Developing Reproducible and Comprehensible Computational Models*

Peter C. R. Lane
Department of Computer Science, University of Hertfordshire,
Hatfield Campus, College Lane, HATFIELD AL10 9AB, UK
Peter.Lane@bcs.org.uk

Fernand Gobet
ESRC Centre for Research in Development, Instruction and Training,
School of Psychology, University of Nottingham,
University Park, NOTTINGHAM NG7 2RD, UK
Fernand.Gobet@nottingham.ac.uk

November 19, 2002

Abstract

Quantitative predictions for complex scientific theories are often obtained by running simulations on computational models. In order for a theory to meet with wide-spread acceptance, it is important that the model be reproducible and comprehensible by independent researchers. However, the complexity of computational models can make the task of replication all but impossible. Previous authors have suggested that computer models should be developed using high-level specification languages or large amounts of documentation. We argue that neither suggestion is sufficient, as each deals with the prescriptive definition of the model, and does not aid in generalising the use of the model to new contexts. Instead, we argue that a computational model should be released as three components: (a) a well-documented implementation; (b) a set of tests illustrating each of the key processes within the model; and (c) a set of canonical results, for reproducing the model's predictions in important experiments. The included tests and experiments would provide the concrete exemplars required for easier comprehension of the model, as well as a confirmation that independent implementations and later versions reproduce the theory's canonical results.

1 Introduction

Computational modelling is a popular scientific methodology which relies on computer programs to simulate a proposed theory of some physical or psychological phenomenon. In a number of disciplines, such as cognitive science, computational models have been developed which summarise

*Accepted for publication as a Reseach Note by *Artificial Intelligence*, AI 1042RN.

findings from a large number of separate experimental settings – these are the 'unified theories of cognition' [23]. Developing unified theories, and implementing them in computer programs, is a useful methodology, as the theories are stated in their most general form, and the implementations permit quantitative predictions to be made. However, probing a unified theory expressed in the form of a computer program presents a number of problems, not only in understanding the nature of the theory, but also in attempting to reproduce its predictions. In this article, we propose a methodology by which the implementation of the theory in the computer program may be made more transparent, enhancing the opportunity for a later modeller to understand and reproduce the theory's central findings. (We focus on issues related to models of cognitive theories, but the general methodology may be generalised to all disciplines where computer models are used to generate quantitative predictions from a theory.)

Our methodology is targetted, in the first instance, at the difficulty in reproducing computational models of theories. The need to reproduce a model is driven by good scientific practice: experimental findings should be verifiable by independent researchers. The process of verification has two parts: replicating the experimental data by re-running the experiment, and reproducing the process by which the theory's predictions are derived. Reproducing a prediction in domains where theories are expressed mathematically, such as astrophysics, requires a researcher to reproduce the mathematical manipulations which led to the prediction. Similarly, in domains where predictions are derived by executing computer programs, such as cognitive science, we should expect that a researcher reproduce the computational processes which led to a particular behavioural prediction. Unfortunately, in almost all cases, this level of reproduction is impossible. An important reason for this failure is that the complexities of the implementation force published descriptions to be at a high level, and this level is inappropriate for an independent programmer aiming to reproduce the original model (for instance, consider Newell's description of the Soar cognitive model [23]).

Secondly, let us consider how accessible the model is to a researcher wanting simply to understand the nature of the processes which led to a particular prediction being made. In general, it is easier to comprehend relevant concrete examples than abstract descriptions [12]. However, a complex scientific theory will evidently require a large number of concrete examples before even its basic processes would be exemplified; even a dozen examples would be prohibitively costly in the printed literature. But more, we would want worked examples of how the model can be used to generate quantitative predictions in domains of interest, to facilitate the development of new sets of predictions.

