In *Software Engineering Journal 9(1)*, 27-38, 1994.

TOWARDS A NEW CONCEPT OF SOFTWARE

J Gerard Wolff

July 1993

School of Electronic Engineering and Computer Systems, University of Wales, Dean Street, Bangor, Gwynedd, LL57 1UT, UK. Telephone: +44 248 382691. Fax: +44 248 361429. E-mail: gerry@sees.bangor.ac.uk.

ABSTRACT

The article develops three inter-related ideas: that a ‘well structured’ software system may be understood as a compressed representation of its inputs and its outputs; that software design may be seen as a process of information compression; and that the execution of software may also be understood as a process of information compression.

Information compression may be achieved by the comparison or matching of patterns, the merging or unification of patterns which are the same, and a search through the space of possible unifications to find the set or sets giving the most compression.

A prototype has been developed of a ‘new generation’ computing system which is dedicated to information compression by pattern matching, unification and search. Examples from the prototype are presented to illustrate the themes of the article.

These ideas can mean increased efficiency in software development and increased quality in the resulting product.

Planned developments of the ideas are briefly described.

1 Introduction

This article describes a new concept of software which is emerging from current research. Elements of the concept may be seen in existing ideas about software, but the concept itself represents a radical departure from established views about what software is and what it does.

There is potential in these ideas for substantial gains in productivity in the processes of software development and maintenance and for similar gains in the quality of software products.

In ascending order of novelty, the main features of the concept are these:

- A 'well structured' function, program or information system may be seen as a compressed representation of its inputs and its outputs.
- The process of designing well structured software may be seen as a process of information compression.
- The execution of software may also be understood as a process of information compression.

These ideas derive from a new theory of computing called 'SP' and a 'new generation' computing system which is based on the theory. The SP theory and the SP system are not yet mature but enough progress has been made to show what these ideas can mean for the development and application of software in the future.

The main sections of the article which follow are:

- Outline descriptions of: the SP theory, a prototype of the SP system which is called SP6, and the SP language which is being developed as part of the system.
- Three sections describing the three ideas summarised above. Examples are given from SP6.
- A brief summary of the implications of these ideas for the practice of software engineering.
- How the ideas may be developed in the future.

Related research is described and discussed at appropriate points throughout the article.

2 Background

This section describes the SP theory, the SP6 prototype and the SP language in outline. More detail, discussion and explanation may be found in [36], [35], [33], Chapter 5, and[34]. Some background thinking is described in [31].

2.1 The SP theory

The central idea in the SP theory is the conjecture that many aspects of computing, not normally regarded in this way, may usefully be seen as *information compression*. Information compression may be seen as a process which increases the *simplicity* of data (by reduction of redundancy) whilst preserving as much as possible of the non-redundant descriptive *power* in the data. Hence the mnemonic ‘SP’ applied to ideas in this area.

The extraction of *redundancy* from information, which is the key to information compression, may be achieved by the comparison or *matching* of patterns, the merging or *unification*[1] of patterns which are the same, together with a *search* through the space of possible unifications to find the set or sets giving most compression.

Since the search space is normally very large (where there is recursion, the search space can be infinitely large), it is usually necessary to use a ‘hill climbing’ technique or something equivalent to maximise the amount of redundancy found for a given search effort.

Examples given later should help to clarify these ideas.

2.2 The SP system

In the SP view of computing, conventional computers do pattern matching, unification and search but do not do them very well. The long-term aim of this programme of research is to develop a system which is dedicated to these three processes, which rationalises them and exploits their potential more fully. When it is fully developed, the SP system will almost certainly exploit high levels of parallelism in processing and it is likely that the hardware architecture will differ radically from the organisation of conventional computers.

The current prototype, called SP6, has been developed as a software simulation on a conventional computer. Although this is an early prototype and has several shortcomings, it already demonstrates several different capabilities, with varying degrees of success. These include logical deduction, probabilistic inference, unsupervised learning from examples, information retrieval and pattern recognition (see [33], Chapter 5). These capabilities have not been individually created. They all flow from the processes of pattern matching, unification and search which are central in the system.

2.3 The SP language

The research programme aims to develop a language for the SP system which has a very simple syntax and a semantics provided by the processes of pattern matching, unification and search which are central in the system. It is also intended that the language should be ‘universal’ - that it should be possible to use the language to represent diverse kinds of knowledge, including ‘software’, in an efficient way.

---

1. *Unification* in this article means a simple merging of identical patterns, close to the non-technical meaning of the word. Although there is some risk of confusion with the more complicated concept of ‘unification’ in logic, the term has been adopted here because no other word captures the intended meaning so well.

These are research goals and it is too soon to say whether they can be achieved. The expectation that it may be possible to design such a language is based on two observations:

- That succinct representation of knowledge is equivalent to information compression.
- That a rather small range of representational devices appear in different guises in a wide variety of artificial languages and notations.

The design of the SP language aims to distil the essentials of the main compression techniques into a notation which can express any kind of knowledge without undue redundancy. The current version of the SP language probably does not meet the ideal. But work done so far suggests that the design objectives can be achieved without radical changes.

2.3.1 *Syntax.* The syntax of the current version of the SP language is shown in Fig. 1.

```
Object -> Ordered-AND-object | Unordered-AND-object | OR-object |
                Simple-object
Ordered-AND-object -> ( )  |  ( Body )
Unordered-AND-object -> [ ]  |  [ Body ]
OR-object -> { }  |  { Body }
Body -> Object Body  |  Object
Simple-object -> Symbol  |  _
Symbol -> Symbol Symbch  |  Symbch
Symbch -> A | ... | Z | a | ... | z | 0 | ... | 9 | % | & | # | ...
```

Fig. 1. The syntax of the SP language.

Informally, the language comprises five kinds of object:

- An ‘Ordered-AND-object’ (OAO) represents a sequence. It is similar to a ‘list’ in languages like Lisp but without the internal head-tail structure. In future developments of the SP language, the OAO construct may be generalised to include two-dimensional (or even higher dimensional) patterns.
- An ‘Unordered-AND-object’ (UAO) represents a collection of objects in which order is not significant. It is essentially the same as a ‘bag’ in logic.
- An ‘OR-object’ (ORO) is similar to a UAO but its constituent objects represent alternatives in the given context. There is some uncertainty at present whether the ORO construct represents an ‘exclusive’ or ‘inclusive’ OR relationship or whether it is needed at all.
- A ‘Symbol’ is a string of one or more characters (bounded by spaces or meta-symbols) which is treated by the system as being atomic.
- A ‘Variable’ (‘_’) is a place marker for missing information, rather like an un-named variable in Prolog.

