A Methodology for Developing Computational Implementations of Scientific Theories

Peter C. R. Lane
School of Computer Science
University of Hertfordshire
College Lane, HATFIELD AL10 9AB, UK
peter.lane@bcs.org.uk

Fernand Gobet
School of Social Sciences
Brunel University
UXBRIDGE UB8 3PH, Middlesex, UK
fernand.gobet@brunel.ac.uk

Abstract

Computer programs have become a popular representation for scientific theories, particularly for implementing models or simulations of observed phenomena. Expressing a theory as an executable computer program provides many benefits, including: making all processes concrete, supporting the development of specific models, and hence enabling quantitative predictions to be derived from the theory. However, as implementations of scientific theories, these computer programs will be subject to change and modification. As programs change, their behaviour will also change, and ensuring continuity in the scientific value of the program is difficult. We propose a methodology for developing computer software implementing scientific theories. This methodology allows the developer to continuously change and extend their software, whilst alerting the developer to any changes in its scientific interpretation. We introduce tools for managing this development process, as well as for optimising the developed models.

1 Introduction

Implementing a scientific theory as a computer program has become increasingly attractive to scientists in many disciplines. It is generally accepted that computational models offer a number of valuable features, including: making all processes concrete, supporting the development of specific models, and hence enabling quantitative predictions to be derived from the theory [10, 20, 21]. More broadly, we can speak of an *architecture* as a computer program which supports development of a range of computational models, or simulations, within a wide-ranging theory.

Cognitive science is one discipline where the development of overarching 'cognitive architectures' has been important in unifying the results from different sub-branches of psychology. For example, Soar [19] captures a range of phenomena from problem solving through to learning. Indeed, Newell [19] was the first to propose that this kind of unification could and should take place. He estimated that there are approximately 1,000-10,000 regularities of immediate behaviour which should be captured by a complete theory. This large number makes it clear that a theory must begin with certain phenomena, and then gradually extend its scope to cover all regularities; the architecture is the mechanism by which the current state of understanding of these regularities is maintained over time.

This paper accepts the importance of constructing theories explaining a range of phenomena, and assumes that these theories and models will be implemented as computer programs. We argue, though, that to date there has been no standardised *methodology* to support the continued development of unified theories, both as scientific theories and as working computer programs.

Some of the problems can be highlighted with cognitive architectures such as ACT-R [1] and Soar [19], which are successful as toolkits for creating models in a range of domains, often simulating human data. The first concern is that, to a large extent, the architectures have become libraries for developing models, with no mechanism for maintaining the connection between models, such as consistency of parameter values. The second concern is that there is no control of the empirical evidence in support of the theory when new versions appear; models which work in version N may or may not work in version N+1. Although we accept that certain practices in these communities alleviate these problems (such as providing default values for parameters), there is still no consistent, formalised methodology which can be identified by an outsider. The third concern is that the implementation *is* the theory – Soar is defined by its latest implementation. Even with the documentation, there is little chance of another modeller *replicating* the implementation of Soar, from scratch, to confirm all details of

how a particular set of empirical results was obtained [4].

To address these concerns, we begin by explicitly recognising that the implemented theory is a computer program, and that the theory, and hence the program, will change over time as we increase both our understanding of the theory and the demands to construct new kinds of models. Developing a computer program in an environment where the specification keeps changing requires a careful management of each new version, ensuring that what was important in the older versions is retained in the new. We propose managing this change with a testing framework tailored to scientific applications [13]. Another aspect of scientific theories is the role of parameters in fitting empirical data. Using our framework as a base, we also propose optimisation techniques based on evolutionary algorithms as a way to locate groups of models satisfying multiple constraints.

In summary, following our methodology requires a modeller to: (1) develop the implementation of a scientific theory as an *architecture* using a *scientific-testing methodology* [13]; (2) optimise the parameters of the architecture, using a suitable search technique on its models [9, 14, 16]; (3) use knowledge-discovery techniques to determine the important parameters in different domains [15]; and (4) use data-analysis and visualisation techniques to compare the performance of architectures from different theories.