Criticisms of computational models along these lines have several precedents in the literature [2, 3, 24]. A typical suggestion for addressing them relies on high-level languages or complete environments for specifying or developing cognitive theories. However, in spite of making the central cognitive mechanisms clearer, these methodologies do not totally remove the difficulties mentioned above: namely, encouraging a model's reproduction and providing adequate examples for easy comprehension. In particular, high-level languages do not assist a programmer who constructs an independent implementation of the cognitive model (or indeed, a new version of an existing model) in demonstrating that the new implementation is faithful to the old. Such a requirement demands that an additional element be included in the published description of the cognitive model, and for this we turn to another discipline: software engineering.

Problems of reliable reproduction are not unique to the computational modelling community. Indeed, they are prevalent in any reasonably-sized software project. Popular systems, such as

Perl [25] or TeX [15], are available in source code format, for recompiling into the user's own computer environment. In addition to their basic specification, each comes with a comprehensive set of *self tests*, which are used to confirm the software's behaviour after it has been compiled. Only if all the tests are passed is the software judged to be a faithful copy of the original. The use of tests for the development and release of a sizable software project is an important element in contemporary software engineering [1, 8]. Tests have the advantage of providing an unambiguous confirmation that the behaviour of the compiled software is as described in the test suite; if the program code has also been developed to match the prescribed specifications, then a user can be assured that their version is a true copy of the original design.

We propose that the testing methodology employed by software engineers should become a routine part of the release of computational models. We suggest that a computational model be released in three components: (a) a well-documented implementation; (b) a set of tests, to illustrate and confirm each of the key processes within the model; and (c) a set of canonical results for reproducing the model's predictions in important experiments. The tests and included results help to remove the above-mentioned problems by providing a *descriptive* definition of the model's behaviour. First, the tests act as a standard against which new implementations, variants, or versions can be tested, encouraging extensions and partial or complete reproduction of the model. And second, a descriptive definition of the model is inherently easier to comprehend than the *prescriptive* definitions based around specifications or high-level program code. The following sections elaborate on these elements.

2 Prescriptive and Descriptive Model Definitions

Writing computer programs of any size is a complex process. Software engineers employ several methodologies for improving the correctness and reliability of their programs, but two are of particular interest in the context of this article: providing a *prescriptive* definition of a program in some higher-order specification language, and providing a *descriptive* definition in the form of a specific set of tests which the program must pass to demonstrate that it meets its specification. We describe each in turn.

2.1 Specifications to prescribe program behaviour

Demonstrating the correctness of a sizable piece of software is a complex process, as no period of correct operation is sufficient to prove correctness in future operation. This realisation led McCarthy [19], Dijkstra [6] and others to develop formal specification techniques, which describe *what* a computer program should be doing by defining its class of desired behaviours. An implementation of *how* to perform this behaviour, in the form of a computer program, can then be proved to meet the specification, and so is likely to perform correctly in all future situations within its prescribed class. As an illustration, consider specifying a routine to sort a set of numbers. The specification should state that a given set of numbers is transformed into an ordered sequence of the same numbers. Formal specification languages such as Z (e.g. Lightfoot [18]) provide a standard within which such specifications can be constructed; the sort example appears in Figure 1. Z specifications are useful when publishing definitions of software, such as Soar [20].

$$a? : [N]$$

$$b! : [N]$$

$$\#a? = \#b!$$

$$\forall i : 1..\#a?, a?(i) \in b!$$

$$\forall i, j : 1..\#b!, i < j \Rightarrow b!(i) < b!(j)$$

Figure 1: Z specification of a sort program. The specification states that, given a list a? of integers, it will output a list b! of integers such that: a? and b! are equal sizes, every member of a? is in b! (i.e. b! contains the same elements as a?), and that b! is ordered.

More recently, methodologies have been developed to semi-automate the process of creating program code from specifications. For example, Morgan [21] demonstrates how a formal specification can be converted into a working program by following a sequence of derivations, with each step formally justified by an appropriate reduction rule. Alternatively, it is possible to use a high-level programming language or graphical environment to directly produce working programs from their specifications. This approach has been employed for cognitive modelling with the Sceptic language [2], and the COGENT system [4]. One of the benefits of using a high-level system is that the model is easier to accept as being a correct implementation of a theory, because the basic cognitive processes are provided as primitives within the language.