2.3.2 Semantics. The semantics of the SP language are provided by the processes of pattern matching, unification and search which form the core of the SP system. In outline, the search process in SP6 works like this:

```
WHILE more extractable redundancy in the given information DO
BEGIN
        1 Search for redundancy in the given information by
                systematically comparing substrings in the
                information and identifying matching patterns.
        2 Select the pair of matching patterns which represents the
                greatest amount of redundancy.
        3 Merge or ‘unify’ the selected pair of matching patterns.
END
```

Fig. 2. The search process in SP6.

This search method unifies patterns in order of decreasing size (which means decreasing amounts of redundancy). The method is quite effective for finding ‘good’ sets of unifications and compressing the data although it is not guaranteed to find the best possible answer. Its main shortcoming is that it is very inefficient and cannot tackle large data sets. Recent work [31] has largely solved the problems of efficiency and related problems of computational complexity in a new simulation of the SP system called SP20. However, this more recent model is not yet as good as SP6 for demonstrating the ideas which are the subject of this article.

In connection with issues of efficiency and computational complexity, it should be stressed that a computer with a conventional architecture is almost certainly *not* the right platform for SP software. What is inefficient on a conventional machine can be highly efficient and effective when the architecture is changed; likewise for issues of computational complexity [31].

3 Software as a compressed representation of inputs and outputs

This section considers three inter-related propositions:

- That any function, program or information system may be defined in terms of its inputs and its outputs.
- That ‘well structured’ software is a ‘model’ of the domain which it serves.
- That a ‘well structured’ function, program or information system is a compressed representation of its inputs and its outputs.

3.1 Defining a function in terms of inputs and outputs

Functions in computing and mathematics are often defined ‘intensionally’ in terms of rules or operations required to perform the function. But functions are also defined ‘extensionally’ by specifying one or more outputs for every input or combination of inputs [30]. This idea applies to functions of various kinds ('total', 'partial' and 'multi') and also to 'programs' and information systems in general.

For example, the 'exclusive OR' function may be defined in a table like this:

Input1	Input2	Output
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 3. Definition of the XOR function.

A thermostat designed to keep a room at a 'medium' temperature may be defined like this:

Temp	Current state	New state
high	off	off
high	on	off
medium	on	on
medium	off	off
low	on	on
low	off	on

Fig. 4. Definition of a thermostat function.

In this example, the room temperature and the current state of the switch may be regarded as inputs while the new state of the switch may be regarded as an output.

Fig. 5 shows how a function for positive square roots may be defined in terms of inputs and outputs.

Input	Output
1.00	1.00
2.00	1.41
3.00	1.73
4.00	2.00
5.00	2.24
6.00	2.45
7.00	2.65
...	...

Fig. 5. Positive values of the square root function.

In the first two examples, principles and practice coincide: it is quite realistic and convenient to define each of these two functions by a table. But a table which gives a comprehensive definition of the square root function contains an infinite number of entries and cannot be accommodated in any practical system. However, this in itself does not invalidate the principle that any function is ultimately a mapping between its inputs and its outputs and that this mapping is, in effect, a table.

3.2 Software design as modelling

The idea that any function may be defined in terms of its inputs and its outputs relates to another idea which has become prominent in software design: that the organisation of well designed software reflects or ‘models’ the organisation of its external world.

In the early days of machine computing - when all programming was done in assembly language or worse - software design was heavily oriented towards the organisation of the computer. Instructions to load registers, rotate bits in a register or jump from one memory location to another were prominent features of software (and are still used at a ‘low’ level in most software systems).

Since the invention of compiled and interpreted high level languages, there has been a trend in software design to disguise the workings of the underlying machine. Three essentially independent developments have converged on the idea that the organisation of well-designed software should reflect or model the external world with which the system interacts:

- *Structured programming*. In his pioneering work on structured programming, Michael Jackson [16] developed the idea that software should be designed to reflect the structure of its inputs and its outputs.
- *Object oriented design (OOD)*. From the development of Simula in 1967 (see, for example, [5]), Smalltalk (e.g., [11]) and several other ‘object-oriented’ languages, there has been a growing recognition of the advantages of designing software so that objects in the real world and classes of such objects are reflected, one for one, by corresponding ‘objects’ and ‘classes’ in software.
- *Entity-relationship (ER) models*. An established principle in the design of data processing applications is to model the ‘entities’ in the domain to be served and ‘relationships’ between those entities (see, for example, [6]).

3.3 ‘Well structured’ software and information compression

If a software system is to ‘reflect’ or ‘model’ its environment, the description which it embodies should be succinct. If the model contains information which is repeated two or more times in different places then it is a poor model. A ‘well-structured’ design is one that minimises redundancy.

There is no conflict between this idea and the well-known uses of redundancy to increase reliability in the storage of software or in its applications:

- Backup copies of software are redundant in the sense that they repeat the information in the software. But this redundancy, which is useful, has nothing to do with redundancy within each copy, which should, as far as possible, be minimised.

- To reduce the impact of bugs, a given software module may be developed three or more times by independent teams and the several versions may be run at the same time, with voting in case of discrepancies [9]. Again, this kind of redundancy is independent of any redundancy which may exist within each version of the design.

The close connection between software design and minimisation of redundancy can be seen in the organisation of programming languages and systems. At the heart of these systems, there are notational devices which can be, and usually are, used to represent information in a compressed form, without redundancy. Of course, the provision of these notational devices does not, in itself, guarantee that they will be used to best effect.

The parallels which can be seen between syntactic forms used in computer programs and coding techniques used in standard data compression methods are discussed more fully elsewhere [31]. The paragraphs which follow briefly describe four coding devices which are prominent in data compression and can also be seen in the organisation of software.

3.3.1 Chunking and tags: Where a pattern of information is repeated in two or more places in a larger body of information, the two or more instances of the repeated pattern may be merged into a single 'chunk'. This saves unnecessary repetition of the pattern but loses information about the locations of all but one of the instances. If information about these locations is to be preserved, the chunk must be given a brief label or 'tag' and a copy of the tag must be placed in every relevant context.