2 Computational Modelling: Three Elements

Scientific knowledge is a notoriously complex entity, residing in the heads of scientists, their writings, the instruments used to probe and measure the external world, and formalisations used to explain and predict specific phenomena. In this paper, we are interested in the use of computer programs to implement scientific theories, so that simulations or *models* of phenomena may be constructed; the models may then be used to predict or otherwise explain particular patterns of behaviour.

We propose a three-level separation of scientific knowledge, into *theory*, *architecture* and *model*; see Figure 1 for how these three concepts relate to each other. We define an *architecture* as the *implementation* of a scientific theory. The architecture should contain processes capturing all of the key elements within the theory. There will be many *models* for a given theory. Each of these models will be based upon processes within the architecture, but extend upon them to make the architecture work within a specific domain. Some of the extensions may be experimental formats or particular methods of processing input data. Finally, we can see the *theory* as sitting above the architecture. Whereas the architecture and models are all concretely implemented as computer programs, the theory has a large verbal element, including all aspects of the scientists' understanding of their theory and its implementations.

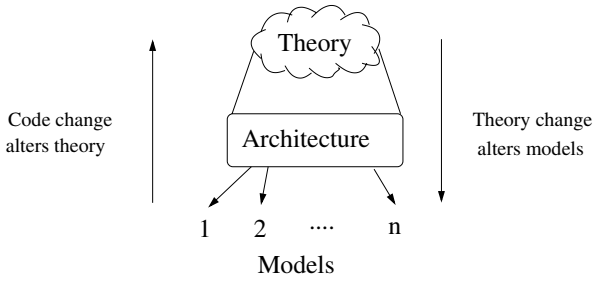

Figure 1. The relation between a theory, architecture and its models.

Our three-level separation also highlights the possible ways in which changes may occur in the theory. The arrows in Figure 1 illustrate that changes may originate from the theory at the top, or from the models at the bottom. Changes from the top will occur when the scientist decides to expand or alter the set of processes within their theory. For example, a cognitive architecture for memory may be expanded with mechanisms to scan visual images. These changes will necessitate the creation of new code in the architecture, along with possible modifications to existing code. Finally, new models will be developed to test the theory's new capabilities. Changes from the bottom can occur when the scientist looks at the implementation of their theory and decides that some of the processes present in the code, which are not articulated at the theory level, should become more prominent. For example, a model of human memory may need a process to compare the similarity of two patterns; similarity testing may not have been part of the theory, but the way it has been implemented may suggest appropriate theoretical constructs.

3 Scientific-Testing Framework

Our methodology recognises that the scientific theory is implemented as a computer program. Hence, we look at how software engineers establish the correctness and behaviour of their programs, paying particular attention to environments in which computer programs must meet changing specifications. Agile programming methodologies (e.g. XP [2]) provide a way to work with software in an environment with changing specifications. Test-driven development (TDD) is one component of such methodologies. TDD requires developers to provide tests for every piece of code which they create. Benefits include an immediate commitment to the description of the process, and concrete examples for each 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" [23].

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

Table 1. Three levels of tests

3.1 Definition

The first step of our methodology is a commitment to TDD: every component within the software will be developed with a set of *tests* designed to demonstrate its behaviour. We separate our tests into different groups, according to their role in the scientific theory; see Table 1.

We define three levels of tests, depending on the relation of the code and its test to the theory. Starting from the bottom, we have the *canonical results*: these indicate the extent to which the models account for the empirical results, and thus support the theory's validity. The canonical results demonstrate the behaviour of the theory in a specific phenomenon. For example, a theory of human memory may provide a model of the recall performance of expert chess players. The *process* tests demonstrate those parts of the software which are reflected in the theory's verbal descriptions. For example, the theory of human memory may define a process comparing two patterns together, determining their similarity. Finally, the *unit* tests confirm implementation details of the software.

A further comment is in order about the role, in the canonical results, of statistical tests in evaluating a model's fit to data. We anticipate that in some cases the work required to design and construct the statistical test will be considerable, and require the expertise of a trained statistician. However, once the nature of the test and the criteria for satisfying it have been determined, the process of applying the test may be automated; it is this last, automated process which forms the substance of the canonical result.

