Pitfalls of Formality in Early System Design

David Robertson
Division of Informatics, University of Edinburgh, Edinburgh, Scotland
Tel: +131-650-2709, Fax: +131-650-6516, Email: D.Robertson@ed.ac.uk

Abstract

One of the advantages of using formal methods in design should be that we can be precise about where our methods fail. However, it is rare to find discussions in the literature of problems in applying formal methods - particularly in the early stages of design. One reason for this is that failures are often caused by the context in which a method is applied, rather than by some purely technical limitation. Using examples from research in which I have been involved I shall describe some of the pitfalls I have encountered and which I have observed frequently in the research of others.

1 Introduction

Berry, in these proceedings [1], advocates the use of appropriately chosen and applied formal methods in the early stages of design lifecycles - observing factors outside the technical considerations of the methods themselves which influence their success. This paper is complementary to Berry's discussion because it looks at some factors which can lead to failure. It has been unfashionable for those using logic-based methods to introspect about causes of failures. This is a pity because without the possibility of failure our research isn't experimental and without the ability to learn from failure we are unlikely to develop robust engineering methods.

To avoid being accused of preying on other unsuspecting researchers, I have taken all the examples of pitfalls from my own research and where others have been involved I have generalised the examples. In none of the projects I describe were the mistakes fatal but this was largely because they were small scale, which made corrections simpler to make. Larger research efforts might not have the manoeuverability to fix these problems at acceptable cost. Once explained, some of the pitfalls may seem obvious but, given how frequently signs of these can be found in the research of others, I suspect that they are much easier to recognise with hindsight than to predict.

In what follows, I use "model" to refer to the set of logical expressions used to describe some problem. This is used instead of the word "specification" because in early design we often use models of problems which are not directly specifications of systems (see [11] or the introduction of [12]). I have also used the phrase "inference system" in a broad sense to denote any system which has been built with the intention of using or synthesising expressions in a mathematical logic.

2 Choice of Inference System

Formal definition takes place within a chosen system of inference. Although in theory there is great deal of overlap between inference systems, so that it should be easy to translate definitions from one to another, in practice our choice of system emphasises particular features of the problem. For example, a modal temporal logic is a natural choice if we are tackling a problem where we need to prove that certain propositions eventually hold of our system model. However, those inference rules which allow us to deal readily with the temporal aspects of problems may be a distraction in problems where we don't need to prove temporal properties. This issue is well known and is being addressed, in part, by those whose interest is in making it easier to combine or translate between inference systems. There remain some subtle difficulties which cannot be solved simply by deepening formal theory.

There are hundreds of specialised formal calculi, for describing certain forms of uncertainty; for expressing temporal information; *etc.* If we are lucky enough to have a problem which obviously suits a particular class of calculi then we may reduce our choice to a few "front runners" but we may have to dig deeply into the theory of each competitor before we can decide on a winner. For example, in [5] it is demonstrated that any problem representable in Dempster-Shafer theory can be represented in the Incidence Calculus and *vice versa.* Unwary readers might be led to believe by this result that the two calculi are "equivalent". This would be a mistake because in [10] a version of Incidence Calculus is described which for some situations involving dependent evidence gives the intuitively correct answer when Dempster-Shafer theory does not. In other words, the comparison between these two inference systems depends on exactly which version of each system we are comparing and on what sort of equivalence we wish to demonstrate. It is hard to understand such comparisons deeply without becoming an expert in all of the inference systems one might want to use - a prohibitively time consuming task since domain problems do not always neatly fit into a small class of related inference systems - but without deep knowledge we can easily misjudge the capabilities of the systems we intend to use.

One reaction to the problem of selecting a specialised logic is to begin projects with a general-purpose logic, perhaps with a commonly accepted computational interpretation. For instance, we might use Horn clauses with the resolution-based inference strategy familiar to logic programmers. We would then write all our formal expressions in the form : $C \leftarrow P_1 \wedge \ldots \wedge P_n$, where C is the (atomic) conclusion and each P is an atomic condition with the special condition, *true*, being used in cases where C is true unconditionally. What happens if we find that we need to represent concepts such as temporal change within our model? There are at least two technical solutions:

- Use the same inference system but add axioms which deal with temporal effects. An example is the Event Calculus [8].
- Use a different inference system containing proof rules which deal with temporal change. An example is the Temporal Logic of Actions [9].

