Languages for Biological Models: Importance, Implications and Challenges - A Work In Progress

Luke Church, Katinka Apagyi, Jasmin Fisher

Computer Laboratory, University of Cambridge, luke@church.name
Department of Biochemistry, University of Cambridge, ka312@cam.ac.uk
Computational Biology Division, Microsoft Research Cambridge, jasmin.fisher@microsoft.com

Keywords: POP-I.C. Biology, POP-I.B. Debugging, POP-I.B Reverse engineering, POP-VI.F Exploratory

Abstract

In this paper we outline a new kind of challenge for the Psychology of Programming research community: how do we build programming languages to support modelling biological systems? We argue that such systems tend to be tightly coupled, partially understood and highly complex, and as such rather similar to modern software. We consider design challenges of such systems, including designing for reverse engineering, how to assist social processes and the importance of translucent abstractions. Finally, we consider how answers to these challenges may assist in the design of other domain specific languages.

Introduction

There's a lot of similarity between aspects of modern biology and engineering large software systems. Both are dealing with a great deal of complexity, both deal in the characterisation of some internal process (one biochemical, one digital) and in both cases the processes are described in an information structure (DNA, source code). In many cases we're more interested in what the code *does* than in the code itself. This is certainly true of software, and is increasingly true of biological systems; progress in sequencing techniques has made the raw data more easily available, what it hasn't done is characterised what those data *mean*.

There are also a number of differences between software engineering and biology. For example: software engineering has traditionally been seen as a discipline of construction. It addresses questions of how one can create code to achieve a desired behaviour. In contrast, most of biology is more concerned with understanding than construction. Typical challenges include characterising a behaviour in terms of how and why it occurs, rather than constructing new behaviour.

However the picture isn't as clear cut as that. Most software isn't written from scratch. Very few pieces of software indeed exist without a supporting apparatus of other software. The software engineer's task is not only to create new software, but to characterise, or reverse engineer, the behaviour of existing software in order to either interact with, or modify it. This characterisation, is a time consuming endeavour, and can be viewed as being at least partially responsible for Brooks' (1975) adage: 'adding resources to a late project, makes it later'. The cost of the engineer understanding the existing software before they can begin productive work is just too high.

So, biology and software engineering seem to share a problem in common. How do we characterise and understand the behaviour of complex systems, be they '*in silico*' or '*in vitro*'? Furthermore, what can tool support do to assist in this endeavour?

Where are we at?

The awareness that there were going to be substantial deviations between what computers were intended to do, and what they did came very early on in the history of computing:

> *"As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs." - Maurice Wilkes, 1949*

Since then, tools for characterising behaviour, such as debuggers, have made substantial progress. Yet the intellectual challenge of understanding how a program works, that is reverse engineering, is still something of a minority field. It is often associated with criminal applications of the technique, such as software piracy. However, we would argue that it is a key part of what a professional software engineer does, especially whilst debugging. (Robbins 2002, Eilam 2005)

The dominance of forward engineering, or the task of constructing software, has resulted in an apparatus of tools designed to create 'knowledge barriers'. A good example of this is the use of *implementation hiding*. Languages such as Java suggest that interfaces be used as 'contracts of behaviour', the idea being that the programmer that is calling a method that implements the interface, does not need to concern themselves with how that method is implemented, only what its outcome is.

Whilst abstracting away from the implementation is undeniably useful in dealing with complex systems, we argue that there are limits on what can be achieved this way. In theories of emergence very complex behaviour can arise out of simple operations (Holland, 2000). As such even the smallest change of a hidden implementation may change the behaviour of a program. An extreme example is that security has been unable to prevent 'covert channels' in which information from the internals of software is 'smuggled out' to an observing process. Even after enormous research, the NSA recommends limiting covert channels, rather than eliminating them completely. (NCSG, 1993)

There is the further more subtle problem that we shall refer to again later, that when one hides an implementation, one makes a judgement as to what is considered to be the important behaviour. For example, programmers are taught that they should not assume that a hidden implementation is going to take the same amount of time in the next release that it does in the current one; precisely because the implementation may have changed. This perspective views the result as important, but not the amount of time taken to compute it. However, it is necessary for developers of performance sensitive applications (of which there are surprisingly many), to be aware of not only the desired result of calling a method, but also more subtle properties like how long it will take. Tools are used to do this; code profilers track 'hot-spots' that appear to be taking a long time, reflective analysis tools and decompilers are used to map the behaviour of APIs. There is a slowly growing acceptance even within the authors of such APIs that such tools are needed, with for example Anderson (2007) advocating the use of such a tool on a framework that he designed.

