© Nottingham University Press
Proceedings of the International Conference on
Computing in Civil and Building Engineering
W Tizani (Editor)

Representational programming for design analysis

Rudi Stouffs & Wei-Tsang Chang
Faculty of Architecture, Delft University of Technology, Netherlands

Abstract

We propose an approach to programming we feel is very powerful for design analysis in the context of design modelling applications. The approach has some similarity to dataflow programming, especially in its visual form. Where the operational nodes in a dataflow program network operate on the data directly, in our approach the nodes operate instead on the representational structures containing the data. For this reason, we term the approach *representational programming*. In this paper, we define representational programming and compare it to dataflow programming and the Grasshopper® graphical algorithm editor. We illustrate representational programming using a constructive approach to representational structures, termed *sorts*, and we exemplify its application to building design analysis. Finally, we show an implementation of a small design analysis tool extending the Autodesk® Maya® modelling environment using visual programming with *sorts*.

Keywords: design analysis, dataflow programming, representational programming, *sorts*, Grasshopper

1 Introduction

The phrase “the designer as tool builder”, coined by Hugh Whitehead of Foster and Partners, reflects on the fact that often designers become confronted with the limitations of modelling software, whether basic geometric or advanced parametric modelling software, and feel forced to learn to script or program in order to extend the capabilities of the particular software. Whether the reason is a tedious repetitive task or a complex creative task, or the need to convert data or information from one application to another, the ability to script or program small application tools can be a great asset to the designer, or engineer for that matter.

However, neither programming nor scripting is as straightforward to the average designer as applying the functionality available within any modelling application. It is not just the lack of an interactive, graphical interface that makes scripting a hard task to learn for many. In fact, graphical or visual programming languages are already available in some modelling-related software, such as Dassault Systèmes’ 3DVIA Virtools™ interactive real-time applications development platform. While visual programming simplifies the actual expression of the script as a series of instructions or operations, these operations still need to be conceived and combined into the script, often with the same accuracy and precision as if one were to use a classical text-based scripting language.

Alternative programming approaches exist, operating at a higher level of abstraction and/or considering a program not as a sequence of instructions but as a network of operations, transformations or processes that exchange data, as in the case of dataflow programming or, more

exactly, the dataflow execution model (Johnston et al., 2004; Arvink and Culler, 1986). We propose an approach that lies somewhere in between the classical imperative approach and the dataflow-based approach, and call it representational programming. While it also emphasizes the data and its transformations, it considers these transformations as applied to the representational structure encoding the data. We believe it is highly applicable to design because design artefact data commonly necessitates large representational structures interrelating geometrical and other artefact-related data and information. Furthermore, design evaluation often relies on the analysis of relevant data extracted from the artefact's data structure. We claim that many evaluation procedures of interest to a designer can be expressed through the identification and extraction and possible recombination of partial data structures, the transformation thereof into new structures and, finally, the augmentation of these structures with functional entities that allow for the computation of data values derived from the existing data. In contrast to dataflow-based approaches, representational programming still mainly considers a program as a straightforward sequence of representational operations, mimicking the step-by-step approach of program development.

2 Dataflow visual programming for design modelling

The Grasshopper® graphical algorithm editor as part of Rhino's 3D modelling tools can be considered to implement a dataflow visual programming approach (using the dataflow execution model), considering a network of operational building blocks or nodes in order to model a complex, parametric geometry (see Figure 1). The nodes contain input and output terminals where an output terminal of one node can be connected to an input terminal of another node and, thus, defines an input to the operation specified by this node (in Grasshopper, input terminals can be found on the left-hand side of the node component, output terminals on the right-hand side). The geometry that results from this network is composed of geometric entities that each can be considered to be a side effect of one of the operational nodes.

