Evolving Good Hierarchical Decompositions of Complex Systems

Rudi Lutz
School of Cognitive and Computing Sciences
University of Sussex
England

Email: rudil@cogs.susx.ac.uk

Technical Report CSRP 519
May 2000

Abstract

Many application areas represent the architecture of complex systems by means of hierarchical graphs containing basic entities with directed links between them, and showing the decomposition of systems into a hierarchical nested "module" structure. An interesting question is then: *how best should such a complex system be decomposed into a hierarchical tree of nested "modules"*? This paper describes an interesting complexity measure (based on an information theoretic *minimum description length principle*) which can be used to compare two such hierarchical decompositions. This is then used as the fitness function for a genetic algorithm which successfully explores the space of possible hierarchical decompositions of a system. The paper also describes the novel crosssover and mutation operators that are necessary in order to do this, and gives some examples of the system in practice.

1 Introduction

Complex systems are often described by means of hierarchical graphs (along the lines of that shown in Figure 1). Examples include the breakdown of a software system into subsystems, submodules, sub-submodules etc. (these will all be referred to as modules and submodules in the rest of the paper), or the hierarchical decomposition of hardware circuitry. In such diagrams the links can represent such things as dependency relationships between the components, or control-flow

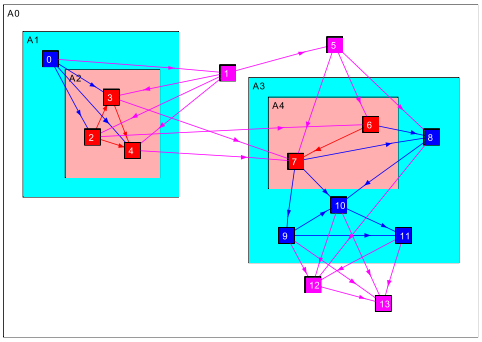

Figure 1: A system with nested modules and sub-modules.

or data-flow. So, for example, Figure 1 shows a system consisting of a top-level module (A0), which consists of four basic entities (labelled 1, 5, 12, and 13) and 2 sub-modules(labelled A1, and A3). Module A3 in turn consists of four basic entities (8, 9, 10, and 11) and contains a sub-module of its own (A4).

Two related issues that often arise when designing such systems are:

- What is the best way to hierarchically break the system up into components?
- Given an existing hierarchically decomposed system to which one is adding structure (e.g. a new component and possibly extra links), should the new component be added to one of the existing modules, and if so which, or should adding the extra component lead to reorganisation of the existing design into a new hierarchical structure?

These issues can arise either during the process of system design, or when trying to reverse engineer the high-level design of some system. Both of these questions can in principle be answered provided we have some method of searching through the space of possible hierarchical decompositions of a system. This paper will describe work which tries to do this using evolutionary techniques. Doing so first requires the answer to a more fundamental question – *what makes one hierarchical modular decomposition (HMD) of a system better than another?* For example, the modular structure shown in Figure 2 appears to most people to be much "nicer" (less complex) than that in Figure 1. In the work reported in this paper, a variant of an information-theoretic metric due to Wood[15] will be used to define when we consider one HMD to be better than another, but in principle any suitable metric could be used. However, the one we use does seem to do quite a good job of matching human intuitions (or at least those of the author!) about what makes

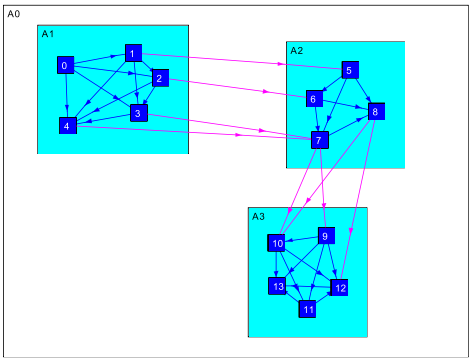

Figure 2: A "better" modular decomposition of the system shown in Figure 1.

a good HMD. Whichever metric is used will then play a fundamental part in the evolutionary algorithm for exploring the space of possible designs, since it provides the basis for defining the "fitness function" for the genetic algorithm.

The paper will begin by describing what is meant by a hierarchical modular decomposition of a system. It will then give some informal definitions needed in order to describe our variant of Wood's metric enabling us to compare hierarchical decomposition. Finally, the evolutionary algorithm will be described, followed by some preliminary results and discussion of future directions for research.

2 Hierarchical Modular Decompositions of Systems

In what follows we will give relatively informal definitions of the main concepts and terminology needed to understand the rest of the paper. More formal definitions can be found in [7]. We will begin by describing what we mean by a *system*, and by a *hierarchical modular decomposition (HMD)* of a system.

2.1 Systems

Following Briand et al.[1], a system will be defined to be a directed graph consisting of some number of nodes (basic entities), and a collection of directed edges(links) between them. An example is shown in Figure 3. All the systems considered here will be of this nature; the work can easily be extended to allow directed multigraphs.

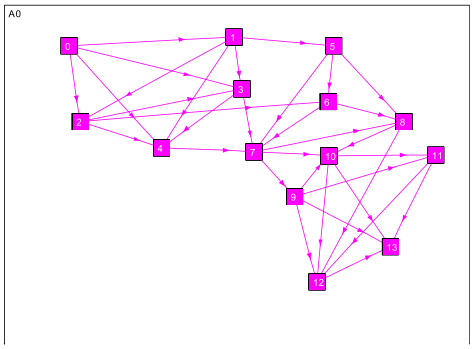

Figure 3: A system (the underlying system for the HMD of Figures 1 and 2).

2.1.1 Terminology

For convenience we define the union $S_1 \cup S_2$ of 2 systems S_1 and S_2 to be the directed graph whose node set is the set-theoretic union[1] of the node sets of S_1 and S_2, and whose edge set is the set-theoretic union of the edge sets of S_1 and S_2.

Given an edge from a node n_1 to a node n_2 in a system, we will refer to n_1 as the *source* of the edge, and to n_2 as the *destination* of the edge. We will refer to the edge as an *incoming* edge of n_2, and as an *outgoing* edge of n_1.

Given a node n in a system, it will also be useful to define the following:

- $inDegree(n)$ = number of edges whose destination is n
- $outDegree(n)$ = number of edges whose source is n
- $inNodes(n)$ = set of nodes with an outgoing edge whose destination is n
- $outNodes(n)$ = set of nodes with an incoming edge whose source is n

2.2 Hierarchical Modular Decompositions (HMDs)

Given a system S a HMD is S together with a *module tree* T. This is a tree whose leaves are the basic nodes of S, and the "internal nodes" correspond to the modules of S, with the root of T corresponding to the outermost "top-level" module of the HMD, and its non-leaf sub-trees are its submodules, and so on. Figure 4 shows a simple HMD, together with its module tree. We can therefore think of a HMD as being a pair $\langle T, E \rangle$, consisting of a module tree T together with a collection of

[1] In the case of multi-graphs this can be generalised to union of *bags*.

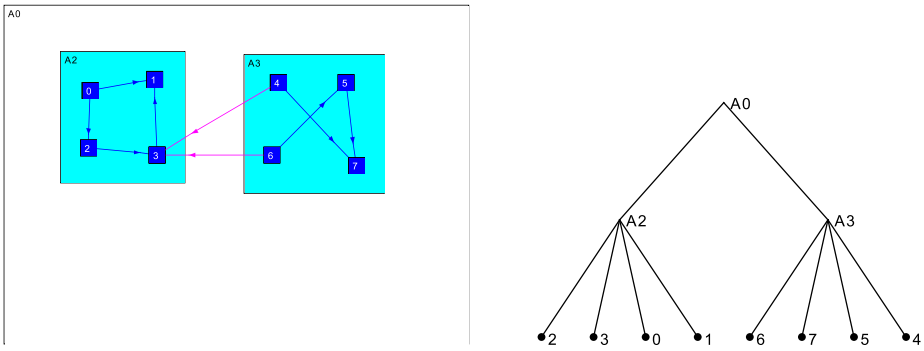

Figure 4: A HMD and its module tree

links E between the basic entities (which is the only other information present in the underlying system which is not implicit in the module tree).