Whichever solution is chosen, there is a cost to the extension which is not purely technical: it is necessary to negotiate the extension to the logic with others working on the project and to re-train them in the tasks which they must now perform in the new style. The reason why this can become a serious problem in early design is that the problems we are modelling often are loosely bounded and it is easy to find complex logical puzzles within them. Why stop at a temporal extension when there are also interesting issues of typing, higher-order function application, *etc* which we can find in the problem if we stare hard enough? Such problems fascinate the logicians in a project so it is easy to slide down a "slippery slope" of increasing language complexity, incurring a cumulative cost in negotiation and education in addition to the normal technical overheads.

3 Boundaries of Formality

Most uses of formality in early design require some degree of interaction with humans in a domain of application. They may be manipulating formal expressions in order to describe a model or assisting in the knowledge acquisition needed to build a domain-specific synthesis system or analysing information deduced from a model during validation. Whatever the point of interaction and regardless of the sophistication of the interface to the formal system, it will be necessary to commit to a human interpretation of at least some of the formal symbols manipulated by the inference system. A system for deciding on the mappings between human and formal language in some domain is often called an ontology. To work perfectly, everyone who needs to use an ontology must follow the same conventions in relating human to formal representation. There are two surprises here: the first is how deep differences in interpretation of ontologies can be; the second is that usable systems can be produced even when such differences occur.

In the knowledge based systems community, ontological research has become a major research theme (see [14] for a survey). The most common strategy is to fix on a bounded domain of application and devise a restricted formal language which those working in the domain use to describe problems. This shared language is used as the basis for sharing information. In our ecological work [13] we attempted to do this for a class of ecological modelling problems - the idea being to use the domain ontology to provide a language in which ecologists describe problems and then to use the formal problem description to control the generation of appropriate ecosystem models. This meant that we had to worry about what words like "biomass" should be taken to mean. Depending on who one speaks to, the answer to this question is different. The most crude definition is "mass of biological material". A more precise definition might be "mass of biological material once all water has been removed". For specialist sub-domains the definition might be "mass of biological material which has been subjected to the following treatment to remove water...". In other words, there is no broadly applicable definition of basic concepts like "biomass" which suits all situations. The search for consensus on such issues quickly took us into waters which were uncharted even for our domain experts. One reaction to this might be to shy away from domain-specific ontologies and rely instead on "domain-independent" terminology from stable technical communities. This route also fails, as we explain below.

An example of a stable technical community is in uncertain reasoning but even here we can find differences in formal interpretation of basic expressions. A manifestation of this appeared in an electronic mail debate which took place on the `uai` mailing list in the early summer of 1998. At issue was the common practice of referring to the X in the expression $P(X = x)$, where P is the probability that X has value x, as a "random variable". This sort of notation is often used formally to represent statements like "The probability that the colour of my car is black", which might be $P(car\_colour = black)$. The difficulty is that from the point of view of classical first-order logic it is hard to think of *car_colour* as a logical variable - one feels obliged to think of it as a function from cars to colours. On the other hand, it is often natural to describe it as a "variable" because it is one of the points in a problem description for which we are interested in variation. This is a different notion of variable than the classical logical one but is no less valid, and has similarities to the use of expressions such as "state variable" in process modelling. Although the use made of variables (of whatever kind) is precise and internally consistent within inference systems, the way we use natural language to refer to them differs across inference systems.

It seems that, no matter what we do, we cannot achieve a perfect ontology. Then why does formal modelling ever succeed? The reason is that we try to deploy these models in situations where the inevitable ontological mismatches will either be checked or will have negligible impact on the task which we are interested in performing. In the ecological modelling work we were careful to avoid building into the model generation mechanism any heuristics which relied on a particular interpretation of "biomass" other than as the name of an attribute of certain objects in our problem description. The price we paid for this was that our generator couldn't provide as much automatic control over model construction as it might if it made more commitments to the meaning of domain-specific concepts. We gladly paid this price in order to avoid the project foundering on arguments about the domain theory which (via the ontology) we would have been forced to embed within our inference system.

4 Fitting in with People in the Domain

A well known problem in early design is in defining who will be expected to maintain and benefit from the inference system we build. For the purposes of this paper I shall ignore that problem - although it is embarrassing to remember how frequently I and others have said "the user" in technical papers instead of some more enlightening description of the people expected to benefit from some applied system. There are other less obvious pitfalls which await the unwary.