The clearest example of this technique in software design is the way two or more identical sequences of program statements may be merged into a named 'function', 'procedure' or 'sub-routine' with the name of the function used as a 'call' in each place that it is needed. The body of the function is a chunk and the name of the function is its tag.

Of course, for such reasons as run-time efficiency, software designers do no always merge identical sequences of program statements in this way. And, as discussed below, functions usually also have 'arguments' or 'parameters'. But the existence of these other features and considerations should not obscure the basic idea that the use of named functions in software design is normally a means of avoiding unnecessary redundancy in the design.

3.3.2 Run-length coding: Where two or more patterns of information are the same, and they are repeated one after the other in one part of a body of information, then they may be reduced to one copy and marked with the number of repetitions. For example, if 'ABC' is repeated 1000 times in a contiguous sequence in a body of information then it may be reduced to (ABC)1000. If precise information about the number of repetitions is not needed, then the pattern may be marked merely for repetition, e.g., (ABC)*.

If a sequence of statements in a computer program is repeated in one part of the program, then it is normally reduced to one instance and the iteration marked with a notation like *while ... do*, *repeat ... until* or *for ... do*. This kind of repetition may also be coded using recursion. Iteration and recursion may both be seen as examples in software design of the run-length coding technique for information compression.

3.3.3 Schema plus correction: If two or more patterns of information are similar but not identical, then the parts which are the same may be recorded once while the other parts of the pattern, which vary from one instance to another, may be given as alternatives at appropriate places or omitted altogether. This kind of structure is a 'schema'. Any particular instance may be described by adding one or more 'corrections' to the schema; these corrections can be used to select amongst alternatives in the schema or to fill in gaps.

A typical program or function can be seen as a schema of this kind:

- Non-conditional statements and constants provide the framework which is the same in all runs of the program.
- All variables within a program represent gaps in the schema waiting to be filled, directly or indirectly, by information supplied as 'data', 'arguments' or 'parameters'.
- Conditional statements (*if ... then ... else* statements or *case* statements) describe parts of the program which vary from one run to another. Information which is supplied as 'data', 'parameters' or 'arguments' is used, on any one run of the program, to select amongst alternatives and thus the to 'correct' the schema.

The object-oriented (OO) concept of a 'class' may also be seen as a schema:

- An OO class is a general framework whose attributes are 'inherited' by every sub-class or instance which it dominates. The attributes of any sub-class or instance of a given class may be regarded as 'corrections' to the schema.
- Inheritance of attributes in an object-oriented hierarchy eliminates the need to repeat information in two or more contexts. It is a powerful aid to information compression.

3.3.4 Negation: The concept of 'negation', which is normally viewed merely as a primitive logical operation, may also be seen as an aid to information compression. For example, if a group of 10 people is the same as the village soccer team with one person missing, it is more economical to describe them as:

Harriers(*not* Fred)

than to list all 10 people individually. This is an example of schema-plus-correction, with the use of negation in the correction. It would take us too far afield to discuss this point in depth but it does seem to be the case that negation is normally used in software where the equivalent coding without the use of negation would be more verbose.

3.4 Mathematical functions as compressed I/O information

A definition of a mathematical function using mathematical notation is usually very much smaller than the corresponding I/O table. A formal definition of a mathematical function is normally a highly compressed representation of the potential inputs and outputs of the function.

Here, for example, is a formal specification of the (positive) square root function:

square-root: $R^+ \rightarrow R^+$
square-root(i: R^+) o: R^+
pre TRUE
post $i = o \cdot o$

In this specification, which uses the conventions of VDM, i is the input and o is the output. To be meaningful, this specification must be interpreted in conjunction with the formal definitions of the number system and the formal definition of multiplication. But even with this additional information, the specification of the function is very much smaller than the (infinitely large) table of inputs and outputs which it generates. The same is also true of the Newton-Raphson specification or any equivalent algorithm used for the actual calculation of square roots.

3.4.1 Recursion in Mathematics: Recursion and iteration, which were identified in section 3.3.2 as examples of the run-length coding technique for information compression are prominent in mathematics. For example, the natural numbers are defined like this [30]:

i) Basis: $0 \in N$.
ii) Recursive step: If $n \in N$, then $s(n) \in N$.
iii) Closure: $n \in N$ only if it can be obtained from 0 by a finite number of applications of the operation s.

In this example, $s(n)$ is the 'successor' function which is a key part of the definition of the number system. Recursive application of this function generates the infinite sequence of natural numbers with which we are familiar. A function like the square root function has an infinite range of values because there is recursion within the number system.

4 The process of software design as information compression

In the ordinary way, software design is a process requiring human skill and judgement. In addition to relevant knowledge, the process of creating a Jackson diagram, or a class hierarchy with inheritance of attributes, or an ER model, requires a good intuitive understanding of the difference between a 'clean', economical description and a less tidy, poorly-structured model.

For example, a class or entity called ‘employee’ may be described with attributes which include ‘name’, ‘address’ and ‘telephone number’; another class or entity called ‘customer’ may have attributes which also include ‘name’, ‘address’ and ‘telephone number’. In that situation, a software designer should recognise that the two classes may, with advantage, be defined as sub-classes of a higher level class (perhaps called ‘person’) and that the shared attributes should be extracted from ‘employee’ and ‘customer’ and defined just once at the higher level. In database terminology, this is ‘normalisation’.

To a large extent, the process of creating well structured software is a process of recognising redundancies and creating a design which reduces or eliminates redundancies. If well-structured software is a compressed representation of inputs and outputs, then software design must include a process of information compression.

This sub-section presents examples to show how SP software may be structured (or ‘designed’) automatically by information compression. The search processes in SP6 are not very sophisticated and certain kinds of useful compression (e.g., run length coding and negation) are beyond what it can do. But the capabilities which SP6 can demonstrate should be enough to show how principles of compression may be used for automatic design of software in the future.

The section ends with a brief review of related research.

4.1 Automatic discovery of ‘functions’ or ‘sub-routines’

The object before the arrow (‘=>’) in Fig. 6 is a sequence of symbols representing the notes in a Welsh folk song[1] and the object after the arrow is the result of processing by SP6. The arrow - which is not part of the SP language - marks the transformation between one object and the other.