2.2.1 More Terminology

The following definitions will turn out to be useful later in the paper:

- Given a HMD $\langle T, E \rangle$ of some system, the *least common ancestor* of two basic entities n_1 and n_2 (leaves in T) is the deepest "module node" M in T such that M is an ancestor of both n_1 and n_2. We will denote this by $lca(n_1, n_2)$. For example, in Figure 1, $lca(n_7, n_4) = A0$, and $lca(n_7, n_{10}) = A3$.

- Given a HMD $\langle T, E \rangle$, if n is a basic entity (leaf node) in T and M is a "module node" of T which is an ancestor of n, then we define the *relative depth of n and M in T*, written $relDepth(n, M)$, to be one less than the number of branches in T on a path from M to n. For example, in Figure 1 $relDepth(n_7, A3) = 1$, and $relDepth(n_7, A4) = 0$. In other words it is a measure of how many modules one must "climb up" to get from the module containing n to M.

3 Choosing Between Hierarchical Modular Decompositions

Given two hierarchical modular decompositions of a system S, how can we choose between them? In software engineering textbooks one often sees guidelines which can be loosely paraphrased as:

- prefer designs with greater cohesion in the modules
- prefer designs with less coupling between modules

These guidelines do not however say anything at all about what to do when they are in conflict with each other,especially in hierarchical systems. The approach taken in this work to choosing between two hierarchical modular decompositions is heavily based on the work of Wood[15], and makes no explicit mention of notions corresponding to coupling and cohesion. Instead it takes the view that the best HMD of a system is the *simplest* (in a sense to be made precise shortly). In practice this seems to give rise to HMDs in which modules are highly connected internally (high cohesion), and have relatively few connections which cross module boundaries (low coupling), and thus seems to achieve a principled trade-off between the coupling and cohesion heuristics without actually involving either.

3.1 The Minimum Description Length Principle

Kolmogorov Complexity theory[6] (sometimes called Algorithmic Information Theory) is a branch of mathematics that tries to answer the question *"how much information does an object contain by virtue of its structure?"*. This is then answered by essentially defining the information content of an object to be the length (measured in *bits*) of the shortest Universal Turing Machine program (consisting of the description of a Turing machine, plus its data) that can generate the object. The *complexity* of an object is then identified with this information content. It can be shown that this gives rise to an essentially invariant (up to an additive constant) quantity, independent of the choice of Universal Turing Machine (and hence, by the Church-Turing thesis, of choice of powerful enough computational formalism). We note that this means that the complexity of an object is given by the length of the shortest two part code, where the first part of the code defines a Turing machine, and the second part defines the data for that machine, such that when the Turing machine is run on that data it generates the object of interest.

Unfortunately, in general, the true minimum description length is an uncomputable quantity[6]. Therefore in applications one usually picks some fixed coding scheme for the objects of interest (equivalent to fixing the Turing machine in the two part code) and defines the complexity to be the length (in *bits*) of the encoding (plus the constant length of the description of the Turing machine). The general idea here is that if one chooses a "reasonable" encoding scheme which seems to us to give reasonably compact encodings of the object, and which is not too complicated to decode (smallish Turing machine), then one will have a fairly good approximation to the true complexity of the objects. An important point to note is that it is important that the encoding be *uniquely decodeable* by the decoding

Turing machine. One can therefore think of the encoding as being a message that is sent to some recipient (the decoder), whose task is then to decode the message. Given that the message will consist of string of bits, it is necessary for the decoder to be able to recognise which substrings of these bits represent which parts of the message. Therefore all the apparatus of Information Theory[11], and in particular the theory of optimal uniquely decodeable codes, can be brought to bear.

Given some data D, one is often faced with a choice of several competing theories which can account for the data. An often used principle in making such decisions is Occam's Razor (attributed to William of Ockham, 1290–1349), which informally states that one should choose the simplest of the alternatives, other things being equal. A modern formalization of Occam's razor is the *Minimum Description Length Principle*[9]. This aims to achieve a principled balance between the complexity (or simplicity!) of the data as described by the theory, and the complexity of the theory itself. It does this by stating[2] that, given some data D, and some set of possible theories $\mathcal{T}$, choose that $T \in \mathcal{T}$ which minimises

$$L(T) + L(D|T)$$

where $L(T)$ is the length (in bits) of the minimal description of the theory T, and $L(D|T)$ is the length (in bits) of the minimum (re-)description of the data D given the theory T.

In the case of HMDs one can think of the underlying system S as being the data, and the space of possible "theories" as being the set of possible HMDs over S. The minimum description length principle would then tell us that the best HMD is the one that minimises:

$$L(T) + L(S|T)$$

Now since each HMD completely encodes the underlying system S then we have that $L(S|T) = 0$, and we are simply looking for that HMD T with the smallest description length.

3.2 The Message Format

We follow the usual practice described above in applications of MDL by *choosing what we hope is a reasonably good encoding* of the objects we are describing, and using the length of this as the basis of our complexity measure. Later in the paper we will give a formula for the length of a message describing a HMD, but first we will have to go into quite a lot of detail about the exact form our messages

[2]Under certain assumptions this can be *derived* within Kolmogorov Complexity Theory. See [6] for further details.

will take. Only by understanding exactly what needs to be sent, and how often various symbols are used in such a message can one understand the complexity formula. We will begin by describing the *high-level format* of the encoding of a HMD, and then discuss encoding this as a string of bits whose length will be taken as the complexity of the HMD. It should also be noted that we will follow the usual practice of allowing "fractional bits"(which give the theoretical lower bounds on the number of bits needed) as this seems to give the best results. Indeed there are coding schemes which can very closely approximate these fractional code lengths on average[14].

In order to communicate the structure of a HMD and its underlying system we need to specify, for each module:

- its basic entities
- its submodules

and for each basic entity we need to communicate what it connects to. This can be done using the message format (expressed as a pseudo-BNF style grammar) shown in Table 1. In this grammar, "+" is a meta-symbol denoting concatenation (it does

HMD	::=	moduleTree + links
moduleTree	::=	entities + submoduleTrees
entities	::=	|unconnected| + |entitySeq| + entitySeq
entitySeq	::=	entity + entitySeq
entitySeq	::=	empty
submoduleTrees	::=	|moduleSeq| + moduleSeq
moduleSeq	::=	moduleId + moduleTree + moduleSeq
moduleSeq	::=	empty
entity	::=	entityId
links	::=	entityLinkInfo + links
links	::+	empty
entityLinkInfo	::=	|connectionSeq| + connectionSeq
connectionSeq	::=	connection + connectionSeq
connectionSeq	::=	empty
connection	::=	|modulesUp| + idSeq
idSeq	::=	entityId
idSeq	::=	moduleId idSeq

Table 1: Pseudo-BNF grammar for encodings of HMDs.

not turn up in the messages), "empty" denotes the empty string, and |entitySeq| denotes the length of the following entitySeq (and similarly for |moduleSeq| and |connectionSeq|). |unconnected| represents the number of unconnected basic entities (i.e. with no links) in a module. The meaning of |modulesUp| will be explained in the next section where we give details of how links are described. It should be noted that we have been very careful about the ordering of information in these messages (links are described after details of which modules there are, and names have been specified for modules and basic entities) to enable a recipient of the message to decode them. By the time they get to link descriptions they will already know all names (codes) for entities and modules. It should also be noted that if a system has some isolated entities (entities with no connections to others), then we do not need to name these, but can simply say how many there are. This is a consequence of the fact that we are only trying to communicate the *structure* of the system, and not any other information such as the arbitrary names a designer might have used for the various components.

3.3 Describing Links

Consider the HMD shown in Figure 1, and in particular consider the link from basic entity n_4 to basic entity n_7. Now if we assume that the names of the basic entities and modules are not necessarily unique as they have been shown in diagrams, but are only unique within the module in which they occur, then we have to use "path names" to describe a link. If we describe each link at the basic entity which is the start of the link, then we need to give a "route" from the current basic entity, up through some number of modules (|modulesUP|), and then down again to the destination of the link. The module one has to "climb up" to is the least common ancestor (in the module tree of the HMD) of the two basic entities involved. So, for the link we are considering, the destination of the link can be described by:

$$2 \;\; A3 \;\; A4 \;\; n7$$

meaning that from node 4 we go up 2 levels in the module hierarchy and then down through A4 and A3 to node 7.

