance, *fillRest* could be equipped with a size check parameter, which skips each filling attempt that would fail because the overall size of *objs* exceeds the overall size of *sizes*. By "injecting" a number of such particular state transformations into the basic algorithm, interactively or automatically, and evaluating the outcome, we might get to a new procedure, which thus would have been obtained as the result of a truely creative process.

Visualizing computed results is, on the one hand, necessary for testing prototypes. On the other hand, implementing corresponding routines seems to contradict the goal of abstraction because one has to cope with machine-dependent I/O primitives. As our case study shows, such a tradeoff can be resolved to a certain extent by using a functor *Display*, which postpones the determination of I/O primitives to the actualization of its parameters.

Our design task involves a further reason for this parameterization. We want to run the filling procedure hierarchically. At first, shelves shall be filled with objects. At second, shelf systems shall be filled with shelves. We get the hierarchy by different actualizations of the functor *Container* (see above). Correspondingly different actualizations of *Display* allow us to visualize the hierarchy. Of course, the representation of a shelf on the screen must be different from the representation of a shelf system containing that shelf!

Display abstracts the necessary I/O primitives into types and values of a structure parameter *Pict* of a signature *Pictures*. Each primitive determines how a specific part of an object shall be represented. *interior* says how its inside part looks like. *left_edge* (*right_edge*, *lower_edge*, *upper_edge*) depicts its leftmost (rightmost, lowest, highest) axes. The *corner* primitive determines

the representation of object corners.[1]

```
signature Pictures
        = sig type obj and objE and cont
              val obj : objE→obj
              val Eq : objE*objE→bool
              val len : cont→int
              val hei : cont→int
              val get : cont*int*int→objE
              val free : cont*int*int→bool
              val interior : obj→int*int→string
              val left_edge : obj→int*int→string
              val right_edge : obj→int*int→string
              val upper_edge : obj→int*int→string
              val lower_edge : obj→int*int→string
              val corner : obj→int*int→string end
functor Display(structure Picts : Pictures)
       = struct open Picts ...end
```

Each of the following two structures SHELF and SHELFSYSTEM provides an actualization of *Container*'s formal parameter *Obj*, a corresponding filling function and a corresponding actualization of *Display*'s formal parameter *Picts*. Shelves may be filled with books or hifi racks. The corresponding top constraint demands that all objects put on top of other objects are books lying on their cover, i.e., their height must not exceed their length.

```
datatype object_type = book|hifi
structure SHELF = struct
structure BookOrHifi
        = struct type obj = int*int*int*object_type
                 fun key(k,_,_,_) = k
                 fun le(_,l,_,_) = l
                 fun he(_,_,h,_) = h
                 fun eq(a,b) = key(a) = key(b)
                 fun topConstraint(_,l:int,h,book)
                     = h≤l
                 |   topConstraint _ = false
                 val Aligned = false end
structure Shelf = Container(structure Obj
                                    = BookOrHifi)
fun fill(objs)(size)
    = let val [c] = Shelf.fill(objs)(1,1,0,0)[size] in c end
structure Strings
        = struct type obj = BookOrHifi.obj
                 open Aux
                 fun interior _ _ = blanks
                 fun upper_edge _ _ = equals
                 val lower_edge _ _ = equals
                 fun left_edge _ _ = "|"^Blanks(2)
                 fun right_edge _ _ = Blanks(2)^"|"
                 fun Str(i:int,_,_,book)
                     = makestring(i)
                 |   Str(_,_,_,hifi) = "HIF"
                 fun corner(a) _
                     = StringM(Str(a))"=" end
structure Display
        = Display(structure Picts = struct
                                 open Shelf
                                      Strings end)
end (*SHELF*)
structure SHELFSYSTEM = struct
structure Shelf
        = struct type obj
                     = int*SHELF.Shelf.cont ref
                 fun key(k,_) = k
                 fun le(_,c) = SHELF.Shelf.len(!c)
                 fun he(_,c) = SHELF.Shelf.hei(!c)
                 fun eq(a,b) = key(a) = key(b)
                 fun topConstraint _ = true
                 val Aligned = true end
structure Shelfsys = Container(structure Obj = Shelf)
fun fill(shs)(size)
    = let val [c] = Shelfsys.fill(shs)(1,1,0,0)[size]
      in c end
```