However, specification languages, proof techniques or supporting software environments still do not eradicate the problems in creating successful reproductions of existing computational models. Working in a different computer environment or with a different programming language can further complicate the already arduous process of checking that a program meets its intended specification. What is required is a performance-related confirmation that a new implementation reproduces the older one. A methodology for achieving just this relies on automated software testing.

2.2 Tests to describe specified program behaviour

An important element in the development of robust and extendable software is the addition of tests to program code. Tests provide a description of how the program is intended to function. For example, the sort program in Figure 1 may be tested by checking that it operates correctly in a number of situations. Figure 2 provides an implementation of Quicksort in Haskell [13],[1] and a set of tests to check that it functions correctly; invoking *runTests* will generate a sequence of dots if the function is correct, or a message if there is a fault. In general, there are several types of tests in use, which differ in their intent, their scope, and their maintenance. For convenience, we group the tests into three categories: system testing, unit testing, and behaviour testing.

System testing: Prior to the final release of a computer program, it is typically subjected to a wide variety of tests [14]. These include: *stress tests*, to ensure the program can handle large

[1] We provide examples in Haskell, rather than pseudo-code, because they are executable and concise.

```
qsort :: (Ord a) ⇒ [a] → [a]
qsort [] = []
qsort (x:xs) = (qsort ys) ++ [x] ++ (qsort zs)
    where ys = [ y | y ← xs, y < x ]
          zs = [ z | z ← xs, x ≤ z ]

runTests = performTests [ (qsort ([] :: [Int]) = [], "qsort"),
                          (qsort [1] = [1], "qsort"),
                          (qsort [3,2,1] = [1,2,3], "qsort"),
                          (qsort ['q','d','c','a'] = ['a','c','d','q'], "qsort") ]

performTests = putStr.concat.(map doTest)
  where doTest (bool, str)
          | bool = "."
          | otherwise = "\n" ++ str ++ "\n"
```

Figure 2: Haskell implementation of Quicksort and some tests of typical behaviour and boundary conditions. Quicksort works by pivoting around an element in the list, recursively sorting those elements less than the pivot and those larger than the pivot, and then concatenating the elements into a whole.

input sets; *regression tests*, to ensure the program works the same as the previous version; and *coverage tests*, to ensure every logical path in the program has been checked. System testing must be comprehensive, to locate as many errors as possible in the program, and is time-consuming; for large projects this level of testing is typically handled by a dedicated team. These are the kinds of tests released for confirming that the program has been correctly installed, e.g. Perl [25] and TeX [16].

Unit testing: Good programmers recognise that writing program code is hard, and that they will have to rewrite their code to enhance clarity and extendability; this rewriting process is called *refactoring* [8], and requires the programmer to create a set of function-level, or unit, tests whilst producing new code. Rewriting code should not affect the program's behaviour, and the unit tests are intended to demonstrate correct behaviour both before and after rewriting. Unit tests are mostly restricted to low-level implementational details, and are removed or maintained by the programmer as required by the current state of the code.

Behaviour testing: When a program is complete, its behaviour must be demonstrated to the customer (who may simply be an end-user). This requires a set of behavioural tests, which the customer may have specified and can easily confirm; these kinds of tests are known as *acceptance tests* [1]. Behavioural tests are intended to be understandable by the customer, who probably is not a programmer, and certainly is not interested in functional details of the implementation. Behavioural tests deal with global properties of the system, and are maintained as part of the current specification of the system. It should be clear that a second team of programmers could meet the same specification requirements and set of behavioural tests, but with an entirely different piece of software (e.g. the sort routine tests in Figure 2

could be passed by an insertion sort algorithm, instead of Quicksort).