---

1. ‘Pa le mae ’nghariad i?’ (‘Where is my sweetheart?’), from *Canu’r Cymry II* by Phyllis Kinney and Meredydd Evans, Gymdeithas Alawon Gwerin Cymru, 1987. The initial letter of each symbol is the name of the note. It is followed by qualifiers: f = flat; q = quaver; c = crotchet; d = dotted (1.5 length); h = semi (0.5 length).

```
(gq bfc gq cc bfqh aqh gcd gc gq cc gq bfq cq aq gcd gc gq bfc gq cc
bfqh aqh gcd gc gq cc gq bfq cq aq gcd gc gqh afqh gq fq efq dc gqh afqh
gq fq efq dc gq bfc gq cc bfqh aqh gcd gc bfq efc cq bfc gq bfc gq fc gq bfc
gq cc bfqh aqh gcd gc)
=>
[([%1
        ([%2
                ([%4 (gq bfc gq)]
                cc bfqh aqh gcd gc)]
        gq cc gq bfq cq aq gcd gc)]
[%1 _] [%3 (gqh afqh gq fq efq dc)] [%3 _] [%2 _] bfq efc cq
        bfc [%4 _] fc [%2 _])]
```

Fig. 6. The discovery of 'functions' or 'sub-routines' by SP6.

The sequence of symbols in the first object is, in effect, a simple program which, with suitable routines for generating sounds, could be used in a computer to create the melody. In terms of I/O relations, the sequence represents only the desired output; this program has no input.

As was described in section 2.2, SP6 searches for redundancy by comparing or matching patterns and reduces redundancy by merging (unifying) patterns which are the same. Each symbol in bold type in the second object is the result of the unification of two or more instances of that symbol.

In this example, repeated patterns have been unified to form four 'chunks', each bracketed inside a UAO with a system-generated 'tag' (which can be recognised as such because its first character is '%'). One such chunk is (**gq bfc gq**) and its tag is '%4'.

In each context where an instance of a chunk has been deleted, the chunk has been replace with the corresponding tag, bracketed with an SP variable to show that there is missing information, like this: [%4 _].

Notice that the '%4' chunk is embedded inside the '%2' chunk which is itself part of the '%1' chunk. Apart from restrictions imposed by available memory or processing power, SP6 is able to find chunks nested inside other chunks through any number of levels.

As described earlier, the formation of a chunk and its labelling with a tag is similar to the declaration of a 'function' or 'sub-routine' in a program and the use of the function name as a 'call' in the contexts where it is required. A function can, of course, contain one or more calls to other functions in the same way that an SP chunk may contain tags for other chunks. In both cases, there is no limit to the number of levels, apart from physical constraints.

4.2 Functions with variables

The example just given shows how functions may be formed when patterns are repeated exactly in different contexts. But, as was discussed in section 3.3.3, patterns are often repeated with variations from one instance to another.

As an example of this kind of variation, imagine someone working at a computer terminal and performing a sequence of operations on files like this:

print A; copy B C; delete D; print E; copy F G; delete H;
print I; copy J K; delete L ...

He or she would quickly tire of this and would form a shell script or 'BAT' file like this:

print *w*; copy *x y*; delete *z;*

where *w*, *x*, *y*, and *z* are variables which may receive file names as their values. To perform the three operations, the name of the script may be given as a command followed by four file names as parameters or arguments.

SP6 has only very limited abilities to detect this kind of pattern-with-variations. SP20 [31] is much better but more work is needed before the abstraction of functions with arguments can be fully demonstrated.

4.3 Automatic discovery of 'sequences' and 'selections'

Fig. 7 shows an example where SP6 has compressed a set of patterns using OROs as well as by the use of a chunk with tags. As before, bold type is used where two or more instances of a symbol have been unified.

The object before the arrow is a small English-French dictionary. As discussed in section 5, below, it may be used for translation of single words between English and French or *vice versa*.

```
[(r e d : r o u g e)
(c o n f e s s : a v o u e r)
(g o o d s : m a r c h a n d i s e)
(f u n n y : i m p a y a b l e)
(p e o p l e : p e r s o n n e s)
(h a g g l e : m a r c h a n d e r)
(c o n t e m p t : m é p r i s)
(n i c e : a g r é a b l e)
(c o n t e n t : s a t i s f a i t)
(t r a d e r : m a r c h a n d)
(p e o p l e : g e n s)]
=>
[(r e d : r o u g e)
(g o o d s [%1 _] i s e)
(h a g g l e [%1 _] e r)
(t r a d e r [%1 _])
[%1 (: m a r c h a n d)]
(p e o p l e : {(p e r s o n n e s) (g e n s)})
({(n i c e : a g r é) (f u n n y : i m p a y)} a b l e)
(c o n
        {(f e s s : a v o u e r)
        (t e
                {(m p t : m é p r i s) (n t : s a t i s f a i t)})})]
```

Fig. 7. The discovery of ‘sequences’ and ‘selections’ by SP6.

In this example, OAOs like (r e d : r o u g e) or (p e r s o n n e s) are ‘sequences’ in the sense of Jackson’s Structured Programming (JSP); OROs like {(p e r s o n n e s) (g e n s)}, are ‘selections’ in JSP terminology.

In terms of the compression principles described earlier, a structure which contains sequences and selections like this may be seen as a ‘schema’. Any one of the original patterns may be recreated by choosing one of the alternatives in each selection. Choices like these may be regarded as ‘corrections’ to the schema.

4.4 *Iteration, recursion and negation*

What is missing in the examples which have been described is the ‘iteration’ construct in JSP. Although iteration, and its close relative, recursion, are useful devices for information compression, SP6 is not able to exploit them. Despite this deficiency in the current model, the ‘run length coding’ technique of recursion can be represented in the SP language, as shown here:

[LOOP (STEP1 STEP2 [LOOP _])]

The variable in this structure represents missing information so that [LOOP _] should unify with [LOOP (STEP1 STEP2 [LOOP _])] giving [LOOP (STEP1 STEP2 [LOOP _])] - and so on without limit. Termination of this kind of loop probably requires an operator like negation, not yet incorporated in the model.

With improved methods for pattern matching, unification and search, and with the inclusion of a negation operator, future versions of the SP model should be able to create structures like this from appropriate information and they should be able to recognise and exploit the recursion in other contexts.

Notice that unrestricted iteration or recursion in a function means that the function is a generalisation of any given finite set of I/O values. Since a function of this kind generates I/O values outside the given set, it represents ‘lossy’ compression of the information in the set.