One of the most difficult human factors to control in a research project of more than a few months duration is when to acquaint collaborators in the domain of application with the formal methods we are using. It has been said (in [4] for example) that the delay in seeing a return for an investment in formal methods is one of the key impediments to their success in industry. I suspect this is true in applications where the problem is clearly identified and the task is to describe it as succinctly and precisely as possible. In this situation we have to wait until the end before we have the "complete" result which is required. However, in early design we are much more likely to be building prototype systems to explore what the tractable problems are. We then have to choose how soon we want to have these ready. Simple prototypes can occasionally be produced extremely rapidly. The fastest I have ever done this is producing a 16-predicate logic program to demonstrate inference of chemical pathways within two hours of meeting a group of plant microbiologists for the first time. The longest time to release a prototype is roughly two years for a system we built to relate codes of practice to a safety shutdown system built using parameterisable components (see [6] for an overview of this).

Taking too long to release a prototype can cause create unpredictable problems because in that length of time the circumstances of industrial partners can change. For example, it would have been better (with hindsight) to have produced more quickly a less impressive prototype system for the safety shutdown domain because at the two year point our industrial collaborators happened to be under greater workload than hitherto, so they had less time to spend with us. Longer gestation periods give more opportunities for accidents like this to happen.

On the other hand, very rapid prototyping can raise expectations too high. Often complicated problems contain a sub-problem which is easy to tackle in an appropriate inference system. To those with little experience of such methods the initial results can seem almost magical and it can be difficult to explain that other essential tasks may be orders of magnitude more difficult. Without careful management, a fast return for investment can be as damaging as a slow return.

Ironically, rapid prototyping can also have an opposing effect. Prototypes invite (constructive) criticism and domain experts are normally good at spotting what they don't like or would wish extended. As we saw in Section 2, it is easy to feel the urge to increase the complexity of an inference system and the feedback from early prototypes may increase this pressure. Changes in support systems (such as visual interfaces) do not necessarily change in harmony with the core inference methods so relatively small changes in the style of inference may require radical overhaul of the prototype implementation (and *vice versa*). This can tip a project into a cycle of rapid prototyping which takes a long time to achieve consensus because there is always some new and exciting variant to build.

5 Education

As Berry points out in these proceedings [1], one aim in early design is to explore the portions of the domain problem which developers don't know and which customers don't know. Normally some form of education is required on each side in order to reach a shared view of the problem which is adequate for the task in hand. There are many pitfalls in education but here I shall take one example of each type - first in the education of customers and then in the education of developers.

A popular way to make inference systems accessible to non-logicians is by providing an environment which helps thos non-logicians understand the system by connecting it to concepts in which they may easily be trained. An example is techniques editing (attributable, among others, to [7] with a survey of applications in [2]) which gives an account of the structure of logic programs in terms of "conceptual structures" corresponding to tasks such as term decomposition in various forms via recursion; term construction and so on. These structural patterns apply across all the clauses in predicate and follow argument position, so predicates can be defined argument by argument according to conceptual structure rather than building clause by clause or in some more serendipitous way. Structure editors have been built which give libraries of patterns and take care of many of the details of applying them. Conveniently, these patterns also correspond to the way predicates are described when teaching logic programming: we explain that a given predicate "decomposes a term in its first argument and constructs a term in its second argument" so it is tempting to think that tools like this help non-logicians learn how to write logic programs. A group of psychologists (then at the University of Loughborough) led by Tom Ormerod ran some tests comparing the performance of undergraduate students using one of our techniques editors within a Prolog programming course to a similar group of students taught using a normal text editor. Those using the techniques editor did indeed write appropriate definitions faster than those without. However, their innate ability to understand example problems in logic programming terms did not seem to be any better than those who hadn't used the editor. One explanation for this is that they had learned to use the tool to build solutions quickly but hadn't learned to think like skilled logic programmers. Sometimes such skills aren't necessary - we may want the inference system to be a mystery to our customers because it would merely distract them. The pitfall here is in thinking that education necessarily comes with tool support.

A more pernicious (and I suspect prevalent) pitfall occurs when we consider how the developers of inference systems should be educated. It is often falsely assumed that for those expert in an appropriate logic the only additional training is in the domain itself - if our domain of application is in potato crop modelling then we need only to talk with potato crop modellers and read a few books on the subject. In fact, there is often need for additional training of the logic experts in knowledge representation. This is because much of the teaching of logic focuses on the semantics of the chosen logic and its proof theory. For these it suffices to use abstract descriptions such as $P \rightarrow \neg Q$. In fact it is often better to use these because it makes abstract notions easier to see, for example that the previous expression is equivalent in classical logic to $\neg P \vee \neg Q$. However, this sort of expertise is different from the expertise necessary to decide on an ontology for a domain and apply that ontology in a way which provides elegant, tractable descriptions of problems. Both forms of expertise are necessary to tackle problems but not all experienced logicians are good at knowledge representation. An example appears in Berry's article in these proceedings. Notice that the sort of expertise in which logicians are often deficient isn't the sort which is taught simply by presenting abstract logic differently - for instance by instantiating the logical implication above to $\mathit{penguin} \rightarrow \neg \mathit{flies}$. These are additional skills, such as the ability to choose good idealisations of problems, which are not guaranteed by an aptitude for abstract logic.