These three types of tests address different aspects of a program's correctness. System testing is devoted to complete coverage of every aspect of how the program functions, and unit testing is devoted to relatively low-level details of how particular elements are implemented; most importantly, these tests are intended to satisfy the programmer (and manager) that the program is doing what the programmer thinks it is doing. Behavioural tests instead aim to convince the customer to accept the claim of the programmers that the program does what they expect. In the context of this article, *the behavioural tests are designed to satisfy the user that the computational model does indeed implement the intended cognitive theory.* We propose that behavioural tests be made an integral part of the release of a computational model. As behavioural tests are basically a set of examples of how to apply the theory, they can be used in learning how the model works, or as a target set of behaviours when developing an independent implementation of the model. The behavioural tests do not replace the need for a complete specification of the program, nor the need for a particular implementation's set of system and unit tests.

What exactly constitutes a 'behavioural test'? The published descriptions of cognitive models typically contain two types of descriptions. The first is a relatively precise definition of the main processes, such as 'if this type of data is seen, then this particular piece of information will be learnt'. The second is a description of how the model can be used to predict a particular phenomenon and how well it matches a given set of experimental data. We address these two types of descriptions by dividing behavioural tests into two groups:

1. *Testing the cognitive processes*: The basic cognitive processes of the model, as described in the theory, must be matched with concrete examples, to verify the operation of the model. Successfully passing these tests should be sufficient to ensure the model is a correct implementation of the theory.

2. *Testing the canonical results*: The model's predictions in important experiments are provided, especially those which support the validity of the cognitive theory. Successfully passing these tests will confirm that the implementation makes the same predictions as previously.

The two sets of behavioural tests complement each other in confirming that the model implements the specified theory. The first set of tests confirms that the cognitive processes identified within the theory have been implemented in the anticipated manner. The second set of tests confirms that these processes, along with other factors within the implementation, combine to make specific quantitative predictions. A useful consequence of this complementarity is that underspecified elements of the theory, which have not been described as key cognitive processes, can be challenged by later researchers. Alternatives may then be explored, and the provision of the canonical results enables these alternatives to be tested against the original supporting empirical evidence. In this manner, the computational implementation of a theory can be developed and refined, whilst retaining a link with all the scientific evidence cited in its support.

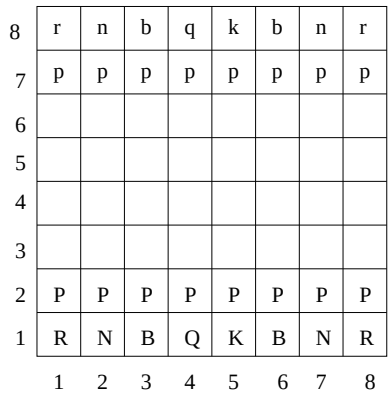

Figure 3: Illustration of chess board pattern.

3 Case Study : Discrimination Networks

As an illustration of our proposed methodology, we describe an implementation of a discrimination-network learning algorithm. Our example is based on the learning mechanisms in the EPAM [7] and CHREST – Chunk Hierarchy and REtrieval STructures – computational models of expert perception [5, 9, 11]. The examples are implemented in Haskell [13] and may be executed in the Hugs98 interpreter (downloadable from `http://www.haskell.org`). The examples are provided to illustrate the distinction between a high-level prescription (program) of the model (which may be difficult for the non-programmer to understand) and the kinds of tests which provide a descriptive definition. We then indicate a likely set of canonical empirical results for this class of cognitive model.

3.1 An implementation of discrimination networks

The learning algorithm constructs a discrimination network consisting of nodes holding chunks of information. There are two important learning operations: *discrimination*, which adds a new node to the network, and *familiarisation*, which adds information to an existing node.

