Journal of Experimental & Theoretical Artificial Intelligence
Vol. 00, No. 00, Month-Month 200x, 1–31

A Theory-Driven Testing Methodology for Developing Scientific Software

Peter C. R. Lane
School of Computer Science, University of Hertfordshire,
College Lane, Hatfield AL10 9AB, Hertfordshire, UK

Fernand Gobet School of Social Sciences, Brunel University,
Uxbridge UB8 3PH, Middlesex, UK

(*Received 00 Month 200x; In final form 00 Month 200x*)

Computer implementations of theoretical concepts play an ever-increasing role in the development and application of scientific ideas. As the scale of such implementations increases from relatively small models and empirical setups to overarching frameworks from which many kinds of results may be obtained, it is important to consider the methodology by which these implementations are developed. Using cognitive architectures as an example, we discuss the relation between an implementation of an architecture and its underlying theory, a relation between a computer program and its description. We argue for the use of an agile development methodology, based around a three-layer scientific test harness and continuous refactoring, as most suitable for developing scientific software. The ideas are illustrated with extended examples of implementing unified theories of human learning, taken from the chunking and template theories.

1 Introduction

Implementing a scientific theory as a computer program leads to a computational view of scientific knowledge and natural processes; the assumption is that computer programs are a suitable way to describe the processes within the theory. It is generally accepted that computational models offer a number of valuable features, including: (a) clear and rigorous specification of the mechanisms and parameters underpinning a theory; (b) derivation of testable predictions; (c) possibility of simulating complex behaviour, both qualitatively and quantitatively, irrespective of the number of variables involved; (d) possibility of systematically manipulating some variables to explore a model, which enables a better understanding of the (often non-linear) dynamics of a system; (e) help in making sense of rich and dense datasets; and (f) provision of explanations that are 'sufficient', in the sense that they can produce the behaviour under study (Gobet and Waters, 2003; McLeod et al., 1998; Newell and Simon, 1972; Pew and Mavor, 1998; Simon and Gobet, 2000). Computational models within one area, such as memory experiments in cognitive science, typically cluster around certain core properties; we can speak of an *architecture* as providing a blueprint or framework from which different kinds of *models* can be developed. For example, a *cognitive architecture* such as Soar (Laird et al., 1987) is used as an implementation framework from which different kinds of models or intelligent agents may be constructed.

Within cognitive science, Newell (1990) was perhaps the first to propose that the different models of various experimental phenomena should be brought together and unified into a single cognitive architecture. The idea was that a single architecture would support the development of models in a range of domains, and the models would thus draw on the accumulated understanding of models across a broad range of cognitive tasks. Newell estimated that a unified theory of human cognition would contain approximately 1,000–10,000 regularities of immediate behaviour. Newell also proposed that attempts to construct this unified system should be done gradually, beginning with some kinds of functionality, and then progressing to others. Newell (1990) made a case for the Soar architecture, which begins from problem solving behaviour. We thus have a picture of a gradual accretion of understanding, as we progress through models

*Corresponding author. Email: peter.lane@bcs.org.uk capturing different regularities and areas of human behaviour, always developing the models within the context of a single cognitive architecture.

Newell did not preclude there being multiple candidate architectures, only that one architecture should attempt to cover as many phenomena as possible. There are now many competing architectures for a cognitive scientist to choose between, partly differentiated by the kinds of behaviour they began to model, for example: ACT-R (Anderson et al., 2004), CHREST (Gobet, 1998), Clarion (Sun et al., 2001), EPAM (Feigenbaum and Simon, 1984), ICARUS (Langley and Choi, 2006), and Soar (Newell, 1990), to name just a few; more complete lists can be found in Langley, Laird, and Rogers (2009) and Samsonovich (2010). As these architectures have developed, a number of issues have arisen in the nature of the scientific process, including: the role of falsification, the presence of task knowledge, barriers to interpretability, and control of multiple versions. Attempts have been made to justify the development of cognitive architectures using variations on standard scientific methodology (e.g. Cooper, 2007; Newell, 1990).

One view, adopted by the creators of the first models of human learning (Feigenbaum and Simon, 1962, 1984), was developed within EPAM (Elementary Perceiver and Memoriser): 'Our theory is an information processing theory, and its precise description is given in the language of a digital computer' (Feigenbaum, 1959). The fact that a cognitive architecture has been implemented as a set of program code means there is a correspondence between the theory and the program implementing it. Recognising this correspondence leads to the idea that theory development is related to program (or software) development, and perhaps software development is a field scientists may use to assist in their development of such architectures.

The argument made in this article is that the implementation of the software supporting a collection of models is important to the scientific theory being captured. Also, the changing nature of scientific theories, and the demands to always construct models in new domains, means the development of the implementation must support the constant changes at the theory level. There are several, relatively new software-development methodologies which emphasise the changing nature of program specifications, user requirements and design decisions. We propose that a form of *agile programming*, emphasising a test-first cycle of code development (Beck, 2003) and extensive refactoring (Fowler, 1999), provides strong scientific gains. Test-first development means that each piece of code within a program has a test, an executable confirmation that the code does what it is intended to do. Refactoring is the process of rewriting code without changing its behaviour in order to improve its design. Both elements are clearly scientifically relevant: tests provide working examples of described behaviour, and rewriting improves the presentation of the theory.

Although our suggestions mostly stem from cognitive science, we further suggest that the goal when implementing several scientific models within one domain should be a coherent and comprehensive framework or architecture, which we term a *scientific architecture*, by analogy with a cognitive architecture. This architecture then provides a process-based description of the implemented scientific theory. Although agile techniques have been used before in scientific programming, (e.g. Pitt-Francis et al., 2008), we believe the nature of the coupling between the software and theory has not been explored. One of the arguments here is that all tests are not equal, and all parts of the software are not equal, when measured against their contribution to the implemented theory; a similar view was raised by Cooper and Shallice (1995). Our main technical innovation is that a three-layer test methodology (Lane and Gobet, 2003) should be used, to distinguish the scientific importance of different tests. During development of the architecture, the necessary changes to these tests can use the test layers to identify any theoretical challenges to the implemented theory.

We continue in Section 2 by explaining some terminology regarding cognitive theories, architectures and models, and exploring some of the motivations for and implications of architecture development. In Section 3 we explain the correspondence between scientific theories as architecture specifications, and hence how scientific theories are similar to program specifications. We explore implications of this perspective, and how considering program specification and program development can give useful gains. Section 4 takes two important techniques in agile software development, describes them, and explains their benefits for developing scientific programs. Section 5 considers tools which can benefit the architecture developer, and introduces an implementation of the scientific-testing framework, which supports our methodology. Section 6 provides a case study in architecture development, and serves as an illustration of the proposed

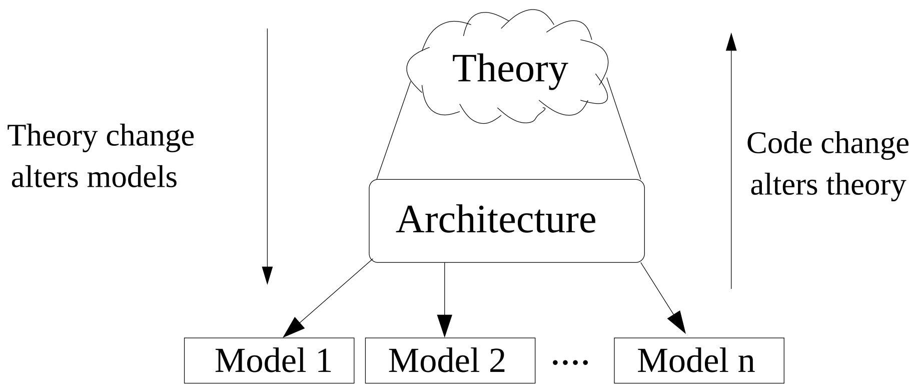

Figure 1. The relation between a theory, architecture and model. The arrows at the side show how change can propagate both from the theory down to the implementation, and from the implementation back to the theory.

techniques. Finally, Section 7 considers some further implications of the proposed methodology and test framework. In particular, we broaden our consideration from cognitive science to other sciences, and discuss how our proposal relates to current development practice in the scientific community.

2 Developing Cognitive Architectures and Models

In this section, we use the field of cognitive science to illustrate the terminology and benefits of our proposed methodology. As described in the introduction, and elaborated upon in the discussion, these ideas will transfer to all sciences with a heavy theoretical element behind their computational implementations.

Within cognitive science, there are varying descriptions of the terms 'theory', 'architecture' and 'model'. One purpose of this section is to clarify what we mean by these terms, and describe how individual theories, architectures or models may develop over time. Our definitions may differ slightly from that of other authors, in particular because we identify our *implementation* in computer software with the terms 'architecture' and 'model', preserving more verbal aspects within the term 'theory'. For instance, some authors adopt the view that each task constitutes a separate theory. However, in the spirit of Newell's call for unification (Newell, 1990), we adopt the view that there is a *single* cognitive theory, and this theory is applicable to a range of domains. Fig. 1 summarises this view of the relation between a cognitive theory, its implementing architecture, and the dependent models.

We distinguish a theory from an architecture by defining the architecture as an *implementation*, in computer code, of its theory. The theory is less concrete, constituting an understanding of the architecture, high-level explanations of how the architecture captures certain phenomena, and the know-how to get the architecture to perform certain tasks. Finally, a *model* is a particular application of the architecture to a task; a model is also a running computer program. An example of this is illustrated in Table 1, where we show this separation for CHREST. The CHREST architecture (Gobet and Lane, 2005; Gobet et al., 2001) has produced a number of models in areas of memory and expertise, and the CHREST architecture itself is an implementation of the template theory (Gobet and Simon, 1996, 2000), which is an extension of the chunking theory (Chase and Simon, 1973). A further example would be an architecture like Soar (Laird et al., 1987), which is derived from a theory of solving impasses in problem spaces, and has spawned a large number of models of different phenomena.

We allow the cognitive theory to be broader in many respects than the architecture which implements it. This is partly because the theory may be expressed verbally, and thus not be a clear enough specification for the architecture. And partly because some design decisions in the architecture, such as choice of programming language, will not be made in the theory. The architecture must, at a minimum, implement all the core mechanisms within the theory. The architecture is also likely to include additional elements, not strictly part of the theory, provided as convenience measures for someone working with the architecture, such as graphical displays, or further utility methods for constructing or evaluating models. A further element in the theory would be what Ritter (2004, p. 23) calls 'a set of conventions that are adhered to when creating or programming the model', such as '[learning] how to use an architecture and how not to

Theory	
chunking and template theories	Chase and Simon (1973); Gobet (1998); Gobet et al. (2001)
Architecture	
CHREST code	version 2.1
	version 3.0
	jChrest
Mosaic	
Models	
Ageing	Smith et al. (2007)
Mental imagery	Waters and Gobet (2008)
Perception	de Groot and Gobet (1996)
Physics problem solving	Lane et al. (2000)
Problem-solving	Gobet (1997)
Recall	Gobet and Waters (2003)
Language acquisition	Freudenthal et al. (2007), Jones et al. (2007)

Table 1. CHREST: its component theory, architecture and some sample models.

misuse it', which have to be learned 'not just individually, but as a community': these verbal, best-practice conventions are not implementable, but are part of the community's knowledge of using the theory and implementation.

The architecture by itself already performs a useful role; it acts as a formal specification of the theory. Creating the architecture's code will force all terms in the theory to be clarified, and also make explicit any supporting mechanisms which were not 'visible' at the level of the theory. Once implemented, the architecture allows a user to query the system dynamically, obtaining responses to certain stimuli. These queries act as worked examples, to help understand the theory in concrete terms. However, the value of the cognitive architecture is not restricted to being a formal specification of the theory; the architecture is also a platform for creating models which perform specific tasks.

A model is both less and more than an architecture. It is less in the sense that a model will only perform one out of a wide range of potential behaviours available to the architecture. It is more in the sense that several aspects of a suitably complex task will not be present in the core architecture, and thus need further programming before the architecture can perform the task. At this stage, it is important to note that model development is done in the context of a given architecture; model development does not affect the architecture in any way.