4.5 Automatic discovery of a ‘class hierarchy’ with ‘inheritance of attributes’

With some kinds of information, SP6 derives an hierarchical structure which has the essential characteristics of a ‘class hierarchy’ in object-oriented design.

The object before the arrow in Fig. 8 shows part of a botanical key[1] as a set of OAOs, each one describing the characteristics of a family of plants. In the derived SP object after the arrow, the eight instances of petals-free have been unified to form one instance which occupies a position at the top of the hierarchy. In this position, it is, in effect, shared or ‘inherited’ by all the structures underneath.

1. Based on the *Excursion Flora of the British Isles*, Second Edition, by A. R. Clapham, T. G. Tutin and E. F. Warburg, Cambridge: Cambridge University Press, 1968.

```
[(petals-free sepals-3 aquatic-plants flower-conspicuous ALISMATACEAE)
(petals-free sepals-3 small-land-plants-of-mossy-appearance
        CRASSULACEAE)
(petals-free sepals-gt-3 stamens-numerous occasional-stipules
        flowers-hypogynous flowers-circa-10-cm-diameter PAEONIACEAE)
(petals-free sepals-gt-3 stamens-numerous occasional-stipules
        flowers-hypogynous flowers-small RANUNCULACEAE)
(petals-free sepals-gt-3 stamens-numerous flowers-perigynous ROSACEAE)
(petals-free sepals-gt-3 stamens-le-petals
        leaves-ternate-not-fleshy ROSACEAE)
(petals-free sepals-gt-3 stamens-le-petals leaves-simple
        eaves-succulent carpels-in-one-whorl CRASSULACEAE)
(petals-free sepals-gt-3 stamens-le-petals leaves-simple
        leaves-not-succulent carpels-spirally-arranged RANUNCULACEAE)]
=>
(petals-free
        {(sepals-3
                {(aquatic-plants flower-conspicuous ALISMATACEAE)
                (small-land-plants-of-mossy-appearance
                CRASSULACEAE)})
        (sepals-gt-3
                {(stamens-numerous
                        {(flowers-perigynous ROSACEAE)
                        (occasional-stipules flowers-hypogynous
                                {(flowers-circa-10-cm-diameter PAEONIACEAE)
                                (flowers-small RANUNCULACEAE)})})
                (stamens-le-petals
                        {(leaves-ternate-not-fleshy ROSACEAE)
                        (leaves-simple
                                {(leaves-succulent carpels-in-one-whorl
                                        CRASSULACEAE)
                                (leaves-not-succulent carpels-spirally-arranged
                                        RANUNCULACEAE)})})})})
```

Fig. 8. The discovery of a 'class hierarchy' by SP6, with 'inheritance of attributes'.

Likewise, at lower levels, multiple instances of patterns like stamens-numerous and (occasional-stipules flowers-hypogynous) in the first object have each been reduced to one instance in the second object, and each one is in a position where it is shared by the structures underneath it. As with chunks and tags, there is no arbitrary limit on the number of levels in the hierarchy which may be formed.

SP6 can only form hierarchies like this when the symbols are ordered in an appropriate way. Future versions of the system will be designed to abstract redundancy from UAOs as well as OAOs and to form class hierarchies as appropriate.

4.6 Related research

In the preceding sub-sections we have discussed how automatic software design may be achieved by the compression of I/O information. Related ideas - associated with terms like 'unsupervised inductive learning', 'concept formation', 'grammar discovery', 'extensional programming', 'programming by example' and 'automatic programming' - have been explored in research in the general area of machine learning. The current proposals relate to this research in the following ways.

The idea of deriving functions from I/O information is quite well established, both in the symbolic tradition (e.g., [29]) and in the way 'back-propagation' neural networks are trained (see, for example, [1]). Compression of I/O information as a means of deriving functions is not fully recognised although related ideas have received some attention:

- In some work on machine learning, economy in coding is used as an organising principle (e.g., [37], [7]).
- 'Minimisation of entropy' - features in some work on artificial neural nets (e.g., [14], [13], [22]).
- There has been some work on the application of artificial neural nets to data compression (e.g., [8]).
- The role of inhibitory mechanisms in economical coding in the nervous system has been recognised for some time (some of the relevant work is reviewed in [31]). These kinds of mechanisms have been synthesised electronically by Mahowald and Mead [21] in their artificial retina.
- Measures based on information theory have been used in [25] and [26] (in the context of learning systems which use negative samples.)

The connection between information compression and the processes of pattern matching, unification and search is not generally recognised.

More generally, there are significant strands of work on the automatic or semi-automatic creation of software designs:

- Several articles covering a variety of approaches are presented in [3]. One of the most interesting of these from the present perspective is Biermann's article[4] describing how restricted classes of program may be synthesised from examples by the merging of primitive functions. In another article, Banerji [2] draws interesting parallels between automatic programming and pattern recognition.
- Other recent work includes [17], [38], [39], [18], [24] and [20].
- There is related work on the discovery of scientific laws [19].

- It is pertinent here to mention that the term ‘automatic programming’ is sometimes used for the automatic generation of executable code from ‘higher’ or more abstract representations of software. Recent examples include [28] and [23]. Automatic programming in this sense means translation from one language or notation to another (with or without the incorporation of library functions) and appears to be rather different from the idea of automatic structuring or restructuring of I/O information which is the focus of interest here.

Learning by the SP system may be classed as ‘unsupervised’ learning, meaning that the system operates on relevant data without the intervention of any kind of human or non-human ‘teacher’, without ‘negative’ examples (examples marked as ‘wrong’) and without constraints on the order in which examples are presented. In this kind of learning, the system decides for itself what structures to create without any external guidance.

The SP system belongs firmly in the symbolic tradition because it does not have a network architecture and, by contrast with neural networks, knowledge in the SP system and the organisation of that knowledge is explicit and open to inspection.

5 The execution of software as information compression

Perhaps the most novel feature of this article is the proposition that the process of executing software may usefully be understood as information compression.

In the conventional view of computing, a ‘program’, ‘function’ or ‘procedure’ or, indeed, a whole information processing system, is seen as being rather like a factory or processing plant which takes one or more streams of information as raw material (‘data’) and releases one or more streams of information as products (‘results’). This ‘data flow’ view of computing is embodied in the terms ‘input’ and ‘output’ and in development methods such as the Structured Analysis and Design Technique (SADT) and the Yourdon method.