As an example domain, we consider learning about, and recalling, chess positions. We represent the board as an 8x8 grid, with pieces placed at various grid locations (refer to Figure 3). The following data structure defines a PIECE data type with fields for the x and y coordinates of the square (which are integers) and the piece (which is a character):

```
data PIECE = PIECE {xCoord, yCoord :: INT, piece :: CHAR}
                   deriving (EQ, SHOW)
```

The **deriving** (EQ, SHOW) statement instructs Haskell to provide default mechanisms for testing the equality of two such objects, and for displaying their values. Board positions are lists of PIECE objects, for example:

```
[PIECE 1 1 'R', PIECE 1 2 'P', PIECE 7 8 'n']
```

The discrimination network is a tree data structure, with nodes and child test links. Each node has an *image*, which is a list of PIECE objects, and a list of *children*, representing the possible links

from this node. The links each have a test (which must be some PIECE) and a child-node. These structures are defined as follows:

```
data NODE = NODE {image :: [PIECE], children :: [LINK]}
                 deriving (EQ, SHOW)
data LINK = LINK {test :: PIECE, node :: NODE}
                 deriving (EQ, SHOW)
```

These structures are sufficient to describe the trees in Figure 4, as in the definition of $tree_1$, $tree_2$ and $tree_3$ in *runTests* below.

Learning within the model is defined by the following function. The function learn combines the sorting process with functions for altering the network where appropriate; a training example is sorted through the network by following those links with matching tests. Taking a node and a training example as arguments, learn applies one of three cases. First, if no further sorting can occur, and the image matches the example, then familiarisation occurs. Second, if no further sorting can occur and the image does not match the example, then discrimination occurs. Finally, if a link exists for further sorting, it is taken. These cases specify the cognitive processes important in learning. The functions in the **where** clause specify how the selection of these processes is implemented. For instance, *validLinks* is the list of links for which the test of the link is an element of the training example (the syntax translates this statement directly).

```
learn (NODE image links) example
        | (null validLinks) ∧ (image `matches` example)
                        = familiarise (NODE image links) example
        | (null validLinks)
                        = discriminate (NODE image links) example
        | otherwise = NODE image newLinks
        where validLinks = [link | link ← links, (test link) `elem` example]
              takenLink = head validLinks
              otherLinks = [link | link ← links, link ≠ takenLink]
              newLinks = (LINK (test takenLink) (learn (node takenLink) example))
                                : otherLinks
              xs `matches` ys = (take (length xs) ys) = xs
```

The familiarise learning mechanism is only applied if the example matches the node's image. Learning adds a new element of the example to the node's image, unless the two are already equal. The definition of *nextItem* refers to the next PIECE in the training example after those PIECES which match the image.

```
familiarise (NODE image links) example
                        | image ≠ example = NODE (image ++ [nextItem]) links
                        | otherwise = NODE image links
                        where nextItem = example !! (length image)
```

The discriminate learning mechanism is applied if the example mismatches the node's image. Learning adds a new link from the given node, with the test on the link indicating the mismatching piece. The definition of *newTest* selects the first mismatching piece between the image and example.

```
discriminate (NODE image links) example = NODE image (newLink : links)
```

```
    where newTest = snd (head (dropWhile (λ(x,y) → x = y)
                                         (zip image example)))
          newLink = LINK newTest (NODE [] [])
```

The preceding description and implementation of a simple learning algorithm is fairly typical. It does not have a formal specification, nor are the motivations for the implementation particularly transparent; the descriptions could be retained in the source code, using the *literate programming* [17] capability of Haskell, in which program code is merely one part of a readable document describing the program's behaviour. However, our example is useable, and could form part of a larger model to provide detailed quantitative predictions of the learning behaviour of humans. The question then is: If we are to release the algorithm as a component of a larger theory, what additional material do we require? As argued in Section 2.2, we propose that the implementation be released with two levels of behavioural tests: cognitive processes, to verify the operation of the model, and canonical results, to verify the predictions made by the model. We provide examples of these two kinds of tests.