6 Evaluation

In research projects involving the construction of experimental, applied inference systems we need some form of evaluation to assess what sorts of applied problems can be tackled with the system and (if we are making usability claims) how easily it can be used by those who work in the domain. We must then worry about the cost of the evaluation effort; the degree to which empirical evaluation is likely to yield meaningful results; and, in the most extreme case, whether it is possible for our system to fail (that is, whether we are doing an experiment at all). In each if these three cases the fact that we are evaluating an inference system can raise special difficulties.

Empirical evaluation is normally costly for any software implementation but inference systems are often particularly expensive to evaluate because, even if we do our best to fit them into standard work practices, they give people new ways of doing their jobs. Describing ecological systems in a domain-specific formal language [13, 3] was new to those who tested our model generation systems. Describing shutdown systems using parameterisable components in the way described in [6] is new to most safety engineers. This makes it difficult to relate old to new work practices. If the inference system has not been embedded carefully within its host organisation then it may rated poorly just because it was badly introduced. If it is cosseted too carefully during field trials then its rating may be artificially high. Attempting to get this right takes time and money. For example, in the evaluations mentioned in Section 5 we were interested in comparing student performance in Prolog programming with and without a techniques editor. Both groups of students (with or without the editor) needed to be taught Prolog. To avoid artificially boosting the editor's performance because it was being introduced by those who built it, the Prolog courses and accompanying testing had to be done by other researchers (in another University). To avoid a misfit between the editor and the Prolog training course it was necessary to construct a variant of the original course into which the editor was dovetailed. To allow comparison between techniques editor and text editor courses some retrospective adaptatation of the original course was needed. All of this takes considerable effort beyond that of the evaluation experiments themselves, and our example concerns an inference system which was built in order to be tested this way. Frequently, the costs of carefully controlled evaluation can be much higher.

Given the difficulty and cost of controlled evaluation it is little wonder that there are few (if any) extensive usability evaluations of larger inference systems. An alternative is to identify parts of the system and evaluate those. Here we meet at least another two pitfalls for those using logical inference systems. It is standard practice to develop these systems in a modular way so that the (internal) inference mechanisms are separable from but interacting with the (external) user interface. This can make it difficult to judge whether some faults turned up by evaluation could easily have been corrected by some adjustment to either (or both) parts of the system. Perhaps a more serious pitfall is in assuming that evaluation is compositional, in the sense that we can evaluate part of a system then combine that with evaluations of other parts to form a broader evaluation. This is seldom possible, even if our inference system is compositional, for two reasons. First, if the separately evaluated components have user interfaces then we must ensure that a good interface for one is consistent with a good interface in the other which may not be the case if they require modes of communication which are mutually antagonistic. For example, we might have a good usability evaluation for a sub-system which displays a graphical proof tree and, separately, a good evaluation of a sub-system which displays structured terms in a similar graphical style but when these are combined we get a poor usability evaluation because people are confused by the uniform visual representation of different formal concepts. The second form of compositionality problem is created by changes in demand for the system as we consider the broader lifecycle of which it must be a part. For instance, we are reasonably confident from our evaluations of novice programmers that techniques editors fit well into Prolog training courses. However, this does not mean that eventually techniques editors will be part of all introductory Prolog courses because for that to happen it would be necessary for them to mesh with the other tools which more experienced Prolog programmers want to use. Our limited evaluation says nothing about that.

Ironically, the success of logicians in producing expressive and internally consistent formal languages makes it easy for engineers to fall into the trap of designing "evaluation experiments" which are not experiments at all because there is no possibility of failure. Some examples of questions for which the answer will almost certainly be "yes", given enough effort and an expressive formal language are:

- Can a problem tackled in system X be tackled by a (similar) system Y? The answer will be "yes" if we work hard enough with system Y. This question is rather like comparing programming languages - only interesting if we can compare the degree of difficulty on clearly defined problems.
- Can system Y be made to work better than system X on a given problem? The answer will be "yes" if we are allowed unlimited adaptation of system Y. This is interesting only if the changes to Y are carefully constrained.
- Can people be trained to use system X? They almost certainly can if we choose them carefully and give them enough resource and incentives. The interesting question is whether the people, resources and incentives are available in any real domain.