A basic entity which has several outgoing links will be described by giving the number of outgoing links (|connectionSeq|), followed by a sequence of such "paths", one for each link. It should be noted that there will be one such "entityLinkinfo" entry for each connected (i.e. named) entity, and these entries will occur in the order in which the entities were named in tyhe previous part of the message giving the module structure and naming the modules and entities.

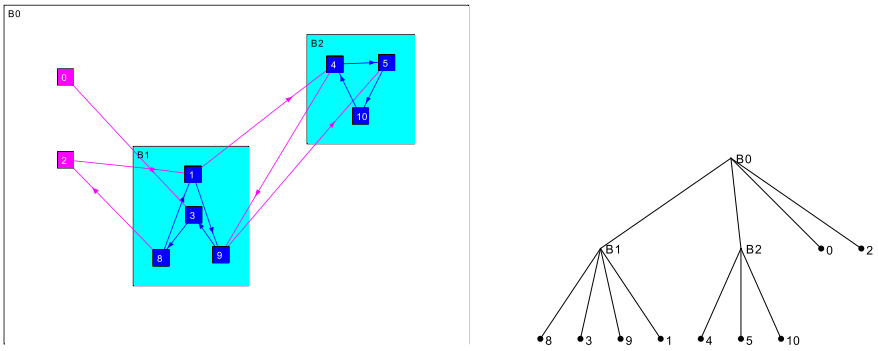

Figure 5: Another HMD and its module tree

3.4 An Example

Consider the HMD shown in Figure 5. This will be described by the message below (indented, and partially commented, for clarity):

0 2 *(0 unconnected entities, 2 basic entities)*
    n0 *(name of node)*
    n2 *(name of node)*
2 *(2 sub-modules)*
    B1 *name of first sub-module)*
        0 4 *(0 unconnected entities, 4 basic entities)*
            n1 *(name of node)*
            n3 *(name of node)*
            n8 *(name of node)*
            n9 *(name of node)*
        0 *(0 sub-modules in B1)*
    B2 *(name of second sub-module)*
        0 3 *(0 unconnected entities, 3 others)*
            n4 *(name of node)*
            n5 *(name of node)*
            n10 *(name of node)*
        0 *(0 submodules in B2)*
1 0 B1 n3 *(entity n0 info : 1 outgoing link, to B1 n3)*
1 0 B1 n1 *(entity n2 info: 1 outgoing link to B1 n1)*
2 1 B2 n4 0 n9 *(entity n1 info: 2 outgoing links)*
1 0 n8 *(entity n3 info: 1 outgoing link)*
2 0 n1 1 n2 *(entity n8 info: 2 outgoing links)*

2 0 n3 1 B2 n5 *(entity n9 info: 2 outgoing links)*
2 0 n5 1 B1 n9 *(entity n4 info: 2 outgoing links)*
1 0 n10 *(entity n5 info: 1 outgoing link from n5)*
1 0 n4 *(entity n10 info: 1 outgoing link from n10)*

3.5 Messages as Bitstrings

Messages in the format just described consist of two types of symbol:

1. integers
2. names for basic entities or modules

We therefore need to find good encodings (as strings of bits) for both of these. A point to bear in mind is that *these encodings need to be decodeable by a recipient of the message*, and that, as we are placing no upper bounds on the size of the systems we are considering, a fixed length encoding for the integers cannot be used.

3.5.1 Code Lengths for Integers

For the integers this simply means that both sender and recipient have to agree on some prefix code[6] for the integers. For our purposes, all we need to know is the length of such a code. One possibility (for positive, non-zero integers n) has length:

$$l(n) = \log_2(n) + 2\log_2\log_2(n+1) + 1$$

which roughly corresponds to sending the binary representation of n, preceded by (for decoding purposes) the length of this binary representation (with extra "doubled up" bits to ensure unique decodeability). Since we need to be able to communicate zero as well, we will send the code for $n+1$ to communicate an integer n, and hence use $l(n+1)$ as the length of the code for integer n.

3.5.2 Code Lengths for Names

Before discussing the lengths of codes for names it should be stressed that we only need to worry about having unique names within modules, since we are using "path names" as described earlier to name entities external to a module from within it. In the usual information theoretic way, we will minimise message length if we assign short codes to frequently occurring names, and longer codes to less frequently occurring ones. It should also be noted that the module and entity names occurring in the above message format before information about the module or entity are there to act as a "name table" to ensure decodeability of the message by a recipient who

does not have prior knowledge of the relative frequencies of the various symbols occurring in the message. So how often is each name used in the message. A little thought will show that the names of a connected basic entity n will occur in the message once for each incoming edge of n, and once to specify the name prior to the edge descriptions, giving a total of:

$$f(n) = indegree(n) + 1$$

Similarly, the name of each module m will occur

$$f(m) = indegree(m) + 1$$

times. Additionally, the first occurrence of each of these names (essentially telling the recipient what name (code) is being used, so they can recognise it in the future, has to be preceded by its length, so the recipient can tell when the name ends.

Information theory tells us that when transmitting a message containing symbols s_i occurring with frequencies f_i, then we need roughly

$$-\log_2\left(\frac{f_i}{f}\right)$$

bits (where $f = \sum_i f_i$) for the code for s_i, and that the message length will be minimised if we use codes of that length. Furthermore, information theory tells us that such codes exist. Furthermore, because of our use of "route descriptions" to name entities connected to a node, we need only guarantee unique names for the basic entities and submodules *within their containing module*. Accordingly, we can take the length of the code for a name to be:

$$-\log_2\left(\frac{f(n)}{\sum_x f(x)}\right)$$

where x ranges over the named basic entities and sub-modules of the module n is in. In other words, the code lengths for names are determined by the *relative frequencies* of the names. The actual frequencies are determined globally, but the relative frequencies are determined by using these global frequencies locally within a module to compute the lengths of the local names.

3.6 The Complexity Formula

In the light of the above discussion the formula for the complexity $\psi(X)$ of a HMD X is given by:

$$\psi(X) = \sum_{m \in \mathcal{M}} \left(\begin{array}{l} l(|U_m|+1) + l(|C_m|+1) + l(|M_m|+1) \\ + \sum_{n \in C_m \cup M_m} \left[l\left(-\log\left(\frac{f_n}{F_m}\right)\right) - f_n \log\left(\frac{f_n}{F_m}\right) \right] \\ + \sum_{n \in C_m} \sum_{n' \in outNodes(n)} l(relDepth(n, lca(n,n'))) \end{array} \right)$$

where $\mathcal{M}$ is the set of all the modules in the HMD X, l represents the code lengths for integers discussed earlier, U_m is the set of unconnected basic entities in module m, C_m is the set of connected basic entities in module m, M_m is the set of submodules of module m, f_n is the frequency of entity n (submodule or connected basic entity) in module m, and $F_m = \sum_{n \in C_m \cup M_m} f_n$. This formula contains terms for the code lengths of all the various integers that need to be sent, the contribution all the occurrences of the various names make, and the extra length information that needs to be sent about each name.

Applying this formula to the HMDs shown in Figures 1 and 2, the first has a complexity value of 398.5 bits, while the second has complexity value 348.7, agreeing with our intuition that Figure 2 is somehow "nicer" (less complex) than Figure 1.

4 Evolutionary Search Through the Space of HMDs of a System

Trying to find the optimal HMD for a system involves a search through a very large space with many local optima. In [15] this was searched using essentially a heuristically based beam search[12]. This proved to be rather slow, and sometimes had trouble finding the best HMD. In the work described here a Genetic Algorithm is used.