3.2 Testing the cognitive processes

Behavioural tests for cognitive processes focus on the key mechanisms in the theory which the program claims to implement. In this case, the underlying theory relates to learning, and contains the two important processes of discrimination and familiarisation. Each of these requires a descriptive test, as illustrated in Figure 4. Code to represent the trees and perform the tests follows:

```
runTests = performTests [ ((tree'1 = tree2), "Familiarise"),
                          ((tree'2 = tree3), "Discriminate") ]
    where tree1 = NODE [PIECE 7 8 'n']
                       [LINK (PIECE 1 2 'P') (NODE [PIECE 1 2 'P'] [])]
          tree2 = NODE [PIECE 7 8 'n']
                       [LINK (PIECE 1 2 'P')
                             (NODE [PIECE 1 2 'P', PIECE 1 1 'R'] [])]
          tree3 = NODE [PIECE 7 8 'n']
                       [LINK (PIECE 1 2 'P')
                             (NODE [PIECE 1 2 'P', PIECE 1 1 'R']
                                   [LINK (PIECE 1 1 'Q')
                                         (NODE [] [])])]
          tree'1 = learn tree1 [PIECE 1 2 'P', PIECE 1 1 'R']
          tree'2 = learn tree2 [PIECE 1 2 'P', PIECE 1 1 'Q']
```

Note that each test simply performs a learning operation, producing the new trees $tree'_1$ and $tree'_2$, and then checks each new tree against its expected form, $tree_2$ and $tree_3$ respectively. These tests provide an example of each cognitive process, and so describe the model's expected behaviour; their execution checks that the model is performing as promised. A practical point is the ease with which the tests may be transformed into alternative languages, if the model should be reimplemented; hence the description can be used to confirm the consistency of two implementations