So far, our scientific picture has been fairly linear: a theory is thought up to explain one or more tasks, an implementation of the theory is created, and then a collection of models is developed within the architecture. However, this tends to be just the first iteration in the development of a theory and, having implemented the theory, we find that it must be changed. There is a natural pattern of scientific enquiry which generates demands on the theory, translating into theory development. We can identify sources of change based at a theoretical level, and at a code level.

At a theoretical level, an important motivation for developing the theory is to make the theory support further tasks. This can be directed top-down, by adding new theoretical assumptions, converting these into additions to the architecture, and so enabling the architecture to support new kinds of models. However, this could also be achieved bottom-up: attempting to model specific tasks forces the modeller to create new mechanisms, which may be adopted into the architecture, and finally into the theory. In both cases, the theory and architecture have been extended, and new models can be developed; Fig. 2 summarises this picture of theory development.

At the code level, there are two factors which can lead to theory development. First, we may find that, as the models become increasingly complex, the additional code developed for each model will also increase in size and complexity, although we are also likely to find similarities in the design of several models. Both the complexity and the similarities suggest that modifying the architecture with additional mechanisms would simplify the job of constructing models. Second, we may improve the architecture's code by modifying its implementation, such as its functions, methods or classes; we discuss this process in more detail in

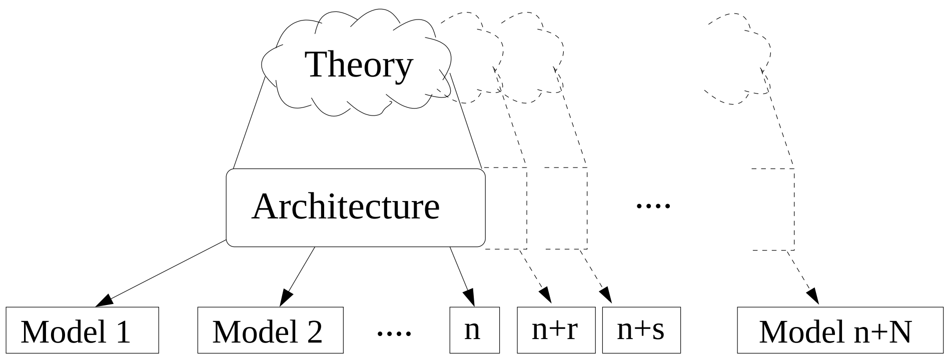

Figure 2. Theory development is driven by the addition of new capabilities in the theory or the implementation, to enable more kinds of models to be created. As a consequence, the theory and architecture grow in size and complexity.

Section 4.2. Modifying the architecture means that the implementation of the theory has changed. We must be sure our theoretical understanding, in the form of our high-level explanations, remains faithful to the revised implementation. An interesting question is whether we allow the changes in the architecture to prompt changes in the theory, a subject we return to later.

In this section we have defined an architecture as an *implementation* of a scientific theory, and argued that theory development and architecture development are intimately linked. Indeed, development of the theory can be driven in two ways: top-down, from the theory itself, or bottom-up, from its architecture implementation. As our theory and architecture develop, we have two scientific issues to consider. First, we must ensure the theory and architecture develop *together*, in the sense that theoretically important constructs and mechanisms are reflected in the architecture, and function in the manner intended by the theory. Second, we want the empirical support for the theory, the set of implemented models, to be preserved; we do not want changes to the architecture to affect previously implemented models, or at least we want to be made aware of any problems that will inevitably arise. We now consider architecture development from the perspective of software development, and show how techniques in software engineering can help manage the development of our theory so that these two scientific considerations are reflected in the development of the scientific architecture.

3 Theories as Program Specifications

In traditional software development, a customer, or user, comes to a developer with a vague description of the functionality that they want the final program to achieve. An important task for the developer is to help the user better understand exactly what functionality is most appropriate; the developer must help the user *specify* the program to be developed. Traditional methods of program development have seen the specification as the start of the process, so that a program is developed to *meet* a specification. However, traditional methods have also been criticised, as projects regularly fail to meet their design, cost or time objectives. A common complaint raised by developers and users is that specifications were inappropriate or, by the end of the project, wrong (Curtis et al., 1988; Lubers et al., 1993).

More recently, these difficulties have been faced by attempting to remove the need for an initial, formal and complete, specification, and by instead supporting a more dynamic relationship between the user's requirements and the developing program. For instance, Naur (1985) argued for a 'Theory Building View' of programming: the program is a formal representation of the programmer's theory of what the user needs. Beck (1999) has designed an entire development methodology, 'eXtreme Programming' (XP), around the co-evolution of the user's understanding of their requirements and the developer's program which implements those requirements. XP consists of twelve related principles, which include ideas such as pair programming, where more than one developer works on the code at the same time; test-driven development, where a test is written for all new functionality before it is added; and having the customer always available, to help with decisions during development. Agile programming is a collective term for related development methodologies, which typically use at least some of XP's principles – we are proposing a form of agile programming using test-driven development and continuous refactoring, as explained in the next section.

```
(defstruct perceptron theta w1 w2)

(defun compute-class (perceptron x1 x2)
  (> (+ (perceptron-theta perceptron)
        (* (perceptron-w1 perceptron) x1)
        (* (perceptron-w2 perceptron) x2))
      0.0))

(defun test-perceptron ()
  (let ((p (make-perceptron :theta 0.5 :w1 0.2 :w2 0.7)))
    (and (compute-class p 0 1)
         (not (compute-class p 0 -1)))))
```

Figure 3. Example of a test: the test code constructs a perceptron with a known set of parameters, and then checks that it correctly classifies two data instances.

The following two points are worth emphasising. First, thinking of programming as an exercise of theory building means that the developed program and the knowledge gained by the developers and users through the construction of the program constitutes a 'theory' of the user's application; that is, a verbalisable description of the objectives and processes underlying the program's functionality. Second, agile programming supports the evolution of this theory through a dialogue between the user and developer, over the lifetime of the program's development. Agile programming does this by encouraging the developer to produce many releases of the program, each with gradually increasing levels of functionality. At any point, the user may change their mind about the direction for the next release, and may even decide that already completed code should be modified. Due to the constant and dynamic exchange of information between the developer and user, this helps to ensure changes are relatively minor, and do not build into a large mismatch between the user's expectations and the developer's delivered program.

Returning now to the process of developing scientific architectures, it is clear that the architecture and its dependent models represent the developed program. The scientific theory, we argue, is the 'specification', the theory of the user's application, which is implemented by the architecture. We can draw an analogy between the evolving understanding of the scientist of their theory, and how this understanding is manifested in the architecture, with the evolving understanding of the user of their application, and how the developer converts this understanding into the program; the difference in the scientific domain is that the scientist is, in many cases, both the user, as a model interpreter, and the developer, as the architecture/model creator. The dynamic, constructive and sometimes revisionary nature of scientific progress finds a natural analogue in the process of developing programs using an agile methodology.

In the previous section, we discussed how a scientific theory progresses by various demands: the requirement to achieve new tasks, to model new data, is primary. This scientific progress means that the architecture must be expanded, which leads to the specification of the architecture being augmented or adapted. The only constant in a science is that changes to the theory will be required, and we must be ready to incorporate such changes, by adopting an appropriate software-development methodology.

4 Proposed Methodology

We propose that scientific architectures are most appropriately developed using a form of agile programming, emphasising test-driven development and continuous refactoring. In this section we discuss how the suite of tests constructed during software development can be separated into groups, with two of these groups linking the implementation to the theory and its empirical support. We also argue that the code-level practice of refactoring can help drive architectural and theoretical changes, improving and expanding the structure of our theory.

Test type	Level	Description
Unit	Algorithmic	implementation details
Process	Functional	theory's core processes
Canonical result	Behavioural	empirical results

Table 2. Three levels of tests found in implementation of a cognitive architecture.

4.1 *Test-driven development*

Test-driven development (TDD) (Beck, 2003) is a relatively recent software development methodology which requires developers to provide tests for every piece of code which they create; TDD is recent when compared with more traditional methodologies dating from the late 1960s, such as the Waterfall Model (which became popular in spite of being criticised in its first description Royce, 1970). A test is an example, an illustration that your code does what it claims; executing a test is the equivalent of taking a car for a test drive. For example, Fig. 3 shows some code to test if a function for a perceptron works correctly. The example shows the typical components of a test: constructing an instance of the object to test; computing a value, by calling a method; and comparing that value with the expected value.

There are two major benefits of TDD. First, the developer is forced to commit to a clear description of the intended process, including how it is to be used and its expected results. Second, writing the test forces the developer to produce a concrete example for that piece of code. TDD is important because it 'replaces traditional top-down, up-front design with a more bottom-up, incremental design approach which drives development forward by passing tests' (Stott, 2003, p. 23). In general, TDD is part of an agile-programming methodology in which program specifications are not constructed in advance, but emerge out of interactions between the users, the developers, and the intermediate versions of the working code.

What we have said so far applies to TDD as a *general* software-development methodology. In principle, this will apply to the development of scientific architectures, simply because such architectures are specific kinds of computer programs. However, we can go further by recognising the specific nature of our architecture and its demands as an implementation of a scientific theory, by creating a dedicated testing methodology. In earlier work (Lane and Gobet, 2003), it was shown that the range of tests produced with an implementation of a scientific theory could be divided into groups (see summary in Table 2). The aim of these groups is, firstly, to allocate some of the existing test types used in software engineering into groups best fitting a scientific project and, secondly, to provide a way to separate tests for implementation details from tests for scientifically important processes. These groups include: *unit tests*, which confirm that the basic implementation of the code is correct; *process tests*, which confirm that the code implements the basic theoretical processes correctly; and *canonical results*, which ensure that the code matches specific empirical data. The unit tests are constructed at an algorithmic level, and are tied to the specific implementation of the theory. An example unit test would be to confirm that a method calculating the average output of the model performed correctly given a specific set of model outputs.

The process tests are constructed at a functional level, and are independent of the implementation, but tied to the current theory. An example process test would be to confirm that a perceptron produced the expected output given a specific set of input and weight values; in other words, the test confirms that the perceptron formula for computing its output activation was followed correctly. Process tests are independent of the implementation in the sense that the same example would be required for the test whether the theory was implemented in Lisp or Java, in a functional or an object-oriented style. Process tests document the core processes within the theory in a natural manner, giving a precise example of the kind of input and the expected form of output for the given process. A complete set of process tests forms a descriptive specification of the theory being implemented. As a concrete, exemplar-based specification, this will be inherently easier to understand by a user than an abstract description of the theory, or even the raw code (see Gravemeijer, 1997, for example).

The canonical-result tests are constructed at a behavioural level. Each canonical result captures some empirical result achieved by the theory. This result may simply be a performance measure, such as 'this theory can play chess to a particular standard,' but we shall focus on providing empirical support for a theory through modelling observed behaviour in some task. For example, in a categorisation task, we

Test	Explanation of change	Affects theory?
Unit	Implementation details	No
Process	Key element of theory	Yes
Canonical result	Empirical support for theory	Yes

Table 3. Relation between test failure and theory development.

may want to model the pattern of errors found in a group of human participants. In order to compare performance in this way, the canonical result must include the following information: the experimental setting, the data collected from the human participants, the measure by which the quality of fit is computed, and possibly a target level of fit to be achieved. In order to meet a canonical result, a *model* of the experiment must be constructed within the architecture. This model will need a certain amount of extra program code, on top of the functionality provided by the architecture.

The canonical-result tests will typically be designed at a high level; they are the kind of results that appear in tables and graphs in papers. For example, the CHREST architecture has the results from de Groot and Gobet (1996) and Gobet and Simon (2000) as canonical results. These tests not only gather a large amount of data, but also require statistical tests to confirm the presence of differences in responses between groups. Depending on the result being tested, this kind of test can require a considerable amount of automation.

The best results for many models are often only achieved after searching for the optimum values for its different parameters. One of the motivations for developing broader theories is that parameter choices for one model can be compared with those for another model. By separating out canonical results as specific tests, we label and encapsulate the critical results, supporting automation of parameter selection; this use of multiple canonical results gets us away from the dangers of 'mere' parameter fitting (c.f. Roberts and Pashler, 2000).