So acceptance is growing that, whilst abstracting away from implementations is necessary for managing large problems, it is also necessary to be able to peer inside these abstractions from time to time. This growing maturity of understanding of the role that abstraction has to play can be seen as typifying the limitations of reductionism in understanding complex systems. There is never just one level of abstraction that is ideal for looking at a system. Indeed, the most reduced form of software is machine code, but this stream of commands says little about the overall purpose of the program.

Biology

Some of the modern scientific approaches to biology share this sentiment. For example:

> *"The reductionist approach has successfully identified most of the components and many of the interactions but, unfortunately, offers no convincing concepts or methods to understand how system properties emerge..." – Sauer, U. et al. (Science, 17th April 2007)*

Hence there is a need to begin building an understanding of how to support this important aspect of complex system comprehension; the ability to view a system at multiple different levels of abstraction. However, despite these fundamental similarities there are a number of differences between software engineering and biology that add to the challenge.

Biology, like any natural science, studies a partially known domain. For example, whereas we can be reasonably confident about the computational model of a simple processor, we are only beginning to see glimpses of the computational model of a cell, let alone an entire biological system. We find plenty of indications that the respective models are very different.

There are also practical differences. In programming, the entity that is being analyzed is generally 'available to the computer'. By this we mean that software can analyse other software running on the same computer, so the source of the data is close to hand. This has various effects, such as being able to manipulate the flow of time in very sophisticated ways; a biologist would love to be able to insert a break-point into a cell and non-destructively analyse what was happening! Instead, biologists often have to use indirect inference, based on analysing parameters extracted from the process (e.g. to measure the length of a DNA fragment, one can't count the base pairs, but rather a 'gel' is run, where the fragments form bars which are compared to fragments of know length, acting as a 'ladder'). Further there is the problem that such experiments are remarkably expensive in terms of time. Performing a series of molecular cloning steps (a standard preparation step of many experiments) may take several weeks, whereas a software bug that takes weeks to reproduce is a thorny one.

The data in biological systems tend to be very noisy. Part of the training of a biologist is to develop an instinct as to which phenomena are important to investigate and which aren't. Without this skill, a biologist working on a partially characterised system would find themselves forever side-tracked into understanding side conditions, or experimental anomalies. This is in contrast to the comparatively low noise data sources that are available to the programmer, such as variable readouts.

Finally, there is the nature of the entity under consideration. Naturally occurring biological systems are (generally) accepted to have arisen via a process of evolution, as opposed to human design. This means that the way the complexity is organised is very different. As we shall see shortly, the lack of human imposed design may be responsible for some of the differences between the social processes that occur in the biology and software engineering.

Social Processes and common vocabularies

Post-modern theory argues that knowledge is socially constructed (Wikipedia, 2007). An example that will shape some of our design requirements is De Millo et al. (1979) who argued that proofs in mathematics are accepted or rejected on the basis of the social processes, rather than deductive logical reasoning. A central aspect of their critique of verification proofs of software is that they are of a form where social processes cannot be applied, vastly decreasing the value of such proofs.

Expert software teams seem to be aware of the social nature of knowledge and often explicitly construct mechanisms to support these social aspects. For example (Whitaker, J. Personal communications, 2006) adopted a 'bug tales' process whilst working for a company performing security testing. At the end of each day, the testers would discuss any interesting behaviours they had found, which they thought might be worthy of investigation.

There is further important point to note here: As discussed above, man-made artefacts have structures implicit in them from the tools that were used to create them. For example in object orientated software, the code is clustered into methods and bound with its data into objects. Many of the social processes of reverse engineering are centred on these constructions. Websites such as dotnet247.com (Barker, 2008) contain lists of the .NET API and discussions around each of the methods. Many of these discussions are considering effects of the 'hidden implementation', such as under what conditions to exceptions arise from the `MeasureCharacterRanges` method.