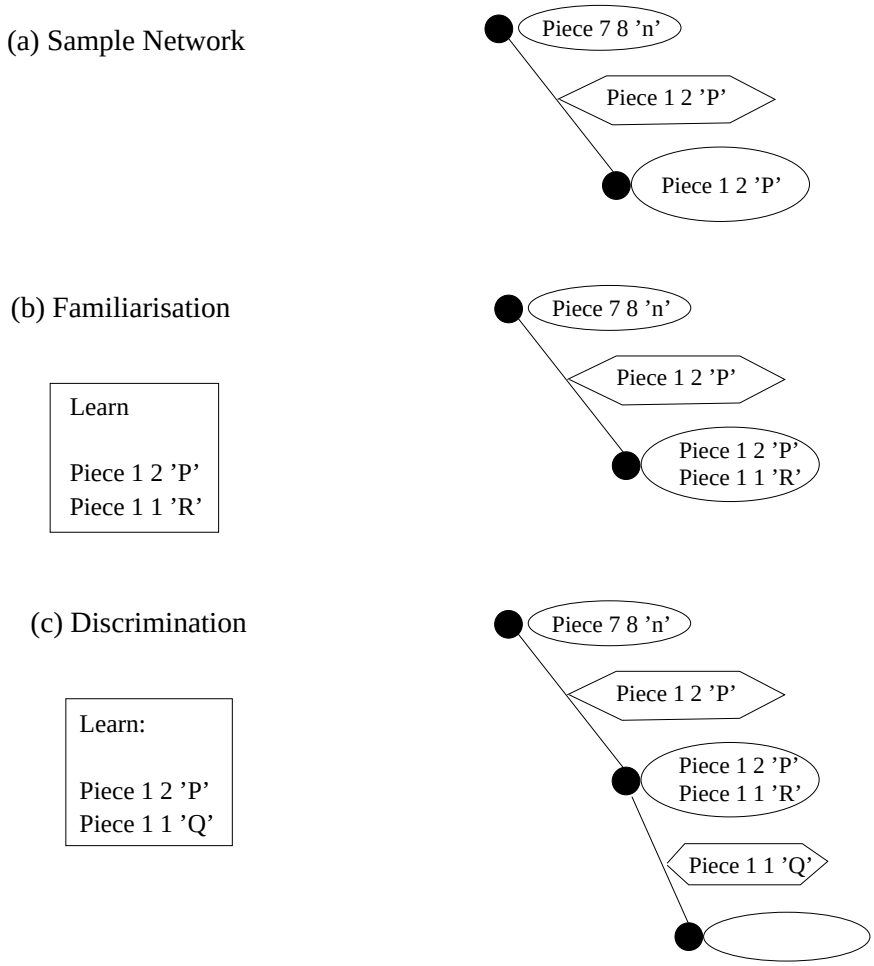

Figure 4: Description of the discrimination-network learning mechanisms showing trees and the results of learning. Each tree indicates the nodes (black circles) and test links (straight lines); the top of the tree is the root node, and patterns are sorted down towards the leaf nodes by matching the tests. Text within hexagons indicates the tests, and text within ellipses the information in node images. (a) represents part of a discrimination network. On being presented with the pattern to the left of (b), familiarisation occurs, leading to the network shown in (b). On further being presented with the pattern to the left of (c), discrimination occurs, leading to the network shown in (c).

quickly and automatically. (The tests may also be placed within a graphical execution environment, to provide a 'guided tour' of the key processes within the model.)

The algorithm, as described above, makes a number of assumptions which are underspecified in the theory. For instance, the default definition of equality of two lists of pieces means that different orderings of the same pieces would not be considered equal. If this is later discovered to be important, either by the original developer or by an independent researcher, then the definition of equality must be refined and made part of the theory's specification; further behavioural tests should then be provided to enforce this decision in later implementations. We mention this to indicate how later work will uncover implementational vs. theoretical decisions, and how behavioural tests enable us to document and enforce a decision on the nature of the theory. Whichever definition of equality we use, we would still expect the learning operations described above to function as described, so these tests would remain permanently in the test suite.

3.3 Testing the canonical results

The third component of an implemented computational model must be a set of canonical results, or its important experimental predictions, along with a description of how to obtain the predictions by running the provided version of the model. A number of important issues need to be specified in these tests, including issues on how the computational model is validated experimentally. For instance, details of the experimental design, how to compute the degree of fit, how much variation will be tolerated in the model's predictions, etc.

For the sake of concreteness, we now assume that the above discrimination-network learning algorithm is a component within a complete implementation of the CHREST [9] computational model. To determine what should form part of CHREST's central canon of results, we must ask two questions: What kinds of experiments have been performed to validate CHREST's claims to be a model of perceptual learning? What are the key results by which we would want to judge all future versions as true to the current definition of CHREST?

A key set of results for the CHREST model is the relative performance of novice and expert chess players in the recall of game and random chess positions [11]. These results can be arranged as executable tests by defining and providing the following:

1. the performance of human participants in the experiment;
2. the set of chess positions used in training the model, and the criteria for their selection;
3. the protocol for running the CHREST model and assessing its performance; and
4. the procedure for comparing the performance of the model and the participants.

Explicitly providing each of these four elements supports the researcher who wants to replicate and explore the model's results. For instance, a new set of experiments may be performed to gather data on the performance of human participants, or the model may be trained with a different set of input data. Finally, details are provided of how the model was used to generate comparative data and the statistical method used to make the comparison.

Our proposed methodology is designed to tackle the related problems of reproducing and understanding the computational logic behind predictions of a scientific theory implemented as a

computational model. As noted in the introduction, models in cognitive science are sometimes described as 'unified theories', in the sense that the models summarise how a common set of cognitive processes can be used to simulate human behaviour in multiple experiments. An important consequence of our proposal is in providing a framework within which scientists can confirm that their own view of a cognitive theory agrees with that of previous researchers. This confirmation is achieved by running the model against a given set of behavioural tests and canonical results. The larger the set of canonical results associated with a theory, and the greater the spread of application areas covered by these results, the greater the chance that the underlying theory is a true reflection of the cognitive processes within the human mind. Similar ideas have been raised by other authors, at the level of theory formation [10] and experimental design [22].