A further benefit of our perspective, using canonical results as tests, is that the canonical results are, in many respects, independent of the source theory. This can be seen by separating each empirical result into three components: the part to implement the model within the chosen architecture, the part to treat the model as a participant in the original experiment, and the part that manages the fit to the empirical data. The second and third components can then be applied to models constructed from different architectures, providing a natural process of architecture comparison and reuse of empirical results. We shall return to this point in Section 7.

We have already discussed the importance of taking into account the development of the architecture over time. It is in the nature of scientific enquiry that any theory will be modified and adapted to suit changing requirements. TDD is a natural tool for assisting a programmer in managing change, because the test suite provides a 'safety-net' informing the programmer of any unforeseen implications in modifying parts of the program. In the context of developing a scientific architecture, we can consider our three kinds of tests and ask, what would be the implications if a modification in the architecture caused a failure in any of these three areas?

Table 3 summarises the three kinds of tests, and suggests an explanation for what the implications may be, in terms of our theory, if a modification caused one of the tests in that group to fail. The first group of tests, the unit tests, are concerned with implementation details. These tests will fail if a method is modified so its behaviour does not conform to previous behaviour. As implied by their name, these tests are simply to catch details of the implementation, and may be freely modified to suit new method behaviour without affecting the theory. The second group of tests, the process tests, are concerned with the implementation of basic processes within the theory. If these now fail, then either our implementation is incorrect and needs fixing, or we need to revisit our understanding of how these processes work. The revision may necessitate modifications to our theoretical position. The third and final group of tests, the canonical results, are concerned with the empirical support for the theory. If these now fail, it may be for two reasons: our architecture no longer allows the model which generated the result to run correctly, or our architecture's processes have changed in unanticipated ways to affect the behaviour of this particular model. In both cases, we need to revisit our understanding of the models of these results, and either revise our theory or implementation so the canonical result again applies, or else remove the canonical result from our list of empirical support for the theory.

We note one potential concern with our proposal: often, software development is seen as an incremental process of continually adding new functionality. In science, changes may require fundamental revisions of previous results, a view particularly associated with Kuhn (1962). Nothing in our methodology precludes such non-monotonic growth. We see it as an advantage that the consequences of such revisions will be reflected in terms of the rewriting of previous work: existing tests will have to be removed, and new tests included.

A natural consequence of TDD is a self-documentation by the architecture of those aspects of its behaviour or empirical support which must be reconsidered as a result of changes to the architecture. Scientifically, this provides many benefits: a rigorous validation of any changes to the architecture, a complete set of valid empirical results, and demonstrable examples of every core process within the theory. This makes our theory more understandable, robust and reproducible, points discussed further in Section 7.

4.2 *Improving the design: Refactoring*

One of the dangers for an extended software project, like a scientific architecture, where new features are continually added, is that the implementation becomes more complex, less understandable, and fragile. However, this need not be the case, as Milewski argues:

> 'We usually expect modifications to add complexity and break structure ... Most programs become less and less maintainable as they go through development cycles. But there is no fundamental law that dooms every software project to keep increasing in entropy until it reaches thermal death. Complexity can be fought by imposing more structure. If every program transformation is accompanied by appropriate restructuring, a software project can keep evolving virtually forever.' (Milewski, 2001, p. 437)

Beck (2003) concurs with this view, stating that every step of development should be accompanied by a process of restructuring. A formal process of program modification and restructuring is *refactoring* (Fowler, 1999), which is aimed solely at improving the design of a program; no new functionality must be added. Refactoring can only be done with a program which has a complete suite of unit tests, because the program's behaviour will be checked against the test suite; if all the tests pass before any change, and all the tests pass after the change, then the program's observable behaviour has remained constant. Examples of changes include: changing variable or method names, splitting large methods into smaller ones, moving methods from one class to another, and modifying the class hierarchy by splitting or merging classes. A catalogue of refactorings and examples for Java can be found in Fowler (1999) or similar standard references.

For a scientific architecture, the constancy of its observable behaviour means that refactoring the code does not alter the theory which the architecture implements. Refactoring allows us to explore the space of possible implementations, without affecting the desired theoretical behaviour. However, the link between the theory and its implementation means that the concepts underlying our understanding of the theory may change as a result of refactoring, or vice-versa.

For example, changing a method name is a simple form of refactoring. Instead of a method being called 'identify', we may change its name to 'recognise'. Why would this change be carried out? Usually, such changes arise because of a better appreciation of the function performed by the method, and so a revised name would better suit its intent. Theoretically, we now have a new name to call this process; the change may be driven either by theoretical demands (we would like to change the name of a fundamental process) or by implementation demands (it is clear the implementation of a process is better described by a new name). In either case, theory and implementation are kept in harmony with each other, and processes are given their most appropriate name.

As a second example, we may have a 'classify' process in our theory. In its implementation we find that the process goes through a large number of steps: perhaps first retrieving some information, then seeking an association with the retrieved information, finally returning the associated information as the classification. A complex process like this would be refactored into three separate methods: 'retrieve', 'locate association', 'return association'. Each method would be smaller than the original, perform a single well-defined process, have associated unit tests, and leave the original classify process to call each of the smaller processes. Having performed this refactoring in the implementation, we then have the question:

should these code-level changes affect our understanding of the theory, or are they simply implementation details? In general, this question can only be answered at the theoretical level, and both answers are possible depending on the level of granularity we wish to have in our process descriptions. However, in this example, it is probably clear that a process like 'retrieve' should be elevated to the level of a core process: its description will then feature in descriptions at the theoretical level, and, using our terminology from above, its associated unit tests will be placed in the category of 'process tests'. The last step ensures that the implementation will not be modified in future in such a way as to lose or modify this key process, at least not without the test suite protesting about unmatched tests.

A fundamental task of refactoring is the elimination of 'code smells', such as duplication of code. The process of developing models from a scientific architecture is a natural place where duplication may occur. For example, we (or two different research groups) may develop two models of categorisation, each using a different set of experimental data. We find that we have duplicated the manner in which large sets of data are passed against the model during training. This process, 'learn from dataset', is not present in the architecture, but is shared by more than one model. Should the process be added to the core architecture?

If we treat the architecture as a library to aid the construction of cognitive models, then the answer is probably 'yes'. All processes or methods common to more than one model can be incorporated into the architecture, and so used as a library to improve the development of further models. Having added these processes to the architecture, do we give them the status of theoretical constructs? In this instance, perhaps not: learning multiple examples from a dataset is a relatively simple process. However, if two models of perception had required a similar method of scanning an image, then that scanning technique would be a suitable candidate for a new theoretical concept.

5 Tools: Support for the Methodology

The methodology proposed above is clearly a development methodology for managing the implementation of a scientific architecture. The programmer's task of conforming to the methodology can be made easier by providing appropriate tools. We have developed a test framework to make it easier to write tests, identify their role according to our classification, and manage the tests within their code. We describe this test framework in the next section, and then summarise other development tools which may be of use to the scientist as developer.

5.1 *Test Framework*

An individual test evaluates some function or method from the program and checks that the result of that function or method is as expected; an example was shown earlier in Fig. 3. If the result is correct, the test passes, else the test fails. The aim of testing is to highlight any failing tests, without placing any burden on the user to interpret passing tests. The framework takes as much of this burden away from the architecture developer as possible. The two main points are exemplified in Fig. 3: first, each test has to be constructed using internal operations, with nothing in the expression to indicate that it is a test; consider `(compute-class p 0 1)`. Second, the tests must be combined in some way, so they can be run as a set, and this is achieved for the perceptron example by using the Lisp `and` construct and a function definition. Both of these points are considered in our test framework.

The framework described here follows a popular output format, displaying a single dot '.' for each passing test, but reporting a message for all failing tests. Fig. 4 shows a typical output from the testing framework: each error is highlighted, but several passing tests are shown only by a sequence of dots. A summary is given to indicate how many tests failed. Notice how the tests are organised into three groups, following Table 2.

Table 4 summarises the basic commands provided by the framework. The commands are divided into three groups for defining individual tests, defining groups of tests, and running the tests. The top section gives functions for defining individual tests. Each individual test is, as described above, a check that a piece of code has performed the task it was intended to perform. The basic command is `(test value`

```
[11]> (run-all-tests)
Running Unit tests: ..
Error 1: bad-dp

=== DONE: There was 1 error in 3 tests
Running Process tests: .
=== DONE: There were 0 errors in 1 test
Running Canonical results: .
=== DONE: There were 0 errors in 1 test
```

Figure 4. Sample output from test framework.

Individual tests	
`(test value [msg])`	Basic test: checks if value is true
`(assert= expected actual [msg])`	Calls `(test (= expected actual) msg)`
`(assert-equalp exp actual [msg])`	Calls `(test (equalp exp actual) msg)`
`(assert-true value [msg])`	Same as test
`(assert-false value [msg])`	Calls `(test (not value) msg)`
Defining test groups	
def-unit-tests	Create a unit test group
def-process-tests	Create a process test group
def-canonical-result-tests	Create a canonical-result test group
Running the tests	
`(run-unit-tests)`	Runs just the unit test groups
`(run-process-tests)`	Runs just the process test groups
`(run-canonical-result-tests)`	Runs canonical-result groups
`(run-all-tests)`	Runs all three groups of tests

Table 4. Summary of provided functions.

`[message])`. The `value` is replaced by a call to some code, and a check of its correctness. The optional message may be provided to give better feedback in the event of a failing test. For example, the call `(test (= 11 (dot-product '(1 2) '(3 4))) "dot-product 1")` would check that the return value of `(dot-product '(1 2) '(3 4))` was equal to 11 – failure would give the message 'dot-product 1'.

The other commands in this group make the task of writing tests easier. There are functions to confirm that a function returned a true value, `assert-true`, or a false value, `assert-false`, as well as a group of commands to check that an expected value equals a computed value, using different equality tests; in the latter case, the error message will mention the expected and actual values. An example of this can be seen in Errors 1 and 2, in Fig. 4.

The second group of functions separates groups of tests into categories; it is this group which is designed specifically to assist a developer using our test framework. A function or method within the architecture's code will be tested not once but many times, with a variety of example data. This set of tests will be collected together into a single test function, which will be evaluated together (rather like the function `test-perceptron` from Fig. 3, which provides tests of the `compute-class` function). Our framework enables the developer to place these test functions into categories; the framework should impose a minimal overhead on the developer.

We have implemented our framework in Lisp, so the mechanics of placing test functions into categories must now be described in Lisp terms.[1] The natural way to group tests is to use a function, which will be evaluated as a single item, just like the `test-perceptron` example. In Lisp, the usual way to define such a function is in the form:

[1]Similar techniques are available in other languages. For example, in Java, either subclassing or annotations may be used to identify test categories.

```
(defstruct perceptron theta weights)

(defun compute-class (perceptron inputs)
  (> (+ (perceptron-theta perceptron)
        (dot-product (perceptron-weights perceptron)
                     inputs))
     0.0))

(defun dot-product (vector-1 vector-2)
  (apply #'+ (mapcar #'* vector-1 vector-2)))

(defun bad-dot-product (vector-1 vector-2)
  (apply #'* (mapcar #'+ vector-1 vector-2)))

(def-unit-tests dp1 ()
  (test (= 11 (dot-product '(1 2) '(3 4))) "test dp 1")
  (assert= 3 (dot-product '(1 0) '(3 4)) "test dp 2")
  (test (= 11 (bad-dot-product '(1 2) '(3 4))) "bad-dp"))

(def-process-tests perceptron1 ()
  (let ((p (make-perceptron :theta 0.5 :weights '(0.2 0.7))))
    (assert-true (compute-class p '(3 4)))))

(def-canonical-result-tests cr1 ()
  (let ((p (make-perceptron :theta 0.5 :weights '(0.2 0.7))))
    (assert-true (and (compute-class p '(0 1))
                      (not (compute-class p '(0 -1)))))))

(run-all-tests)
```

Figure 5. Example of using the test framework to test a simple perceptron model.

```
(defun function-name () (test ...) (test ...))
```

Our framework replaces the generic `defun` to define a function with either of `def-unit-tests`, `def-process-tests` or `def-canonical-result-tests`, as appropriate for the tests being created. These new ways of defining functions do the same task of defining a function, but record that function as a test in the appropriate category.