In the SP view, the sausage-machine idea of computing is recast as a process of discovering and extracting redundancy in the body of information represented by the ‘input’ data and the ‘program’ together. The ‘output’ of an SP ‘function’ comprises parts of the resulting structure which have been ‘marked’ by the unification process.

5.1 Computing with a function represented as an I/O table

The XOR function, described earlier, provides a simple example to illustrate how computing may be seen in these terms. In the SP language, the XOR function can be represented by the second, third, fourth and fifth OAOs in Fig. 9.

```
[
        (0 1 _)
        (0 0 0)
        (1 0 1)
        (0 1 1)
        (1 1 0)
]
=>
[
        (0 0 0)
        (1 0 1)
        (0 1 1)
        (1 1 0)
]
```

Fig. 9. The XOR function in SP notation with ‘input’. The SP object after the arrow is the result of processing by SP6.

The first two digits in the first OAO at the top of the figure represent ‘input’. The variable (‘_’) in that OAO is a place marker for the missing ‘output’. Given this structure, the SP system searches for repeating patterns and unifies the ones giving most compression. In this example, the result of this pattern matching and unification is the SP object which follows the arrow.

This new object is essentially the same as the XOR definition, except that the first two digits in the third OAO are now unified with the ‘0 1’ sequence in the ‘input’. As in previous examples, bold type is used to show where symbols have been unified. The effect of this unification is to ‘select’ the third OAO in the structure, including the ‘1’ digit in the position corresponding to ‘output’. As a by-product of its search for the best possible unification of patterns, the SP system achieves ‘table lookup’ and the XOR structure delivers the required result.

This example is very simple but it captures the essentials of what appears to be a very general model of the computing process. The key features of the model are these:

- As previously discussed, any function may be understood as a structure which links inputs with corresponding outputs.
- To find an output for any given input, it is necessary to search for a match between the given input and patterns in the structure. If no match can be found, then the input is not valid. If a match is found, then the linked output is also found.

The generality of the model lies in the generality of these features. They apply not merely when a function is expressed as an I/O table like the XOR function in Fig. 9 but they apply also, as shown below, to functions which have been compressed in the way that was discussed in section 4 above.

5.2 *Computing with compressed functions*

The discussion above of how SP 'executes' a function assumes that the function has the form of a table. But if the table contains redundancy then, as described in section 4, the SP system should compress it. How can a compressed version of an SP 'function' operate to process new input?

```
[((f u n n y : _ _ _ _ _) _ _ _ _)
(r e d : r o u g e)
(g o o d s [%1 _] i s e)
(h a g g l e [%1 _] e r)
(t r a d e r [%1 _])
[%1 (: m a r c h a n d)]
(p e o p l e : {(p e r s o n n e s) (g e n s)})
({(n i c e : a g r é) (f u n n y : i m p a y)} a b l e)
(c o n
        {(f e s s : a v o u e r)
        (t e
                {(m p t : m é p r i s) (n t : s a t i s f a i t)})})]
=>
[(r e d : r o u g e)
(g o o d s [%1 _] i s e)
(h a g g l e [%1 _] e r)
(t r a d e r [%1 _])
[%1 (: m a r c h a n d)]
(p e o p l e : {(p e r s o n n e s) (g e n s)})
({(n i c e : a g r é) (f u n n y : i m p a y)} a b l e)
(c o n
        {(f e s s : a v o u e r)
        (t e
                {(m p t : m é p r i s) (n t : s a t i s f a i t)})})]
```

Fig. 10. The processing of 'input' by SP6, using a compressed SP 'function'

At the top of Fig. 10, the symbols in the OAO ((f u n n y : _ _ _ _ _) _ _ _ _) represent 'input', while the variables are place markers for expected 'output'. The object is bracketed with the structure from Fig. 7, which was created by SP6 from 'raw' information. That structure may be seen as a compressed version of an SP 'function'.

As with the XOR example, SP6 unifies the input with appropriate parts of the function giving a new structure shown after the arrow. The characters 'f u n n y :' have been unified with the matching characters in (f u n n y : i m p a y), giving (**f u n n y :** i m p a y) as a result. The 'residual', non-unified characters ('... i m p a y'), together with residual characters ('...a b l e') in the higher level OAO, may be regarded as the 'output' of the function.

In this example, the output is, in effect, the plain, un-unified, characters in the OAO ((**f u n n y :** i m p a y) a b l e).

This object has the same form as the object

({(n i c e : a g r é) (f u n n y : i m p a y)} a b l e)

but the ORO has been replaced by the OAO within it which has been 'selected' by the input.

As another example, when (p e o p l e : _) is added to the structure, SP6 creates the unified OAO

(**p e o p l e :** {(p e r s o n n e s) (g e n s)})

and thus delivers the ORO

{(p e r s o n n e s) (g e n s)}

as the 'output'. As one would wish, this is an 'indeterminate' result giving the two synonyms of the input word.

In the example from Fig. 10, it has been necessary to put inner brackets in appropriate places in the input and to put a variable ('_') in each position where a symbol is expected in the output. This is necessary because the search methods in SP6 are not good enough to find the appropriate unifications without them. In future versions, it is intended that inner brackets may be omitted from the input in a case like this and that each group of variables may be replaced by a single variable representing zero or more symbols in the expected output.

Another shortcoming of SP6 in this area is that it cannot 'de-reference' tags in an appropriate way. In future versions, it should be possible to supply ': m a r c h a n d' as 'input' and for the system to find 'g o o d s ... i s e', 'h a g g l e ... e r' and 't r a d e r' as alternative outputs.

5.3 Direction of the computing process

With systems like Prolog it is possible to write programs which can be run 'backwards' or 'forwards' depending on the data which is supplied. But most other established software systems have a distinct direction in the computing process which cannot be reversed.

The SP model differs from the conventional view in that the distinction between 'input' and 'output' is not an intrinsic part of any function but is defined by the data supplied when a function is run. As described in the preceding two sub-sections, any sub-sequence of the symbols in a pattern may be supplied as input and the complementary sub-sequence of the pattern (the non-unified symbols) will be identified as output.

Because of this movable boundary between input and output, any function may be run 'forwards' or 'backwards' at will. For example, if (3.00 _) is applied as 'input' to an SP version of the square root function shown in Fig. 5 it will deliver ‘1.73’ as the answer. Conversely if (_ 1.73) is provided as input, the square of the number will be found. In general, the ‘input’ and the ‘output’ symbols may appear in any position as, for example, when (r _ d _ r _ u _ e) is unified with (r e d : r o u g e) from Fig. 10 giving (**r** e **d** : **r** o **u** g **e**).

