Software engineering and the SP theory of intelligence

J Gerard Wolff*

November 15, 2017

Abstract

This paper describes a novel approach to software engineering derived from the *SP theory of intelligence* and its realisation in the *SP computer model*. These are the bases of a projected industrial-strength *SP machine* which, when mature, is anticipated to be the vehicle for software engineering as described in this paper. Potential benefits of this new approach to software engineering include: the automation of semi-automation of software development, with non-automatic programming of the SP system where necessary; allowing programmers to concentrate on 'real-world' parallelism, without worries about parallelism to speed up processing; the ambitious long-term goal of programming the SP system via written or spoken natural language; reducing or eliminating the distinction between 'design' and 'implementation'; reducing or eliminating operations like compiling or interpretation; reducing or eliminating the need for verification of software; reducing the need for an explicit process of validation of software; no formal distinction between program and database; potential for substantial reductions in the number of types of data file and the number of computer languages; benefits for version control; and reducing technical debt.

Keywords: SP theory of intelligence, software engineering, automatic programming, natural language processing, compiling, interpretation, verification, validation, parallel processing, version control, technical debt.

*Dr Gerry Wolff, BA (Cantab), PhD (Wales), CEng, MBCS, MIEEE; CognitionResearch.org, Menai Bridge, UK; jgw@cognitionresearch.org; +44 (0) 1248 712962; +44 (0) 7746 290775; *Skype*: gerry.wolff; *Web*: www.cognitionresearch.org.

1 Introduction

This paper is about a novel approach to software engineering with potential advantages over standard approaches. It is a considerable revision, expansion and development of preliminary ideas in [16, Section 6.6], and it draws together some ideas that are scattered in other publications.

The novelty of the approach is because it derives from the *SP system*, a radically new alternative to conventional computers that comprises the *SP theory of intelligence* and its realisation in the *SP computer model*. It is envisaged that both those things will provide the basis for an industrial-strength *SP machine* which would be the vehicle for software engineering as described in this paper.

There is an outline description of the SP system in Appendix A with pointers to where fuller information may be found.

The potential advantages of the SP system in software engineering are described and discussed in most of the sections that follow. But first, the next section relates concepts in software engineering to concepts in the SP system.

2 How concepts in the SP system relate to concepts in ordinary computers

Superficially, the workings of the SP system, as outlined in Appendix A.1, is quite different from the workings of an ordinary computer. But this section demonstrates with some simple examples that, with appropriate SP patterns, most of the concepts that are familiar in ordinary programming may be modelled in the SP system. For readers that are not already familiar with the SP system, it would probably be useful to read Appendix A.1 before reading this section.

2.1 Programs, procedures, functions, subroutines, calling of subroutines, parameters, and conditional statements

To illustrate how several programming concepts may be modelled by the SP system, we begin with a simple example: the kinds of things that need to be done in preparing a meal in a restaurant or cafe in response to an order from a customer, excluding any advance preparation.

Figure 1 shows an SP *grammar*, comprising a collection of SP patterns, which may be seen as a highly simplified program for the preparation of a meal to order.

The first pattern in the figure, '`PM ST #ST MC #MC PD #PD #PM`' describes the overall structure of the procedure for preparing a meal. It is identified by the pair

of symbols 'PM ... #PM', mnemonic for 'prepare meal'.

The main steps are the preparation of a starter ('ST ... #ST'), the preparation of the main course ('MC ... #MC'), and the preparation of a pudding ('PD ... #PD'). Corresponding patterns are shown in the second and subsequent rows in the figure: there are three kinds of starter, five kinds of main course, and four kinds of pudding.

```
PM ST #ST MC #MC PD #PD #PM  | Prepare meal
ST 0 mussels #ST             | Starter: prepare a dish of mussels
ST 1 soup #ST                | Starter: prepare a bowl of soup
ST 2 avocado #ST             | Starter: prepare an avocado dish
MC 0 lassagne #MC            | Main course: prepare a lassagne dish
MC 1 beef #MC                | Main course: prepare a beef dish
MC 2 nut-roast #MC           | Main course: prepare a nut-roast dish
MC 3 kipper #MC              | Main course: prepare a kipper
MC 4 salad #MC               | Main course: prepare a salad
PD 0 ice cream #PD           | Pudding: prepare ice cream
PD 1 apple crumble #PD       | Pudding: prepare apple crumble
PD 2 fresh fruit #PD         | Pudding: prepare fresh fruit
PD 3 tiramisu #PD            | Pudding: prepare tiramisu
```

Figure 1: An SP grammar comprising a set of SP patterns representing, in a highly simplified form, the kinds of procedures involved in preparing a mean for a customer in a restaurant or cafe. To the right of each SP pattern is an explanatory comment, after the symbol '|'.

To see how this grammar functions in practice, consider the SP-multiple-alignment shown in Figure 2.[1]

This SP-multiple-alignment is the best one created by the SP computer model with the New pattern, 'PM 0 4 1 #PM', processed in conjunction with Old patterns shown in Figure 1. Here, the New pattern may be seen as an economical description of what the customer ordered: a starter comprising a dish of mussels represented by the short code '0'; a main course chosen to be a salad represented by the short code '4'; and a pudding which in this case is apple crumble, represented by the code '4'.

Assuming that each of the symbols 'mussels', 'salad', and 'apple_crumble', represents the execution of instructions for preparing the corresponding dish, the whole SP-multiple-alignment may be seen to achieve the effect of preparing what the customer has ordered.

This example shows how SP system may model several of the concepts which are familiar in ordinary computer programming:

[1] Just to confuse matters, this SP-multiple-alignment has been rotated by 90° compared with the SP-multiple-alignment shown in Figure 12. These two versions of an SP-multiple-alignment are entirely equivalent. The choice between them depends entirely on what fits best on the page.

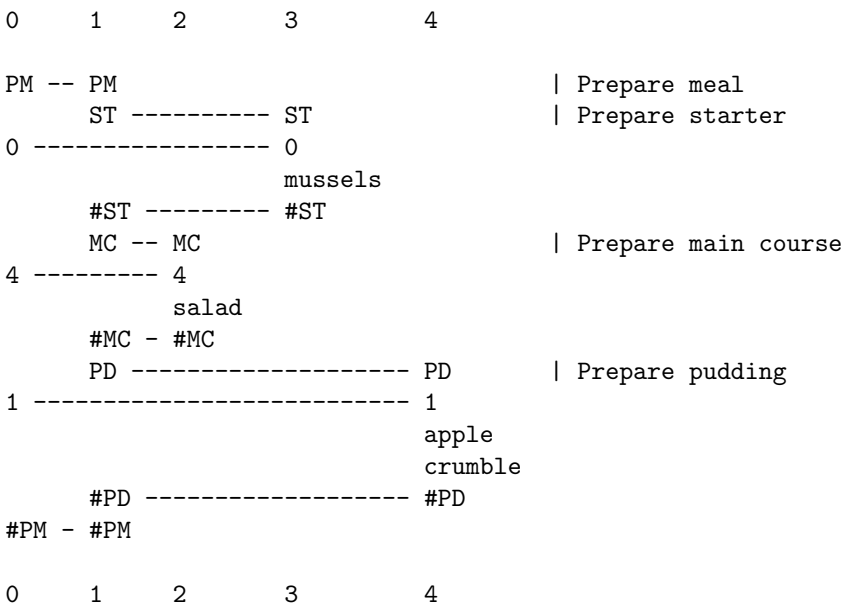

Figure 2: The best SP-multiple-alignment created by the SP computer model with the New pattern, '`PM 0 4 1 #PM`', and the set of Old patterns shown in Figure 1. Comments are shown on the right, each one following the symbol '`|`'.

- As mentioned earlier, the whole grammar in Figure 1 may be seen as a *program* for preparing a meal to meet a given order.
- Each of the remaining patterns in Figure 1, '`ST 0 mussels #ST`', '`ST 1 soup #ST`', and so on, may be seen as a *procedure*, *function*, or *subroutine* that is *called* from the higher-level procedure '`PM ST #ST MC #MC PD #PD #PM`'.

  Although the example does not illustrate the point, it should be clear that each subroutine may call zero or more lower-level subroutines, and so on through as many levels as may be required. As we shall see in Section 2.6, recursion is also possible.
- Each of the code symbols '`0`', '`4`', and '`1`', in the New pattern '`PM 0 4 1 #PM`', may be seen as a *parameter* to the program.
- Since the code symbol '`0`' has the effect of selecting the pattern '`ST 0 mussels #ST`' from the set of patterns '`ST 0 mussels #ST`', '`ST 1 soup #ST`', and '`ST 2 avocado #ST`', the process of selection may be seen to achieve the effect of a *conditional statement* or *if-then rule* in an ordinary computer program, something like this: *if (parameter_1 == 0) (ST 0 mussels #ST)*, meaning "If the value of parameter_1 is '0', perform the subroutine

'ST 0 mussels #ST' ". Much the same may be said, *mutatis mutandis*, about the code symbols '4', and '1'.

2.2 Unordered parameters

Parameters to a conventional program sometimes work like the example in Section 2.1, where the values for parameters must be given in the right order to ensure that each one is applied in the right place. But in other kinds of program, there is flexibility in the ordering of parameters because a label for each value ensures that it is applied correctly.

This kind of flexibility may be modelled in the SP system, as illustrated by the grammar shown in Figure 3 and the SP-multiple-alignment shown in Figure 6.

```
PM ST #ST MC #MC PD #PD #PM          | Prepare meal
ST <P1> 0 </P1> mussels #ST          | Starter: prepare a dish of mussels
ST <P1> 1 </P1> soup #ST             | Starter: prepare a bowl of soup
ST <P1> 2 </P1> avocado #ST          | Starter: prepare an avocado dish
MC <P2> 0 </P2> lassagne #MC         | Main course: prepare a lassagne dish
MC <P2> 1 </P2> beef #MC             | Main course: prepare a beef dish
MC <P2> 2 </P2> nut-roast #MC        | Main course: prepare a nut-roast dish
MC <P2> 3 </P2> kipper #MC           | Main course: prepare a kipper
MC <P2> 4 </P2> salad #MC            | Main course: prepare a salad
PD <P3> 0 </P3> ice cream #PD        | Pudding: prepare ice cream
PD <P3> 1 </P3> apple crumble #PD    | Pudding: prepare apple crumble
PD <P3> 2 </P3> fresh fruit #PD      | Pudding: prepare fresh fruit
PD <P3> 3 </P3> tiramisu #PD         | Pudding: prepare tiramisu
```

Figure 3: Another SP grammar comprising a set of SP patterns representing, in a highly simplified form, the kinds of procedures involved in preparing a meal for a customer in a restaurant or cafe. *Key:* '<P1>' = parameter 1 ; '<P2>' = parameter 2; '<P3>' = parameter 3; '<P4>' = parameter 4. As in Figure 1, there is an explanatory comment to the right of each SP pattern, after the symbol '|'.