Fig. 5 gives examples of how these functions are used to group collections of tests. The figure begins by defining the functions to be tested: `dot-product` and `compute-class`. In order to test these functions, we need to evaluate them with given values, comparing the result against expected values: the individual tests are written using functions from the first group, such as `test`, `assert=` and `assert-true`. The individual tests are grouped by category, into unit, process and canonical-result tests.

Finally, the developer must have some means of running the tests as a group. Although individual tests can be evaluated by calling their individual function, a complete architecture will have many thousands of such tests, and these must be evaluated as a whole. As shown in the lower part of Table 4, there are four functions provided to run test groups. `(run-all-tests)` will evaluate every test in the complete system, providing output from each category separately. A developer may choose to run just the unit or process tests by themselves, and probably the canonical-result tests will be evaluated less frequently, as a systems-level test. The example output in Fig. 4 illustrates the kind of output from the test runs. Errors are reported as they occur, and a summary of the number of detected errors is provided.

The important element of this framework is the definition of the terms `def-XXX-tests`. When Lisp encounters these terms, it creates a function for the group of tests, just as if `def-XXX-tests` were written as `defun`. But then our framework places the name of that function into a list appropriate to the

XXX in the definition. The functions which run the tests then look at these lists to determine which functions to run. For the developer, it is convenient that only a few functions must be remembered, in order to develop tests within this framework. Also, our framework makes it easy to run any group or all tests at any point of development, encouraging the frequent check of new code against the tests. (The code for this framework is available from the web page referred to in the Appendix; a version suitable for testing software running on the Java Virtual Machine has been implemented as a rubygem, see `https://rubygems.org/gems/modellers_testing_framework`.)

5.2 *Further tools*

Considering the scientific architecture as a piece of code means that a variety of code-level analysis and manipulation tools are immediately available to support the development of our architecture. Here, we summarise a few of the more apparently useful tools. Of course, the applicability and ease-of-use of such tools will depend on the programming environment used to develop the architecture.

Code coverage. A suite of tests is said to 'cover' a piece of code if that code is executed when the tests are run. The more frequently a piece of code is executed, the more important that code is to the implementation. Development tools often support a means of 'profiling' a piece of code, by reporting back how often and for how long the system executed each method. This profile can be used to analyse the implementation of the architecture, determine which parts of the implementation are most important, and perhaps direct attention at ways to simplify or extend the architecture. For example, Licata et al. (2003) analysed the evolution of feature signatures in two programming languages, and found that tests either had an impact on isolated parts of the implementation, or had an impact on almost every part of the implementation, if they were part of the core processes in the language. Similar analyses with scientific architectures could show which parts of the architecture were important for particular groups of models, perhaps suggesting ways in which the theory could be improved by ignoring or promoting processes in the architecture.

Test documentation. One advantage of providing a comprehensive test suite, with a separation into unit, process and canonical-result tests, is that demonstrating the architecture and its models to a user becomes trivial. The tests, being concrete examples of how the architecture functions, document the behaviour of the theory in an example-led manner. Using a suitable tool, we can extract lists of tests and their documentation straight from the source code and provide an easily navigable set of linked pages to demonstrate the architecture's behaviour. This could be provided in the form of a web interface, which may enable the user to locate and select the name of any method or test, view its documentation and, if required, its source code. The programmer can ensure that only the public parts of the code, the parts affecting the theory, appear within the documentation.

Automated refactoring. Many development tools, e.g. Eclipse and Netbeans, support automated refactoring tools. These tools are very useful for ensuring that certain kinds of refactoring are applied consistently to an entire program, and their ability to deal with larger more complex changes is growing.

6 Methodology in Practice: Developing Mini-Chrest

As an example of our methodology, we will develop mini-Chrest, a simple but effective cognitive architecture based on the more comprehensive CHREST (Gobet, 1998; Gobet et al., 2001). Mini-Chrest includes many of CHREST's basic features, such as its core long-term memory operations and short-term memories; missing are templates and the detailed model of perception, making mini-Chrest reminiscent of CHREST's predecessor, EPAM (Feigenbaum, 1959; Feigenbaum and Simon, 1984). We also create models in mini-Chrest of results from a verbal-learning experiment (Bugelski, 1962) and from a categorisation task (Medin and Smith, 1981). The aim of our example is to show how a version of mini-Chrest created for the first

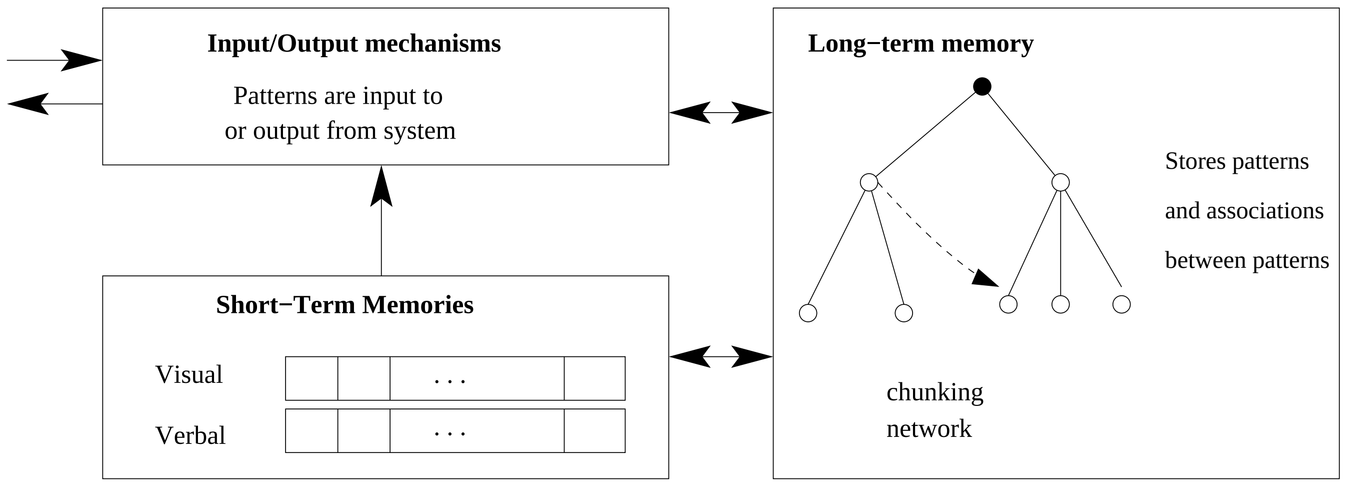

Figure 6. CHREST architecture.

domain can be generalised and extended to accommodate the second domain as well, using our testing framework to retain all the theoretical processes and canonical results along the way.

At the highest level, mini-Chrest is a system for learning *patterns* which are presented to it in a symbolic format, such as a list of letters making up a word. Mini-Chrest will first locate a *chunk* within its memory based on the provided pattern; this chunk will typically contain a subset of the information in the presented pattern, and represent the *familiar* part of the pattern. Much of the machinery behind mini-Chrest is devoted to creating chunks based on the presented patterns, and also to make links between chunks in memory.

We trace the development of mini-Chrest through three distinct versions:

(i) An implementation of the long-term memory mechanisms to learn and recognise patterns. This version supports the construction of models of verbal-learning.
(ii) A rewrite (refactoring) of version 1.0, abstracting out the verbal-learning pattern so that the architecture can be used on different types of data. This includes separate short-term memories for visual and verbal information.
(iii) A further extension to include cross-modal links between visual patterns and verbal labels. This extension enables mini-Chrest to be used to model categorisation data.

The final version of mini-Chrest is quite powerful, and can form the basis of models in a number of different domains. By using the testing framework, we ensure that the empirical evidence and our understanding of the core processes is retained throughout the lifetime of the architecture. Mini-Chrest also forms a good introduction to the more comprehensive CHREST architecture.

6.1 *Verbal-learning model – Version 1.0*

The original memory experiments which led to the development of EPAM, the precursor of CHREST, involved the memorisation of sequences of three-letter ‘words’. Observations of how humans learned such sequences produced descriptive laws explaining these observations. For example, Bugelski (1962) explored the time required to learn lists of nonsense syllables. He required experimental participants to learn lists, such as:
<D A G > <B I F> <Q O F> ...

Each list was repeated a number of times, and the amount of time each item was shown would be varied. Bugelski demonstrated that a constant time per syllable was required by the human learners. Feigenbaum and Simon (1962, 1984) showed that their EPAM model provided an explanation for this descriptive law (Simon, 2000). We now use our methodology to create a model of this experiment.

We begin by considering what the desired input and output to the model should be. As in the original experiment, we use lists of three letters as our input data: e.g. <B I F>. To test the model’s recall

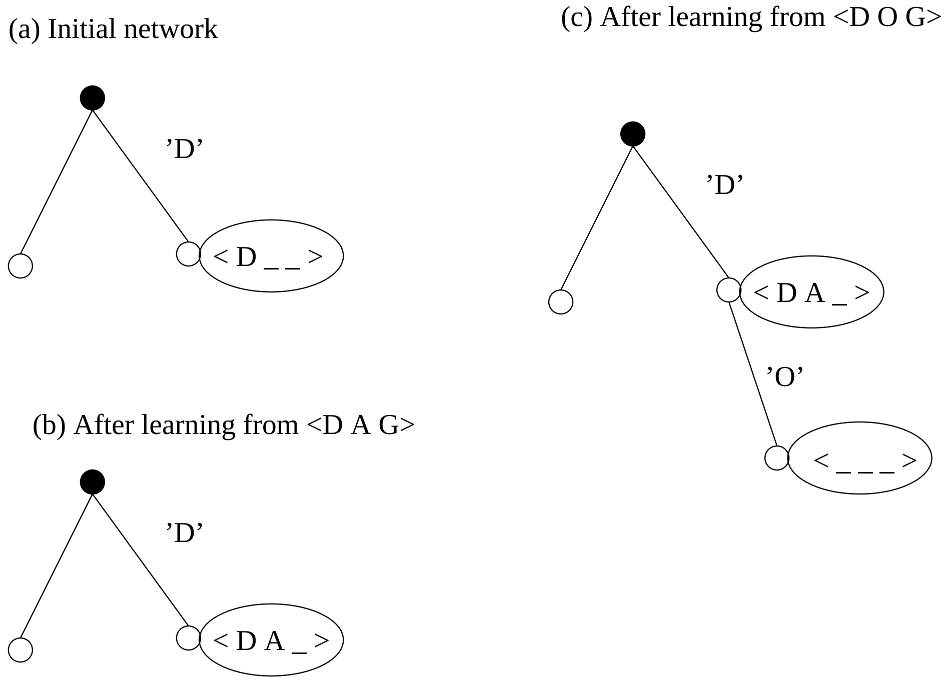

Figure 7. The two learning mechanisms within mini-Chrest. (a) Shows the initial state of the network. (b) Shows one step of familiarisation. (c) Shows the result of discriminating a new pattern.

performance, we will ask it to output a list of letters given some input: if the model 'knows' the input, it should be able to recall the input correctly; if not, the model will recall some partial information, or even perhaps get something wrong. Our basic model should have the following two processes:

(i) recognise: given an input word, return the information accessed in memory by this input.
(ii) recognise and learn: given an input word, sort it through the memory, and then modify memory so as to improve the amount of the input which will be recalled in future.

6.1.1 ***Memory structure.*** The experiment is designed to probe how long-term memory is constructed, because the lists are too long to be held in short-term memory (the architecture does not have a short-term memory yet, but when it gets one, the memory will be limited to a small number of items). Hence, modelling this experiment requires a theory of long-term memory. The theory is that the input patterns, the nonsense words, are held in the nodes of a discrimination network. Each node in the network will hold the partial or complete information of a single pattern. There are two key learning operations which affect the state of the network. *Familiarisation* extends the information (known as the *image*) held at the node. Familiarisation occurs when a node is retrieved and its image is found to match the pattern being sorted; a new part of the sorted pattern is added to the image. *Discrimination* extends the network by adding a new test and node to the network. Discrimination occurs when a node is retrieved and its image mis-matches the pattern being sorted; a new test is added below the retrieved node. The two learning operations are complementary: familiarisation improves the amount of information *recalled* about a given pattern, and discrimination increases the *number* of different patterns which can be identified. Fig. 7 illustrates these two learning operations.