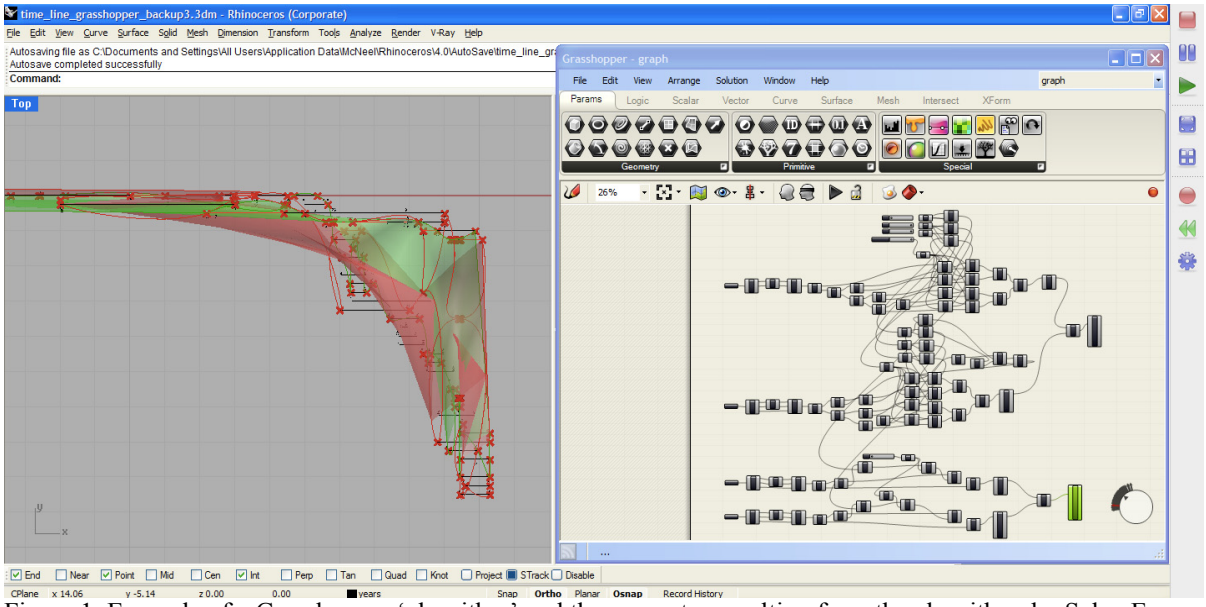

Figure 1. Example of a Grasshopper 'algorithm' and the geometry resulting from the algorithm, by Selen Ercan.

The advantage of dataflow visual programming is at least two-fold. Extensively emphasized in literature is the advantage of dataflow programming for parallel computing (Johnston et al., 2004; Nikhil, 1991). Unlike the classical imperative programming approach, the program no longer defines

a linear sequence of instructions that need to be executed in the order they are specified. In dataflow programming, the execution of the nodes in the dataflow network is not predefined but dependent on the availability of the inputs; a node executes when all its inputs are available. The second advantage is the semi-exploratory nature of building the visual graph. Nodes can be connected, disconnected and reconnected in different ways, leading to different results. Naturally, the ability to connect any two nodes is dependent on the input and output types of the nodes. In the case of Grasshopper, input and output values are mainly numerical values, geometric elements, and arrays (or lists) thereof.

3 Representational programming for design analysis

While Grasshopper is a powerful tool for building parametric and generative geometric models, its support for design analysis, while useful, is rather limited. There are mainly two reasons for this, one is the focus of Rhino's 3D modelling tools on geometric modelling rather than building information modelling, the second is the limitation of input and output values to individual values (or modelling objects) and arrays (or lists) thereof. While Grasshopper allows for multi-dimensional arrays, operations on arrays always apply to one-dimensional arrays. Even considering multi-dimensional arrays, it lacks the ability to specify, maintain and operate on property relationships as can be found in building information modelling. Support for hierarchical or graph data structures would greatly improve the power of Grasshopper. On the other hand, limiting data structures to arrays enables simple operational handling of multiple input arrays, either in a scalar (product) way (pairing array elements that share the same index) or in a Cartesian way (pairing any element from one array with any element from the other array). Allowing more complex data structures will either complicate the processing thereof –simple traversal algorithms will not take into account the variations in data kinds and data types at various levels within the data structure– or require special kinds of data structures with build-in support for processing these.

3.1 Sorts

Stouffs (2008) describes a semi-constructive algebraic formalism for design representations, termed *sorts*, that provides such support. It presents a uniform approach for dealing with and manipulating data constructs and enables representations to be compared with respect to scope and coverage, and data to be converted automatically, accordingly. *Sorts* can be considered as hierarchical structures of properties, where each property specifies a data type, properties can be collected and a collection of one or more properties can be assigned as an attribute to another property. *Sorts* can also be considered as class structures, specifying either a single data type or a composition of other class structures.

