Design, Complexity and Abstraction

John R. Woodward

School of Computer Science, University of Birmingham, B15 2TT, UK Email: J.R.Woodward@cs.bham.ac.uk

1 Introduction

The process of design can be considered a searching for the solution to a design problem. One approach we may use in design is to decompose the overall design down into simpler component parts which can each be designed in isolation, and reassembled into complete design solution at a later stage. For example, we may design a car by designing the shape of the body, the braking system, the dashboard layout and the engine in isolation, and the individual parts are assembled at a later stage into a complete car.

There are a number of different ways of representing the description of the design. We point out that representations which are capable of expressing reuse of component parts have an interesting property; the complexity of an object with respect to that type of representation is independent of the primitives used in the representation. These types of representation can express the operation of composition and we equate this with the process of abstraction.

One possible approach to the design of a system with reusable components is to use cooperative co-evolution. The parts of the system are evolved, and the overall product is tested and assignened a fitness score. We mention two papers. The first examines a number of different methods to tackle the credit assignment problem. The second evolves specifications for component parts and once a suitable decomposition is evolved, solutions to the subcomponents are found.

2 Primitives

We begin with a set of 'primitives'. A set of primitives is a collection of objects which are taken as given. These are considered to be atomic and can be taken as our starting point in the design process (i.e anything we design consists of these primitives). A design space consists of all the designs that can be constructed using these primitives (called the search space [1]). Primitives are treated as black boxes, and we may claim ignorance of any internal workings of the primitives. We give examples of primitives in the following section.

Children's toys of the construction variety (e.g. Lego or Meccano) consist of parts which maybe connected together to build more complex parts. These initial parts (plastic blocks or strips of metal) are considered as atomic units. New constructions can, once constructed, be considered as units (i.e. they can

physically be moved around as if it were a single part and can be plugged together with other parts).

Straight lines and a method of constructing angles could be used to construct more complicated drawing which consist of straight lines (see [2] for examples of what can and cannot be drawn with a unit line and a compass). For example we could construct a square with sides of a certain length, and squares could now be considered as a unit to be used in further drawings. We could supplement this with a method of drawing curves (e.g. arcs of a circle or arcs of an ellipse) and with this we could draw new pictures we could not draw before. Again new shapes can be constructed, and once constructed can be thought of as primitives for further drawings (i.e. we do not have to go back to the basic description of straight lines and edges each time).

In Genetic Programming [1], a set of primitives are supplied (the function set and the terminal set), and new functions are synthesises by taking the output of one primitive and feeding it into the input of another primitive. In this way new functions are created and are represented as tree structures, where the output of a subtree feeds into nodes further up the tree. Solutions to the design problem are specified using the primitive set. In the representation permits modules (i.e. the ability to construct a function once and refer to it when needed), we can in effect add new primitives to our given primitive set. The best know of these modular methods is Automatically Defined Functions [1].

In each of the systems above, new objects can be constructed and these can be treated as units themselves (i.e. new primitives can be created on the fly).

3 Complexity

The complexity of an object is often identified with Kolmogorov complexity [3] which is defined as the size of the shortest program (where is size is measured in the number of bits to express the program) with reference to a given universal machine (which could be a specific Universal Turing Machine or a programming language e.g Java). Other definitions of complexity exist, but what makes this definition particularly useful is that it is invariant (up to an additive constant). However it does have the drawback that it is incomputable. Generally, we can define complexity as the size of the shortest description with reference to some description method. Note that different definitions of size will typically change the measure of size a multiplicative amount (i.e. the number of instruction in a computer program will be a multiple of the size of the program expressed in bits).

Let us consider the partial recursive functions (PRFs) which is the set of computable functions. PRFs are generated from simple functions (the zero, successor and projection functions) and three other functions (recursion, composition and minimisation). Primitive recursive functions are a subset of PRFs and are defined without minimisation (i.e. can be defined in terms of the zero, successor and projection functions with recursion and composition). Automatically Defined Functions [1] can be thought of as a set of functions using composition

alone to construct new functions. In [4] we proved that the complexity of a function is independent of the primitive set (up to an additive constant) if our method of description allows the expression of composition. In a forthcoming paper we prove a similar result for description methods of primitive recursive functions. This leads to a general result; given two equally expressive representations, provided the representations are capable of expressing composition (i.e. reuse of component parts), then the complexity of an object is independent of the actual representation within an additive constant.