The aim of Bugelski's experiment was to probe the amount of time it required for a learner to acquire each word. In order to give our model a sense of time, we need to indicate how long each process in the system will take. For simplicity in mini-Chrest, we only provide timings for the familiarisation operation (2 seconds) and the discrimination operation (10 seconds). These timings constrain the rate of learning, as further learning cannot take place until the previous operation has completed. The complete CHREST architecture provides timings for further processes, such as the time to sort a pattern through the discrimination network. These timings act as *parameters* in our theory, and may be modified. There is a methodological point here, about the danger of tuning parameters to fit specific experiments; as Simon (2000) explains, the

```
  (i) Let the current node be the root node
 (ii) Let the potential children be the children of the current node
(iii) If the list of potential children is empty: return the current node
 (iv) If the input pattern matches the link test of the first potential child:
        a) Let the current node be the link child of the first potential child
        b) Continue from step 2.
  (v) Remove the first potential child from the list of potential children.
 (vi) Continue from step 3.
```

Figure 8. The steps within the *recognise* process, used within mini-Chrest to find a chunk matching the input pattern.

parameters used in EPAM to model Bugelski's data were obtained from prior experiments with EPAM, and not specially tuned to the new experiment. This fixing of parameters in one experiment to act as constants in the next is one of the chief benefits of using many sets of experimental data to constrain the development of a general architecture and its later models (Gobet and Ritter, 2000) (cf. discussion in Section 4.1 on parameter fitting).

6.1.2 ***Implementation.*** The implementation is divided into three parts: the representation and use of the patterns, the representation and use of the long-term memory, and the representation and use of the complete system.

Patterns. We shall use the same internal and external representation for the data: a simple list of symbols `'(B I F)`. We can use Lisp's functions for testing equality of lists to check if two patterns are the same. The learning processes for deciding whether to familiarise or discriminate require the model to test if one pattern 'matches' a second pattern: for example `'(B I)` will match `'(B I F)` because it is a presequence of the second pattern. We need to implement a function `matching-patterns-p` to make this comparison; see Appendix A.1.1.

Long-term memory. The memory, as explained above, is arranged as a network of nodes. Our data structure for a node will be a simple structure, with three fields: contents, a record of the tests required to reach this node; image, the information stored at this node; and the children, an ordered list of links following from this node. Each link is again a structure with two fields: a test, which the pattern must satisfy to follow this link; and a child node, which is reached by following this link.

The long-term memory is modified by adding information to the image of a node, through the `familiarise` operation, and by adding a new child link to a node, through the `discriminate` operation.

System. The mini-Chrest system is represented as a structure, holding a reference to its internal long-term memory, as well as other information, such as its internal clock and values for the timing parameters. Operations on the system define the external interface, used by modellers to develop models using mini-Chrest. The key operations are *recognise* and *recognise-and-learn*. We give a summary of their operation here.

The *recognise* process requires an individual model to sort the provided input pattern through the memory, beginning from the root node. The process is shown in Fig. 8.

(In an experiment, the image of the returned node should be used to measure the amount of detail recalled, and an extra function `recall-pattern` is provided in the architecture for this purpose.)

The *recognise and learn* process requires the model first to sort the provided input pattern through memory, using the above process. Then the returned node will be modified. If the image of the node retrieved matches the input pattern, then familiarisation will occur, calling `familiarise` on the retrieved node. If the image of the node does not match, or if the retrieved node is the root node, then discrimination will occur, calling `discriminate` on the retrieved node.

6.1.3 ***Tests.*** The tests must check the operation of code in all three areas of the implementation. These tests are divided into the prescribed three groups: unit tests determine whether the code is implemented correctly; process tests determine whether the code supports all the key processes of the target theory; and canonical-result tests determine whether the complete set of code meets the desired target behaviour.

Patterns. There are three things that occur with patterns: they are created, they can be compared for equality, and they can be compared to see if they match. Of these three, only the last required us to produce new code, so this last process must be tested. Although we rely on the correct functioning of the underlying implementation language, it can be helpful to provide tests for basic operations like equality, to provide a concrete example of the operation. We don't do this for the unit tests, as these are implementation details, but would do so for key processes, no matter how trivial the code.

The first question we must ask is: what kind of test do we want? Is this operation, to check if two patterns match, a canonical result or a key process in our theory? In this case, we would argue neither, as the matching operation is not part of the theory's description. So we create unit tests to check the functioning of this method. Appendix A.1.1 gives the source code and tests for this function. Notice how the tests check some boundary conditions of the function, meaning empty patterns, and also make sure both possible values (true or false) are returned. Also notice that the tests would not need changing if we changed the way the function was implemented; the tests capture the *intention* of the function, how it is specified to work.

Long-term memory. There are only two operations which can affect the long-term memory, and these are the core learning processes: familiarise and discriminate. To test these mechanisms we need to create a sample node and check that its image responds appropriately to the `familiarise` function and that a new child is created appropriately for a `discriminate` function. These tests are core to the theory, and so are defined as *process* tests. The language used in the tests is at a high level, appropriate to the theory:

```
(assert-null (node-image node))
(familiarise node '(B I F))
(assert-equalp '(B) (node-image node))
```

These lines check that the node's image is empty, call the function to familiarise a node with a pattern, and then check that the image of the node is now of the expected value. The abstract structure of this test is clearly derived from the theory's requirements of the familiarisation function, and we would expect a similar test to appear in *any* implementation of the theory. The full source code and tests for the long-term memory can be found in Appendix A.1.2.

An aside on the kinds of values we use for testing. It is important to remember that these functions will *only* be called under appropriate circumstances, and we only need to test the functions under these circumstances. For example, the `familiarise` function is only called by the `recognise-and-learn` function when the input pattern matches the node's image: this precondition is checked by the `assert` statement in the function's definition. However, we do not need to test for this condition, as it will only occur if the code calling `familiarise` did so incorrectly.

System. The complete model has two main functions: *recognise* and *recognise-and-learn*. We need to test both of these functions. One of the problems with testing a function like *recognise* is that we need an example model on which to test it; this example model must be constructed by hand in the test. Although this process can be tedious initially, the benefits for automated testing once it has been constructed are great, as should become apparent as we further extend the model's implementation. Appendix A.1.3 contains the tests for `recognise-pattern`. These tests check that a sample model returns the correct node given different patterns to sort. The tests for `recognise-and-learn` use the function to build up a model for some given patterns, and then test that the expected pattern is recalled. Additional tests are used to confirm that the internal clock is updated during learning, and also that updates are only permitted when the model is free.

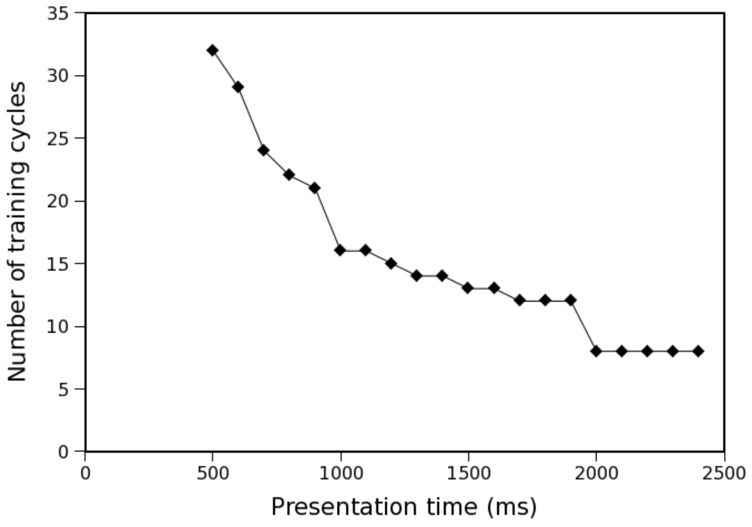

Figure 9. Plot of number of training cycles against presentation time.

Canonical result. Bugelski (1962) showed that the total time required to learn a paired-associate list is not affected by presentation rate, which is the amount of time each word is visible before the next one appears. Another way of stating this result is that the number of times the list must be repeated before it is completely learned is inversely proportional to the amount of time each item is presented for.

We will not precisely replicate this experiment here, as it requires participants to predict the next stimulus in the list, and the ability to form associations is only added to mini-Chrest in version 3.0. Instead, our criterion of success will be that every stimulus is completely learned by the model: when asked to recall the stimulus, it will return the complete stimulus. We will count the number of cycles to reach the success condition, and see how the number of cycles varies with the presentation time of each item.

We trained mini-Chrest using the list of nonsense words:
`(D A G) (B I F) (G I H) (J A L) (M I Q) (P E L) (S U J)`

Mini-Chrest was shown each word in the list, asked to learn it, and, at the end of the cycle, was checked whether it knew every word in the list correctly. If not, mini-Chrest was trained again through another cycle. The presentation time for each item was varied, from 500ms to 2400ms in 20 steps of 100ms. Fig. 9 shows a plot of the number of training cycles against the presentation time. The correlation between these two values is -0.927.

This highly negative correlation supports Bugelski's results, and so mini-Chrest provides a model of these data. We now capture this result and conclusion in a test:

```
(def-canonical-result-tests constant-learning-rate ()
  (assert-true
    (<
     (compute-pearson-correlation-coefficient
       (do-bugelski 500 100 20
                    '((D A G) (B I F) (G I H) (J A L)
                      (M I Q) (P E L) (S U J))))
     -0.9)))
```

This demonstrates the typical pattern for an empirical test:

(i) The experiment must be run on the model, or series of models. In this case, the function `do-bugelski` uses the list of items for the input, and indicators of the number of experiments to perform (20), what presentation time to begin with (500ms) and the amount to increase the time by between experiments (100ms).

(ii) The results are passed to a function for computing a correlation coefficient, a quantity for judging the quality of the model.
(iii) Finally, the computed quantity is compared against a target value. The test passes if the computed quantity is at least as good as the target value.

In real settings, the statistical calculation may also compare the model's performance directly with some human data. These calculations will be decided upon and judged by the experimenter, not the architecture. Once the experimenter has determined the best way of calculating the performance of their model, and so confirming that it is a model of the data, these calculations and quantitative check can be converted into an automated test, which can then be run as a single function, a canonical result.

6.2 *General architecture – Version 2.0*

The main aim in moving to Version 2.0 is the desire to support more than one type of pattern; we want our architecture to support models in domains other than verbal learning. To implement this, we have three separate aims. First, we want to extend the nature of a 'pattern'. In particular, patterns may be visual or verbal data, and may come from different domains, so contain different kinds of features. Second, we want the modality of the pattern, whether it is visual or verbal, to be identified. Third, we require a short-term memory system which can store pointers to nodes in long-term memory; a different short-term memory will be provided for each input modality.

The most important part of this process is the arrangement for the different pattern types. As every pattern must have the same set of methods, we will adopt an object-based strategy, with each pattern type being a subclass of one of the basic `visual-pattern` or `verbal-pattern` classes; this division meets the second aim. We now need to locate in our existing implementation every function which relates to the patterns, and turn that function into a generic name. The details of this are not important, but the final result for the verbal patterns is contained in Appendix A.2.1.[1]

The result of generalising the pattern type is that the long-term memory and model functions now call generic operations, applicable to many pattern types. For example, whereas with Version 1.0 equality of patterns could be checked using the built-in `equalp` function, now equality of patterns is checked using the generic `equal-patterns-p` method. Also, because the model could be trained on any type of pattern, a special pattern type has been created to represent the root node in memory. All of these details belong to the world of programming and implementation. What is important for our theory is that the new version of mini-Chrest *passes (almost) the same set of tests as before*! The 'almost' is because we now have to use our new pattern classes and associated methods to construct patterns and compare the results in our tests; the same underlying data is used for testing, and the same function names are used at the model level. A comparison of the tests in Appendix A.1.1 and Appendix A.2.1 for the methods `matching-patterns-p` and `familiarise` illustrates how the intention of the method (its specification) has not changed, although its syntactic implementation has.