In the grammar, each of the possible values for the 'ST ... #ST' slot is labelled with the pair of symbols '<P1> ... </P1>' (meaning 'parameter 1'), and likewise for the possible values in the other two slots: values for 'MC ... #MC' are labelled with '<P2> ... </P2>' ('parameter 2'), and values for 'PD ... #PD' are labelled with '<P3> ... </P3>' ('parameter 3').

When the SP computer model is run with a set of New patterns like this: 'PM', '<P3> 1 </P3>', '<P1> 0 </P1>', '<P2> 4 </P2>', and '#PM'—and since the SP computer model treats such a set as being unordered—the labelling of '1', '0', and '4', ensures that each pattern is assigned to its proper place in the SP-multiple-alignment shown in Figure 4. This is despite the fact that the order of the values in

the set of New patterns is different from the order of the corresponding structures in the 'master' pattern '`PM ST #ST MC #MC PD #PD #PM`'.

```
0       1     2       3         4

PM ---- PM                              | Prepare meal
        ST -------------------- ST      | Prepare starter
<P1> -------------------------- <P1>
0 ----------------------------- 0
</P1> ------------------------- </P1>
                                mussels
        #ST -------------------- #ST
        MC -- MC                        | Prepare main course
<P2> -------- <P2>
4 ----------- 4
</P2> ------- </P2>
              salad
        #MC - #MC
        PD ---------- PD                | Prepare pudding
<P3> ---------------- <P3>
1 ------------------- 1
</P3> --------------- </P3>
                      apple
                      crumble
        #PD --------- #PD
#PM --- #PM

0       1     2       3         4
```

Figure 4: The best SP-multiple-alignment created by the SP computer model with the set of New patterns: '`PM`', '`<P3> 1 </P3>`', '`<P1> 0 </P1>`', '`<P2> 4 </P2>`', and '`#PM`', and the set of Old patterns shown in Figure 3. As in Figure 2, comments are shown on the right, each one following the symbol '`|`'.

2.3 Variables, values, and types

In addition to the programming concepts already considered, the concepts 'variable', 'value', and 'type' may be modelled in the SP system.

Consider, for example, the SP grammar shown in Figure 5. This is an expansion of the 'salad' main course entry in the grammar shown in Figure 1. Instead of simply giving the name of the dish, this grammar provides for choices of ingredients in four categories: salad leaves ('`L ... #L`'), root vegetables ('`R ... #R`'), garnish ('`G ... #G`'), and dressing ('`D ... #D`').

When the SP computer model is run with the New pattern '`MC 2 1 0 1 #MC`'

```
MC salad L #L R #R G #G D #D #MC
L 0 lettuce #L
L 1 beetroot leaves #L
L 2 water cress #L
L 3 spinach #L
R 0 carrot #R
R 1 potato #R
R 2 parsnip #R
G 0 nuts #G
G 1 saltanas #G
D 0 mayonnaise #D
D 1 vinaigrette #D
D 2 thousand island #D
```

Figure 5: An SP grammar comprising a set of SP patterns representing in simplified form the kinds of things that may go in a salad. *Key:* MC = main course; L = salad leaves; R = root vegetables; G = garnish; D = dressing.

and Old patterns comprising the patterns shown in Figure 5, the best SP-multiple-alignment created by the SP computer model is the one shown in Figure 6.

This example illustrates the three concepts, 'variable', 'value', and 'type', as follows:

- Within the pattern '`MC salad L #L R #R G #G D #D #MC`', each of the pair of symbols '`L #L`', '`R #R`', '`G #G`', and '`D #D`', may be seen to represent the concept *variable* because they are slots where values may be inserted.
- The effect of the SP-multiple-alignment is to assign '`water_cress`' to the first slot, '`potato`' to the second slot, '`nuts`' to the third slot, and '`vinaigrette`' to the fourth slot. Those four things are of course *values*, each one assigned to an appropriate variable.
- For each of the four variables, its *type*—meaning the range of values that it may take—may be seen to be defined by the grammar shown in Figure 5: possible values for the variable '`L #L`' may be seen to be '`lettuce`', '`beetroot_leaves`', '`water_cress`', and '`spinach`', and likewise for the other three variables.

2.4 Structured programming

An established feature of software engineering today, which is now partly but not entirely subsumed by object-oriented programming (next), is 'structured programming' [2] in which the central idea is that programs should comprise well-defined

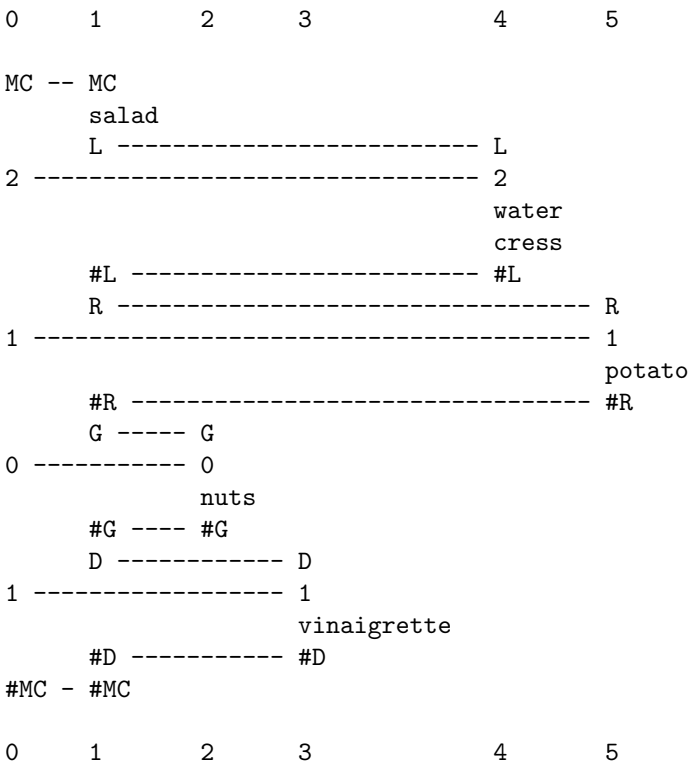

Figure 6: The best SP-multiple-alignment created by the SP computer model with the New pattern '`MC 2 1 0 1 #MC`' and the set of Old patterns shown in Figure 5.

structures which should reflect the structure of the data that is to be processed and should *never* use the 'goto' statement of an earlier era.

To a large extent, the SP system incorporates that principles of structured programming, since unsupervised learning in the SP system creates structures that reflect the structure of incoming data. And other kinds of processing in the SP system, such as pattern recognition or reasoning, is achieved by recognising and processing similar structures in new data, without the use of anything like a 'goto' statement.

2.5 Object-oriented design or programming

From its introduction in the *Simula* computer language [1], 'object-oriented design' (OOD) and the closely-related 'object-oriented programming' (OOP) have become central in software engineering and in such widely-used programming languages as C++ and Java.

Key ideas in OOD/OOP are that the structure of each computer program should reflect the objects to which it relates—people, packages, fork-lift trucks, and so on—and the classes and subclasses in which each object belongs. This not only helps to make computer programs easy to understand but it means that the features of any specific object may be 'inherited' from the classes and subclasses to which it belongs.

The same is true for all the other objects in a given class, meaning that there is an overall saving or compression of information compared with what would be needed without inheritance. In this respect, OOD/OOP is very much in keeping with the central importance of information compression in the SP system.

2.5.1 Example: a class-inclusion hierarchy in the SP system

In Figure 7, the SP-multiple-alignment produced by the SP computer model shows how a previously unknown entity with features shown in the New pattern in column 1 may be recognised at several levels of abstraction: as an animal (column 1), as a mammal (column 2), as a cat (column 3) and as the specific cat 'Tibs' (column 4). These are the kinds of classes used in ordinary systems for OOD/OOP.

From this SP-multiple-alignment, we can see how the entity that has now been recognised *inherits* unseen characteristics from each of the levels in the class hierarchy: as an animal (column 1) the creature '`breathes`' and '`has-senses`', as a mammal it is '`warm-blooded`', as a cat it has '`carnassial-teeth`' and '`retractile-claws`', and as the individual cat Tibs it has a '`white-bib`' and is '`tabby`'.