Each *sort* has a behavioural specification assigned, governing how data entities combine and intersect, and what the result is of subtracting one data entity from another or from a collection of entities from the same *sort*. This behavioural specification is a prerequisite for the uniform handling of different and a priori unknown data structures and the effective exchange of data between various representations. The behavioural specification forms part of the predefined template of a *sort* and is based on a part relationship on the entities of a *sort*, with the *sortal* operations of addition, subtraction, and product defined in accordance to this part relationship. A functional *sort* allows the specification of data (analysis) functions that automatically apply to *sortal* structures through tree traversal.

3.2 Sortal example

Consider the following example: given a room adjacency graph, where the nodes in the graph represent rooms and the edges represent room adjacencies, how can we derive a floor adjacency graph, where the nodes represent floors and the edges floor adjacencies? Basically, we are interested

in knowing how many floors there are, which rooms are on which floor, what floors are below/above/next to a given floor, etc. We assume that rooms have attribute information specifying the floor level or, otherwise, complete geometric information allowing us to extract floor level information.

From a programming point of view, the derivation of a floor adjacency graph from a room adjacency graph is not all that complex, but without proper programming knowledge, the task can still be very challenging. We show how one might approach this problem with representational programming. First, we need to define the representational structure we will use as a starting point.

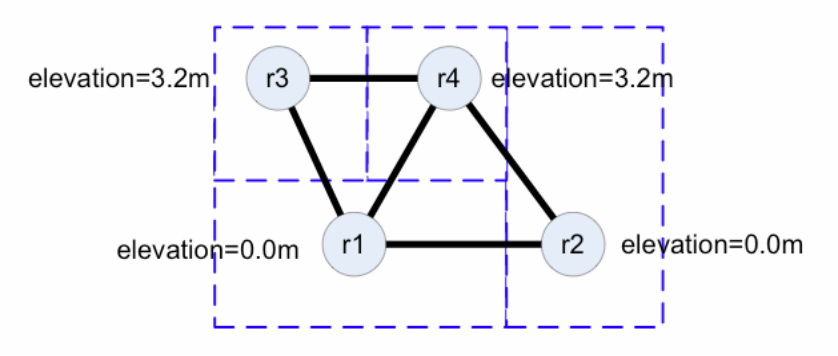

Figure 2. A room adjacency graph with elevation information.

Figure 2 illustrates the data that may be present in the room adjacency graph. We ignore the format in which the data may be provided, and instead consider the basic data entities that are required. Firstly, we need to represent the rooms themselves, e.g., "r1", "r2", "r3" and "r4". We can do so by their name, a string. We define a primitive *sort* with `Label` as *sortal* template:

```
sort rooms : [Label];
form $rooms = rooms: { "r1", "r2", "r3", "r4"};
```

The first line defines the *sort* `rooms`, with template `Label`. The second line defines an exemplary data *form* of *sort* `rooms` and referenced by the *sortal* variable `$rooms`. It defines a collection of rooms or, more specifically, room labels "r1" through "r4". Similarly, we can represent the room elevations as numbers with *sortal* template `Numeric`:

```
sort elevations : [Numeric];
form $elevations = elevations: { 0.0, 3.2 };
```

Finally, we need to represent the adjacency relations. For this, we use the `Property` template. Unlike previous templates, the `Property` template requires two primitive *sorts* as arguments, and defines not one but two new primitive *sorts*:

```
sort (adjacencies, rev_adjacencies) : [Property] (rooms, rooms);
```

The two arguments define the representational structure for the tails and heads of the adjacency relationship. Since the `Property` template applies to directional relationships, we consider two resulting *sorts*: `adjacencies` and `rev_adjacencies` (reverse adjacencies). An example is given below.