The final aim, the addition of a short-term memory, is easily achieved. We provide a short-term memory structure with two fixed-length queues, one queue for visual patterns and one queue for verbal patterns. When a node is retrieved in the recognise-pattern function, a pointer to that node is placed onto the top of the queue appropriate to the modality of the node's image. We only allow one pointer to each node in the queue, and pointers which drop off the bottom of the queue are forgotten. The addition of the short-term memories introduces two new parameters into the system, the size of each of the memories. These sizes have been explored experimentally (Gobet and Clarkson, 2004), and a value of 3 or 4 is used in most of our work with more extensive CHREST models. New tests are provided in the implementation to confirm the operation of the short-term memory.

Very little had to be done to the part of the code using the architecture to model Bugelski's experiment,

[1]For ease of description, we are not discussing all the details here, although this step is the most important and also the most complex of all the transitions to be made to the architecture. In a real cycle of development, it could not be made in isolation. A more plausible development step would be to use the target of the categorisation experiment more directly as a means for generalising the pattern functions. A technique for doing so using tests is Beck's 'Triangulation', see Chapter 3 of Beck (2003).

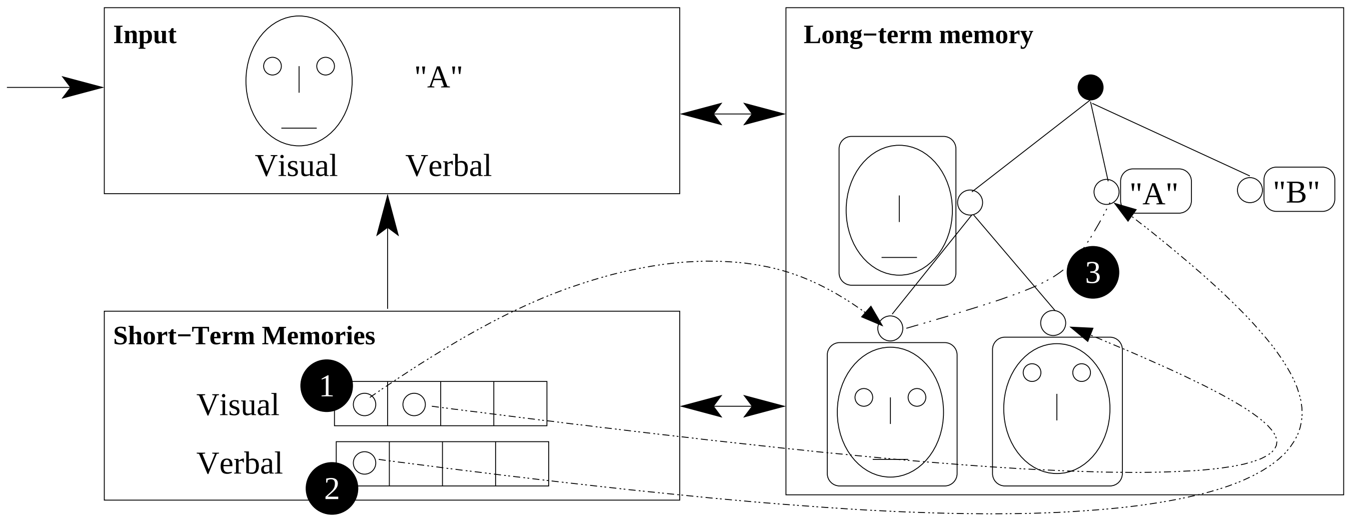

Figure 10. Learning a 'naming link' across two modalities. (1) The visual pattern is sorted through LTM, and a pointer to the node retrieved placed into visual STM. (2) The verbal pattern is sorted through LTM, and a pointer to the node retrieved placed into verbal STM. (3) A 'naming link' is formed between the two nodes at the top of the STMs.

and so to check the empirical evidence for the theory. The changes were two minor, syntactic ones: a conversion of the input patterns to the new object style, and use of the general equality test in the function to check whether the model had finished learning. After these small changes, the test passed, assuring us that even with so many complex changes to the underlying implementation of the architecture, the overall behaviour has not been affected.

6.3 *Categorisation model – Version 3.0*

The aim of Version 3.0 of the mini-Chrest architecture is to support the development of models in domains such as categorisation, where a stimulus must be named. A classic example of such a problem is the five-four task of Medin and Schaffer (1978); Medin and Smith (1981). Our architecture so far supports most of the ingredients for this domain, in particular the ability to learn from more than one input modality and store the retrieved nodes in separate short-term memories. Remaining is just the ability to associate a stimulus in one modality with a response in a second.

6.3.1 ***Implementing cross-modal links.*** In order to implement cross-modal links, we must extend the definition of a node. We follow the scheme introduced by Gobet (1996) and used by Lane et al. (2003), where each node may support a link to another node in the network. These links are cross-modal links if the source node is a visual pattern and the target node a verbal pattern; such a link enables the verbal name to be retrieved after sorting a visual pattern. The formation of such a link occurs when the two patterns have been sorted and nodes retrieved and placed into their respective short-term memories. A link is then formed between the node at the top of the visual short-term memory, and the node at the top of the verbal short-term memory. Fig. 10 illustrates this mechanism in operation.

6.3.2 ***Model of the five-four task.*** Smith and Minda (2000) describe a collection of thirty previous experimental results using the five-four structure of Medin and Smith (1981), and analyze a number of mathematical models of behaviour. Although instantiated in different ways, the basic experiment uses the structure illustrated in Table 5. There are four binary attributes: different interpretations for each stimulus are created by varying the meaning of the attributes. Examples of category A are typically those closer to having all four attribute values set, whereas examples of category B are typically those closer to having all four attribute values unset. Example E2 is interesting because it is the only one with two set values which is in category A.

	Attribute (A)					Attribute (A)			
Example	A0	A1	A2	A3	Example	A0	A1	A2	A3
A examples					Transfer items				
E1	1	1	1	0	E10	1	0	0	1
E2	1	0	1	0	E11	1	0	0	0
E3	1	0	1	1	E12	1	1	1	1
E4	1	1	0	1	E13	0	0	1	0
E5	0	1	1	1	E14	0	1	0	1
B examples					E15	0	0	1	1
E6	1	1	0	0	E16	0	1	0	0
E7	0	1	1	0					
E8	0	0	0	1					
E9	0	0	0	0					

Table 5. The five-four structure used in categorisation experiments.

These data can be used to create different kinds of objects, depending on the interpretation given to the attributes. For example, by making A0 eye height, A1 eye separation, A2 nose length, and A3 mouth height, we obtain the face experiment performed by Medin and Smith (1981), and simulated by Gobet et al. (1997). The thirty different experiments discussed by Smith and Minda (2000) used different interpretations of the attributes, and varying instructions for the participants.

For mini-Chrest, our target will be to match the experimental data from one of these different experiments. The experimental data is obtained from a pool of participants. Each participant is shown the examples on the left of Table 5 and told the category of the instance. After becoming familiar with the labelled examples, each participant is shown all sixteen instances in the table, and asked to place each one into category A or B. Across the group of participants, we can obtain the probability that each instance is placed into category A.

The first issue for mini-Chrest's implementation is how to represent the visual and verbal patterns. The categories are represented as verbal labels, using a `name-pattern` class. The actual instances are presented as visual patterns. We find that the visual patterns can be represented as lists of bits, e.g. instance E1 could be represented as `'(1 1 1 0)`. This pattern is just like the verbal-learning pattern used before. We make a refactoring in the code here to generalise the name of the class used to represent the verbal-learning pattern: instead of `vl-pattern`, we now use `list-pattern`. Along with this change, we make the syntactic change to the test and other relevant code to reflect the new name. After this change, we check that all the tests still work, confirming that our refactoring has not broken anything in any part of mini-Chrest's behaviour.

The second issue for mini-Chrest is that the architecture so far has been deterministic; we have no natural way to produce the different results required to model a population of individuals. Following Gobet et al. (1997), we introduce a parameter, ρ, which is the probability that learning will occur (ρ will take values in the range 0.0 to 1.0, inclusive). The system at present will always learn a new stimulus if it is not otherwise busy; the addition of ρ means learning only occurs if mini-Chrest is not busy and if a randomly generated number is smaller than ρ. For example, if ρ is at its default value of 0.7, then learning will occur 70% of the time when possible. Setting up the experiment is now relatively trivial. A pool of 100 models is created, all are given the examples on the left of Table 5 to learn from (learning is repeated for ten cycles), and then the responses of the 100 models to all the instances are collated to obtain the probability that each instance is placed into category A.

The code for the experiment is wrapped up as the function `do-categorisation-expt`, and the canonical-result test checks that the output probabilities from the population of models are correlated with the predicted results:

```
(def-canonical-result-tests categorisation-expt ()
  (assert-true (> (compute-correlation (do-categorisation-expt) AVG)
  0.65)))
```

Again, we have made some changes to mini-Chrest's operation, with the addition of ρ. We find that the default value of ρ still satisfies the canonical result for Bugelski's experiment. However, for the process tests involving learning, the tests assume that the learning operation does occur, and so for these tests we set ρ to be 1.0, which forces learning to occur – the canonical results all use the same set of parameters to construct instances of mini-Chrest.

6.3.3 ***Returning to Bugelski's experiment.*** The machinery for learning a cross-modal link can be extended to learn 'sequence' links, that is a link between a stimulus and a response in the same input modality, where the response is presented after the stimulus. The extension is to form such a link between the top two items of the short-term memory for one modality. With this extension, mini-Chrest supports better models of the verbal-learning experiments, and can properly replicate Bugelski's experimental format. With this example, we see the benefit of having a cognitive architecture from which to develop our models, as each model can draw on the lessons learned from the other domains.

6.4 *Summary*

In this section, we have presented an illustrative development of mini-Chrest. The development proceeded incrementally, beginning with a simple model of the verbal-learning task, and ending with a generic architecture, capable of managing multiple kinds of visual and verbal patterns. This last version required some extensive rewriting and refactoring when compared with the first version: first the patterns were turned into a class-based representation, and then the names of the patterns were changed to reflect the more generic sets of patterns. All these changes were made to the code, using the tests created for the first version as a safety-net. Everytime anything was changed, the code was checked against these tests. The importance of the tests is to demonstate that the empirical support and conceptual understanding of version 1.0 has been retained in version 3.0.

In the final version, we have 101 tests, as shown in the following output:

```
* (run-all-tests)
Running Unit tests: .....................................
...............................
=== DONE: There were 0 errors in 69 tests
Running Process tests: ..............................
=== DONE: There were 0 errors in 30 tests
Running Canonical results: ..
=== DONE: There were 0 errors in 2 tests
NIL
```

The 2 canonical results confirm that mini-Chrest obtains the target results in both the Bugelski experiment and the categorisation experiment. The 30 process tests check the key processes within the architecture, and the 69 unit tests check the lower-level implementation details. The final system has about 670 lines of code, with 400 of these representing the tests (including the descriptions of the models).

7 Discussion

This article has proposed that an agile development methodology with a three-layer scientific test harness makes an effective methodology for developing scientific software, as it couples the theory with its implementation. All changes in the theory must be reflected in the implementation, and changes in later versions with previous behaviour will be identified by the tests. A (weak) case can be made for the value of tests simply as a software-development methodology:

'No studies have categorically demonstrated the difference between [test-driven development] and any of the many alternatives in quality, productivity, or fun. However, the anecdotal evidence is overwhelming, and the secondary effects are unmistakeable.' (Beck, 2003, pp. 202–203)

A stronger case can be made for the value of test-driven development in linking the processes within the code with the functional descriptions contained in the tests. This is important in maintaining the link between our scientific theory and general understanding of the system with the actual code implementing that understanding within the architecture and its component models. This link makes our methodology a potential solution to some criticisms and issues that have arisen specifically in the development of symbolic cognitive architectures. In this discussion, we suggest ways in which our proposed methodology can support or be adapted into a more general-purpose methodology for computational modelling.

7.1 *Reproducible and comprehensible theories*

One of the advantages of providing tests for code is that every process within the system becomes associated with a concrete example of what that process is meant to do; the test captures the *intention* of the code, and the code captures the *process*. A user of the system can use the tests to understand the intent behind each part of the code without necessarily understanding the mechanism. For example, in Appendix A.1.1 some Lisp code is provided to implement a function `matching-pattern-p`. Understanding this requires the reader to know Lisp, understand functions like `car` and `cond`, and appreciate the idea of recursion. But looking at the tests for this function, the reader can quickly see what kinds of patterns are supposed to match, and, as the test can be verified to run, the reader can confirm that the code does do what it is intended to do. An adventurous reader could even try new examples, to clarify their own understanding against the architecture's. We suggest that providing a complete set of tests with our scientific architectures, particularly divided into groups to highlight the important processes of the system, will go a long way to addressing the recurring complaint of computational implementations that the theory cannot be understood from the code.