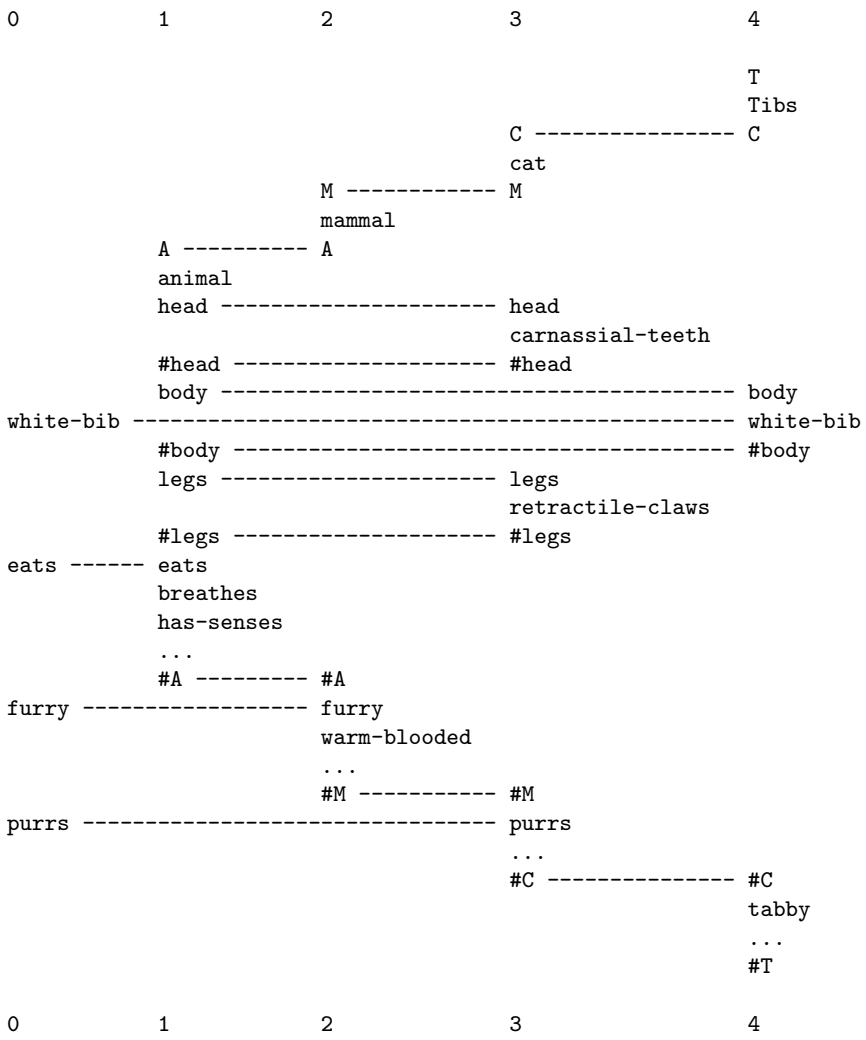

Figure 7: The best SP-multiple-alignment found by the SP model, with the New pattern '`white-bib eats furry purrs`' shown in column 1, and a set of Old patterns representing different categories of animal and their attributes shown in columns 1 to 4. Reproduced with permission from Figure 6.7 in [10].

2.5.2 Example: a part-whole hierarchy in the SP system

An idea that is related to the concept of a class-inclusion hierarchy is that any given object or class of object may be divided into parts and sub-parts etc through as many levels as may be required.

Figure 8 shows how this kind of hierarchy may be accommodated in the SP system. Here, a pattern representing the concept of a car is shown in column 2, with parts such as '`<engine>`' '`<body>`', and '`<gearbox>`'. The pattern in column 1 shows parts of an engine such as '`<cylinder-block>`', '`<pistons>`', and '`<crankshaft>`'. The pattern in column 3 shows how the body may be divided into such things as '`<steering-wheel>`', '`<dashboard>`', and '`<seats>`'. The pattern in column 5 divides the dashboard into parts that include '`<speedometer>`' and '`<fuel-gauge>`'. And the pattern in column 4 divides the speedometer into '`<speed-dial>`', '`<speed-pointer>`', and more.

The kind of structure shown in Figure 8 exhibits a form of inheritance, much like inheritance in a class-inclusion hierarchy. In this case, recognition of something as a '`<speed-dial>`' suggests that it is likely to be part of a '`<dashboard>`', which itself is likely to be part of the '`body`' of a '`<car>`'. This kind of inference is the kind of thing that crime investigators will do: search for a missing body when a severed human arm has been discovered.

2.5.3 Seamless integration of class-inclusion and part-whole hierarchies

A neat feature of the SP system is that, because all kinds of knowledge are represented within the framework of SP-multiple-alignments, there can be seamless integration of diverse kinds of knowledge in any combination (Appendix **??**).

This aspect of the SP system means that there can be seamless integration of class-inclusion and part-whole hierarchies, as described and illustrated in [12, Section 9.1, Figure 16].

2.6 Recursion

The SP system does not provide for the repetition of procedures via these kinds of statement: *while ...*, *do ... while ...*, *for ...*, or *repeat ... until ...*. But the same effect may be achieved via recursion, as illustrated in Figure 9.

In the figure, the symbols '`a6 b1 b1 b1 c4 d3`' in the New pattern in column 0 ('`pg a6 b1 b1 b1 c4 d3 #pg`') may be seen as parameters for the SP 'program' or grammar for this example (not shown on this occasion), much as in Section 2.1.

One point of interest here is that the pattern '`ri ri1 ri #ri b #b #ri`' (which appears in columns 5, 7, and 9) is recursive because it is self-referential, and this is because the pair of symbols '`ri #ri`' within the larger pattern '`ri ri1`

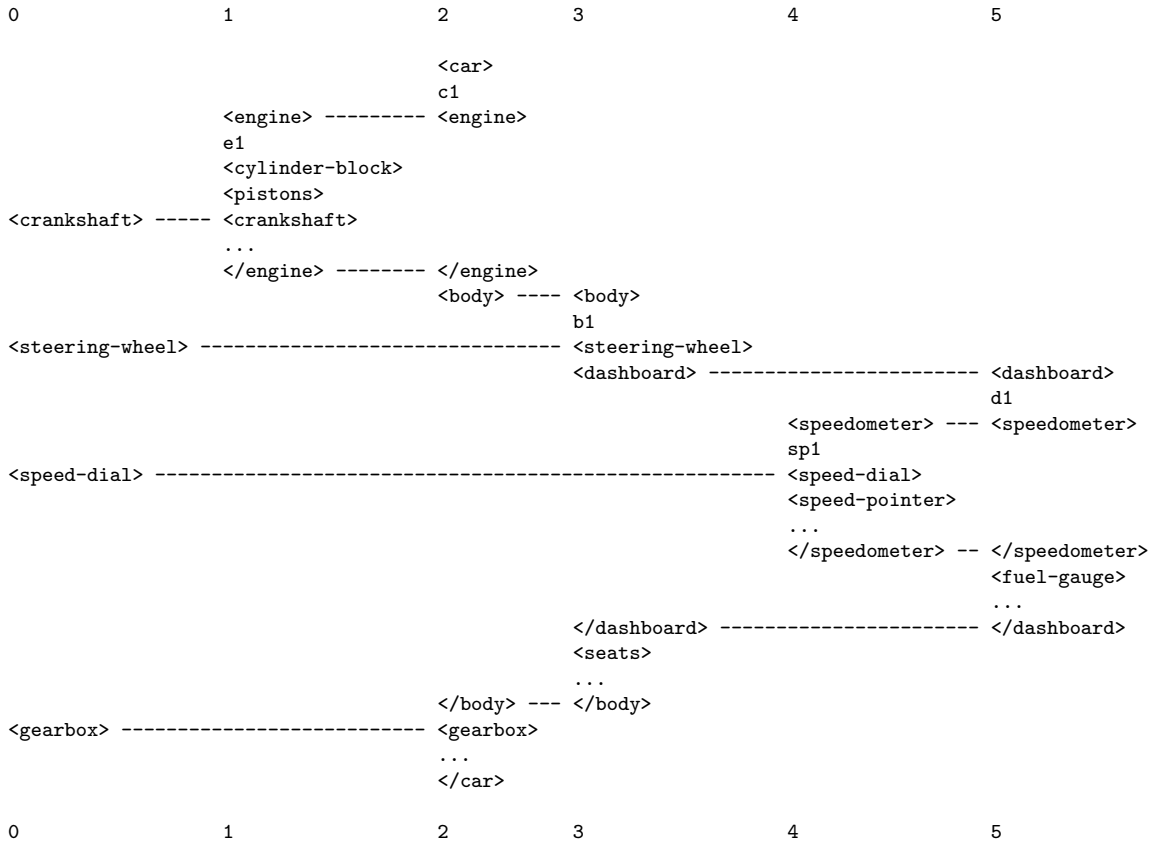

Figure 8: The best SP-multiple-alignment found by the SP model, with the New pattern '<crankshaft> <steering-wheel> <speed-dial> <gearbox>' shown in column 1, and a set of Old patterns representing different parts and sub-parts of a car, shown in columns 1 to 5.

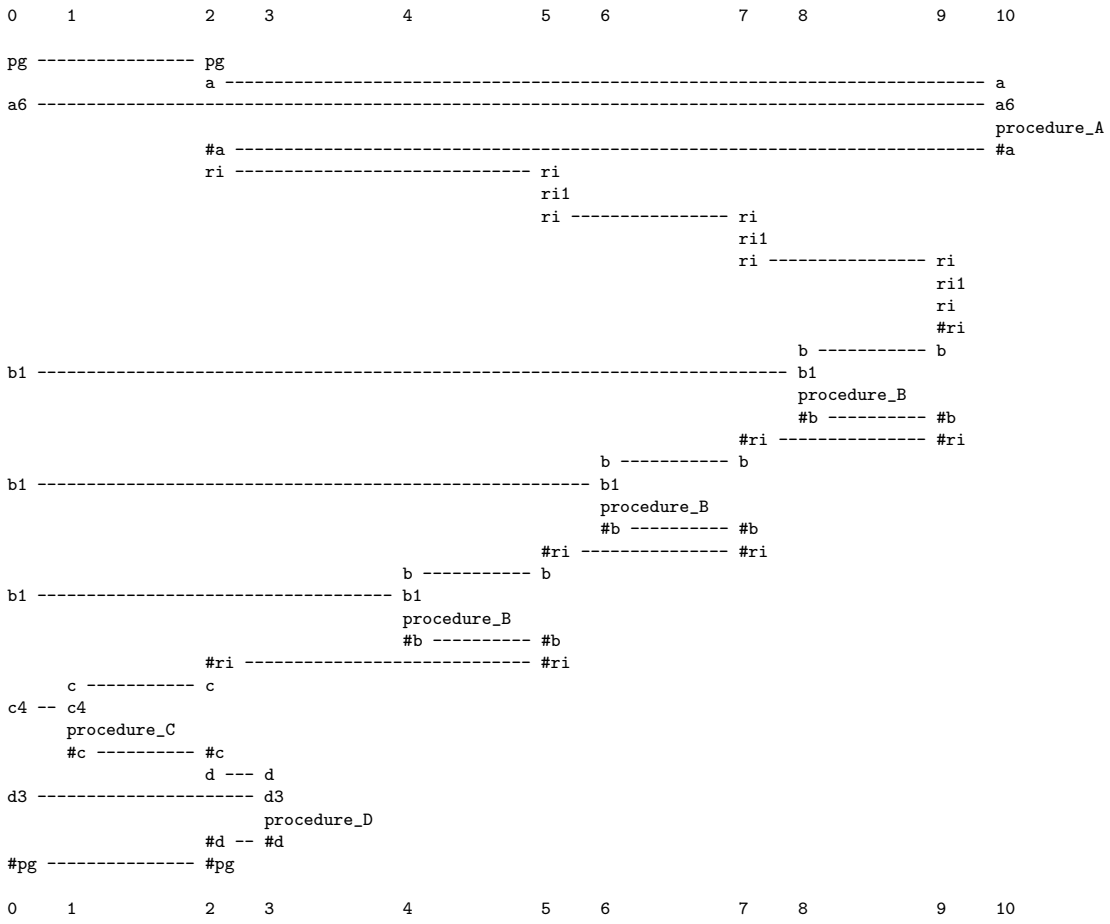

Figure 9: An SP-multiple-alignment produced by the SP model showing how, via recursion mediated by a self-referential pattern (which is '`ri ri1 ri #ri b #b #ri`' in this example), the SP system may model the repetition of a procedure or function, which in this examples is shown as '`procedure_B`'.

ri #ri b #b #ri' may be matched and unified with the same two symbols at the beginning and end of that larger pattern. Hence, the larger pattern contains a reference to itself.