A complex representational structure is defined as a composition of primitive representational structures. *Sortal* structures offer us two compositional operators: an attribute operator, `^`, specifying a subordinate, conjunctive relationship between *sortal* data, and a sum operator, `+`, specifying a co-ordinate, disjunctive relationship. Considering the room adjacency graph, we can define a corresponding *sortal* structure as follows:

```
sort input : rooms ^ (elevations + adjacencies + rev_adjacencies);
```

Rooms have elevations, adjacency relationships and reverse adjacency relationships as attributes. A corresponding data form would be defined as:

```
form $input = input:
{ #me-rooms-1 "r1"
  { (elevations): { 0.0 },
    (adjacencies): { me-rooms-2, me-rooms-3, me-rooms-4 } },
  #me-rooms-2 "r2"
  { (elevations): { 0.0 },
    (adjacencies): { me-rooms-4 } },
  #me-rooms-3 "r3"
  { (elevations): { 3.2 },
    (adjacencies): { me-rooms-4 } },
  #me-rooms-4 "r4"
  { (elevations): { 3.2 } }
};
```

Of course, this data form may be generated from the original data format, rather then specified in textual form. `#me-rooms-1` is a reference ID for "r1" that can be used later, in the form `me-rooms-1`, to reference "r1" in an adjacency relationship from a different room. The specification of reverse adjacency relationships is optional; the *sortal* interpreter will automatically generate these.

Similarly, we can define a representational structure for the output we need to produce. Consider the goal to group rooms with the same elevation into floors. For this, we can consider elevations with rooms as attributes; the rooms themselves may still have (reverse) adjacency relationships as attributes:

```
sort output : elevations ^ rooms ^ (adjacencies + rev_adjacencies);
form $output = output: $input;
```

The second line defines a variable of *sort* `output` with `$input` as data. Since `$input` is defined of *sort* `input`, the data must be converted to the new *sort*. This conversion is done automatically based on rules of semantic identity and syntactic similarity. The result is:

```
form $output = output:
{ 0.0
  { #me-rooms-1 "r1"
    { (adjacencies): { me-rooms-2, me-rooms-3, me-rooms-4 } },
    #me-rooms-2 "r2"
    { (adjacencies): { me-rooms-4 },
      (rev_adjacencies): { me-rooms-1 } } },
  3.2
  { #me-rooms-3 "r3"
    { (adjacencies): { me-rooms-4 },
      (rev_adjacencies): { me-rooms-1 } },
    #me-rooms-4 "r4"
    { (rev_adjacencies): { me-rooms-1, me-rooms-2, me-rooms-3 } } }
};
```

This is a collection of floor elevations, with for each floor elevation a list of rooms, with room adjacency relationships (and reverse relationships). The specification of functional entities can result in the number of floors, the number of rooms per floors, etc (see figure 3). The example can also be further expanded considering the complete geometric information of the rooms instead of only the elevation attribute (see also figure 3). In that case, it will also be possible to derive the geometric information of each floor as a composition of rooms (figure 4).

In order to emphasize the relative simplicity of the approach, figure 5 shows a Java program resolving the same problem. For reasons of simplicity, the exemplar Java program does not define any classes, nor does it consider geometric (shape) information. While the Java program is fairly short,

any reformulation of the problem, e.g., adding geometric (shape) information, may have an important impact on the code. Furthermore, the definitions of *sorts* and forms, both as hierarchical structures, can also be considered graphically, such that the exact syntax of these statements is no longer important.

```
#SDL V1.1a [me]

// consider the derivation of a floor adjacency graph from a room adjacency graph

sort rooms : [Label];
sort elevations : [Numeric];
sort (adjacencies, rev_adjacencies) : [Property] (rooms, rooms);
sort shapes : [PlaneSegment];

sort input : rooms ^ (shapes + elevations + adjacencies + rev_adjacencies);
sort output : elevations ^ rooms ^ (shapes + adjacencies + rev_adjacencies);

form $input = input:
{   #me-rooms-1 "r1"
        { (elevations): { 0.0 },
          (adjacencies): { me-rooms-2, me-rooms-3, me-rooms-4 },
          (shapes): { <<(0,0,0), (6,0,0)>, <(6,0,0), (6,6,0)>, <(6,6,0), (0,6,0)>, <(0,6,0), (0,0,0)>> } },
    #me-rooms-2 "r2"
        { (elevations): { 0.0 },
          (adjacencies): { me-rooms-4 },
          (shapes): { <<(6,0,0), (9,0,0)>, <(9,0,0), (9,6,0)>, <(9,6,0), (6,6,0)>, <(6,6,0), (6,0,0)>> } },
    #me-rooms-3 "r3"
        { (elevations): { 3.2 },
          (adjacencies): { me-rooms-4 },
          (shapes): { <<(0,0,3.2), (3,0,3.2)>, <(3,0,3.2), (3,6,3.2)>, <(3,6,3.2), (0,6,3.2)>, <(0,6,3.2), (0,0,3.2)>> } },
    #me-rooms-4 "r4"
        { (elevations): { 3.2 },
          (shapes): { <<(3,0,3.2), (6,0,3.2)>, <(6,0,3.2), (6,6,3.2)>, <(6,6,3.2), (3,6,3.2)>, <(3,6,3.2), (3,0,3.2)>> } },
};

form $output = output: $input;

// additional information we may derive from this floor adjacency graph

sort counts : [Function];
sort number_of_rooms : counts ^ elevations ^ rooms;

// func count(x) = c : {c(0) = 0.0, c(+1) = c + 1};
ind $count = number_of_rooms: count(rooms.length) $output;

sort rooms_per_floor : elevations ^ counts ^ rooms;

form $count_per_floor = rooms_per_floor: $count;

sort floors : elevations ^ shapes;

form $floors = floors: $output;
```