The lack of such man-made structures and their naming schemes causes problems with the social structures of biology. Typically the problem occurs that multiple people working on the same system, say a transcription product of a gene, give the product different names. This can inhibit discussion until a standard is selected. Further issues occur around the use of diagrams; biologists are typically interested in dynamic systems and make heavy use of diagrams to communicate about such systems. These diagrams have evolved rather than being designed and standardised. This has a number of interesting effects, such as allowing the creators of diagrams to pick rather arbitrary abstraction levels

to suit the question in hand. Let's look at one of the diagrams and associated legend. There is no universal standard for such diagrams and they rely heavily on implicit semantics, agreed by social convention. To highlight this point, some of the implicit semantics have been informally overlaid.

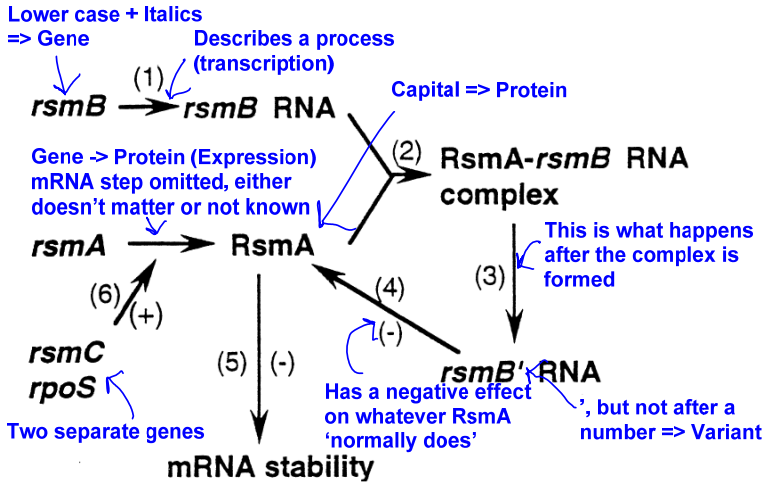

Fig. 8. A tentative model depicting the RsmA–*rsmB* regulatory scheme controlling mRNA stability. The proposed model postulates the following events: the transcription of *rsmB* produces the primary RNA (step 1). *rsmB* RNA binds RsmA (step 2), thereby depleting the pool of free RsmA. The formation of RsmA–*rsmB* RNA complex (step 3) creates a favourable ribonucleo-protein conformation facilitating RNA processing. *rsmB'* RNA, the processed product, negatively controls RsmA levels (step 4) by affecting transcription and/or translation of *rsmA* or turnover of RsmA. The decrease in RsmA pool contributes to the stability of mRNA species, such as $hrpN_{Ecc}$, *pel-1, peh-1 and ohll* transcripts (step 5). *rsmA* expression also is positively controlled by *rsmC* and *rpoS* (step 6).

Figure 1 - Diagram and Caption from Liu et al. (1998), additional, implicit semantics overlayed in blue (not in original)

It should be stressed that this is in no way critical of the particular diagram, or even of the process, it is rather just to aid consideration of the effects of the use of such diagrams. The ability to choose the abstraction level presents an interesting trade-off: It allows key concepts to be communicated, despite large amounts of the system being unknown, but in doing so, it does not highlight to the users of the diagram that this information is unknown. This may result in assumptions being made that aren't justified. For example, the arrows on the above diagram that describe up regulation or inhibition of processes in the system: How fast does this process occur? Does it occur proportionally? What outside influences apply? None of these questions are answered in the diagram. Note that this is a trade-off; even if such information was known, the inclusion of it all on the diagram would decrease *Hidden Dependencies*$_{<CD>}$ but increase *Diffuseness*$_{<CD>}$. (Note we are using The Cognitive Dimensions framework (Green, 1989) and Edge and Blackwell's notational form (2006))

As we shall see later, this combination of imprecision, implicit semantics and complex social constraints makes the introduction of computer support challenging. Let us first look at some reasons why we might want to do so, and then consider the difficulties.