A related issue, which the tests help to solve, is that scientific results should be *replicable* by another scientist. In large cognitive architectures such as Soar, this may be practically difficult (recreating 50,000 lines of code is a long task), but in principle we would like to support such an endeavour. Cooper and Shallice (1995) discuss some difficulties in replicating the behaviour of Soar from its verbal descriptions. A similar problem arises in large-scale software projects, such as when developing alternative implementations of programming languages. A case in point is the development of jruby, an alternative implementation of the ruby language for the Java Virtual Machine. In response to such efforts, a comprehensive test suite for the ruby language has been created, and jruby's success as a replication of ruby is judged based on its ability to match that test suite. (A further behavioural test for jruby was its ability to run the important ruby application, which is rails.) Well documented tests make good specifications of complex software projects, and scientific software would similarly benefit.

The use of our scientific-testing framework would support a scientist creating a new version of a given architecture. The process and canonical-result tests provide the target tests for the new implementation, and, once passed, a new implementation can be claimed to satisfy all the concrete demands of the previous architecture. By separating out unit and process tests, we clearly identify those behaviours in our implementation which are critical to the theory, and those which are merely part of the coding, an important issue raised by Cooper and Shallice (1995). For example, Appendix A.1.1 provides *unit* tests for `matching-pattern-p`, so this part of the code is an implementation detail; Appendix A.1.2 provides *process* tests for `familiarise`, so this part of the code is critical to the theory. Such uses of the testing framework have been invaluable in transforming a Lisp implementation of CHREST into a Java version, jChrest.

7.2 *Comparing and creating theories*

The testing methodology is not simply a way to write robust programs, but can also help drive scientific understanding forward by supporting quantitative comparisons of different theories. The high-level behaviour captured in our canonical-result tests are in the right form for supporting optimisation of our models and providing comparison between architectures. A published set of canonical-result tests would form a target for modellers using other architectures who aim to construct alternative models for the same phenomena. As the need for empirical comparisons is more widely recognised (Biegon, 2006; Cragin et al., 2010), such published sets of data have become increasingly common, e.g. in life sciences there is GenBank (`http://www.ncbi.nlm.nih.gov/genbank/`), and Myung and Pitt (2010) have recently started a similar site for psychology. We anticipate the role of such repositories in comparing implemented theories will become increasingly important, and that our proposed definition of a canonical-result could be used to provide a common structure to published datasets, perhaps as a form of meta-data.

When discussing the canonical results, we argued that the basic test is independent of any particular model or architecture. Each canonical result is a test which can be applied to different models, developed in the same or different architectures. For example, the five-four task has been used in a large number of experimental studies, and these data can be used to compare models from different theories (Gluck et al., 2001; Smith and Minda, 2000). We can exploit these tests in a formal methodology for comparing architectures using a specially adapted optimisation technique (Lane and Gobet, 2005a,b, 2007). The idea is to generate the best models possible with parameters which fit across all the provided sets of data. Then, these best models from each contending theory are compared, to see which theory's models do best on which sets of empirical data.

Theory-independent tests are also important in efforts to automatically generate models or theories. In unrelated work, Frias-Martinez and Gobet (2007) have shown how genetic programming can be used to generate theories, judging the quality of each theory by how well its models fit a standard set of experimental data. These experimental data, in the terminology of this article, would form the set of canonical results for the developed theories. A related idea, which may be applicable to architectures such as ACT-R (Anderson et al., 2004), would be for the genetic program to construct alternative models within a single architecture, attempting to optimise their fit to a set of canonical results. Gluck and Pew (2005) report on the variations found when different teams construct models for the same set of empirical data, so this would seem an interesting area for further exploration.

7.3 *Scientific methodology*

Cognitive scientists have debated the nature of the process of constructing computational models and architectures and how this relates to a broader consideration of its scientific nature. One of the problems with a cognitive architecture is that it is hard to prove that a particular target phenomenon cannot be modelled; a different model could hide or compensate for some of the deficiencies in the architecture. This problem means that a Popperian view of falsifiable theories becomes harder to maintain, and several prominent authors (Cooper, 2007; Feigenbaum and Simon, 1984; Newell, 1990) have suggested that Lakatos (1970) provides an alternative perspective which better fits cognitive science.

Lakatos (1970) proposed that a scientific theory should be divided into two aspects: the first aspect, the *core*, is the part of the theory which the scientist is committed to; the second aspect, the *periphery*, acts as a protective belt of subsidiary hypotheses. Although our methodology is not derived directly from such considerations, there is a correspondence to be drawn between what we call the architecture with Lakatos' core assumptions, and what we call the models with Lakatos' peripheral hypotheses.

However, another view may present itself, if we concur with Feigenbaum (1959) and adopt the strong view that the implementation is the formal specification of the theory. The implementation makes concrete much of what is important in the scientific theory. Our methodology allows this implementation to grow and evolve over time in response to changing empirical demands on the implemented theory. We do not preclude the possibility that this evolution may lead to the modification or removal of certain key processes; our concern so far has just been that these changes be clearly flagged by the tests. It would be an interesting exercise to trace the evolution of an architecture's actual implementation over many versions, with many supported models, using our methodology to explore how much of the central implementation remains constant. This picture of a gradually evolving computer program may give a better model of how scientific understanding progresses in response to empirical demands.

8 Conclusion

Scientists in many disciplines use computers to represent and model processes believed to occur within the world. Scientific understanding of these processes will develop over time, and so will their implementation. We have argued that a tight link between our scientific knowledge and the structure of the computer program can be obtained through the use of an agile programming methodology and a three-layer scientific test harness. With the examples of developing CHREST, we have demonstrated the value of tests in ensuring that previously important behaviour is preserved whilst developing new versions of the architecture. We claim that the three levels of tests, expressing the theoretical relevance of every piece of code, will make scientific software more understandable and also, if necessary, easier to replicate. In this way, the formal specification of a theory can be separated into the two halves of an ever-evolving program: the process-based specification is contained in the code, and the behavioural specification is contained in the tests.

Appendix A: Source code for system and tests

The material in these appendices is a subset of the complete set of code, available from: `http://chrest.info/software.html` under mini-Chrest.

A.1 *Recall model: Mini-Chrest version 1.0*

A.1.1 *Pattern Implementation.*

```
(defun matching-pattern-p (pattern-1 pattern-2)
  "Pattern-1 matches pattern-2 if it is a presequence: check this by recursion"
  (cond ((null pattern-1)
         t)
        ((null pattern-2)
         nil)
        ((equalp (car pattern-1) (car pattern-2))
         (matching-pattern-p (cdr pattern-1) (cdr pattern-2)))
        (t
         nil)))

(def-unit-tests matching-pattern-tests ()
  (assert-true (matching-pattern-p () ()))
  (assert-true (matching-pattern-p () '(a b c)))
  (assert-false (matching-pattern-p '(a b c) ()))
  (assert-true (matching-pattern-p '(a) '(a)))
  (assert-true (matching-pattern-p '(a b) '(a b)))
  (assert-true (matching-pattern-p '(a) '(a b c)))
  (assert-true (matching-pattern-p '(a b) '(a b c)))
  (assert-false (matching-pattern-p '(a b c) '(a b))))
```

A.1.2 *Long-term memory implementation.*

```
(defstruct node contents image children)
(defstruct link test child)

(defun familiarise (model node pattern)
  "Extend image of node with a new item from pattern"
  (assert (matching-pattern-p (node-image node) pattern))
  (when (> (length pattern) (length (node-image node)))
        (incf (chrest-clock model) (chrest-familiarisation-time model))
(setf (node-image node)
      (append (node-image node)
      (list (nth (length (node-image node)) pattern))))))

(defun discriminate (model node pattern)
  "Add a new child to node, with a new item from pattern,
   taken from node contents"
  (assert (eq (recognise-pattern nil pattern node) node))
  (assert (> (length pattern) (length (node-contents node))))
  (incf (chrest-clock model) (chrest-discrimination-time model))
  (let ((new-item (nth (length (node-contents node)) pattern)))
    (push (make-link :test new-item
                     :child (make-node
                              :contents (append (node-contents node)
                                                 (list new-item))
                              :image nil
                              :children nil))
          (node-children node))))

(def-process-tests familiarise-tests ()
  (let ((node (make-node :contents nil :image nil
                         :children nil))
      (model (create-chrest)))
    (assert-null (node-image node))
    (familiarise model node '(B I F))
    (assert-equalp '(B) (node-image node))
    (familiarise model node '(B I F))
    (assert-equalp '(B I) (node-image node))
    (familiarise model node '(B I F))
    (assert-equalp '(B I F) (node-image node))
    (familiarise model node '(B I F))
    (assert-equalp '(B I F) (node-image node))))

(def-process-tests discriminate-tests ()
  (let ((node (make-node :contents '(B I) :image '(B I F)
                         :children nil)))
    (discriminate (create-chrest) node '(B I H))
    (assert-false (null (node-children node)))
    (assert-false (eq node (recognise-pattern nil '(B I H) node)))
    (assert-equalp '(B I H)
                   (node-contents
                     (recognise-pattern nil '(B I H) node)))))
```

A.1.3 *Model tests.*

```
(def-process-tests recognise-tests ()
  (let ((empty-model (create-chrest)))
    (assert-eq (chrest-ltm empty-model)
(recognise-pattern empty-model '(B I F))))
  (let* ((node-1 (make-node :contents '(B I) :image '(B I)
                              :children nil))
         (link-1 (make-link :test '(I) :child node-1))
         (node-2 (make-node :contents '(B) :image '(B I F)
                              :children (list link-1)))
         (link-2 (make-link :test '(B) :child node-2))
         (node-3 (make-node :contents '(I) :image '(I F)
                              :children nil))
         (link-3 (make-link :test '(I) :child node-3))
         (root-node (make-node :contents nil :image nil
                                 :children (list link-2 link-3)))
         (model (make-chrest :ltm root-node)))
    (assert-eq node-1 (recognise-pattern model '(B I F)))
    (assert-eq node-2 (recognise-pattern model '(B)))
    (assert-eq node-2 (recognise-pattern model '(B E F)))
    (assert-eq node-3 (recognise-pattern model '(I)))
    (assert-eq root-node (recognise-pattern model '(G)))))

(def-process-tests recognise-and-learn-tests ()
  (let ((model (create-chrest))
        (pattern-a '(B I F))
        (pattern-b '(X A Q)))
    (dotimes (n 4)
   (recognise-and-learn-pattern model pattern-a)
   (recognise-and-learn-pattern model pattern-b))
    (assert-equalp pattern-a (recall-pattern model pattern-a))
    (assert-equalp pattern-b (recall-pattern model pattern-b))))
```