Figure 3. A complete Sorts Description Language (SDL) file, including the specification of the room geometries (shapes) and operations to count the number of rooms (total and per floor) and to determine the geometry of each floor. The function *count* is pre-defined in the *sortal* interpreter but, otherwise, could be specified as shown in the last comment (preceded by '//'). The output is shown, partially, in figure 4.

3.3 Implementation

We've built a small design analysis tool extending the Autodesk® Maya® modelling environment using visual programming with *sorts*. A few different icons (node types) provide access to the design model data from Maya, allow object filtering based on layering information or a MEL script filter, allow data filtering and conversion to retrieve surface or volume information, provide arithmetic calculations for further data processing (e.g., calculate cost information), and output the results. The example in figure 6 selects the surface objects from the model facing North, retrieves their surfaces and computes the total surface area times a multiplication factor (e.g., expressing cost information).

```
ind $count = number_of_rooms: count(rooms.length) = 4.0
 { 0.0
    { "r1",
      "r2" },
  3.2
    { "r3",
      "r4" } };

form $count_per_floor = rooms_per_floor:
{ 0.0
   { count(rooms.length) = 2.0
      { "r1",
        "r2" } },
  3.2
   { count(rooms.length) = 2.0
      { "r3",
        "r4" } } };

form $floors = floors:
{ 0.0
   { <<(0,0,0), 6(0,1,0)>, <9(1,0,0), 3(3,2,0)>, <6(1,0,0), 9(1,0,0)>, <6(1,1,0), 3(3,2,0)>, <(0,0,0), 6(1,0,0)>, <6(0,1,0),
6(1,1,0)>> },
  3.2
   { <<16/5(0,0,1), 2/5(0,15,8)>, <2/5(15,0,8), 2/5(15,15,8)>, <16/5(0,0,1), 1/5(15,0,16)>, <2/5(0,15,8), 1/5(15,30,16)>,
<1/5(15,0,16), 2/5(15,0,8)>, <1/5(15,30,16), 2/5(15,15,8)>> } };
```

Figure 4. The output of the SDL file of figure 3, omitting the floor adjacency graph already shown in the text.

```java
import java.util.*;

public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // simple room adjacency graph
        String[] rooms = { "r1", "r2", "r3", "r4"};
        double[] room_elevations = {0, 0, 3.2, 3.2};
        int[][] room_adjacencies = {{0, 1, 1, 1},
                                    {1, 0, 0, 1},
                                    {1, 0, 0, 1},
                                    {1, 1, 1, 0}};
        int number_of_rooms = rooms.length;

        // identify the different floors by their elevations
        TreeSet<Double> floors = new TreeSet<Double>();
        for (double i : room_elevations)
            floors.add(new Double(i));
        ArrayList<Double> floor_elevations = new ArrayList<Double>(floors);

        // retrieve the number of floors
        int number_of_floors = floor_elevations.size();
        System.out.println("number of floors: " + number_of_floors);

        // link the rooms to the floors
        ArrayList<ArrayList<Integer>> room_floors = new ArrayList<ArrayList<Integer>>();
        for (int i = 0; i < number_of_floors; i++)
            room_floors.add(new ArrayList<Integer>());
        for (int i = 0; i < number_of_rooms; i++)
            room_floors.get(floor_elevations.indexOf(new Double(room_elevations[i]))).add(new Integer(i));

        // retrieve the number of rooms per floor
        int[] rooms_per_floor = new int[number_of_floors];
        System.out.println("number of rooms per floors: ");
        for (int i = 0; i < number_of_floors; i++) {
            rooms_per_floor[i] = room_floors.get(i).size();
            System.out.println(floor_elevations.get(i) + " -> " + rooms_per_floor[i]);
        }
    }
}
```