Computation for modelling and communication in domain specific languages

This section will outline potential advantages of the introduction of computing to the process of biological modelling. Whilst biological system characterisation will remain our primary example, we believe that our observations have implications for other domain specific languages. This section is speculative, and represents a set of research aims and hypothesises than a completed agenda.

Modelling

The first reason to introduce computing is to support modelling. As suggested above, biological testing is incredibly time consuming and therefore expensive. A talented software tester working with mature tools might take a couple of minutes to set up a test. An analogous test in a biological system might take a talented researcher a couple of *weeks* to set up. As such, the number of tests that can be performed is low. It may take months of work to answer a simple question.

If we had a language in which we could model the processes, and associated tools for interrogating the model, we may be able to ease the problem. For example, if we had some hypothesis about a gene knockout (disabling a gene) and its effect on the system we may be able to simulate experiments as to

what would happen to the system with and without the gene to confirm in advance that the results are likely to be sufficiently different to make it worth while doing the experiment '*in vitro*'.

More sophisticated use of modelling may involve using a computer to verify whether a model can explain all of the data gathered from experiments, this has already been used by Fisher et al. (2007) to discover a new biological characteristic of *C. elegans*.

Generalisation over organisms

Computational models also offer easy generalisation. Biology has centred on the extensive characterisation of a number of 'model organisms'. A common research approach is: 'I see some evidence that suggests that mechanism x from a model organism explains the phenomenon that I am investigating. Let's look for similarities and differences between the mechanism in my organism and the model organism'

A computational model would assist in this process by allowing the data extracted from the organism under study to be quickly checked against the computational model for the model organism. This poses an interesting design constraint on the modelling language: We wouldn't want the modelling environment to just reply with 'CONSISTENT' or 'INCONSISTENT', but rather with an explanation of the consistency. This requires that the biologist have an understanding, all be it an approximate one, of the checking procedure so that they can determine the severity of a reported inconsistency.

This is a difficult problem in many modern programming languages, where compiler error messages only weakly correlate with the cognitive cause of the problem, but one that will become increasingly important to domain specific languages, as the average technical expertise of the programmer decreases and they think more in the domain that is being modelled as opposed to in the computer's domain.

Communication

If we're going to perform generalisations across organisms, we're going to need up to date computational models of model organisms, this will involve communication of such models. However the importance of communication in science should not be understated, if a tool increases the generation of information, but does so in a form where the data cannot be communicated, the tool might as well not exist. As such, we should look for ways in which to build tools that facilitate communication, we believe that a computational modelling tool might do just that.

A computational model may allow fuller hypotheses to be communicated; instead of a static description of a hypothesised process, a dynamic implementation can be shared. Biologists can then perform 'exploratory understanding' on the behaviour of this 'executable hypothesis' to understand some of its characteristics, without having to run costly lab experiments. Furthermore, it may be possible to share such models far more frequently, allowing improved communication between labs.

Consider an example where two separate laboratories are working on the same mechanism in an organism, and are maintaining computational models of the mechanism. If, at the end of every week, each group could check their new data with the hypothesised mechanisms the time lag could be shortened from the multi-year publication cycle. Further, the model could provide a common vocabulary between groups for discussing the mechanism.

Technical and Psychological Challenges

We are under no illusions that these aims are easy to achieve. They would require new forms of programming languages with explicit design support for comprehension and communication, easy and rapid modification and dealing with partially understood domains. However the languages also need to be of a form which is accessible to computers so they can assist in the modelling process by simulation or verification. Further, to support adoption the languages need to be easy to use for biologists, rather than computer scientists. Let us briefly look at some of the challenges associated

with designing such languages. We believe that many of these problems generalise to other domains and as such may help motivate design of languages for other applications.

Abstraction and Domain Specificity

We use programming languages to instruct the computer to perform tasks on our behalf. The Cognitive Dimensions tell us that the notations and environments make some tasks easy or hard. As a result of this there are a number of different programming languages and tools in everyday use, each suitable for its specific purpose. For example, biologists will often use Excel for data analysis, but Perl for genomic data processing.