Genetic Algorithms (GAs)[3, 8, 4] are a stochastic search technique that often work well in difficult search landscapes. Such algorithms work by maintaining a population of individuals, each of which encode a possible solution to some problem. Inspired by ideas taken from biology and natural evolution, A GA essentially works by repeatedly:

1. pick some individuals to reproduce, based on their "fitness"
2. produce some "children" from the chosen individuals, using processes of "crossover" and "mutation" to mix features of the "parents" solutions, and to introduce some random variations
3. replace one or more members of the original population by one or more of the resulting children

Typically GAs are considered to have genomes (the encoding of the solution to the problem) which are string-like, often consisting of a string of bits giving a binary encoding of the solution. In the closely related field of Genetic Programming (GP) [5] this restriction is removed allowing genomes with a tree-like or even graph-like structure. This work draws quite strongly on some of the work in the GP tradition.

Another distinction often made in the GA literature is between the "classical" generational GA , and what have come to be known as distributed "steady-state" GAs[2]. In these the population is spatially distributed (usually over a 2-dimensional grid), and individuals only "mate" with those in some neighbourhood of themselves. Additionally children are put back in the population immediately they are produced, rather than waiting until an entire new population of children has been produced. In this work we use a distributed steady state GA as described below.

So let S be some system for which we are trying to find the best HMD. The genome for an individual X in the GA simply consists of a HMD for S. The fitness of X is given by:

$$f(X) = \frac{1}{\psi(X)}$$

where $\psi(X)$ is the complexity of the HMD represented by X. Maximizing this fitness will therefore minimize the complexity of the HMD. The population of individuals is arranged on an $N \times N$ toroidal grid. The algorithm then proceeds as follows:

repeat until decide to stop
  with uniform probability pick a random individual on the grid (parent p_1)
  find its fittest neighbour (parent p_2)
  with probability P_c:
    produce a child c from p_1 and p_2 by crossover (randomly choosing p_1 or p_2 to be primary parent)
  otherwise with probability $1 - P_c$
    let the child c be either p_1 or p_2 (randomly chosen)
  with probability P_m mutate c
  replace (in the population) the least fit of p_1 and p_2 by c
endrepeat

In all the examples given later in this paper, the crossover probability P_c was 0.5, and the mutation probability P_m was 1.0, as these values seem to give good results. Further work needs to be done to find the optimal values of these parameters.

In what follows we will often refer to a "generation" of this GA, meaning the process of N^2 iterations of the outer loop in the above algorithm i.e. we have put back into the population a number of children equal to the original population size. It should be noted that this does not imply that every member of the population has been replaced. Indeed the replacement strategy we are using (children replace weakest parent) implicitly implies a certain type of elitism in that the fittest individ-

uals in a neighbourhood can never be replaced until there is another fitter individual in the neighbourhood.

4.1 Crossover

Defining a suitable crossover operation which produces a new HMD from two "parent" HMDs is not entirely trivial. Given a fixed underlying system S we can think of a HMD as simply being a tree structure over the nodes of S. We do not need to take the links into consideration as they are the same for all HMDs over S (of course we need to consider the links when computing the *fitness* of the HMD their description lengths form the dominant part of ψ). Since each genome can therefore be encoded by a tree structure, we have based our crossover operation on the tree-based crossover operation used by Koza[5] in his Genetic Programming work. However, using Koza's original tree crossover is not guaranteed to give a legal HMD of S, and we have to repair the result so that it is legal. This is easily done using the concatenation operator described below.

4.1.1 Concatenating Two HMDs

Given two HMDs H_1 and H_2 for two underlying systems S_1 and S_2 one can define a family of concatenation operators $\circ_M$ (one for each module M occurring in H_1) for "gluing together" the two HMDs to form a new HMD $H_1 \circ_M H_2$ for the system $S = S_1 \cup S_2$. Although these concatenation operators were originally defined [7] in order to prove various theoretical properties of our measure (in particular that it satisfies a minor variant of Weyuker's axioms [13] for a complexity measure), it turns out that we can define our crossover operator in terms of them.

Defining the concatenation operators is not entirely trivial because the two original HMDs may share some of their basic entities, and these may be in different places in the module hierarchy in each of the two HMDs. When this situation happens the question arises about where in the resulting HMD these basic entities should go. The basic idea is to treat the first HMD as primary, and all its structure is carried over into the new HMD, which then gets *whatever is necessary* from the second HMD to complete the process. To do this we first concatenate the module trees of the two HMDs, and then turn the resulting module tree back into a HMD by defining its edge set to be the union of the edge sets from the original two HMDs.

4.1.2 Concatenating Two Module Trees

Let S_1 and S_2 be two systems, and let T_1 and T_2 be module trees for S_1 and S_2 respectively. Let M be a module of S_1, corresponding to some non-leaf node M' in

T_1. Define $S_1 \circ_M S_2$ (read *the concatenation of S_2 onto S_1 at M*) by the following process:

1. Let T be a copy of T_1, and T_2' a copy of T_2.
2. Delete from T_2' any basic entity nodes corresponding that also occur in T. This may leave some "module nodes" with no children.
3. **while** T_2' *has module nodes with no children* **do**
   delete such nodes from T_2'
4. Add all remaining children (immediate subtrees) of T_2' (basic entities and modules) as extra children of node M in T
5. Form $E = E_1 \cup E_2$.
6. The resulting concatenated HMD is given by $\langle T, E \rangle$.

For example if T_1 is the tree shown in Figure 4, and T_2 is the tree in figure 5, then T_2' after step 3 is shown in Figure 6 (note that for this example Step 3 itself was not really necessary).

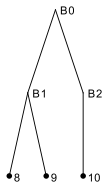

Figure 6: T_2', after step 3 of concatenating the module trees in Figures 4 and 5.

Figure 7 shows the final result of concatenating the HMD shown in Figure 5 onto the HMD shown in Figure 4 at A0. Similarly, Figure 8 shows the results of concatenating the HMD of Figure 5 onto the HMD of Figure 4 at A2.

4.1.3 The Crossover Operation

The crossover operator is most easily considered as an operation on the module trees of the two HMDs involved. Figure 9 shows a HMD (actually the same as in Figure 1), together with its module tree. Similarly, Figure 10 shows another HMD and its module tree for the same underlying system.

Let the two module trees be T_1 and T_2. Then the crossover operator works as follows:

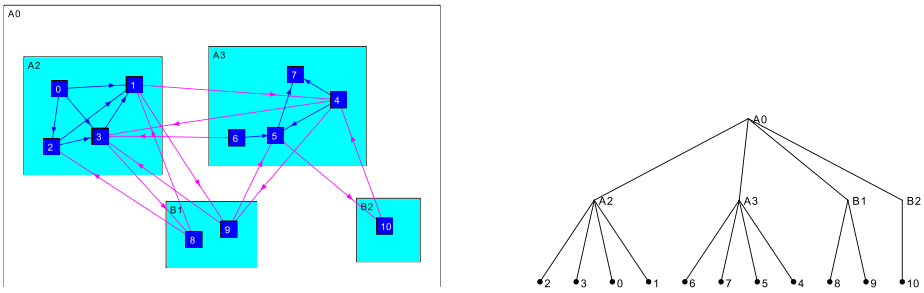

Figure 7: Concatenation of Fig. 5 onto Fig. 4at A0

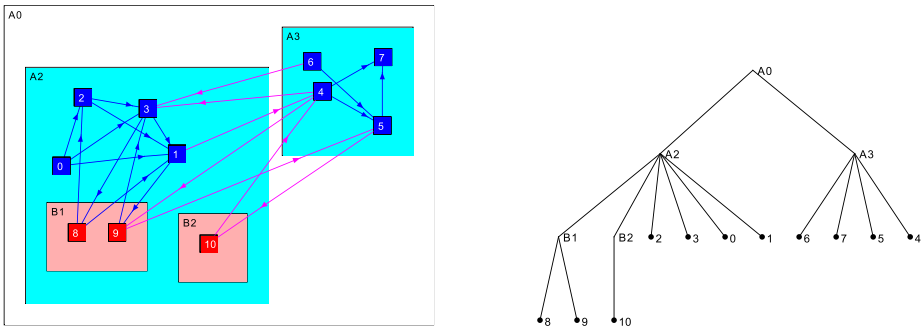

Figure 8: Concatenation at A2

1. Pick a random node (subtree) S_1 in T_1, and pick a random node S_2 in T_2.
2. Remove the subtree whose root is S_1 from T_1.
3. Concatenate the resulting system with the system with module tree S_2, at the ancestor of node S_1.
4. Concatenate the resulting system with T_1 at any node (we use the root).