Another point of interest is that the pattern 'ri ri1 ri #ri b #b #ri', and its connected pattern 'b b1 procedure_B #b', each occur 3 times in the SP-multiple-alignment in Figure 9 although each of them only occurs once in the set of Old patterns for this SP-multiple-alignment.

The overall effect of the recursion is to provide for 3 occurrences of the symbol 'procedure_B' via a grammar in which that symbol only occurs once. Hence, the grammar is relatively compressed compared with what would be needed if all possible repetitions were stored explicitly.

With any kind of recursion, something is needed to tell the system when to stop the repetition. In some cases it may be something like "Keep on doing press ups until I tell you to stop", or "Do press ups and stop after you have done 10". In our example, the number of repetitions is specified explicitly by the three instances of the symbol 'b1' within the New pattern, much as in unary arithmetic.

Readers may, with some justice, object that, with this example, the recursive pattern 'ri ri1 ri #ri b #b #ri' may be dropped because information about repetition is contained within the New pattern 'pg a6 b1 b1 b1 c4 d3 #pg'. This is largely true with this example but would not be true if the number of repetitions were not specified in the New pattern. Also the recursion will accommodate any number of repetitions of 'procedure_B', or any of its alternatives within the grammar, not just the three repetitions in our example.

3 The full or partial automation of software development

This and the following main sections aim to describe potential advantages of the SP system in software engineering compared with software engineering with conventional computers.

This section considers the full or partial automation of software and the associated issue of generalisation in software, and how to avoid under- and over-generalisaton.

3.1 Automation of software development

Assuming that the SP machine (Appendix A.5) has been developed to the stage where it has robust abilities for unsupervised learning with both one-dimensional and two-dimensional SP patterns, and assuming that residual problems in that area have been solved (Appendix A.4), the SP machine is likely to prove useful

in both the automatic and semi-automatic creation of software. The former is discussed here and the latter is discussed in the next subsection.

At least three things suggest that such possibilities are credible:

- There is already a considerable body of research related to the automatic or semi-automatic creation of software.[2]
- As noted in Section 2.4 and Appendix **??**, it has been recognised for some time, in connection with the concept of 'structured programming', that the structure of software should mirror the structure of the data that it is designed to process [2]. This fits well with the observation that in forms of unsupervised learning such as grammatical inference, the structure of the resulting grammar reflects the structure of the data from which it was derived.
- And in the same vein, in connection with 'object-oriented design' and 'object-oriented programming' (Section 2.5), it is well-established that a well-structured program should reflect the structure of entities and classes of entities that are significant in the workings of the program. This fits well with the observation that unsupervised learning in the SP system appears to conform to the 'DONSIC' principle [12, Section 5.2]: *the discovery of natural structures via information compression*—where 'natural' means aspects of our environment such as 'objects' which we perceive to be natural.

3.1.1 Example: learning in an autonomous robot

Perhaps the best example of how the SP system may facilitate automatic programming is in autonomous robots that learn continually via their senses, much as people do [14]. Here, the robot's ever-increasing store of knowledge, together with any in-built motivations, provide the basis for many potential inferences ([10, Chapter 7], [12, Section 10]) and, perhaps more important in the present context, the creation of one or more plans ([10, Chapter 8], [12, Section 12]), each one of which may be regarded as a program to guide the robot's actions.

The potential for this kind of development raises important issues about how much autonomy should be granted to any robot and how external controls may be applied. Pending the resolution of such issues, there is potential in the SP system for more humdrum kinds of automatic programming, as described in the next two subsections.

[2]See, for example, 'Programming by example', *Wikipedia*, bit.ly/2uMyVYP; 'Programming by demonstration', *Wikipedia*, bit.ly/2v3phy8; 'Inductive logic programming', *Wikipedia*, bit.ly/2ttwBpn; 'Inductive programming', *Wikipedia*, bit.ly/2uMAgyP; and 'Automatic programming', *Wikipedia*, bit.ly/2vnzAxc; retrieved 2017-07-20.

3.1.2 Example: processing data received by the SKA

The fully automatic creation of software should be possible in situations where there is a body of data that represents the entire problem or a realistically large sample of it. An example is the large volumes of data that will be gathered by the Square Kilometre Array (SKA)[3] when it is completed.

With data like this, unsupervised learning by the SP machine should build grammars that represent entities and classes of entity—such as stars and galaxies—in two dimensions at least, and possibly in three dimensions [13, Sections 6.1 and 6.2]. And its grammars should also embrace 'procedural' or 'process' regularities in the time dimension.

Any such grammar may be seen as a 'program' for the analysis of similar kinds of data in the future. A neat feature of the SP system is that the SP-multiple-alignment construct serves not only in unsupervised learning but also, without modification, in such operations as pattern recognition, reasoning, and more (Appendix **??**).

With an area of application like the processing of data received by the SKA, it may of course happen that significant structures or events—such as supernovas or gamma-ray bursts—do not appear in any one sample of data. For that reason, unsupervised learning should be an ongoing process, much as in people, so that the system may gain progressively more knowledge of its target environment as time goes by.

Some more observations relating to this example are described in Section 3.1.4.

3.1.3 Example: programming by demonstration

Another situation where the SP machine may achieve fully-automatic creation of software is with a technique for programming robots called 'programming by demonstration'.[4]

As an example, a person who is skilled at some operation in the building of a car (such as paint-spraying the front of the car) may take the 'hand' of a robot and guide it through the sequence of actions needed to complete the given operation. Here, signals from sensors in various parts of the robot's arm, including the robot's actuators or 'muscles', would be recorded and the record would constitute a preliminary kind of 'program' of the several positions of the arm and actuators that are needed to complete the operation.

[3] See, for example, 'Square Kilometre Array', *Wikipedia*, bit.ly/2t16xxW, retrieved 2017-07-15.

[4] See, for example, 'Programming by demonstration', *Wikipedia*, bit.ly/2v3phy8, retrieved 2017-07-15.

Any such preliminary program may be processed by the SP system to convert it into something that more closely resembles an ordinary program, with the equivalent of subroutines, repetition of operations, and conditional statements. To allow for acceptable variations in the task, there should also be appropriate generalisation from the raw data, as described in Section 3.3.

Some more observations relating to this example are described next.

3.1.4 Possible augmentations

An assumption behind the two examples just described is that the grammar or program created via unsupervised learning would do everything that is needed.

Probably, in many cases, this would be true. This is because of a neat feature of the SP system: that the SP-multiple-alignment subsystem is not only an important part of unsupervised learning but is also the key to such operations as pattern recognition, several kinds of reasoning, retrieval of information, and problem solving (Appendix **??**). With the SKA example (Section 3.1.2, these kinds of operations may be all that is required. With the programming-by-demonstration example (Section 3.1.3), the program created via unsupervised learning may function directly in controlling the robot arm.

But the user of the SKA system might want to do such things as showing stars in red, galaxies in green, and so on. And the user of the programming-by-demonstration system might want to add some bells and whistles such as playing musical sounds as the robot works.

Clearly, such augmentations fall outside what could be created automatically via unsupervised learning. They take us into to the realm of semi-automatic creation of software, discussed next.

3.2 Semi-automatic creation of software

With some kinds of application, it seems unlikely that the creation of relevant software could be fully automated in the foreseeable future. One example is the kinds of augmentation to an automatically-created program described in Section 3.1.4. Another example is the kind of software that is needed to manage a business—with knowledge of people, vehicles, furniture, packages, warehouses, relevant rules and regulations, and so on.

With the latter kind of problem, there appears to be potential for the system to assist in the refinement of human-created software by detecting redundancies in any draft design, and inconsistencies from one part of the design to another. On the assumption that the software is developed using SP patterns and is hosted on an SP machine (as outlined in Section 4), then the SP machine may be a vehicle for verification and validation of the software as described in Sections 9 and 10.

At some point in the future, it is conceivable that knowledge about how a business operates may, at some stage, be built up by an intelligent autonomous robot of the kind described in [14] that is allowed to explore different areas of the business, observing the kinds entity and operation that are involved, asking questions, and so on. But for the foreseeable future, it seems likely that any software that may have been created by such a robot would need to be augmented and refined by people.

3.3 Generalisation and the avoidance of under- and over-generalisation

As a rule, any given computer program is more general than any set of examples that it may process. For example, an ordinary spreadsheet program can work with millions or perhaps billions of different sets of data, far more than it would ever be used for in practice.

Since we have been considering the possibility that software may be created automatically or semi-automatically in the manner of unsupervised learning (Sections 3 and 3.2), we need to consider how the system would generalise correctly from the examples it has been given, without either under-generalisation (sometimes called 'overfitting') or over-generalisation (sometimes called 'underfitting').

The SP system provides an answer outlined in [12, Section 5.3],[5] with some supporting evidence. In brief, it appears that correct generalisation may be achieved, without either under- or over-generalisation, like this:

1. Given a body of raw data, **I**, compress it as much as possible via unsupervised learning with the SP machine or something equivalent.

2. Divide the resulting compressed version of **I** into two parts: a *grammar*, **G**, which represents the recurring features of **I**, and an *encoding*, **E**, of **I** in terms of **G**.