- Will people become better at solving task T using system X? If task T is of interest and system X is relevant and people use it then they are very likely to get better at the task just because they are getting practice (see Berry's comments on the "second time" phenomenon in this proceedings). The interesting question is whether they get better faster than they would have done by normal means.

None of the above are necessarily the wrong thing to do. The pitfall here is in thinking of these as giving some useful measure of empirical fitness of the method without controlling the conditions under which the measure was obtained.

7 Conclusions

Mathematical method is a prerequisite to precise experimental method. However, the former does not guarantee the latter. In preceding sections I have given examples of mistakes which mathematical method alone cannot help us to avoid. These are summarised below:

- The choice of inference system often changes during a project and this requires either alteration of the current inference machinery or its substitution by a new system. It can be difficult to make the right choice of specialist inference system and, even if it is the right choice, there can be high cost in re-education of co-researchers which accompanies each change. Too many such changes cause failure to a project because of the cumulative cost.

- The cost of producing an ontology is not just in inventing the domain-specific formal language but in maintaining it once the system is deployed, since perfect ontologies cannot be guaranteed. Over-commitment to perfecting an ontology causes failure either during development (through irreconcilable arguments over what the ontology should be) or after deployment (through inappropriate human interpretation of inference system inputs or outputs).

- Formal methods are often criticised because they take too long to yield results but this isn't necessarily true in early design. Ironically, problems such as inflated expectations and perpetual prototyping can be caused by the ability of some inference systems to tackle isolated parts of problems rapidly.

- Education is required to bring logicians and domain specialists closer together. We sometimes assume that the tools we produce will help educate domain specialists and that training in abstract logic is enough formal preparation for tackling problem domains. Both assumptions can be false.

- Through force of circumstance or naivete we may under-evaluate our systems, because we can't afford the cost; or we lack a framework for structuring the evaluation; or because our programme of research was not in essence experimental.

References

[1] D. Berry. Formal methods: The very idea, some thoughts about why they work when they work. In *Proceedings of the 1998 ARO/NSF Monterey Workshop on Increasing the Practical Impactof Formal Methods for Computer-Aided Software Development*, Carmal, California, 1998.

[2] A. W. Bowles, D. Robertson, W. W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying Prolog Programming Techniques. *International Journal of Human-Computer Studies*, 41(3):329–350, September 1994. Also as Research Paper 641, Dept of Artificial Intelligence, University of Edinburgh.

[3] A Castro. *A Techniques-based Framework for Domain-specific Synthesis of Simulation Models.* PhD thesis, University of Edinburgh, 1998.

[4] G. Cleland and D. MacKenzie. Inhibiting factors, market structure and the industrial uptake of formal methods. In *Workshop on Industrial-Strength Formal Specification Techniques*, pages 46–60, Boca Raton, Florida, 1995.

[5] F. Correa da Silva and A. Bundy. On some equivalence relations between Incidence Calculus and Dempster-Shafer theory of evidence. In *Proceedings of the 6th Conference on Uncertainty in Artificial Intelligence*, 1990.

[6] J. Hesketh, D. Robertson, N. Fuchs, and A. Bundy. Lightweight formalisation in support of requirements engineering. *Journal of Automated Software Engineering*, 5(2):183–210, 1998.

[7] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for Prolog programming. Tr 89-170, Case Western Reserve University, 1989.

[8] R.A. Kowalski and F. Sadri. Reconciling the situation calculus and event calculus. *Journal of Logic Programming*, 31:39–58, 1997.

[9] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[10] W. Liu and A. Bundy. A comprehensive comparison between generalised incidence calculus and the dempster-shafer theory of evidence. *International Journal of Human-Computer Studies*, 40:1009–1032, 1994.

[11] D.L. Parnas. Using mathematical models in the inspection of critical software. In M. Hinchey and J. Bowen, editors, *Applications of Formal Methods*. Prentice Hall, 1995. ISBN 0-13-366949-1.

[12] D. Robertson and J. Agusti. *Software Blueprints: Lightweight Uses of Logic in Conceptual Modelling*. Addison Wesley, 1999. in press.

[13] D. Robertson, A. Bundy, R. Muetzelfeldt, M. Haggith, and M Uschold. *Eco-Logic: Logic-Based Approaches to Ecological Modelling*. MIT Press (Logic Programming Series), 1991. ISBN 0-262-18143-6.

[14] M. Uschold and M. Gruninger. Ontologies: Principles, Methods and Applications. *Knowledge Engineering Review*, 11(2):93–136, 1996. ISSN 0269-8889.