The effect of performing crossover on the HMDs in Figures 9 and 10 at A3 and B3 is illustrated in Figure 11. It should be noted that, unlike most crossover operations on string-like genomes (e.g. 1-point, 2-point, or uniform crossover), crossing a genome with itself does not in general result in children identical to the parents. For example, consider the HMD shown in Figure 9. The result of crossing it with at itself at A1 and A2 is shown in Figure 12.

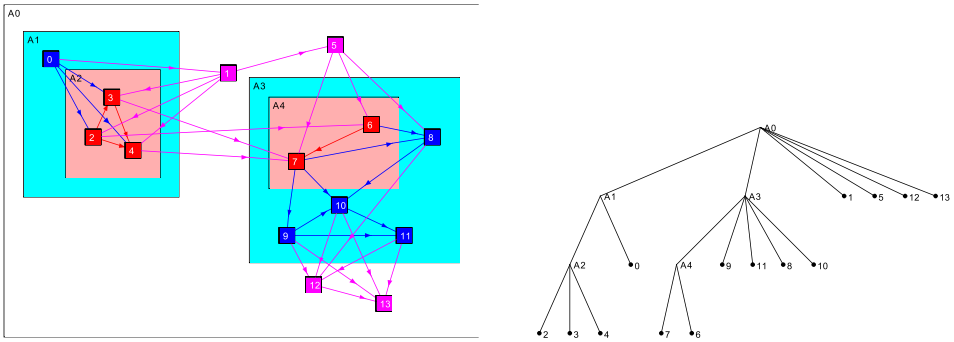

Figure 9: The HMD of Figure 1 together with its module tree

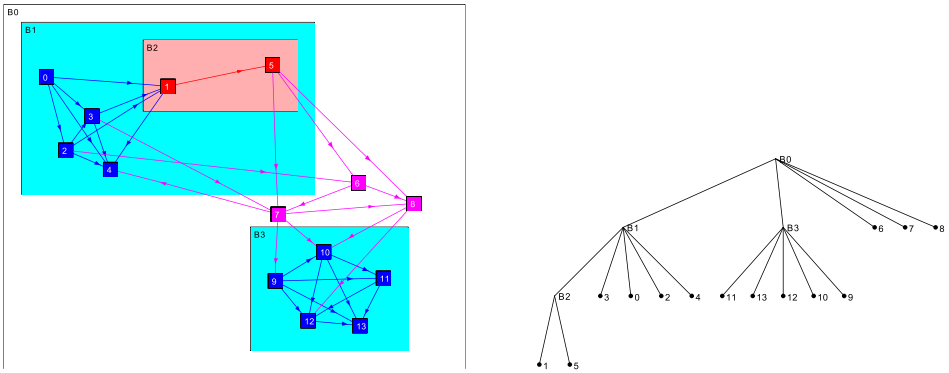

Figure 10: Another HMD for the system shown in Figure 3

4.2 The Mutation Operator

When a HMD is mutated, three different operations are applied (each with some appropriate probability). These three operations can all be thought of as operations on the module tree for the HMD. They are:

- **moving** a randomly chosen node (which can be a basic entity or a submodule) from where it is in the tree into another randomly chosen module of the tree, subject to the caveat that this must not introduce cycles into the module tree i.e. it must remain a tree. This can be guaranteed if the second randomly chosen module is not a daughter (in the module tree) of the first. Figure 13 shows the result of taking the HMD of Figure 9 and moving Module A2 into A4. Similarly Figure 14 shows the effect of starting with the same HMD, and moving node 0 into A2.

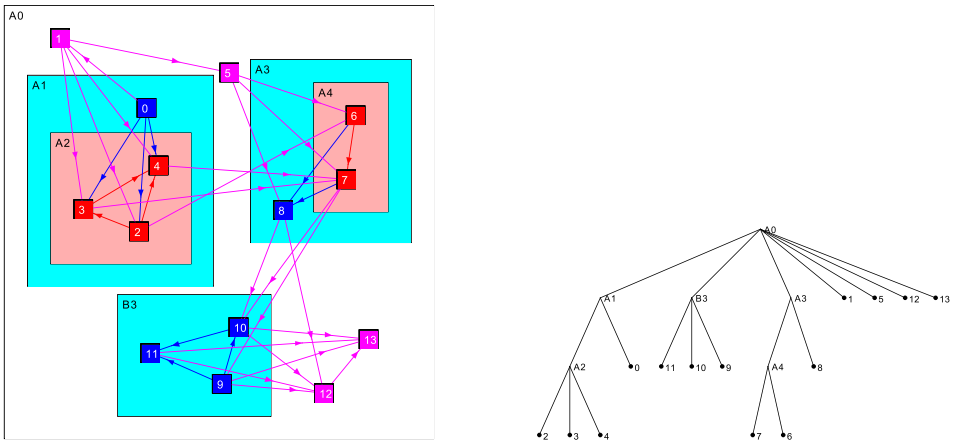

Figure 11: HMD and its module tree resulting from crossover at A3 and B3

- **modularise** the nodes of some randomly chosen module. This involves making a new module containing just the basic entities of the chosen module, and making the new module a sub-module of the original module. Figure 15 shows the effect of "modularising" the basic entities in A3 on the HMD of Figure 9.

- **remove** a module "boundary". This involves randomly choosing a (non-top-level) module in the HMD, and making all its immediate daughters (sub-modules and basic entities) daughters of its parent module. Figure 16 shows the effect of doing this to module A2 of the HMD of Figure 9.

In all the examples described later in this paper modularising the nodes of some module was done with probability 0.8, followed by module boundary removal with probability 0.05, finally followed by moving a component elsewhere in the module tree with probability 1.0.

4.3 The Initial Population

The initial population for the GA consists of N^2 (the grid world is $N \times N$) randomly created genotypes, all based on the system for which we are trying to discover a good HMD. Each of these individuals is created by randomly mutating some number of times (using the mutation operators just discussed) a particular "seed" individual[3].

[3]In all the experiments reported here each individual in the initial population was created by mutating the seed indidual n_{mut} times where n_{mut} was a randomly generated (newly generated for each new individual) number between 1 and 3

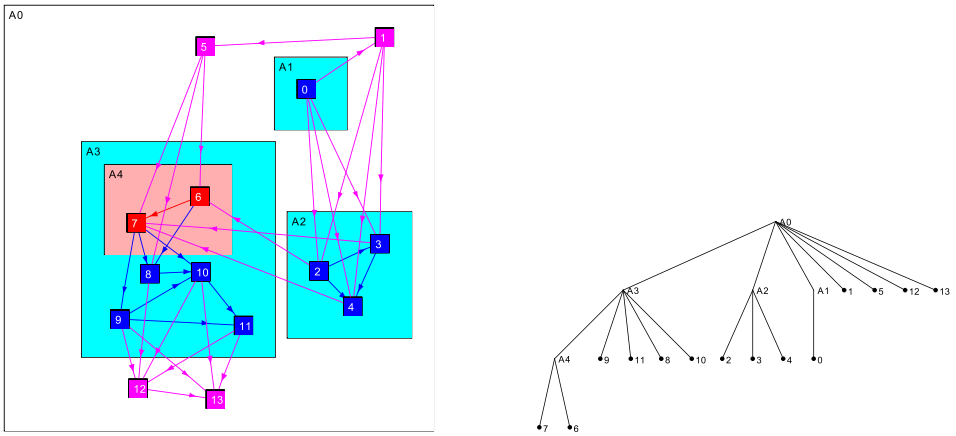

Figure 12: HMD resulting from crossing the HMD of Figure 9 with itself at A1 and A2

The initial "seed" individual is constructed by "modularising" the initial system. This is done by "wrapping" an extra module around every basic node and module of the system. The resulting "initial individual" corresponding to the system shown in Figure 3 is shown in Figure 17. This structure gives mutation in particular a great deal to work with as it moves components around in the system.