Strictly, it is meaningless to talk about the complexity of an object without referring to a representation. The complexity of an object can be high using one representation, but using powerful enough primitives the complexity can be made arbitrarily small (i.e. 1 as the primitive expresses the object). In some situations we may have access to the raw complexity. For example, with a system of drawing a point is the smallest unit that we can mark a page with, and lines are simply sets of points. We could also define complexity with respect to the smallest Universal Turing Machine. In systems like those described in [4] we can define raw complexity with respect to the smallest minimal primitive set (e.g. $NAND$ is the smallest logically complete function set). If we are talking about physical objects, we actually cannot get down to the lowest level of description (as we do not know what it is, i.e. are quarks atomic). The point is that we only need to describe an object in sufficient detail with the tools we have for the job at hand.

4 Abstraction

The word abstraction comes from the Latin, abs meaning 'away from' and trahere, meaning 'to draw'. Abstraction is the ability to hide unnecessary complexity in the form of new primitives. Thus its utility is in the fact that we do not need to concern ourselves with unnecessary detail and can focus on the important issues with appropriate level of description. Thus when we abstract away from a description we can absorb any unnecessary detail into a primitive and ignore it. Hence abstraction can be thought of as the ability to define new primitives (in terms of currently existing ones) and talk about an object at a new level of less detail. We may therefore talk about systems being equivalent (or similar) at some abstract level.

5 Design and co-evolution

Design is the problem of describing a system to achieve a target task, which may be thought of as an optimisation problem. Evolution is often used to tackle design problems [5, 6]. Given the results above concerning complexity, it would advantageous to have a design system, which, once it has found a solution to a component part of the overall design task, and be reused and thus avoids the problem of having to reinvent the wheel. One approach to evolving component parts is cooperative co-evolution (as opposed to competitive co-evolution), but

one problem is credit assignment (i.e how can credit be given to individual components of the system?). For example, consider a car with a good engine but square wheels. It would be advantageous to be able to recognise that while the overall design is poor (i.e. the car will not move), there are useful component parts (i.e the engine is fine) which may be used in other designs.

It is not easy to see how a reward can be directly assignet to a component. Typically a fitness score can be given to a whole design, but it can be difficult to see how individual parts contribute to the overall design. In [7] a number of different methods to this problem are described. They empirically examine selection pressure, pool size and credit assignment. and conclude that an optimistic approach is generally the best mechanism for collaboration credit assignment. There are of course many other methods of attempting this.

In [8] we use a novel approach to producing components part of an overall solution. All other methods represent component parts (i.e. modules) in terms of the primitives at all times in the evolution ([1] section 10.2). Our method differs fundamentally in that initially component parts are represented by specifications which are not represented in terms of primitives, but just as a description. Once a suitable description for a component part (called a module in [8]) is found, it can then be isolated and solved separately from the overall system. The methods outlined in [7] could be used together with the ideas in [8].

6 Summary

A method of representing our proposed designs in needed in the design process. We point out that if the representation is capable of expressing composition then there are interesting results concerning the complexity of the description of a design. The reader may identify composition with modules used in some work on evolutionary computation. Composition can also avoid the problem of reinventing the wheel (as a design solution can be expressed once and referred to when needed). We equate the process of abstraction with composition, which can allow us to concentrate on the more interesting aspects of the design rather that being preoccupied with unnecessary detail which may be packaged into the primitives of the description language.

Given that we have a system capable of expressing modules for our design process, the question then is how do we go about constructing candidate designs. One approach is cooperative co-evolution where component parts are evolved together. We mentioned that there are a number of different approaches to the evaluation of the component parts of designs [7]. We also mentioned a novel way of evolving specifications for modules rather than implementations (which is easier to do) and later producing designs for the component parts (which reflects the design process).

References

1. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applica-

tions. Morgan Kaufmann, dpunkt.verlag (1998)
2. Fraleigh, J.B.: A First Course in Abstract Algebra. 5 edn. Addison Wesley (1994)
3. Li, M., Vitanyi, P.M.B.: An Introduction to Kolmogorov Complexity and Its Applications. Springer-Verlag, Berlin (1993)
4. Woodward, J.R.: Modularity in genetic programming. In: Genetic Programming, Proceedings of EuroGP 2003, Essex, UK, Springer-Verlag (2003)
5. Bentley, P.J., Corne, D.W., eds.: Creative evolutionary systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
6. Bentley, P.J.: Evolutionary Design by Computers with CDrom. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
7. Wiegand, R.P., Liles, W.C., Jong, K.A.D.: An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In: Genetic and Evolutionary Computation Conference (GECCO) 2001, Morgan Kaufmann Publishers (2001) 1235–1245
8. Woodward, J.R.: Function set independent genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004), Workshop on Modularity, Regularity, and Hierarchy in Evolutionary Computation, Seattle, USA, Morgan Kaufmann (2004)