3.2 Implemented 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. 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, and a minimal burden on the programmer, when creating such tests.

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. Figure 2 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 reports any failures.

```
Running Unit tests: ..
Error 1: Expected 7 got 3.
Error 2: Expected 7 got 3. sorting error
...
=== DONE: There were 2 errors in 7 tests
Running Process tests: .
=== DONE: There were 0 errors in 1 test
Running Canonical results: .
=== DONE: There were 0 errors in 1 test
```

Figure 2. Sample output from test framework

Three groups of commands are provided for: defining individual tests, defining groups of tests, and running the 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 [message])`. The `value` is replaced by a call to some code, and a check of its correctness. The optional message provides better feedback in the event of a failing test. For example, the call `(test (= 4 (adder 2 2)) "adder")` would check that the return value of `(adder 2 2)` was equal to 4 – failure would give the message "adder". Some simplifying functions are provided, for example (`assert-true`) and (`assert-equal`).

The second group of functions separates tests into categories; it is this group of functions which is designed specifically to assist a developer using our test framework. An individual function or method will likely be tested with a variety of example data, and a collection of tests will be created for that example. We group these tests into a function, which will be evaluated as a unit. In Lisp, the usual way to define such a function is: `(defun function-name () (test ...) (test ...))`. Our framework replaces the generic `defun` with either of `def-unit-tests`, `def-process-tests` or `def-canonical-result-tests`, as appropriate.

The final stage is to evaluate the tests. The framework provides separate functions to evaluate tests in each category, as well as `(run-all-tests)` to run every test in the complete system. The example output in Figure 2 shows a typical result: note how the tests for each category are executed separately, and in turn. Errors are reported where they occur, and a summary of how many errors were detected provided. 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.

The key 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 onto a list ap-

propriate to the XXX in the definition. The functions which run the tests then look at these lists to determine which functions to run. (This framework is readily implemented for languages other than Lisp.)

3.3 Benefits of Test Framework

Comprehensible Theories A recurring complaint of computational implementations is that the theory cannot be understood, unless you are an expert programmer, and even then it is notoriously hard to understand another programmer's software. Use of a test-driven methodology removes this problem, because every aspect of the program is accompanied with a simple, well-defined test. Understanding the *intent* of each piece of the program is made easier by the concrete example; understanding the *mechanism* by which this intent is realised can be left to the programmer.

Reproducible Theories An important goal for any theory is for its empirical results to be reproducible by other scientists. A complex theory implemented as a large computer program (Soar contains around 50,000 lines of code) would require a dedicated researcher to attempt a reimplementation, but we should, as diligent scientists, support the possibility. The process and canonical-result tests provide just this support. Reimplementing the underlying architecture is permitted, but the new version must run the same set of (perhaps translated) process and canonical-result tests to be acceptable as an example of the same theory. (A real example of this was the reimplementation of the Soar cognitive architecture from Lisp into C.)

Developing Theories The tests play a key role in ensuring that each version of the software retains the scientific results and the relation to the expected theory of the previous version. There are two ways in which theories may be developed. First, new functionality may be added to the theory, so the architecture will support new kinds of model. As the architecture is altered, new code will be added, and probably existing code will be modified. The tests act as a safety net. Every time the code is changed, the tests are checked. If a test fails, the type of failing test tells us how to respond to the failure. A failing unit test is the simplest case to handle: unit tests have no relation to the implemented theory, and so can be removed or changed to suit the new circumstances. A failing process test means that something critical to our understanding of the theory has changed. Further modifications may be needed to restore the previous behaviour, or our understanding of this process at a theoretical level may need reviewing. Finally, a failing canonical-result test means that the empirical support for the previous version of our theory no longer holds. Documenting these changes between versions would provide a useful commentary on the theory's development, as well as pointing the way to critical areas of the theory requiring further work.