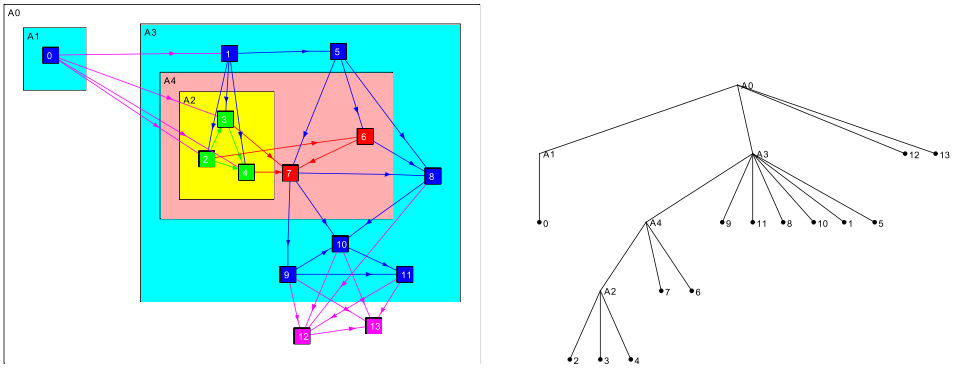

Figure 13: Mutation – moving module A2 from A1 into A4

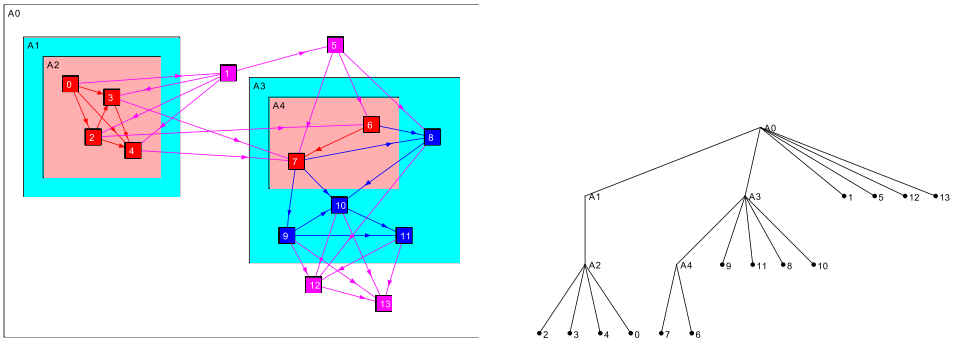

Figure 14: Mutation – moving node 0 from A1 into A2

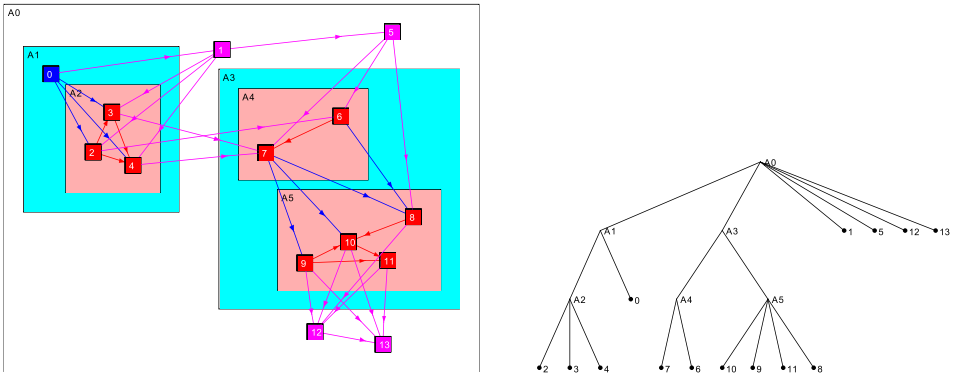

Figure 15: Mutation – making a new module from nodes of A3

5 Some Examples

5.1 Example 1

We will begin by looking at the rather simple system S_1, shown in Figure 18. This system has complexity 208.37 bits. An initial population of 100 (10×10 grid world) random individuals created from this system as described earlier was created, and the GA then allowed to run. The best individual after 2 "generations" is shown in Figure 19, and has complexity 249.91 bits (all early stages in the evolution of a good HMD have higher complexity than the original due to all the extra modules in the individuals in the initial population). This particular system is simple enough that already signs of grouping starting to occur can be seen. The best individual after 10 "generations" is shown in Figure 20 (complexity 230.0 bits), where the two main groupings are clearly apparent, and by 20 generations (shown

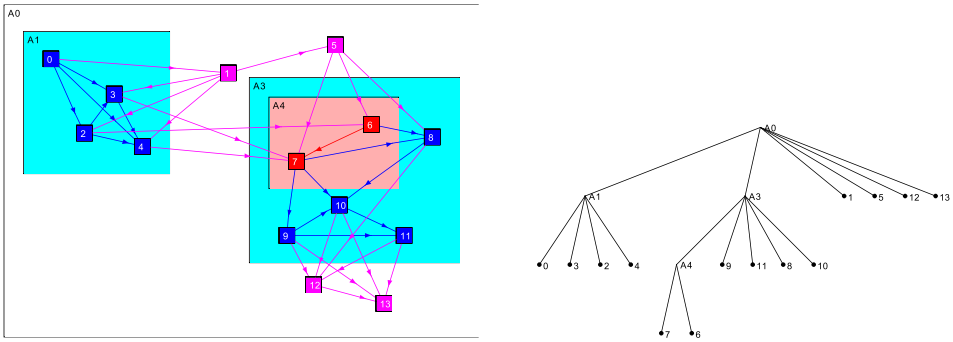

Figure 16: Mutation – "removing the boundary" of module A2

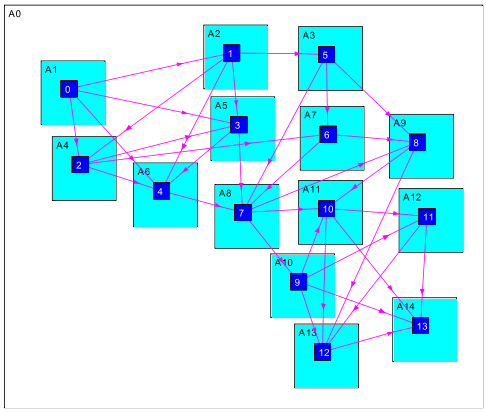

Figure 17: The initial "modularised individual" corresponding to the system in Figure 3.

in Figure 21) the system has found what we believe to be the optimal HMD for this system, with complexity 202.3 bits. Note that this is lower than the complexity of the original system. The fact that the original flat system is rather small makes this flat system a local optimum that is quite close in fitness to the final best HMD. This is basically the reason why it is not until quite late in the evolutionary process that a better HMD is found – normally this happens much sooner.

5.2 Example 2

We will now look at the slightly larger system that we have used several times in this paper (shown in Figure 3). We will refer to this system as S_2, and it has a

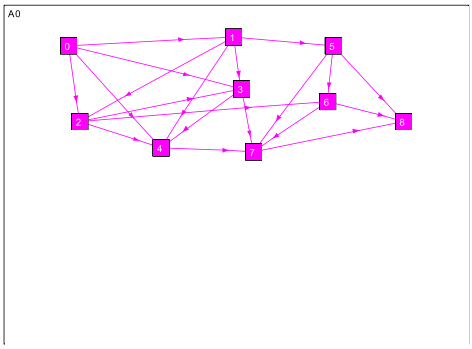

Figure 18: A rather simple system S_1.

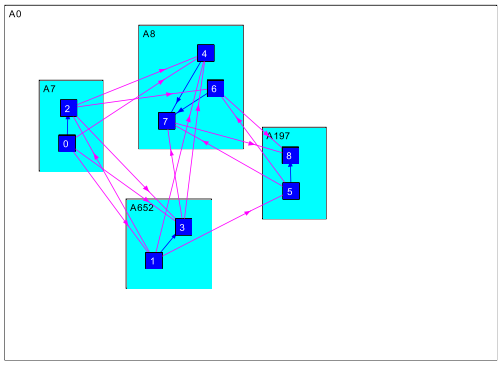

Figure 19: S_1 after 2 generations.