Figure 5. The same adjacency graph problem (omitting geometric data, i.e., shapes) resolved in Java.

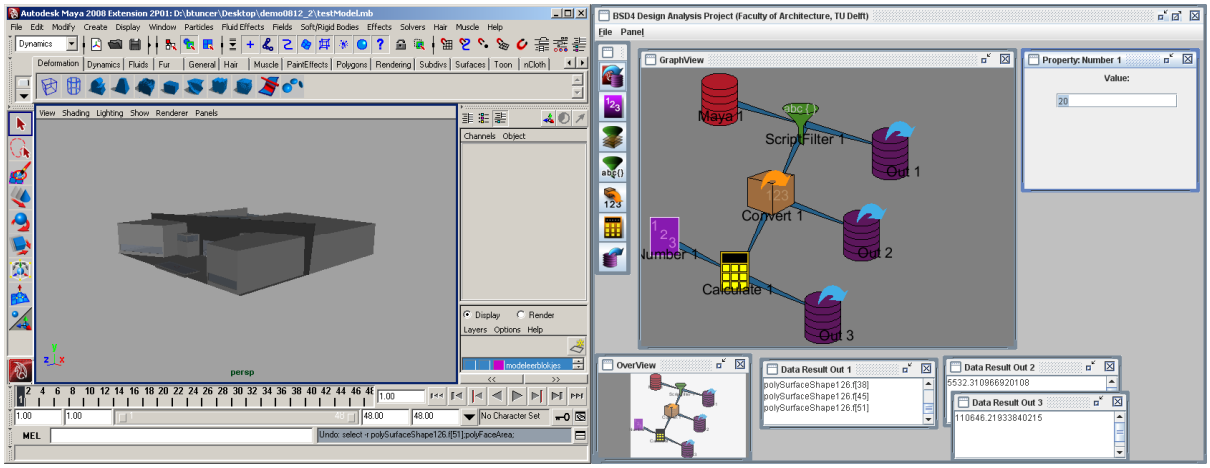

Figure 6. Example of a visual *sortal* program (right) and the Maya model (left) providing the input data for the tool.

Conclusion

Representational programming considers operations on data structures to yield data transformations and computations. It adopts the dataflow execution model for visual programming. *Sorts* offer support for representational programming. They present a uniform approach for dealing with and manipulating data constructs and enable data representations to be compared with respect to scope and coverage, and data to be converted automatically, accordingly. These characteristics make *sorts* very appropriate for data manipulation and analysis in the context of design modelling. In principle, *sorts* could complement and be integrated into a dataflow visual programming approach as considered in Grasshopper.

Acknowledgements

The authors wish to thank Georg Suter for providing us with the example problem, and the reviewers for their feedback and suggestions.

References

ARVIND and CULLER, D.E., 1986. Dataflow architectures. *Annual Reviews of Computer Science*, 1, 225-253.

JOHNSTON, W.M., HANNA, J.R.P and MILLAR, R.J., 2004. Advances in dataflow programming languages. *ACM Computing Surveys*, 36 (1), 1-34.

NIKHIL, R.S., 1991. Dataflow Programming Languages. Computation Structures Group Memo 333, Laboratory for Computer Science, MIT, Available online: csg.csail.mit.edu/CSGArchives/memos/Memo-333.pdf. Last accessed: January 2010.

STOUFFS, R., 2008. Constructing design representations using a sortal approach. *Advanced Engineering Informatics*, 22, 71-89.