A.2 *Generalised patterns: Mini-Chrest version 2.0*

A.2.1 *Object-based pattern implementation.*

```
(defclass pattern () ())
(defclass visual-pattern (pattern) ())
(defclass verbal-pattern (pattern) ())

(defgeneric make-pattern-for (pattern))
(defgeneric empty-pattern-p (pattern))
(defgeneric equal-patterns-p (pattern-1 pattern-2))
(defgeneric matching-patterns-p (pattern-1 pattern-2))
(defgeneric get-next-item (source-pattern target-pattern))
(defgeneric combine-patterns (source-pattern target-pattern))

(defclass verbal-pattern (visual-pattern)
  ((data :accessor get-data :initarg :data :initform ())))

(defmethod equal-patterns-p ((pattern-1 verbal-pattern)
                             (pattern-2 verbal-pattern))
  (and (= (length (get-data pattern-1)) (length (get-data pattern-2)))
       (matching-patterns-p pattern-1 pattern-2)))

(defmethod matching-patterns-p ((pattern-1 verbal-pattern)
                                (pattern-2 verbal-pattern))
  "Pattern 1 matches pattern 2 if its data is a presequence"
  (presequence-p (get-data pattern-1) (get-data pattern-2)))


(def-unit-tests verbal-matching-pattern-tests ()
  (let ((a (make-instance 'verbal-pattern :data ()))
        (b (make-instance 'verbal-pattern :data '(a)))
        (c (make-instance 'verbal-pattern :data '(a b)))
        (d (make-instance 'verbal-pattern :data '(a b c))))
    (assert-true (matching-patterns-p a a))
    (assert-true (matching-patterns-p a d))
    (assert-false (matching-patterns-p d a))
    (assert-true (matching-patterns-p b b))
    (assert-true (matching-patterns-p c c))
    (assert-true (matching-patterns-p b d))
    (assert-true (matching-patterns-p c d))
    (assert-false (matching-patterns-p d c))))

(defun familiarise (model node pattern)
  "Extend image of node with a new item from pattern"
  (assert (matching-patterns-p (node-image node) pattern))
  (unless (equal-patterns-p pattern (node-image node))
    (incf (chrest-clock model) (chrest-familiarisation-time model))
    (setf (node-image node)
          (combine-patterns (node-image node)
                    (get-next-item pattern (node-image node))))))

(def-process-tests familiarise-tests ()
  "Use verbal patterns to test familiarisation"
  (let ((model (create-chrest))
        (node (make-node :contents (make-instance 'vl-pattern)
                         :image (make-instance 'vl-pattern)
                         :children nil))
        (pattern (make-instance 'vl-pattern :data '(B I F))))
    (assert-true (empty-pattern-p (node-image node)))
    (familiarise model node pattern)
    (assert-true
      (equal-patterns-p (make-instance 'vl-pattern :data '(B))
      (node-image node)))
    (familiarise model node pattern)
    (assert-true
      (equal-patterns-p (make-instance 'vl-pattern :data '(B I))
      (node-image node)))
    (familiarise model node pattern)
    (assert-true (equal-patterns-p pattern (node-image node)))
    (familiarise model node pattern)
    (assert-true (equal-patterns-p pattern (node-image node)))))
```

References

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebière, C., and Qin, Y. L. (2004). An integrated theory of the mind. *Psychological Review*, 111(4):1036–1060.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change.* Reading, MA: Addison-Wesley.

Beck, K. (2003). *Test-Driven Development: By Example.* Boston, MA: Pearson Education, Inc.

Biegon, D. (2006). Project store: Biosciences report. http://hdl.handle.net/1842/1414.

Bugelski, B. R. (1962). Presentation time, total time, and mediation in paired-associate learning. *Journal of Experimental Psychology*, 63:409–412.

Chase, W. G. and Simon, H. A. (1973). Perception in chess. *Cognitive Psychology*, 4:55–81.

Cooper, R. P. (2007). The role of falsification in the development of cognitive architectures: Insights from a Lakatosian analysis. *Cognitive Science*, 31:509–533.

Cooper, R. P. and Shallice, T. (1995). Soar and the case for unified theories of cognition. *Cognition*, 55:115–49.

Cragin, M. H., Palmer, C. L., Carlson, J. R., and Witt, M. (2010). Data sharing, small science and institutional repositories. *Philosophical Transactions of the Royal Society A*, 368(1926):4023–4038.

Curtis, B., Krasner, H., and Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, 31:1268–87.

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

Feigenbaum, E. A. (1959). An information processing theory of verbal learning. The RAND Corporation Mathematics Division, P-1817.

Feigenbaum, E. A. and Simon, H. A. (1962). A theory of the serial-position effect. *British Journal of Psychology*, 53:307–320.

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

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Reading, MA: Addison-Wesley.

Freudenthal, D., Pine, J. M., Aguado-Orea, J., and Gobet, F. (2007). Modelling the developmental patterning of finiteness marking in English, Dutch, German and Spanish using MOSAIC. *Cognitive Science*, 31:311–341.

Frias-Martinez, E. and Gobet, F. (2007). Automatic generation of cognitive theories using genetic programming. *Minds and Machines*, 17:287–309.

Gluck, K. A. and Pew, R. W., editors (2005). *Modeling Human Behavior With Integrated Cognitive Architectures: Comparison, Evaluation, and Validation.* Mahwah, NJ: Lawrence Erlbaum.

Gluck, K. A., Staszewski, J. J., Richman, H., Simon, H. A., and Delahanty, P. (2001). The right tool for the job: Information-processing analysis in categorization. In Moore, J. D. and Stenning, K., editors, *Proceedings of the Twenty-Third Annual Conference of the Cognitive Science Society*, pages 330–335. Mahwah, NJ: Erlbaum.

Gobet, F. (1996). Discrimination nets, production systems and semantic networks: Elements of a unified framework. In *Proceedings of the Second International Conference of the Learning Sciences*, pages 398–403. Evanston, IL: Northwestern University.

Gobet, F. (1997). A pattern-recognition theory of search in expert problem solving. *Thinking and Reasoning*, 3:291–313.

Gobet, F. (1998). Expert memory: A comparison of four theories. *Cognition*, 66:115–52.

Gobet, F. and Clarkson, G. (2004). Chunks in expert memory: Evidence for the magical number four... or is it two? *Memory*, 12:732–47.

Gobet, F. and Lane, P. C. R. (2005). The CHREST architecture of cognition: Listening to empirical data. In Davis, D. N., editor, *Visions of Mind: Architectures for Cognition and Affect*, pages 204–224. Hershey, PA: Information Science Publishing.

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

Gobet, F., Richman, H., Staszewski, J., and Simon, H. A. (1997). Goals, representations, and strategies in a concept attainment task: The EPAM model. *The Psychology of Learning and Motivation*, 37:265–290.

Gobet, F. and Ritter, F. E. (2000). Individual data analysis and Unified Theories of Cognition: A methodological proposal. In Taatgen, N. and Aasman, J., editors, *Proceedings of the Third International Conference on Cognitive Modelling*, pages 150–57. Veenendaal, The Netherlands: Universal Press.

Gobet, F. and Simon, H. A. (1996). Templates in chess memory: A mechanism for recalling several boards. *Cognitive Psychology*, 31:1–40.

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

Gobet, F. and Waters, A. J. (2003). The role of constraints in expert memory. *Journal of Experimental Psychology: Learning, Memory & Cognition*, 29:1082–1094.

Gravemeijer, K. (1997). Mediating between concrete and abstract. In Nunes, T. and Bryant, P., editors, *Learning and teaching mathematics: An international perspective*, pages 315–45. Hove, UK: Psychology Press.

Jones, G. A., Gobet, F., and Pine, J. M. (2007). Linking working memory and long-term memory: A computational model of the learning of new words. *Developmental Science*, 10:853–873.

Kuhn, T. (1962). *The structure of scientific revolutions.* Chicago: University of Chicago Press.

Laird, J., Newell, A., and Rosenbloom, P. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64.

Lakatos, I. (1970). Falsification and the methodology of scientific research programmes. In Lakatos, I. and Musgrave, A., editors, *Criticism and the growth of knowledge.* Cambridge: Cambridge University Press.

Lane, P. C. R., Cheng, P. C.-H., and Gobet, F. (2000). CHREST+: Investigating how humans learn to solve problems with diagrams. *AISB Quarterly*, 103:24–30.

Lane, P. C. R. and Gobet, F. (2003). Developing reproducible and comprehensible computational models. *Artificial Intelligence*, 144:251–63.

Lane, P. C. R. and Gobet, F. (2005a). Discovering predictive variables when evolving cognitive models. In Singh, S., Singh, M., Apte, C., and Perner, P., editors, *Proceedings of the Third International Conference on Advances in Pattern Recognition, part I*, pages 108–117. Berlin: Springer-Verlag.

Lane, P. C. R. and Gobet, F. (2005b). Multi-task learning and transfer: The effect of algorithm representation. In Giraud-Carrier, C., Vilalta, R., and Brazdil, P., editors, *Proceedings of the ICML-2005 Workshop on Meta-Learning.*

Lane, P. C. R. and Gobet, F. (2007). Developing and evaluating cognitive architectures with behavioural tests. In Kaminka, G. A. and Burghart, C. R., editors, *Proceedings of the AAAI-07 Workshop on Evaluating Cognitive Architectures*, pages 36–39.

Lane, P. C. R., Sykes, A. K., and Gobet, F. (2003). Combining low-level perception with expectations in CHREST. In Schmalhofer, F., Young, R. M., and Katz, G., editors, *Proceedings of EuroCogsci*, pages 205–210. Mahwah, NJ: Lawrence Erlbaum Associates.

Langley, P. and Choi, D. (2006). A unified cognitive architecture for physical agents. In *Proceedings of the American Association of Artificial Intelligence.*

Langley, P.; Laird, J. E.; and Rogers, S. 2009. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research* 10:141–160.

Licata, D. R., Harris, C. D., and Krishnamurthi, S. (2003). The feature signatures of evolving programs. In *IEEE International Symposium on Automated Software Engineering.*

Lubers, M., Potts, C., and Richter, C. (1993). A review of the state of the practice in requirements modeling. In *Proceedings of the International Requirements Engineering Symposium.* California, USA: Los Alamitos.

McLeod, P., Plunkett, K., and Rolls, E. T. (1998). *Introduction to Connectionist Modelling of Cognitive Processes.* Oxford, UK: Oxford University Press.

Medin, D. L. and Schaffer, M. M. (1978). Context theory of classification learning. *Psychological Review*, 85:207–238.

Medin, D. L. and Smith, E. E. (1981). Strategies and classification learning. *Journal of Experimental Psychology: Human Learning and Memory*, 7:241–253.

Milewski, B. (2001). *C++ in action: Industrial-strength programming techniques.* Upper Saddle River, NJ: Addison Wesley.

Myung, I. J. and Pitt, M. A. (2010). Cognitive modeling repository. In *Proceedings of the Thirty-Second Annual Meeting of the Cognitive Science Society*, page 556.

Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming*, 15:253–261.

Newell, A. (1990). *Unified Theories of Cognition.* Cambridge, MA: Harvard University Press.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall.

Pew, R. W. and Mavor, A. S., editors (1998). *Modeling human and organizational behavior: Applications to military simulations*. Washington, D. C.: National Academy Press.

Pitt-Francis, J., Benabeu, M. O., Cooper, J., Garny, A., Momtahan, L., Osborne, J., Pathmanathan, P., Rodriguez, B., Whiteley, J. P., and Gavaghan, D. J. (2008). Chaste: Using agile programming techniques to develop computational biology software. *Philosophical Transactions of the Royal Society A*, 366:3111–3136.

Ritter, F. E. (2004). Choosing and getting started with a cognitive architecture to test and use human-machine interfaces. *MMI-Interaktiv*, pages 17–37.

Roberts, S. and Pashler, H. (2000). How persuasive is a good fit? A comment on theory testing. *Psychological Review*, 107:358–367.

Royce, W. (1970). Managing the development of large software systems. In *Proceedings of IEEE WESCON 26*, pages 1–9.

Samsonovich, A. 2010. Toward a unified catalog of implemented cognitive architectures. In *Proceedings of the 2010 Conference on Biologically Inspired Cognitive Architectures*, 195–244. IOS Press.

Simon, H. A. (2000). Discovering explanations. In Keil, F. C. and Wilson, R. A., editors, *Explanation and Cognition*, pages 21–59. Cambridge, MA: The MIT Press.

Simon, H. A. and Gobet, F. (2000). Expertise effects in memory recall: Comments on Vicente and Wang. *Psychological Review*, 107(3):593–600.

Smith, J. D. and Minda, J. P. (2000). Thirty categorization results in search of a model. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 26:3–27.

Smith, R. L., Gobet, F., and Lane, P. C. R. (2007). An investigation into the effect of ageing on expert memory with CHREST. In *Proceedings of The Seventh UK Workshop on Computational Intelligence*.

Stott, W. (2003). Extreme programming: Turning the world upside down. *IEE Computing and Control Engineering*, pages 18–23.

Sun, R., Merrill, E., and Peterson, T. (2001). From implicit skills to explicit knowledge: A bottom-up model of skill learning. *Cognitive Science*, 25:203–244.

Waters, A. J. and Gobet, F. (2008). Mental imagery and chunks: Empirical and computational findings. *Memory and Cognition*, pages 505–517.