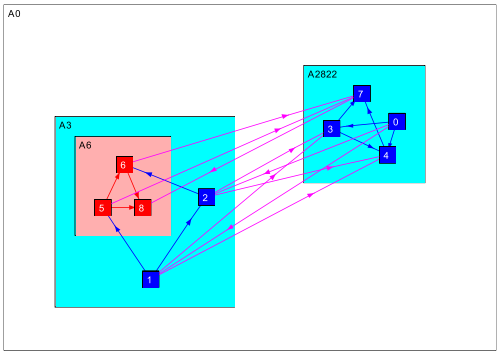

Figure 20: S_1 after 10 generations.

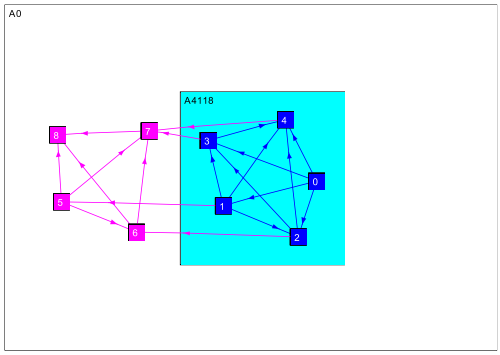

Figure 21: S_1 after 20 generations.

complexity of 372.2 bits. After 50 generations we have found quite a reasonable modular structure (shown in Figure 22 with a complexity of 350.7 bits (better than the original flat system, confirming our intuition that this is not a bad HMD for this system), and by 120 generations (shown in Figure 23) the system has again found what we believe to be the optimal modularisation for this example. This final HMD has a complexity of 342.1 bits. It should be noted here that the HMD for this system shown in Figure 2 has a complexity of 348.7 bits, which may account for why we tend to think it is a better hierarchical modular decomposition for this system than that in Figure 22. Note though that module A2 in Figure 2 is not big enough for it to be worth having as a separate module, given also that there are only two other modules in the system, and hence Figure 23 is better.

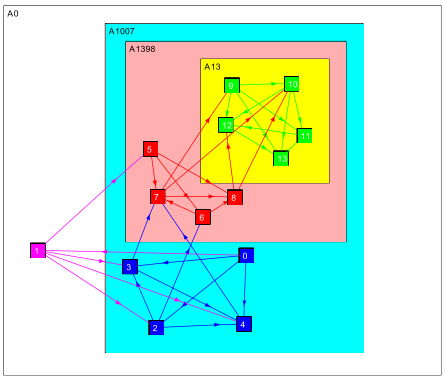

Figure 22: S_2 after 50 generations

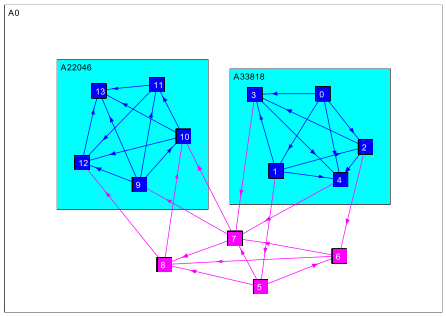

Figure 23: S_2 after 120 generations

5.3 Example 3

The next example S_3 makes it clear that the system can easily find more deeply nested submodules if necessary. Figure 24 shows an elaboration of the earlier example, in which we have added some extra structure to the system of Figure 3. After 200 generations the system has found the HMD shown in Figure 25.

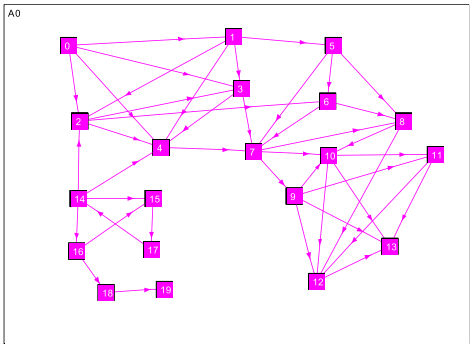

Figure 24: The initial system S_3.

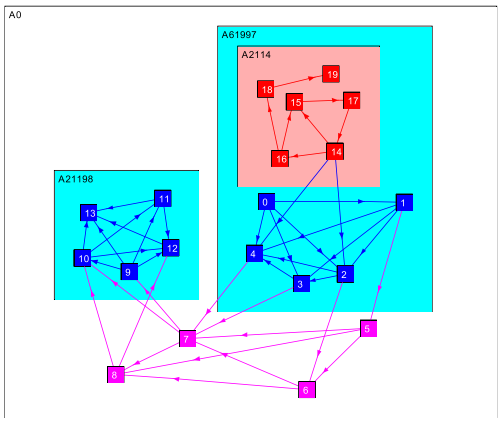

Figure 25: S_3 after 200 generations

5.4 A Software Design Example

All the previous examples have been rather abstract in nature, not corresponding to any real artifact that we might actually be really interested in, although they do show the potential power of the complexity metric together with evolutionary

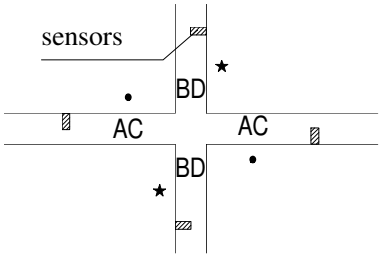

Figure 26: The traffic lights example

techniques. In this last example we will look at one (fairly small) example of a real software design, published in [10]. The requirement is to control the flow of traffic over a cross-roads, see Figure 26. The sensors provide information about waiting traffic, and the lights are only changed if traffic is waiting and the current pair of lights have been green for some pre-defined interval.

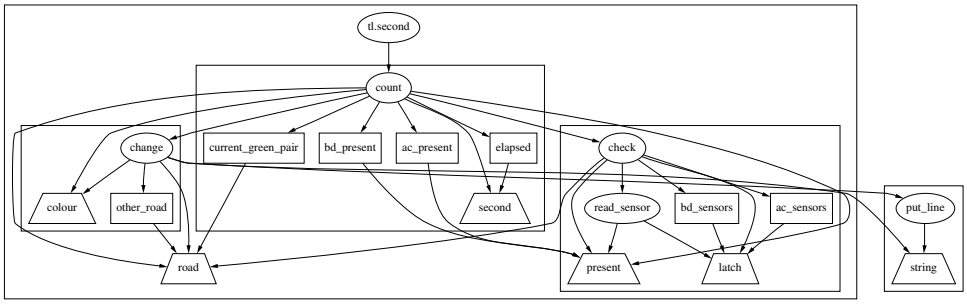

Figure 27: The original HOOD design for the traffic lights example.

We took the original design as proposed by Robinson[10], shown in Figure 27 and input it to our system as shown in Figure 28. Since the original design has an external "library" module (shown on the right of Figure 27) containing *put_line* and *string* we have had to artifically add this as an extra module of the system, with an extra module reprenting the rest of the system[4]. Furthermore we reversed the direction of all links in the original design. This is because our encoding of HMDs rather arbitrarily chose to associate link information with the source node of each link, giving the system a bias towards destination nodes contributing more to the description length(names of sources are used once, while names for destina-

[4]We are currently working to remedy this defect of our current system.