3. Discard **E** and retain **G**.

Here, **G** may be seen to be a program for processing **I** and for processing many other bodies of data with the same general characteristics as **I**, without either under- or over-generalisation.

[5] That account only mentions over-generalisation but it appears that the same procedure will apply to the avoidance of under-generalisation.

4 Non-automatic programming of the SP system

If or when the automatic creation of software is not feasible, or if something more than small revisions are needed with software that has been created semi-automatically, then something like ordinary programming will be needed.

In principle, this can be done using SP patterns directly. But, mainly for reasons of human psychology, some kind of 'syntactic sugar' or other aids may be helpful for programmers. Here are four possibilities:

- With a pattern like '`PM ST #ST MC #MC PD #PD #PM`' in row 1 of Figure 1, it may he helpful if, when the first symbol ('`PM`') has been typed in, the programming environment would automatically insert the balancing last symbol ('`#PM`').

- Unless or until programmers become used to how things are done in the SP system, it may be helpful to create a programming environment in which SP concepts are presented in a manner that resembles ordinary programming concepts, as described in Section 2.

- Instances of the object-oriented concept of a class-inclusion hierarchy (Section 2.5), and instances of any part-whole hierarchy (dividing an object into its parts and subparts) may be represented graphically and implemented with equivalent sets of SP patterns.

There will be a need for programmers to specify aspects of parallel and sequential processing, as described in Section 5, next.

5 Parallel processing in the real world and in the SP machine

With the SP system, there is a need to distinguish sharply between two kinds of parallel processing:

- *Real-world parallel processing.* This is the kind of parallel processing that a cook might use when he or she prepares the icing for a cake at the same time as the cake is baking in the oven, or that a pianist uses when he or she plays with both hands. This kind of parallel processing would be visible to the user of the SP machine and would be part of their thinking about the task in hand.

  It appears that this kind of parallelism may be represented and executed in the SP system via the use of two-dimensional SP patterns, as described in [14, Sections V-G, V-H, and V-I, and Appendix C].

- *Parallel processing in the workings of the SP machine.* To speed up the SP machine, it is envisaged that parallel processing will be employed in the workings of the SP machine, in such tasks as searching for matches between patterns and building multiple alignments. This kind of parallel processing would be invisible to the user of the machine who would not normally need to think about it.

In this connection, potential advantages of the SP system compared with conventional computers are:

- With conventional computers, programmers normally have to worry about parallelism that is reflected in the real world and parallelism that is needed to speed up processing—and with the latter kind of parallelism, some tricky issues can arise.[6] By contrast, programmers with the SP system would only need to worry about real-world parallelism (more below).

- With the SP system, there is potential for the representation and processing of real-world parallelism, as described in [14, Sections V-G, V-H, and V-I, and Appendix C].

With regard to the programming of real-world parallelism, there will be a need for programmers to specify where operations are to be done in sequence or in parallel. And, where real-world processes are to be done more slowly than is implied by the speed of the SP machine, there will be a need for a clock and for a means for programmers to specify timings of operations.

6 Programming via natural language

One of the strengths of the SP system is in the processing of natural language, mentioned in Appendices **??** and **??**, and described in more detail in [12, Section 8] and [10, Chapter 5].

There is clear potential in the SP system for developing human-level processing of writing, and ultimately speech, but there will be some difficult hurdles to overcome, probably requiring a two-pronged attack: working on problems in the processing of natural language together with problems in the unsupervised learning of syntactic knowledge, semantic knowledge, and syntactic/semantic associations [8, Sections 9 and 10].

If or when these problems are solved, there is potential for programming the SP system using written or spoken natural language, in much the same way that

[6] For a description of some of the issues, see 'Parallel computing', *Wikipedia*, bit.ly/1MPI5kA, retrieved 2017-08-10.

people can be given written or spoken instructions. However, achieving human levels of understanding is an ambitious goal and is not likely to be realised soon.

7 Bringing 'design' closer to 'implementation'

It has been established for some time that, in conventional development of software, one should begin with a relatively abstract high-level design (which is often represented graphically) and then translate that into a working program. There seem to be three main reasons for this approach:

- With any kind of design, it is often useful to establish a relatively abstract 'big picture' before filling in details.
- For the kinds of reasons described in Section 4, it may be useful to disguise the details of a program behind syntactic sugar that is more congenial for programmers.
- Even with 'high' level programming languages such as C++, Python, or Java, or 'declarative' systems such as Prolog, it is often necessary to pay attention to the details of how the underlying machine will run a program, details that are not relevant to the more abstract 'design' of the software, with its focus on entities and processes that are significant for the user.

The SP system probably makes no difference to the first and second of the above points, but it is likely to be helpful with the third. This is because, in the manner of declarative programming systems, it will probably allow programmers to specify 'what' computations are to be achieved, and to reduce or eliminate the need to consider 'how' the computations should be done.

8 Possible reductions in the need for operations like compiling or interpretation

At first sight, the SP system eliminates the need for anything like compiling or interpretation. This is because it works entirely via searches for full or partial matches between SP patterns, or parts of patterns, with corresponding unifications.

But it is likely that, in the development of the SP machine, indexing will be introduced to record the first match between a given symbol and any other symbol, and thus speed up the later retrieval of the zero or more matching partners of the given symbol [8, Section 3.4]. And it is likely that similar measures will be

introduced into the computer model for SP-neural [8, Section 13.2], a version of the SP theory expressed in terms of neurons and their interconnections.

Indexing of that kind is similar in some respects to the use of compiling or interpretation in a conventional computing system. Hence it would be misleading to suggest that the SP system would eliminate the need for such operations. But there are potential gains in this area, especially if, at some later stage, it became feasible to introduce very fast and highly-parallel searching for matches between patterns which may reduce or eliminate the need for indexing.

9 Verification

The SP system has potential to reduce the need for 'verification' of software—meaning processes designed to reduce or eliminate 'bugs' in software via dynamic testing or static analysis—and there is corresponding potential for improvements in the quality of software. The main reasons for these potential benefits are:

- *The potential of the system for automatic or semi-automatic creation of software* (Sections 3 and 3.2). To the extent that automatic or semi-automatic creation of software is possible, it should reduce or eliminate human-induced errors in software.

- *Potential reductions in the sizes of software systems.* The potential of the system for reductions in the overall sizes of software systems (Section 12) means that there are likely to be fewer opportunities to introduce bugs into software, and, probably, less searching would be required in the detection of bugs via static analysis of software.

- *Bringing 'design' closer to 'implementation'.* To the extent that 'design' and 'implementation' may be merged (Section 7), and in particular to the extent that SP software may concentrate on 'what' the user needs and reduce or eliminate details of 'how' the underlying machine may meet those needs, there is potential to reduce the numbers of bugs in programs.

10 Validation

In addition to its potential with verification, the SP system has potential to strengthen the process of 'validation' in software development—meaning checking to ensure that each body of software fulfills its intended purpose.

As with verification, the potential of the SP system for the automatic or semi-automatic creation of software means elimination or reduction of the kinds of human error that may send a program off track.

Also, the potential of the SP system to bring 'design' and 'implementation' closer together (Section 7) can mean fewer opportunities for a program to drift away from its original conception.

11 Seamless integration of 'software' with 'database'

In the SP system, *all* kinds of knowledge are represented with arrays of atomic *symbols* in one or two dimensions (Appendix A.1), and *all* kinds of processing is achieved via the matching and unification of patterns. For these two reasons, and because of the system's potential for *general, human-like artificial intelligence* (GHLAI) [21], there would be no distinction in the SP system between 'software' and 'database', as there is normally in conventional software engineering projects.

A potential benefit of this kind seamless integration of software and database is elimination of awkward incompatibilities between different kinds of knowledge and elimination of the need for translations where incompatibilities exist.

How the SP system may meet some of the objectives of software engineering is discussed at various points in [11].

11.1 Hybrid systems

A qualification to the foregoing is that, pending provision of arithmetic processing in the SP system [12, Section 3.3], there may be a need to provide such capabilities via a hybrid of a conventional arithmetic co-processor working in conjunction with an SP machine. It is envisaged that such an arrangement would be a stop-gap pending fuller development of the SP system.

12 An overall simplification of computing applications

With the SP system, there is potential for an overall simplification of applications compared with what is required with ordinary computers [16, Section 5].

In broad terms, this potential arises because of the way in which conventional software contains often-repeated procedures for searching amongst data, and similar 'low level' operations needed to overcome shortcomings in conventional CPUs. In an SP system, the 'CPU' is relatively complex but with fewer of the shortcomings of conventional CPUs, so that that relative complexity is, probably, more than offset by simplifications in software. That relative advantage is likely to grow, roughly in proportion to the numbers of applications and their sizes.

This kind of idea is not new. In the early days of databases, each database had its own procedures for searching and for retrieval of information, and it had its own user interface and procedures for printing, and so on. People soon realised that it would make better sense to develop a general-purpose system for the management of data, with a user interface and system for retrieval of data, and to load it with different bodies of data according to need. There was a similar evolution in expert systems, from bespoke systems to general-purpose 'shells'.

13 Reducing the variety of formats and formalisms in computing

As noted in Section 11, the SP system has potential for *general, human-like artificial intelligence* (GHLAI). What this means in the SP programme of research, and how the concept of a GHLAI differs from alternatives such as the concept of a universal Turing machine, is discussed in [21].

If this expectation is born out, and the evidence is strong, there is clear potential for the use of the SP machine to clean up the curse of variety in the thousands of different formats and formalisms which exist for the representation of data, and the hundreds of different computer languages for describing how data may be processed (Appendix C).

14 Version control

In a typical software engineering project, there is a need to keep track of the parts and sub-parts of the developing program. At the same time, there is a need to keep track of a hierarchy of versions and subversions. And, associated with each part or version, there may be several different kinds of document, including a statement of requirements, a high-level design, a low-level design, and notes. To avoid awkward inconsistencies, these things should be smoothly integrated.[7]

The SP system provides a neat solution to the problem of integrating a class-inclusion hierarchy with a part-whole hierarchy, as described in [12, Section 9.1] and [10, Section 6.4].[8] Although these sources do not demonstrate the point, it

[7] The problem of integrating a class-inclusion hierarchy with a part-whole hierarchy—a problem that arose in connection with version control in a project to develop an 'Integrated Project Support Environment' (IPSE) when I was working as a software engineer with Praxis Systems plc—was one of the main sources of inspiration for the development of the SP system.