The second kind of development is more subtle. After writing the architecture and a few models, we may review the code and rewrite it, to improve its design: this process is known as *refactoring* [7]. As above, the tests act as a safety net, ensuring none of the earlier behaviour is broken; Fowler and other authors on this topic insist that refactoring is impossible without a complete suite of tests. Intelligent refactoring holds the promise of systematically improving the clarity of a computer program's design. From a scientific perspective, refactoring may lead us to change the status of certain processes within the software. For example, common processes in different models may be moved into the architecture, to remove code duplication, and from there we may want to give them theoretical status. If such a change in status occurs, then the tests would be moved from the group of unit tests to the group of process tests.

Important in both these cases is the role played by the tests in keeping track of all the prior understanding of how the software implements the target scientific theory. The process tests ensure that each process within the theory is reflected in the code, and the canonical-result tests preserve the empirical support for the theory in a readily reproducible format. As the theory develops, the tests will tell us, by their group, what the impact of any changes made has been to the developing theory. We now proceed to the next set of tools, which use the canonical-result tests to optimise the theory's models and parameters, and also to support semi-automated comparisons between theories.

4 Optimising Models

Any model created to capture the results in a single experiment will have a number of parameters, which may take a range of values. The performance of the model on each canonical result can be used as a target for optimising the parameter values. We formalise this process by defining a space $\mathcal{M}$ of models, based on the set of possible parameter settings; a specific model, $m \in \mathcal{M}$, is defined by a specific set of parameter values. Each canonical result is defined as a function, $f_i(m)$, which produces the fitness of a model for that task.

The presence of multiple constraints, f_i, makes the problem a multi-criteria optimisation problem. One of the key challenges is to define 'optimal', because two models may outperform each other on different constraints. Our aim is to obtain the *set* of models which are not worse in *all* constraints than any other model (sometimes known as the non-dominated, or pareto-optimal set). Formally, we say that model m_1 *dominates* model m_2 if m_1 does at least as well as m_2 in all constraints, and there is at least one constraint in which m_1 does better.

5 Comparing Theories

We similarly define the set of non-dominated models from multiple theories. We allow the space $\mathcal{M}$ of models to be the union of all the possible models from all the competing theories. Thus, a model, $m \in \mathcal{M}$, will be a specific set of parameter values for one of the classes of theories. As above, the problem is a multi-criteria optimisation problem, because there are multiple canonical results to fit. However, one problem of standard optimisation techniques is that the competition can lead to examples of inferior theories being driven completely out of the population; it is useful to see the very best that each theory could offer, so that the scientist, not the optimisation technique, could decide to rule out the theory.

We use a modified form of non-dominated sorting genetic algorithm (NDSGA) [14, 16] to locate the set of models which are not outperformed on all tasks (taken from categorisation experiments) by any other. Our approach uses a standard sorting process [11, 22], using fronts of non-dominated models to bias the creation of each new population towards the non-dominated models. We extend this algorithm (to form the *Speciated*-NDSGA) by using a separate population for each theory. The property of domination is computed across all the populations, but a complete population is maintained for every theory.

The chief benefit of our algorithm may be seen by considering one of the worst-case scenarios: what would happen if every model in one of the theories was dominated by models in another? In regular NDSGA, there would be no new models from this dominated theory. In our variation, a new population will be created, still preferring candidates from set 2 over set 3. Once evolution is complete, extracting the non-dominated models from all the theories may mean some theories have no models. However, we will still have evolved a set of models for that theory, and can analyze their performance. A related benefit is that some theories require longer to evolve good values for their parameters: our approach allows these theories to 'catch up', and produce non-dominated models at a later period, whereas NDSGA would have removed them from consideration.

We have developed some software which locates optimal sets of models drawn from multiple theories and tested against multiple experimental results. The software allows the user to visualise individual models, run optimisation experiments across single or multiple sets of data and single or multiple theories, and plot some graphs or output statistics on the performance of groups or individual models. In an initial study, we evolved models from four different theories on data from 29 categorisation experiments, and compared their performance with those found previously in the literature; we routinely found a 40% improvement on other published results [14], and employed other knowledge-discovery techniques to locate scientifically useful parameter values [15]. Although only a small study, these results suggest the potential of automated, large-scale optimisation of models for better understanding scientific theories.