Similarly, there are lots of different modelling tasks that we may wish to perform. For example, the authors have worked on problems including modelling inter-cellular communications (related to Fuqua et al, 1994), understanding the processes that shape the fate of cells (Fisher et al, 2007), and characterising where proteins migrate to inside cells. These are all different tasks, and as such we can probably achieve better usability by designing multiple modelling languages and tools.

This argument is by no means unique to biological systems and is one of the key arguments in favour of domain specific languages. There is an interesting tradeoff in that too many different languages would harm communication, whilst too few would result in increased difficulty for any given modelling domain. We believe that characterising this set of tradeoffs is an interesting problem that is relevant both to the design of domain specific languages, as well as more subtle cases such as the customisation of existing languages with frameworks. It also increases the pressure on making such languages and tools easy and quick to learn.

Formality

Computers are incapable of being informal. Ambiguity can only arise from the manner in which humans instruct computers to perform tasks. Nardi (1993) argues that this is a principle cause of the failure of conversational programming forms, but observes that it doesn't preclude end user programming. She observes that humans are capable of interacting with formal systems in real life, it's a matter of how user-friendly the formalisms are. For example spreadsheets are in more wide-spread use than 'conventional programming languages' and yet are an example of a formal notation.

This results in a number of challenges. Formal systems to date have not been particularly helpful in the way in which they deal with unknowns. Consider the earlier diagram, much of its semantics were imprecise, but because it was being interpreted by a human, the message could still be conveyed.

Software to date has treated imprecise information somewhat differently. Because software generally cannot reason without perfect information, it has a tendency to either fill the gaps with assumptions that might surprise even the programmer (e.g. initialisation of undeclared variables in C++ to random junk in the memory), or to refuse to work at all until the information is supplied.

For biological tools to be useful, some level of ambiguity must be accepted, as the universe is not perfectly known. Therefore the computer must be able to reason with ambiguity. The language must be designed very carefully in this respect, to prevent the user's mental model diverging from what the computer is actually. This is especially true if such computational models are to be used for inter-group communication.

We believe this challenge is general to many domain specific languages, as users will wish to model systems that contain elements of ambiguity that humans are accustomed to dealing with in every day life.

Design determinism and error models

As we have mentioned at several points throughout this paper, abstraction systems in computing are human designed and require that entities be modelled in a particular way. In software systems that have undergone long term use, designed abstraction starts to look like a less appropriate model.

Overtime the coupling tends to grow, and subtle side effects emerge, leading to the truism that ‘everything points to everything’.

It is of little surprise that the same issues affect biology, but generally to a much larger degree. Systematics, the attempt to classify organisms has been hugely problematic, even basic definitions, such as what is meant by a species, are not agreed upon. Similarly, evolution has resulted in an immensely coupled system, and one that often exhibits very subtle emergent side effects, such as the biochemistry of a cell changing depending on the number of similar cells in the vicinity.

It is unlikely that we will be able to design perfect abstractions with which to model this world and the choice of abstraction imposes a view onto the world, and limits what we can and cannot understand. For example the abstractions in the diagrams previously enabled us to understand some of the macro behaviour, but not the micro behaviour of how it happened.

This design determinism increases the requirement for multiple different domain specific languages for modelling different mechanisms, but also means that we need a way of systematically understanding the blindnesses that any particular abstraction framework will impose.

Such error models are in still in their infancy for understanding the effects of programming languages. By analogy to C#, the kind of problem we need to understand and predict is not that people forget to call ‘`dispose`’, but rather what effects does modelling a car in a single-inheritance framework cause?

Furthermore, we suggest that due to the tight coupling of biological systems, and the low likelihood of creating matching abstractions, we need more transparent abstractions so it is easy to look inside them if it is suspected that there is a mismatch between the abstraction and the biological reality.

We believe that both these requirements are important in designing end user programming languages generally, not just biological ones.

Adoption

The final challenge is how to encourage adoption of the tools? Attention investment (Blackwell, 2002) effects are substantial. The tools need to offer benefits that are proportional to the effort invested in using them, rather than requiring long learning periods before conferring any benefits, otherwise they will simply not be adopted.