[8] The solution also applies to class-inclusion heterarchies, meaning class-inclusion hierarchies with cross-classification.

appears that the SP system also provides for each version or part to have one or more associated documents, as outlined above.

Also relevant to these issues is a brief discussion of how to maintain multiple versions and parts of a document or web page in [16, Section 6.10.3].

15 Technical debt

As noted in [16, Section 6.6.6], the SP system has potential to reduce or eliminate the problem of 'technical debt', meaning the way in which software systems can become progressively more unmanageable with the passage of time, owing to an accumulation of postponed or abandoned maintenance tasks, or a progressive deterioration in the design quality or maintainability of the software via the repeated application of 'fixes' in response to short-terms concerns, without sufficient attention to their global and long-term effects.

The SP system may reduce or eliminate the problem of technical debt by streamlining the process of software development via automatic or semi-automatic automation of software development, by reducing the gap between design and implementation, by streamlining processes of verification and validation, and other facilitations described in preceding sections.

16 Conclusion

This paper describes a novel approach to software engineering derived from the *SP theory of intelligence* and its realisation in the *SP computer model*. It is anticipated that the SP theory and the SP computer model, together, will be the basis for the development of an industrial-strength *SP machine*. And a mature version of the SP machine is seen as the likely vehicle for software engineering as described in this paper.

Although concepts such as SP-multiple-alignment, which are central in the SP system, seem far removed from concepts associated with software engineering, Section 2 describes how many of those concepts map quite neatly into elements of the SP system.

Potential benefits of this new approach to software engineering include:

- *The automation of semi-automation of software development.* Taking advantage of the SP system's strengths and potential in unsupervised learning, there is clear potential for the automation or semi-automation of software development.

- *Non-automatic programming of the SP system.* Where it is not possible to create software automatically, or when human assistance is needed, there is clear potential for programming the SP system in much the same way as a conventional system.

- *Programming via natural language.* An ambitious goal, which is not likely to be realised soon, is to bring the SP system to a point where it has human levels of understanding and production of natural language, so that the SP system may be 'programmed' in much the same way that people can be given written or spoken instructions.

- *Reducing or eliminating the distinction between 'design' and 'implementation'.* By contrast with conventional systems, there is potential in the SP system to reduce or eliminate the distinction between 'design' and 'implementation'. This is because elements of software design such as structured programming and object-oriented design with inheritance of attributes may be expressed directly with SP patterns.

- *Reducing or eliminating operations like compiling or interpretation.* The SP system has potential to reduce or eliminate operations like compiling or interpretation. This is because the system works directly on 'source' code by searching for patterns or parts of patterns that match each other. But it seems likely that indexing of matches between symbols will speed up the system, and the compiling of such an index may be seen to be similar to what is entailed in conventional compiling or interpretation.

- *Reducing or eliminating the need for verification of software.* The need for verification of SP software may be reduced or eliminated: via the potential of the SP system for the automatic or semi-automatic creation of software; because compression of software is likely to reduce the opportunities for bugs to be introduced; and because there is likely to be a reduced need to bridge the divide between design and implementation.

- *Reducing the need for an explicit process of verification of software.* The SP system also has potential to help ensure that software meets its objectives, thus reducing the need for an explicit process of verification. This is because of the system's potential for automatic or semi-automatic creation of software and because of the way in which design and implementation may be brought closer together or merged.

- *No formal distinction between program and database.* Unlike conventional systems, where 'programs' and 'databases' are distinguished quite sharply, there is no formal distinction of that kind in the SP system because all kinds

of knowledge are expressed with SP patterns. This can mean useful simplifications on occasion, and it can reduce or remove awkward incompatibilities.

- *Potential for an overall simplification of computing applications.* Despite the fact that the processing 'core' of the SP system is, almost certainly, more complex than the CPU of a conventional computer, there is potential with the SP system for an overall simplification of computing applications, considering hardware and software together.

- *Potential for substantial reductions in the number of types of data file and the number of computer languages.* Because of the SP system's potential as a *general, human-like artificial intelligence* (GHLAI), there is potential to reduce the many thousands of types of data file to one, and to reduce the hundreds of different computer languages to one. A possible qualification here is that users may wish to create sub-types of data file or sub-types of computer language, each sub-type containing information about this or that specialised domain such as physics or finance.

- *Allowing programmers to concentrate on 'real-world' parallelism, without worries about parallelism to speed up processing.* With a mature version of the SP machine, it is intended that parallelism that is designed only for the purpose of speeding up processing will be built into the system, so that programmers need not worry about it. They would be free to concentrate on parallelism in the real world, perhaps with assistance from unsupervised learning.

- *Benefits for version control.* The SP system has potential to help organise all the knowledge associated with any given software development project, with provision for: the representation of hierarchies of versions of the software; the representation of parts and sub-parts of the software; the seamless integration of version hierarchies with part-whole hierarchies; and, for any given version or part, the representation of the one or more kinds of information associated with that element. It provides for cross-classification where that is required.

- *Reducing technical debt.* The potential of the SP system to increase the efficiency of software development can mean reductions in 'technical debt', meaning the way in which software systems can become progressively more unmanageable with the passage of time, owing to short-term fixes and the postponement or abandonment of maintenance tasks.

A Outline of the SP system

To help ensure that this paper is free standing, the SP system is described here in outline with enough detail to make the rest of the paper intelligible.

The *SP theory of intelligence* and its realisation in the *SP computer model* is the product of a unique extended programme of research aiming to simplify and integrate observations and concepts across artificial intelligence, mainstream computing, mathematics, and human learning, perception, and cognition, with information compression as a unifying theme.[9]

The latest version of the SP computer model is SP71. Details of where the source code and associated files may be obtained are here: www.cognitionresearch.org/sp.htm#ARCHIVING.

It is envisaged that the SP computer model will provide the basis for the development of an industrial-strength *SP machine*, described briefly in Appendix A.5, below.

The SP system is described most fully in [10] and quite fully but more briefly in [12]. Other publications from this programme of research are detailed, many with download links, on www.cognitionresearch.org/sp.htm.

A.1 Overview

The SP theory is conceived as a brain-like system which receives *New* information via its senses and stores some or all of it in compressed form as *Old* information, as shown schematically in Figure 10.

Both New and Old information are expressed as arrays of atomic *symbols* in one or two dimensions called *patterns*. To date, the SP computer model works only with one-dimensional patterns but it is envisaged that it will be generalised to work with two-dimensional patterns.

In this context, a 'symbol' is simply a mark that can be matched with any other symbol to determine whether they are the same or different. No other result is permitted. Apart from some distinctions needed for the internal workings of the SP system, SP symbols do not have meanings such as 'plus' ('+'), 'multiply' ('∗'), and so on. Any meaning associated with an SP symbol derives entirely from other symbols with which it is associated.

[9]This ambitious objective is in keeping with Occam's Razor. And as a means of solving the exceptionally difficult problem of developing general, human-level artificial intelligence, it is in keeping with "If a problem cannot be solved, enlarge it", attributed to President Eisenhower; it chimes with Allen Newell's exhortation that psychologists should work to understand "a genuine slab of human behaviour" [6, p. 303] and his work on *Unified Theories of Cognition* [7]; and it is in keeping with the quest for 'Artificial General Intelligence' (*Wikipedia*, bit.ly/1ZxCQPo, retrieved 2017-08-15).

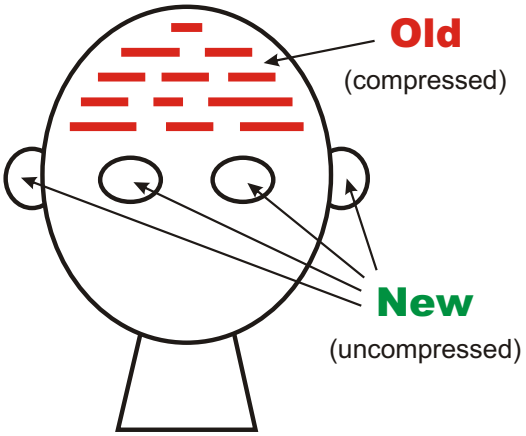

Figure 10: Schematic representation of the SP system from an 'input' perspective. Adapted with permission from Figure 1 in [12].

A.2 Multiple alignments in bioinformatics

At the heart of the SP system is a process of finding patterns that match each other and merging or 'unifying' multiple instances to make one, a process which is referred to here and elsewhere as 'Information Compression via the Matching and Unification of Patterns' (ICMUP). More specifically, a central part of the SP system is a concept of *multiple alignment*, borrowed and adapted from bioinformatics.[10]

The original concept of multiple alignment is an arrangement of two or more DNA sequences or sequences of amino acid residues, in rows or columns, with judicious 'stretching' of selected sequences in a computer to bring matching symbols, as many as possible, into line. An example of such a multiple alignment of five DNA sequences is shown in Figure 11.

A.3 SP-multiple-alignments in the SP system

In the SP system, multiple alignments are sufficiently different from those in bioinformatics for them to be given a different name: *SP-multiple-alignments*.[11] The distinctive features of an SP-multiple-alignment are:

- Normally, one New pattern is shown in row 0 (or column 0 when patterns

[10]Six variants of ICMUP and how they may be realised via SP-multiple-alignment are described in Appendix B.

[11]This name has been introduced fairly recently to make clear that there are important differences between the two kinds of multiple alignment.

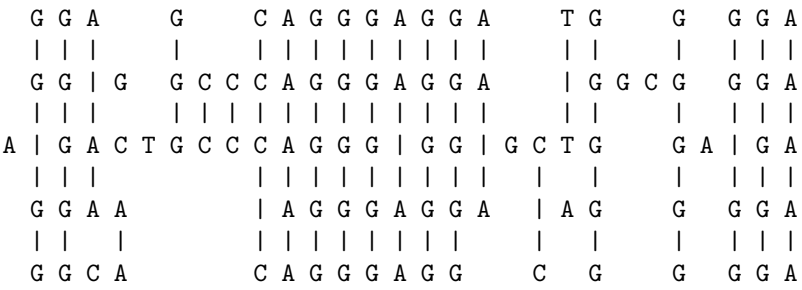

Figure 11: A 'good' multiple alignment amongst five DNA sequences. Reproduced with permission from Figure 3.1 in [10].

are arranged in columns as shown in Figure 2). Sometimes there is more than one New pattern in row 0 or column 0.