6 Discussion

Adopting our methododology requires the scientist to make the implementation and its development part of the knowledge that makes up the scientific theory. We may ask how this relates to other aspects of scientific methodology. There is a tradition within cognitive science of using Lakatos [12] as a source [5, 6, 19]. Lakatos proposed that scientists develop a core set of hypotheses and elaborate upon these with some peripheral assumptions to develop concrete models. When a theory is challenged, it is the peripheral assumptions which are modified, not the core. The separation proposed here is consistent with the Lakatosian framework, as the architecture represents the core hypotheses of the scientist's theory, and the models encapsulate the peripheral assumptions; however, nothing that we propose constrains a scientist to adopt any particular methodological position, beyond a commitment to representing theories in the form of computer programs.

We have argued that using a scientific-testing framework enhances the implementation of a scientific theory, making it easier to reproduce and understand. An alternative technique is to use prescriptive definitions, such as formal specification languages or a tailored development environment [3], to define our program. The advantage of a formal language, such as Z [17], is that an implementation may be proven to meet the specification. Version 5 of the Soar cognitive architecture was formally defined in this manner [18]. However, there are several problems in using such a framework for scientific theories. The first is that a formal framework provides a barrier to non-specialists wanting to understand the implementation. The formal notation must be understood, and so provides few, if any, benefits in comprehending the theory over the computer program itself. The use of process tests, as proposed here, provides an easily understood set of demonstrations of the implementation's behaviour. The second problem is that we assume the scientific theory to be in a constant state of flux. We anticipate any evolving theory to continually add new phenomena and processes. It is easy to check whether a new version of our implementation meets prior results using our testing framework, but hard to do so with a formal specification. Indeed, the formal specification of Soar mentioned above does not appear to have been updated, even though Soar is now at version 8. Similar objections apply to tailored environments, such as [3].

The role of canonical results is not restricted to an individual theory. As in the section on comparing models from

different theories, a canonical result represents a common target, an important empirical result, for theorists and modellers wanting to demonstrate the value of their own theory. An extension of this idea is to use the canonical results as a target for the automatic generation of theories. In unrelated work, Frias-Martinez and Gobet [8] have shown how theories may be generated automatically using genetic programming. The quality of the theories is judged based on their fitness to a standard set of experimental data – this standard data, in our terminology, would form the canonical results of the resultant theories. A related idea is to use genetic programming to construct alternative models from a single architecture to suit a set of canonical results.

7 Conclusions

This paper has described a methodology and a set of supporting tools to aid a scientist when using computer programs to implement and test theories. The methodology is based around the use of tests to confirm the behaviour of a running computer program. These tests are subdivided into groups, based upon their importance to the theory or the scientific domain being modelled. With these mechanical tests in place, the scientist is free to modify their implementation or theory, with the tests providing feedback on whether the modifications have affected the previous understanding or results. The second stage of the methodology uses one group of tests, the canonical results; we show that models can be optimised using automatic techniques, and that the optimisation should proceed across as many canonical results as possible. In addition, we have provided a customised optimisation technique which enables a scientist to develop and compare models drawn from multiple theories on the same set of canonical results.

We propose that the general picture of scientific theory construction and development makes our methodology applicable to many other domains in computational simulation and modelling. Both stages of our methodology, the scientific-testing framework and the optimisation techniques, are supported by software tools; see:
`http://homepages.feis.herts.ac.uk/~comqpcl`

References

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

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

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

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

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

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

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

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

[9] F. Gobet and P. C. R. Lane. A distributed framework for semi-automatically developing architectures of brain and mind. In *Proceedings of the First International Conference on e-Social Science*, 2005.

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

[11] D. E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

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

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

[14] P. C. R. Lane and F. Gobet. Applying multi-criteria optimisation to develop cognitive models. In *Proceedings of the UK Computational Intelligence Conference*, 2005.

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

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

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

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

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

[20] A. Newell and H. A. Simon. *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall, 1972.

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

[22] N. Srinivas and K. Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2:221–248, 1994.

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