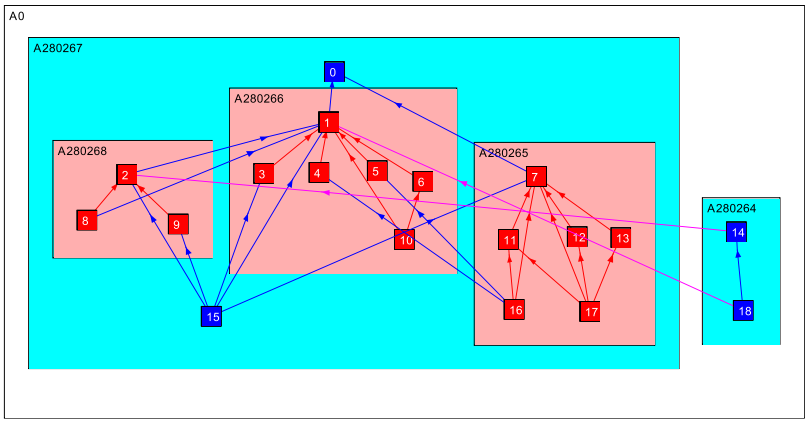

Figure 28: The "reversed" HMD graph corresponding to the design shown in Figure 27.

tions are used once for each time the node is a destination). Whether or not this is appropriate depends to some extent on the semantics of the links. In the original HOOD design a link from n_1 to n_2 is interpreted as "n_1 depends on n_2". We prefer to reverse them and interpret them as "n_2 provides services to n_1". In this way we essentially penalise the user of services, rather than the supplier. The complexity of this HMD is 415.18 bits. We then ran our system on the "flat"system underlying this version of Robinson's design (shown in Figure 29), to produce a new HMD shown in Figure 30 with complexity 398.5 bits. Because our system as currently implemented has no knowledge of constraints imposed on real software systems (such as the fact that library objects such as *put_line* and *string* cannot be moved around inside the system as they are external) we then modified the resulting system by hand to produce the HMD shown in Figure 31. This has complexity 410.27 bits, which is still lower than our version original. The new HMD is better than the original in a variety of ways:

- The top level module contains all the entities concerned with managing time (the datatype *second*, *count*, *elapsed* as well as the top-level "start" node *tl.second*).
- The module named A22249 in our new HMD contains everything that was in the original module contianing *check*, but has had the two variables *ac_present* and *bd_present* moved into it. These are both of type *present* which is also in that module (node number 16).
- The variables *other_road* and current_green_pair of type *road* have been moved

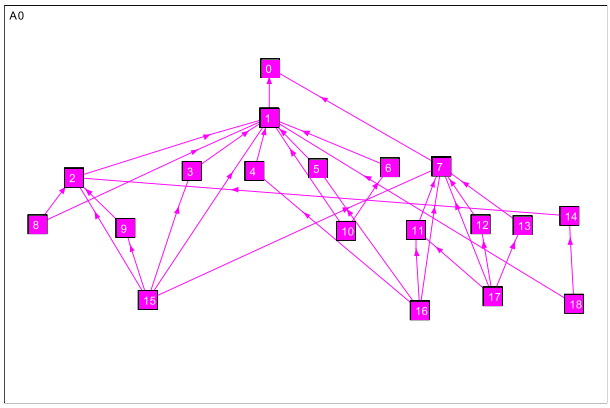

Figure 29: The flat system graph corresponding to the design shown in Figure 27.

into the other module together with type *road* itself., and have been grouped rather nicely with the operation *change*(responsible for changing the colour of the lights) and the datatype *colour* which depends on the roads and on the datatype.

Thus it would appear that the system has managed to find a more logical grouping of the components of the design than the original.

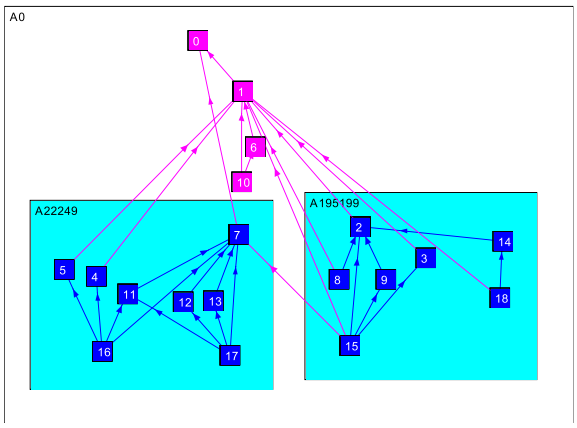

Figure 30: The initial evolved traffic lights design.

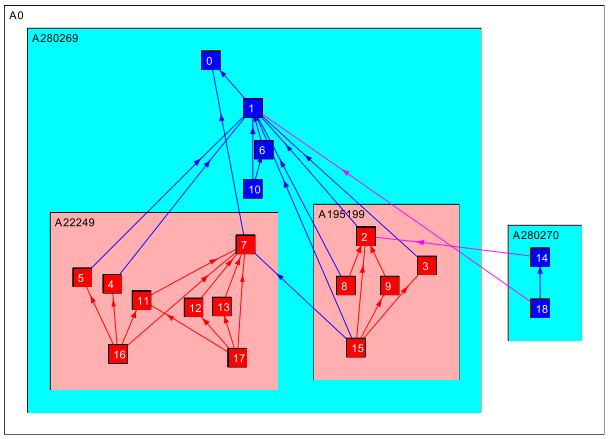

Figure 31: The evolved traffic lights system after "hand adjustment".

6 Discussion and Future Work

In this paper we have described an evolutionary system which finds good hierarchical decompositions of complex systems. Part of this work has involved (as is common in GA applications) developing novel domain specific mutation and crossover operators. However, the real bulk of the work has really been in defining a suitable fitness function, and in this paper we have described a particular information theoretic function for the complexity of a HMD based on a minimum description length principle. We have implemented the system as a JAVA program, and indeed all the diagrams in this paper (except for Figure 27) have been produced by the system[5].

Initially this research used the complexity measure devised by Wood[15], theoretically investigated further in [7]. However, on many occasions this measure did not give as good results as were hoped, and in this paper a new improved measure has been described. Unlike Wood's original measure the new measure is sensitive to the directionality of the links, giving it much greater domain of applicability. Furthermore, it is not "reluctant" to let one form modules with more than about five components.

Like most research, the work described here has opened up many new avenues of research. Some of these involve things to do with the internals of the GA itself-

[5]The system allows one to move components around "on screen" by means of a mouse, and the layout of many of the diagrams was therefore improved "by hand". However (except for the case of the HOOD design as discussed earlier) this was always done so as to leave the hierarchical module structure intact, and only improved the appearance of the diagrams, not their fundamental structure.

such as finding the best values for the various parameters involved, investigating the complexity of the algorithm, and so on. More interestingly though, future research that we are undertaking will be into *applications*. Most of what has been described has been in terms of abstract graphs, and hence ought to be applicable in many domains where such graphs are used to describe systems. One area we are starting to investigate is the applicability of our system to *reverse engineering* problems (both for software systems, and for hardware systems). We are also interested in seeing if the system can be applied to helping designers *improve* their designs. Our early experiments with the traffic lights example described in this paper seem very promising.

References

[1] Briand, L.C., Morasca, S., and Basili, V.R. (1996) Property-based software engineering measurement: Refining the additivity properties. *IEEE Transactions on Software Engineering*, 22(1):68–86.

[2] Collins, R. and Jefferson, D. (1991) Selection in massively parallel genetic algorithms. *Proceedings of the Fourth International Confernce on Genetic Algorithms, ICGA-91* Belew, R.K. and Booker, L.B. (eds.), Morgan Kaufmann.

[3] Goldberg, D.E. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

[4] Holland, J.H. (1975) *Adaptation in Natural and Artificial Systems*. Now published by MIT Press.

[5] Koza, J.R. (1992) *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

[6] Li, M. and Vitanyi, P. (1997) *An Introduction to Kolmogorov Complexity Theory and Its Applications*. Springer-Verlag.

[7] Lutz, R. and Wood, J.A.(2000) A Minimum Description Length Approach to Measuring the Structural Complexity of Software Design Graphs. CSRP-517, School of Cognitive and Computing Sciences, University of Sussex.

[8] Mitchell, M. (1996) *An Introduction to Genetic Algorithms*. MIT Press.

[9] Rissanen, J. (1978) Modelling by the shortest data description. *Automatica-J.IFAC*, 14, pp.465–471.

[10] Robinson, P.J. (1992) *HOOD: Hierarchical Object-Oriented Design*. Prentice-Hall Object-Oriented Series, Prentice Hall.

[11] Shannon, C.E. (1948) The mathematical theory of communications. *Bell System Technical Journal* 27:379–423, 623–656.

[12] Thornton, C.J. and du Boulay, B. (1992) *Artificial Intelligence Through Search*. Intellect, Oxford, England.

[13] Weyuker, E.J. (1988) Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, SE-14(9), pp. 1357-65.

[14] Witten, I.H., Neal, R.M., and Clear, J.G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

[15] Wood, J.A. (1998) Improving Software Designs via the Minimum Description Length Principle. Ph.D. Thesis, University of Sussex.