- The Old patterns are shown in the remaining rows (or columns), one pattern per row (or column).

- As with the original concept of multiple alignment, the aim in building multiple alignments is to bring matching symbols into alignment. More specifically in SP-multiple-alignments, the aim is to create or discover one or more 'good' SP-multiple-alignments that allow the New pattern to be encoded economically in terms of the Old patterns. How this encoding is done is described in [10, Section 2.5] and in [12, Section 4.1].

An example of an SP-multiple-alignment is shown in Figure 12.

In this SP-multiple-alignment, a sentence is shown as a New pattern in row 0. The remaining rows show Old patterns, one per row, representing grammatical structures including words. The overall effect is to analyse (parse) the sentence into its parts and subparts. The pattern in row 8 shows the association between the plural subject of the sentence, marked with the symbol '`Np`', and the plural main verb, marked with the symbol '`Vp`'.

Because, with most ordinary multiple alignments or with SP-multiple-alignments, there is an astronomically large number of ways in which patterns may be aligned, discovering good multiple alignments means the use of heuristic methods: building each multiple alignment in stages and discarding all but the best few multiple alignment at the end of each stage. With this kind of technique it is normally possible to find multiple alignments that are reasonably good but it is not normally possible to guarantee that the best possible multiple alignment has been found.

The concept of SP-multiple-alignment has proved to be extraordinarily powerful: in the representation of knowledge (Appendix **??**), in aspects of intelligence

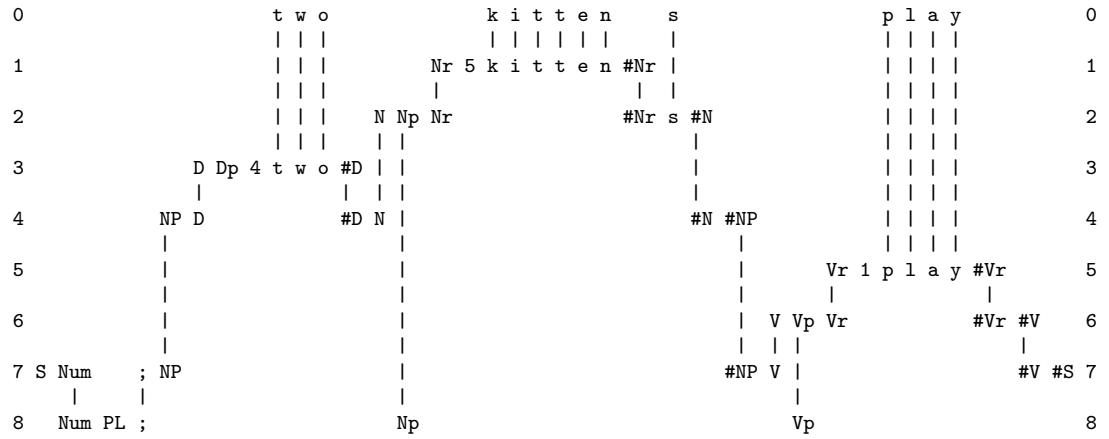

Figure 12: The best SP-multiple-alignment created by the SP computer model with a store of Old patterns like those in rows 1 to 8 (representing grammatical structures, including words) and a New pattern (representing a sentence to be parsed) shown in row 0. Adapted with permission from Figures 1 in [11].

(Appendix **??**), and in the seamless integration of diverse kinds of knowledge and diverse aspects of intelligence in any combination (Appendix **??**). It could prove to be as significant for an understanding of intelligence as is DNA for biological sciences: it could be the 'double helix' of intelligence.

There is more about the concept of SP-multiple-alignment in Appendix B.

A.4 Unsupervised learning

Unsupervised learning in the SP system is described quite fully in [10, Sections 3.9 and 9.2]. The aim with unsupervised learning in the SP system is, for a given set of New patterns, to create one or two *grammars*—meaning collections of Old SP patterns—that are effective at encoding the given set of New patterns in an economical manner.

The building of SP-multiple-alignments is an integral part of unsupervised learning in the SP system. It provides a means of creating Old patterns via the direct assimilation of New patterns into the set of Old patterns, and via the splitting of New patterns and pre-existing Old patterns to create additional Old patterns. And it provides a means of evaluating candidate grammars in terms of their effectiveness at encoding the given set of New patterns in an economical manner.

As with the building of SP-multiple-alignments, the creation of good grammars requires heuristic search through the space of alternative grammars: creating grammars in stages and discarding low-scoring grammars at the end of each stage.

The SP computer model can discover plausible grammars from samples of English-like artificial languages. This includes the discovery of segmental structures, classes of structure, and abstract patterns.

At present, the program has two main weaknesses outlined in [12, Section 3.3]: it does not learn intermediate levels of abstraction or discontinuous dependencies in data. However, it appears that these problems are soluble, and it seems likely that their solution would greatly enhance the performance of the system in the learning of diverse kinds of knowledge.

To ensure that unsupervised learning in the SP system is robust and useful across a wide range of different kinds of data, it will be necessary for the system, including its procedures for unsupervised learning, to have been generalised for two-dimensional patterns as well as one-dimensional patterns (Appendix A.1).

A.5 The SP machine

As mentioned earlier, it is envisaged that an industrial-strength *SP machine* will be developed from the SP theory and the SP computer model [8]. Initially, this will be created as a high-parallel software virtual machine, hosted on an existing high-performance computer. An interesting possibility is to develop the SP machine as a software virtual machine that is driven by the high-parallel search processes in any of the leading internet search engines.

Later, there may be a case for developing new hardware for the SP machine, to take advantage of optimisations that may be achieved by tailoring the hardware to the characteristics of the SP system. In particular, there is potential for substantial gains in efficiency and savings in energy compared with conventional computers by taking advantage of statistical information that is gathered by the SP system as a by-product of how it works ([15, Section IX], [14, Section III], [8, Section 14]).

A.6 Distinctive features and advantages of the SP system

Distinctive features of the SP system and its main advantages compared with AI-related alternatives are described in [18]. In particular, Section V of that paper describes thirteen problems with deep learning in artificial neural networks and how, with the SP system, those problems may be overcome. The SP system also provides a comprehensive solution to a fourteenth problem with deep learning—'catastrophic forgetting'—meaning the way in which new learning in a deep learning system wipes out old memories.[12]

[12] A solution has been proposed in [4] but it appears to be partial, and it is unlikely to be satisfactory in the long run.

The main strengths of the SP system are in its versatility in the representation of several kinds of knowledge (Appendix **??**), its versatility in several aspects of intelligence (Appendix **??**), and because these things all flow from one relatively simple framework—the SP-multiple-alignment concept—they may work together seamlessly in any combination (Appendix **??**). That kind of seamless integration appears to be essential in any system that aspires to general human-level artificial intelligence.

A.7 Potential benefits and applications of the SP system

Potential benefits and applications of the SP system are described in several peer-reviewed papers, copies of which may be obtained via links from www.cognitionresearch.org/sp.htm: the SP system may help to solve nine problems with big data [15]; it may help in the development of human-like intelligence in autonomous robots [14]; the SP system may help in the understanding of human vision and in the development of computer vision [13]; it may function as a database system with intelligence [11]; it may assist medical practitioners in medical diagnosis [9]; it provides insights into commonsense reasoning [17]; and it has several other potential benefits and applications described in [16]. And, of course, this paper describes how the SP system may be applied in software engineering.

B Six Variants of ICMUP and their realisation via SP-multiple-alignment

In writing about the SP system, it has proved useful in [20, Section 2.1] and [19, Section 2.2] to refer to six variants of ICMUP, which are summarised below.

The purposed of this section is to describe, with examples from the body of this paper, how those six variants of ICMUP, may be realised via the concept of SP-multiple-alignment.

Accordingly, forms of ICMUP should be seen as instances of the more general principle of information compression via SP-multiple-alignment. And for reasons given in Appendix **??**, all six variants may be integrated seamlessly within the SP-multiple-alignment framework, in any combination.

B.1 Basic ICMUP

The simplest version of ICMUP is where two patterns match each other and they are merged or 'unified' to make one. This may be modelled very simply and directly in the SP-multiple-alignment framework with a simple match between all

or part of a New pattern (such as '`t w o`' in Figure 12) and one Old pattern (such as '`D Dp 4 t w o #D`' in the same figure).

B.2 Chunking-with-codes

The basic idea here is that, with each unified 'chunk' of information, give it a relatively short name, identifier, or 'code', and use that as a shorthand for the chunk of information wherever it occurs.

An example of a chunk of information with an associated code is the SP pattern '`PD 1 apple crumble #PD`' in Figure 1. Here, '`apple crumble`' may be seen as a chunk of information (that represents what needs to be done to prepare a serving of apple crumble), while the symbols '`PD`' and '`1`' may be seen as code symbols which, together, distinguish the given chunk from other chunks of information in Figure 1.

In this context, the symbol '`#PD`' is not essential but it has a useful purpose as a marker for the end of the chunk, much like such symbols as '`</H1>`' and '`</LI>`' in HTML, and similar symbols in versions of XML.

B.3 Schema-plus-correction

Schema-plus-correction is like chunking-with-codes but the unified chunk of information may have one or more variations or 'corrections' on different occasions.

This idea is may be illustrated with another example from Figure 1. The pattern '`PM ST #ST MC #MC PD #PD #PM`' may be seen as a chunk of information with the code symbol '`PM`', much as in Section B.2. But here, the chunk is a schema containing three slots, '`ST #ST`', '`MC #MC`', and '`PD #PD`', each of which may be assigned values like '`ST 0 mussels #ST`', '`MC 4 salad #MC`', and '`PD 1 apple crumble #PD`', as shown in Figure 2.

The value assigned to each slot, which may vary from one occasion to another, may seen to be a 'correction' to the schema.