5.4 Indeterminacy in computing

If the XOR function shown in Fig. 9 is run backwards in the way just described, an input pattern like (_ _ 1) has two valid outputs: ‘1 0’ and ‘0 1’. In other words, the result is indeterminate. SP6 can only deliver one of the answers for a case like this (and which one it gives when they are both equally valid is essentially arbitrary). Apart from cases like the (p e o p l e : _) example in section 5.2, SP6 has only a limited ability to find alternative answers in cases where there is more than one valid answer. Future versions should be better.

5.5 Is this really what computing is about?

The generality of the principles described at the beginning of this section should be sufficient to establish that any kind of computing operation may be cast in this mould. The concept appears to be general enough to accommodate real time systems and ‘interactive’ programs as well as more conventional algorithms. But readers may feel, nonetheless, that the view of computing which has been presented is unfamiliar and difficult to reconcile with the conventional styles of computing and mathematics.

Any difficulty of this kind may be due to cosmetic factors, at least in part. There is nothing in the SP concept of computing to say that inputs and outputs must be displayed in the style of the examples in this article. Inputs and outputs may be displayed in any way which is convenient, including conventional styles. For example, the computing operation shown in Fig. 10 could be shown in a more conventional way like this:

f u n n y : → [Fig. 10 function] → i m p a y a b l e

When the SP model is more fully developed, a programme of work will be started to analyse established concepts from computing and mathematics in terms of pattern matching, unification and search. In the light of the general principles which have been presented, there is every reason to believe that operations with the number system such as addition, multiplication and so on can be analysed in these terms. Presentation of these analyses should increase confidence in the relevance and applicability of the SP ideas.

6 Implications for the practice of software engineering

There is potential in the ideas which have been described for substantial gains in productivity in the production of software - both ‘development’ and ‘maintenance’ - and for substantial gains in the quality of software products. This section briefly describes some of the possibilities.

With conventional computing systems, software development typically requires the use of three or more languages or notations (including diagrammatic notations) with translations from one notation to another and checking (‘verification’) that each translation has been performed correctly. With a mature version of the SP system, functional requirements may be recorded in the SP language and then executed directly. There should be no need for any translation, compiling or verification. Like some established languages (e.g., Prolog and Miranda™), the SP language may function as an executable specification language. Simplifying the development path in this kind of way should speed up development and it should offer fewer opportunities for errors to be introduced.

If an efficient system for automatic compression can be developed along the lines which have been described, then the process of software design may be automated. There are potential benefits in reduced effort and cost, shorter time scales and in the elimination of errors associated with the current practice of ‘hand crafting’ software.

In some cases, the information to be compressed may be derived directly from the problem domain. An example of this is the existing practice of programming a robot arm for, say, paint spraying by recording relevant coordinates as the ‘hand’ of the arm is moved in the required way by a skilled human operator. The sequence of coordinates may then be used as a program to reproduce the movement of the arm. Compression of this information should produce a program with ‘functions’ and other structures discussed earlier. If ‘lossless’ compression is used, the compressed program should reproduce the movement as accurately as the original list of coordinates. There are other applications, particularly in the area of control engineering, where the information needed for a program may be obtained directly from the environment.

In other cases, relevant information may exist only in the head of the client and must be obtained by interviews and by direct specification by the client in the traditional manner. In this kind of situation, a mature SP system may be useful in four main ways:

- The SP language is extremely simple and should be easy for non-specialists to learn.
- The client may concentrate on defining I/O relationships without worrying about the internal structure of the program. I/O specifications produced by the client may be compressed by the system to create a program.
- The system should help to detect ambiguities, contradictions and redundancies in the specification.
- Like other executable specification languages, the system may be used for rapid prototyping with recognised benefits in assisting the client to specify his or her requirements [12].

There are other potential benefits in the modifiability and maintenance of software, re-use of software, configuration management and the user interface. These are discussed in [33], Chapter 6.

7 Future developments

The development of these ideas is proceeding on four main fronts:

- The core processes of pattern matching, unification and search are being developed with the following aims:
    - To improve computational efficiency and scaleability to large data sets.
    - So that alternative answers to problems can be found (rather than one 'best' answer as in SP6).
    - So that recurrent patterns can be found which are 'discontinuous' or 'fragmented' and interspersed with other material.
    - So that the system can identify recursive structures and can exploit negation as a coding principle.
- Attention is being given to the ways in which traditional concepts in mathematics, logic and computing may (or may not) be interpreted in terms of SP principles.
- When the basic model is more refined, it will be implemented on an established SIMD or MIMD parallel-processing system.
- At all stages, the potential applications of the system will be kept in view and developed as far as possible.

As previously noted, substantial progress has been made in developing the core processes as described above [31] but SP6 is still the best model to illustrate the main themes of this article.

8 Conclusion

Elements of the concept of software described in this article may be seen in many of the related areas of research which have been described. But the central importance of information compression in the organisation and execution of software and the potential benefits of this viewpoint do not yet seem to be clearly recognised. I hope that the arguments and examples given in this article will help to crystallise this idea and lead to fruitful new developments in the future.

9 Acknowledgements

I am grateful to Mark Lawson and Tim Porter (School of Mathematics, Bangor), Mark

Searl and Graham Stephen (School of Electronic Engineering and Computer Systems, Bangor) for useful discussions and for constructive comments on earlier drafts of this article. I am also indebted to anonymous referees for useful comments and suggestions for improvements in the article.

The work reported in this article was supported in part by a grant to Dr. J G Wolff from the UK Science and Engineering Research Council (GR/G51565) for the 'Further development of SP: integration of inductive learning, inference and information retrieval'.

10 References

[1] ALEKSANDER, I and MORTON, H: 'An introduction to neural computing' (Chapman and Hall, 1990).

[2] BANERJI, R. B.: 'Some insights into automatic programming using a pattern recognition viewpoint', in [3], pp. 483-502.

[3] BIERMANN, A. W., GUIHO G., and KODRATOFF, Y. (Eds.): 'Automatic program construction techniques', (Macmillan, 1984).

[4] BIERMANN, A. W.: 'Dealing with search', in [3], pp. 375-392.

[5] BIRTWISTLE, G. M., DAHL, O-J., MYRHAUG, B., and NYGAARD, K.: 'Simula begin'. (Van Nostrand Reinhold, 1973).