4 Conclusion

We have argued in this article that software engineering can provide a methodology for developing more readily reproducible and comprehensible computational models. The basis of our proposal is that a model should be released as three elements: (a) a core implementation, (b) a set of tests indicating the key cognitive processes, and (c) a set of canonical results, which define the key empirical phenomena supporting the model's validity. Our proposal is compatible with those of earlier writers, such as Cooper *et al.* [2], who have called for the development of high-level specification languages for developing cognitive theories. However, our proposal goes further, by seeing computational modelling as part of the process of developing quantitative predictions from a theory. Including concrete tests in the published form of the model will support later researchers in understanding and possibly reproducing the computational processes by which predictions have been made, thus enhancing the scientific validity of the theories being modelled.

Acknowledgements

The authors would like to thank two anonymous reviewers and Daniel Freudenthal for useful comments on an earlier draft of this article.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley, 1999.

[2] R. Cooper, J. Fox, J. Farringdon, and T. Shallice. A systematic methodology for cognitive modelling. *Artificial Intelligence*, 85:3–44, 1996.

[3] R. Cooper and T. Shallice. Soar and the case for unified theories of cognition. *Cognition*, 55:115–49, 1995.

[4] R. Cooper, P. Yule, J. Fox, and D. Sutton. COGENT: An environment for the development of cognitive models. In U. Schmid, J. F. Krems, and F. Wysotzki, editors, *A Cognitive Science*

Approach to Reasoning, Learning and Memory. Lengerich, Germany: Pabst Science Publisher, 1998.

[5] A. D. de Groot and F. Gobet. *Perception and Memory in Chess: Heuristics of the Professional Eye*. Assen: Van Gorcum, 1996.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., 1965.

[7] E. A. Feigenbaum and H. A. Simon. EPAM-like models of recognition and learning. *Cognitive Science*, 8:305–336, 1984.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

[9] F. Gobet, P. C. R. Lane, S. J. Croker, P. C-H. Cheng, G. Jones, I. Oliver, and J. M. Pine. Chunking mechanisms in human learning. *Trends in Cognitive Science*, 5:236–243, 2001.

[10] F. Gobet and F. Ritter. Individual data analysis and unified theories of cognition: A methodological proposal. In N. Taatgen and J. Aasman, editors, *Proceedings of the Third International Conference on Cognitive Modelling*. Veenendaal, The Netherlands: Universal Press, 2000.

[11] F. Gobet and H. A. Simon. Five seconds or sixty? Presentation time in expert memory. *Cognitive Science*, 24:651–82, 2000.

[12] K. Gravemeijer. Mediating between concrete and abstract. In T. Nunes and P. Bryant, editors, *Learning and teaching mathematics: An international perspective*. Hove, UK: Psychology Press, 1997.

[13] S. Peyton Jones. Haskell 98: A non-strict, purely functional language. Available from http://www.haskell.org/, 2001.

[14] B. W. Kernighan and R. Pike. *The Practice of Programming*. Reading, MA: Addision-Wesley, 1999.

[15] D. E. Knuth. *The TeXbook*. Reading, MA: Addison-Wesley, 1984.

[16] D. E. Knuth. The errors of TeX. *Software – Practice and Experience*, 19:607–85, 1989.

[17] D. E. Knuth. *Literate Programming*. Center for the Study of Language and Information: Stanford, California, 1992.

[18] D. Lightfoot. *Formal Specification Using Z*. Basingstoke, UK: Palgrave, 2001.

[19] J. McCarthy. A mathematical theory of computation. In *Proc. Western Joint Comp. Conf. Los Angeles, May 1961, and Proc. IFIP Congress, 1961*, pages 225–38. Amsterdam, Holland: North Holland Publishing Co., 1963.

[20] B. Milnes. The specification of the Soar cognitive architecture using Z. Technical report: CMU-CS-92-169, Carnegie-Mellon University, 1992.

[21] C. Morgan. *Programming from Specifications*. Prentice-Hall, Inc., 1998.

[22] A. Newell. *You can't play 20 questions with nature and win: Projective comments on the papers of this symposium*. New York: Academic Press, 1973.

[23] A. Newell. *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press, 1990.

[24] W. Reitman. *Cognition and Thought*. New York: Wiley & Sons, 1965.

[25] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly, 1996.