This is particularly the case when producing tools for users who aren’t accustomed, in the same manner as programmers, with having to be patient whilst learning how to use tools in the knowledge that the benefit will come further down the line. Further such users are potentially less willing to work around problems caused by their toolset.

Again, such properties seem to be important for domain specific languages generally. To achieve adoption such tools have to offer a more gentle attention investment profile than many current programming languages.

Conclusion

We have seen that some of the problems biologists deal with are closely related to problems that software engineers face, especially engineers who are working within a large and unknown software environment. We believe this is a fairly common case.

Therefore we are beginning to frame the kind of programming language that we need to be able to design when comprehension and characterisation of existing systems becomes as primary a task as creating new code. We believe that such systems may help to alleviate the current problems of debugging, reverse engineering and systems integration. Further we have started to characterise some of the properties that will be important for domain specific programming languages intended for end user programmers.

We believe that these problems are central to the task of creating programming environments for use in modelling biological systems. We need programming languages whose abstractions offer smooth attention investment tradeoffs and are highly *role expressive*$_{<CD>}$. Further, they must be sufficiently precise that they can be expressed in a way that a computer can do useful things to assist in the modelling process, yet allow intuitive mental models to be constructed. Further the environments must offer low *viscosity*$_{<CD>}$ and high *progressive evaluation*$_{<CD>}$ to support exploratory design. Finally, as the program, not its result, is the important artefact they must be easy to communicate and aid existing social processes.

That sounds like a nice challenge, and one that is potentially important in the design of many modern programming languages.

Acknowledgements

We thank Alan Blackwell and Thomas Green for their valuable comments. Luke's work is supported by The Eastman Kodak Company. Katinka's work is supported by the BBSRC.

References

Anderson, C. (2007) Essential Windows Presentation Foundation, Addison Wesley, Reading, Mass.

Barker, M. (2008) .NET 247, http://www.dotnet247.com/247reference/default.aspx, visited 13/01/08

Blackwell, A.F. (2002) First step in programming: A rationale for Attention Investment models. In Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments, 2-10

Brooks, F.P. (1975) The Mythical Man Month and Other Essays on Software Engineering, Addison Wesley, Reading, Mass.

De Millo, R.A., Lipton, R.J. and Perlis, A.J (1979) Social processes and proofs of theorems and programs, Volume 22, Issue 5 of the Communications of the ACM, 271-280

Edge, D., Blackwell, A. (2006) Correlates of the cognitive dimensions for tangible user interfaces

Eliam, E. (2005) Reversing: Secrets of Reverse Engineering, John Wiley & Sons, NY

Fisher, J., Piterman, N. Hajnal, A. Henzinger, T.A. (2007) Predictive Modeling of Signal Crosstalk during *C. elegans* Vulval Development, PLoS Computational Biology. 3(5):e92

Fuqua W.C., Winans S.C. Greenberg, E.P. (1994) Quorum sensing in bacteria: the LuxR-LuxI family of cell density responsive transcriptional regulators. J Bacteriol; 176: 269-275

Green, T. R. G. (1989) Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay (Eds.) *People and Computers V*. Cambridge, UK: Cambridge University Press, pp 443-460

Holland, J.H. (2000) Emergence: From Chaos to Order, Oxford University Press, Oxford

Liu, Y., Cui, Y., Mukherjuee, A. and Chatterjee, A.K. (1998) Characterisation of a novel RNA regulator of *Erwinia cartovora* ssp. *cartovora* that controls production of extracellular enzymes and secondary metabolites, Molecular Microbiology 29(1), 219-234

Nardi, B.A. (1993) A Small Matter of Programming, MIT Press, Cambridge, MA

NCSG (1993), A Guide to Understanding Covert Channel Analysis of Trusted Systems (Light Pink Book), NCSG-TG-030, United States Department of Defence, Rainbow Series

Robbins, J. (2002) Debugging .NET and Windows Applications, 2nd Edition, Microsoft Press, Redmond, WA

Wikipedia (2007) Social Constructionism, http://en.wikipedia.org/wiki/Social_Constructionism, visited 13/01/08