[6] BOWERS, D. S.: 'From data to database'. (Van Nostrand Reinhold, 1988).

[7] COOK, C. M., and ROSENFELD, A.: 'Some experiments in grammatical inference', in 'Computer Oriented Learning Processes', J. C. Simon, Ed., (Noordhoff, 1976), pp. 157-174.

[8] COTTRELL, G. W., MUNRO, P., and ZIPSER, D.: 'Learning internal representations from gray-scale images: an example of extensional programming', in 'Proceedings of the ninth annual conference of the cognitive science society, Seattle', 1987, pp. 461-473.

[9] ECKHARDT, D. E., CAGLAYAN, A. K., KNIGHT, J. C. LEE, L. D. MCALLISTER, D. F.VOUK, M. A., AND KELLY, J. P. J.: 'An experimental evaluation of software redundancy as a strategy for improving reliability', *IEEE Transactions on Software Engineering*, 1991, **17**, (7), pp. 692-702.

[10] FOSTER, J. R., JOLLY, A. E. P., and NORRIS, M. T.: 'An overview of software maintenance', *British Telecom Technology Journal*, 1989, **7**, (4), pp.37-46.

[11] GOLDBERG. A.: 'Smalltalk-80: the language and its implementation', (Addison-Wesley, 1983).

[12] HEKMATPOUR, S., and INCE, D.: 'Rapid software prototyping', Open University Technical Report 86/4, (The Open University, 1986).

[13] HINTON, G. E., and SEJNOWSKI, T. J.: 'Learning and relearning in Boltzmann machines', In D. E. Rumelhart and J. L. McLelland, Eds., 'Parallel distributed processing, vol. I', (MIT Press, 1986), pp. 282-317.

[14] HOPFIELD, J. J.: 'Neural networks and physical systems with emergent collective

properties', *Proceedings of the National Academy of Science, USA*, 1982, **79**, pp. 2554-2558.

[15] IEEE Standard glossary of software engineering terminology (1983).

[16] JACKSON, M. A.: 'Principles of program design'. (Academic Press, 1975).

[17] KAMANI, M. K., and RAMAKRISHNA, R. S. 'Predicate-formation for synthesising LISP code', *IEEE Transactions on Systems, Man, and Cybernetics*, 1990, **20**, (1), pp. 530-533.

[18] KINBER, E. B., and BRAZMA, A. N.: 'Models of inductive synthesis', *Journal of Logic Programming*, 1990, **9**, pp. 221-233.

[19] LANGLEY, P. SIMON, H. A., and BRADSHAW, G. L.: 'Heuristics for empirical discovery', in 'Computational models of learning', L. Bolc, Ed., (Springer-Verlag, 1987), pp. 21-54.

[20] LIEBERHERR, K. J., BERGSTEIN, P., and SILVA-LEPE, I.: 'From objects to classes: algorithms for optimal object-oriented design', *Software Engineering Journal*, 1991, **6**, (4), pp. 205-228.

[21] MAHOWALD, M. A., and MEAD, G.: 'The silicon retina', *Scientific American*, 1991, **264**, (5), pp. 40-47.

[22] MOED, M. C., and SARIDIS, G. N.: 'A Boltzmann machine for the organization of intelligent machines', *IEEE Transactions on Systems, Man, and Cybernetics*, 1990, **20**, (5), pp. 1094-1102.

[23] NISHIDA, F., TAKAMATSU, S. FUJITA, Y., and TANI, T.: 'Semi-automatic program construction from specifications using library modules', *IEEE Transactions on Software Engineering*, 1991, **17**, (9), pp. 853-871.

[24] PHAN, T. H., and WITTEN, I. H.: 'Accelerating search in function induction', *Journal of Experimental and Theoretical Artificial Intelligence*, 1990, **2**, pp. 131-150.

[25] QUINLAN, J. R.: 'Induction of decision trees', *Machine Learning*, 1986, **1**, pp. 81-106.

[26] QUINLAN, J. R.: 'Learning logical definitions from relations', *Machine Learning*, 1990, **5**, pp. 239-266.

[27] SHRAGER, J., and LANGLEY, P. (Eds.): 'Computational models of scientific discovery', (Morgan Kaufmann, 1990).

[28] SMITH, D. R.: 'KIDS: a semiautomatic program development system', *IEEE Transactions on Software Engineering*, 1990, **16**, (9), pp. 1024-1043.

[29] SMITH, D. R.: 'The synthesis of LISP programs from examples: a survey', in [3].

[30] SUDKAMP, T. A.: 'Languages and machines, an introduction to the theory of computer science', (Addison-Wesley, 1988).

[31] WOLFF, J. G.: 'Computing as compression: SP20'. Submitted for publication.

[32] WOLFF, J. G.: 'Computing, cognition and information compression', *AI Communications*, 1993, **6**, (2), pp. 107-127.

[33] WOLFF, J. G.: 'Towards a theory of cognition and computing', (Ellis Horwood, 1991).

[34] WOLFF, J. G.: 'On the integration of learning, logical deduction and probabilistic inductive inference', Proc. First International Workshop on Inductive Logic Programming, Vienna do Costello, Portugal, 1991, pp. 177-191.

[35] WOLFF, J. G., and CHIPPERFIELD, A. J.: 'Unifying computing: inductive learning and logic', in Proc. Expert Systems 90: T. R. Addis and R. M. Muir, (Eds.), 'Research and development in expert systems VII', (Cambridge University Press, 1990), pp. 263-276.

[36] WOLFF, J. G.: 'Simplicity and power: some unifying ideas in computing', *Computer Journal*, **33**, (6), pp. 518-534.

[37] WOLFF, J. G.: 'Learning syntax and meanings through optimization and distributional analysis', in Y Levy, I M Schlesinger and M D S Braine (Eds.), 'Categories and processes in language acquisition', (Lawrence Erlbaum, 1988), pp. 179-215. Reprinted in Chapter 2 of [33].

[38] XU, Z., HOLMES, G., and ROGERS, W. J.: 'Automatic generation of frame-based expert systems', *New Zealand Journal of Computing*, 1989, **1**, (2), pp. 44-51.

[39] ZHANG, Y., and ORLOWSKA, M. E.: 'An improvement on the automatic tool for relational database design', *Information Systems*, 1990, **15**, (6), pp. 647-651.