For displaying a filled shelf system S the position of each shelf in S must be passed over to the function *picture* of *Display*. For this purpose, the key of a shelf is changed from a simple number into the coordinates of its leftmost-lowest corner in S. Hence we come up with a further actualization of *Container*:

```
structure ShelfOut
        = struct type obj = int*int*Shelf.obj
                 fun key(x,y,_) = (x,y)
                 fun le(_,_,c) = Shelf.le(c)
                 fun he(_,_,c) = Shelf.he(c)
                 ...end
structure ShelfsysOut
        = Container(structure Obj = ShelfOut)
fun transform(c,attrs)
    = fold(ShelfsysOut.Add)(triples(c,attrs))
        (ShelfsysOut.New(attrs))
and triples(c)
    = map(fn(a)⇒
          let val (x,y) = Shelfsys.coord(c,a)
          in ((x,y,a),x,y) end)(Shelfsys.Objs(c))
```

transform translates the modification of shelves into a modification of shelf systems. The new representation of shelves allows us to display them at their correct position with the shelf system:

```
structure Strings
        = struct type obj = ShelfOut.obj
                 fun interior (shelf)(i,j)
                     = let val (x,y,(_,ptr)) = shelf
                           val (i,j) = (i-x+1,j-y+1)
                       in SHELF.
                          Display.picture(!ptr)(i,j)
                       end
                 fun upper_edge _ _ = "+++"
                 val lower_edge = interior
                 fun left_edge _ _ = "# "
```

[1]The actual display specification given in (Padawitz, 1993) admits a finer distinction between the parts of an object.

```
                fun right_edge _ _ = " #"
                fun corner _ _ = "+++" end
structure Display
        = Display(structure Picts = struct
                                open ShelfsysOut
                                     Strings end)
end (*SHELFSYSTEM*)
```

Here is the result of filling 9 books and 3 hifi racks into 4 shelves:

```
19 |                    #+++++++++++++++++++#
18 | #+++++++++++++++++##                   #
17 | #  8====8         ##  3====3   14==14  #
16 | #  8====8         ##  3====3=6=|    |  #
15 | #  1=======1      ##  HIF===HIF|    |  #
14 | #  |       |      ##  |       ||    |  #
13 | #  |       |=7=   ##  |       ||    |  #
12 | #  |       || |   ##  |       ||    |  #
11 | #  |       || |   ##  HIF===HIF14==14  #
10 | #  1=======1=7=   ##+++++++++++++++++++#
 9 | #+++++++++++++++++##  10=====10        #
 8 | #  HIF===HIF      ##  |       |        #
 7 | #  |       |      ##  10=====10        #
 6 | #  |       |      ##  5==========5     #
 5 | #  |       |      ##  5==========5     #
 4 | #  |       |      ##  2=============2  #
 3 | #  |       |      ##  HIF=========HIF  #
 2 | #  |       |      ##  |             |  #
 1 | #  HIF===HIF      ##  HIF=========HIF  #
   ----------------------------------------
     1  2  3  4  5  6  7  8  9  10 11 12 13
```

References

Appel, A. W. and MacQueen, D. B. 1991. Standard ML of New Jersey, *Proc. 3rd Int. Syposium on Programming Language Implementation and Logic Programming*, Springer LNCS 528.

Belew, R. and Booker, L. *Proc. 4th Int. Conference on Genetic Algorithms*, Morgan Kaufmann (1991)

Ehrig, H, and Mahr, B. 1985. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Springer.

Goel, A., and Chandrasekaran, B. 1989. Functional Representation of Design and Redesign Problem Solving, *Proc. 11th Int. Joint Conference on Artificial Intelligence*, pp. 1394-1399.

Goguen, J. A., Thatcher, J. W. and Wagner, E. G. 1978. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, *in* R. Yeh (ed.), *Current Trends in Programming Methodology IV: Data Structuring*, Prentice Hall, pp. 80-144.

Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems*, Ann Arbor, University of Michigan Press.

Navinchandra, D., Sycara, K. P. and Narasimhan, S. 1991. Behavioral Synthesis in CADET, a Case-Based Design Tool, *Proc. of the 7th Conference on Artificial Intelligence Applications* pp. 217-221.

Padawitz, P. 1988. *Computing in Horn Clause Theories*, Springer.

Padawitz, P. 1992. *Deduction and Declarative Programming*, Cambridge University Press.

Padawitz, P. 1992a. EXPANDER, User's Manual, *Report No. 444/1992*, Fachbereich Informatik, Universität Dortmund.

Padawitz, P. 1993. Building Shelf Systems: A Case Study in Structured ML Programming, *Report No. 460*, Fachbereich Informatik, Universität Dortmund.

Reitman, W. R. 1964. Heuristic Decision Procedures, Open Constraints, and the Structure of Ill-Defined Problems, *in* M. W. Shelly, G. L. Bryan (eds), *Human Judgements and Optimality*, Wiley.

Rittel, H. and Webber, M. 1973. Dilemmas in a General Theory of Planning, *Policy Sciences* 4: 155-179.

Simon, H. A. 1973. The Structure of Ill-Structured Problems, *Artificial Intelligence* 4: 181-204.

Wirsing, M. 1990. Algebraic Specification, *in* J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier, pp. 675-788.