B.4 Run-length coding

The variant called 'run-length coding' may be used with any sequence of two or more copies of a pattern. In that case, it is only necessary to record one copy of the pattern, with something to define the length of the sequence, such as the number of copies of the pattern, or tags to mark the start and end of the sequence.

With the SP system, run-length coding may be achieved via recursion, as described in Section 2.6. In general, this means the use of a self-referential pattern like the pattern '`ri ri1 ri #ri b #b #ri`' in Figure 9, as described in the accompanying text.

B.5 Class-inclusion hierarchies with inheritance of attributes

Here, there is a hierarchy of classes and subclasses, with 'attributes' at each level. At every level except the top level, each subclass 'inherits' all the attributes of all the higher levels.

The way in which a class-inclusion hierarchy may be modelled in the SP system is described in Section 2.5.1, with an illustration in Figure 7.

B.6 Part-whole hierarchies with inheritance of contexts

This is like class-inclusion hierarchies with inheritance of attributes except that the structure represents the parts and subparts of some entity. Each subpart may be seen to inherit its place in larger structures.

The way in which a part-whole hierarchy may be modelled in the SP system is described in Section 2.5.2, with an illustration in Figure 8.

C The curse of variety in computing and what can be done about it

Wikipedia lists nearly 4,000 different 'extensions' for computer files, representing a distinct type of file.[13] A scan of the list suggests that most of these types of file are designed as input for this or that application. Each application is severely restricted in what kinds of file it can process—it is often only one—and incompatibilities are rife, even within one area of application such as word processing or the processing of images. And a program that will run on one operating system will typically not run on any other, so normally a separate version of each program is needed for each operating system, and, with some exceptions, each version needs its own kind of data file.

This kind of variety may also be found within individual files. In a Microsoft Word file, for example, there may be text in several different fonts and sizes, information generated by the 'track changes' system, equations, WordArt, hyperlinks, bookmarks, cross-references, Clip Art, pre-defined shapes, SmartArt graphics, headers and footers, embedded Flash videos, images created by drawing tools, tables, and imported images in any of several formats including JPEG, PNG, Windows Metafile, and many more.

Excess variety is also alive and well amongst computer languages. Several hundred high-level programming languages are listed by Wikipedia, plus large

[13] Details may be seen in 'List of filename extensions', *Wikipedia*, bit.ly/28LaT4v, retrieved 2016-08-16.

numbers of assembly languages, machine languages, mark-up languages, style-sheet languages, query languages, modelling languages, and more.[14]

C.1 Problems arising from excessive variety in computing

Excessive variety in computing is so familiar that we think of it as normal—part of the 'wallpaper' of computing. But, although some may see that variety as evidence of vitality in computing, it is probably more accurate to see it as a symptom of a deep malaise in computing as it is today.

Much of this excessive variety is quite arbitrary, without any real justification, and the source of significant problems in computing such as:

- *Bit rot.* The first of these, bit rot, is when software or data or both become unusable because technologies have moved on. Vint Cerf of Google has warned that the 21st century could become a second 'Dark Age' because so much data is now kept in digital format, and that future generations would struggle to understand our society because technology is advancing so quickly that old files will be inaccessible. See, for example, "Google's Vint Cerf warns of 'digital Dark Age'", *BBC News*, 2015-02-13, bbc.in/1D3pemp.

- *Difficulties in extracting value from big data.* With big data—the humongous quantities of information that now flow from industry, commerce, science, and so on—excessive variety in formalisms and formats for knowledge and in how knowledge may be processed is one of several problems that make it difficult or impossible to obtain more than a small fraction of the value in those floods of data [3, 5]. Most kinds of processing—reasoning, pattern recognition, planning, and so on—will be more complex and less efficient than it needs to be [15, Section III]. In particular, excess variety is likely to be a major handicap for data mining—the discovery of significant patterns and structures in big data [15, Section IV-B].

- *Inefficiencies in the development of software.* Excessive variety in computing also means inefficiencies in the labour-intensive and correspondingly expensive process of developing software and the difficulty of reducing or eliminating bugs in software.

- *Safety and security.* And excess variety in computing means potentially serious consequences for such things as the safety of systems that depend on computers and software, and the security of computer systems. With

[14] There is more information in 'List of programming languages', *Wikipedia*, bit.ly/1GTW05W, retrieved 2016-08-16; and also in 'Computer language' and links from there, *Wikipedia*, bit.ly/2aZ2kag, retrieved 2016-08-17.

regard to cybersecurity, Mike Walker, head of the Cyber Grand Challenge at DARPA, has said that it counts as a grand challenge because of, *inter alia*, the sheer complexity of modern software. A relevant news report is "Can machines keep us safe from cyber-attack?", *BBC News*, 2016-08-02, bbc.in/2aLGwOu.

C.2 A potential solution to the problem of excessive variety

The SP system provides a potential solution to the kinds of problems described in Appendix C.1. It arises from the following three features of the SP system:

- *Versatility of the SP system in the representation of knowledge.* The SP system has already-demonstrated versatility in the representation of diverse kinds of knowledge (Appendix **??**), with reasons to think that it may serve in the representation of *any* kind of knowledge (Appendix **??**).

- *Versatility of the SP system in aspects of intelligence.* The SP system has already-demonstrated versatility in aspects of intelligence (Appendix **??**), with reasons to think that it provides a relatively firm foundation for the development of general, human-like artificial intelligence (Appendix **??**).

- *Potential of the SP system to perform any kind of computable process or procedure.* As described in Appendix **??**, the SP system has potential, via learning or programming, for any kind of computation that is not ruled out by problems with computational complexity.

An implication of the foregoing is that, instead of the great variety of kinds of input file for programs that prevails in computing today, we need only one: a type of computing file that contains SP patterns, as described in Appendix A.1.

In a similar way, there is potential to replace all the many different computer languages with one language composed entirely of SP patterns to be processed by the SP machine.

A possible qualification to the idea that there might be only type of data file and one type of computer language is that, in both cases, users may wish to create sub-types of data file and sub-types of computer language by incorporating domain-specific knowledge in any given sub-type. For example, information about physics might be incorporated in a special-purpose language for use by physicists, and information about finance might be incorporated in a special-purpose language for that domain.

References

[1] G. M. Birtwistle, O-J Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur, Lund, 1973.

[2] M. A. Jackson. *Principles of Program Design*. Academic Press, New York, 1975.

[3] J. E. Kelly and S. Hamm. *Smart machines: IBM's Watson and the era of cognitive computing*. Columbia University Press, New York, Kindle edition, 2013.

[4] J. Kirkpatrick. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences of the United States of America*, 114(13):3521–3526, 2017.

[5] National Research Council. *Frontiers in Massive Data Analysis*. The National Academies Press, Washington DC, 2013. ISBN-13: 978-0-309-28778-4. Online edition: bit.ly/14A0eyo.

[6] A. Newell. You can't play 20 questions with nature and win: projective comments on the papers in this symposium. In W. G. Chase, editor, *Visual Information Processing*, pages 283–308. Academic Press, New York, 1973.

[7] A. Newell, editor. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Mass., 1990.

[8] V. Palade and J. G. Wolff. Development of a new machine for artificial intelligence. Technical report, CognitionResearch.org, 2017. Submitted for publication. bit.ly/2tWb88M, arXiv:1707.0061.

[9] J. G. Wolff. Medical diagnosis as pattern recognition in a framework of information compression by multiple alignment, unification and search. *Decision Support Systems*, 42:608–625, 2006. bit.ly/1F366o7, arXiv:1409.8053.

[10] J. G. Wolff. *Unifying Computing and Cognition: the SP Theory and Its Applications*. CognitionResearch.org, Menai Bridge, 2006. ISBNs: 0-9550726-0-3 (ebook edition), 0-9550726-1-1 (print edition). Distributors, including Amazon.com, are detailed on bit.ly/WmB1rs.

[11] J. G. Wolff. Towards an intelligent database system founded on the SP theory of computing and cognition. *Data & Knowledge Engineering*, 60:596–624, 2007. bit.ly/1CUldR6, arXiv:cs/0311031.

[12] J. G. Wolff. The SP theory of intelligence: an overview. *Information*, 4(3):283–341, 2013. bit.ly/1NOMJ6l, arXiv:1306.3888.

[13] J. G. Wolff. Application of the SP theory of intelligence to the understanding of natural vision and the development of computer vision. *SpringerPlus*, 3(1):552–570, 2014. bit.ly/2oIpZB6, arXiv:1303.2071.

[14] J. G. Wolff. Autonomous robots and the SP theory of intelligence. *IEEE Access*, 2:1629–1651, 2014. bit.ly/18DxU5K, arXiv:1409.8027.

[15] J. G. Wolff. Big data and the SP theory of intelligence. *IEEE Access*, 2:301–315, 2014. bit.ly/2qfSR3G, arXiv:1306.3890. This paper, with minor revisions, is reproduced in Fei Hu (Ed.), *Big Data: Storage, Sharing, and Security (3S)*, Taylor & Francis LLC, CRC Press, 2016, pp. 143–170.

[16] J. G. Wolff. The SP theory of intelligence: benefits and applications. *Information*, 5(1):1–27, 2014. bit.ly/1FRYwew, arXiv:1307.0845.

[17] J. G. Wolff. Commonsense reasoning, commonsense knowledge, and the SP theory of intelligence. Technical report, CognitionResearch.org, 2016. bit.ly/2eBoE9E, arXiv:1609.07772.

[18] J. G. Wolff. The SP theory of intelligence: its distinctive features and advantages. *IEEE Access*, 4:216–246, 2016. bit.ly/2qgq5QF, arXiv:1508.04087.

[19] J. G. Wolff. Evidence for information compression via the matching and unification of patterns in the workings of brains and nervous systems. Technical report, CognitionResearch.org, 2017. Submitted for publication. bit.ly/2ruLnrV, viXra:1707.0161v2, hal-01624595, v1.

[20] J. G. Wolff. On the "mysterious" effectiveness of mathematics in science. Technical report, CognitionResearch.org, 2017. Submitted for publication. bit.ly/2otrHD0, viXra:1706.0004, hal-01534622.

[21] J. G. Wolff. Strengths and potential of the sp theory of intelligence in general, human-like artificial intelligence. Technical report, CognitionResearch.org, 2017. bit.ly/2z3rM8b, viXra:1711.0292, HAL.