Thinking tools for the future of computing science

Cliff B Jones

Department of Computing Science,
University of Newcastle
NE1 7RU, UK
e-mail: cliff.jones@ncl.ac.uk

Abstract. This paper argues that "formal methods" can (continue to) provide the thinking tools for the future of computing science. Every significant engineering or scientific discipline has advanced only with systematic and formally based notations. To see just how ubiquitous the need for notation is, one can look beyond the sciences and observe the importance of written notation in the development of music. Map making is another area where the importance of notation and the understanding of the need for (levels of) abstraction is recognised. Formal methods provide notations to navigate the future of computing science.

Please cite the original publication: *Thinking tools for the future of computing science*, Cliff B Jones, pp112–130, In *Informatics – 10 Years Back, 10 Years forward*, (ed.) Reinhard Wilhelm, Springer-Verlag *Lecture Notes in Computer Science 2000*, 2000

1 Background

All engineering disciplines make progress by employing mathematically based notations and methods. Research on "formal methods" follows this model and attempts to identify and develop mathematical approaches that can contribute to the task of creating computer systems (both their hardware and software components). Clearly, mathematical calculation will not solve all of the problems faced in the creation of large systems: neither design skills, nor the problems inherent in organising large groups of people, seem obvious candidates for formalisation. Equally clearly, to believe that hugely complex computer systems can be designed without any formalism other than the running code is to emulate Icarus and court disaster after a hugely successful first half century of computing.

The development of ever faster and cheaper micro-electronic devices has made computing widely available but it is software which has brought about the IT revolution. There is much to be proud of in the systems that are deployed today and would have been unthinkable even twenty years ago. Progress in software has come from increased functionality in operating and windowing systems, from better interfaces and improved models of linking such as objects and components, from procedures for review and testing, and (as will be suggested below) by the selective adoption of some formal methods ideas. It would be stupid to under value the contribution of Moore's law, but it is one sided to contribute today's widespread use of computing to the exponential improvement in performance: but for software, hardware circuits could switch arbitrarily quickly and make no impact on society.

Much then has already been achieved; is there a need for research on formal methods? It is not really the place of the current paper to argue this case in detail; it seems reasonable to accept that the design of any huge artifact needs all the help it can get whether from better management or from notations which make it possible to calculate –rather than postulate– an outcome. If the use of formally based methods can help –as in other engineering disciplines– it is surely unacceptable to ignore them when society's wealth, smooth running and even existence are increasingly reliant of software products.[1]

There is another key argument and that is that the amount of reinvention in software is huge and unsustainable. There are estimates of between one and two million people in the US being involved in some form of programming. It is unlikely that even the majority of these people are well-trained software engineers; furthermore the growth in the percentage of the population involved in this activity cannot be sustained. Carefully documented specifications and reliable implementations (possibly including their justification) are a prerequisite for widespread reuse. In addition, it is necessary that any components of reuse have a carefully designed interface and fit into architectures which facilitate

[1] One of the interesting discussions at the Dagstuhl anniversary conference was on the potential for liability lawsuits against software producers; Fred Schneider made the important point that in many arenas the producer is deemed negligent if "best practice" has not been employed.

composition. The role of notation in design and architecture is now accepted but the level of formality in widely used notations such as UML is low. This is both regrettable and avoidable. A key (Popper-like) test for any notation is the extent to which inconsistent texts in the notation can be detected: completely automatic checks –as with type checking– are invaluable; checks which can be the subject of calculation or proof are almost as good; but the value of any "notation" whose texts can only be the subject of debate is limited.

In general, this section sets out the current situation, comments on a few pieces of past research and their state of adoption and draws some lessons from the past. Subsequent sections of this paper turn to future challenges.

1.1 Current use of formal methods

Various approaches which are recognised as formal methods are used in the design of safety critical systems. Indeed, for such systems where malfunction can result in loss of life, there have been moves to require the use of formalism. It is the genesis of one of the challenges for the future that many safety critical systems are concurrent programs. Apart from such easily identified use of model checking or verification ideas, there is much wider use of ideas that derive from research into formal approaches. An obvious example is the use of loop invariants even in informal discussions of programs. As is made clear below, it is often the case that ideas which originate as part of formal methods research are no longer seen as formal when they are adopted into wider use. Furthermore, avoiding emphasis on formality can often be a key to adoption. Good examples are Michael Jackson's influential writings (e.g. [Jac83,Jac00]) and the wider use of model checking tools than verification technology. Personally, I have moved from a position where formal methods were presented as an all-or-nothing technology to a position branded as "formal methods light" in [Jon96b].

So, there is some direct –although not necessarily acknowledged– use of formal methods in general computing practice as well as in the narrower safety-critical arena. But it has to be conceded that the vast majority of software available today has not been designed using formal methods. Regrettably, one can go much further than this and state with some confidence that most software is not developed using anything like "best practice". An issue which is tackled in Section 5 is how the better developers might come to create a larger percentage of software and how formalism might influence the programming paradigm employed.

The most important long-term contribution of studying theoretical underpinnings of computing has been the insight gained into fundamental concepts. Examples like the distinction between non-determinacy and under-determined characterisations in specifications, the role of unbounded non-determinacy and the fundamental distinction between synchronous and asynchronous communication would all justify lengthy discussions of their own.[2] The purpose here is

[2] The fundamental understanding on the complexity of classes of algorithmic tasks is a major intellectual achievement but its discussion can safely be left to experts who were well represented at the Dagstuhl anniversary.

to look at new challenges which will be done after a brief review of what can be learned from some examples of earlier research into formal methods.

1.2 Historical comment on syntax description

The choice of topics here is made to illustrate key points and is in no way a measure of the intrinsic importance of the material. Many other selections could have been made and the choice of say concurrency theory or theorem proving would have made a more compelling justification of past research; here the purpose is to see what can be learned for the future.

The first illustrative historical note relates to the description of the syntax of programming languages. The original work of John Backus and Peter Naur goes back over forty years and the description of the (context-free) syntax of languages is now standard practice. Several observations can however be made. The use of a precise notation makes the question of syntactic validity unambiguous: there is no argument as to whether a program is syntactically correct; it is only necessary to see if it can be generated from the grammar for the language. Formally based notations avoid contradictions.

The understanding of context-free grammars facilitated the automatic creation of parsers. Precise notations are also essential for the creation of meaningful tools. This observation becomes important when discussing the relative advantages of informal (often graphical) notations below.

The adoption of BNF was not without difficulties. Although it is hard to believe today, some language designers saw even this level of formality as unnecessary. Furthermore there were genuine questions about what constituted "syntax": were the (context-sensitive) constraints which govern declaration and use of variables to be regarded as syntactic in nature because they were concerned with structure or were they to be considered "static semantics" because they were beyond Chomsky Type 2 grammars?

But the most important realisation is that, with the adoption of syntactic formalisms into everyday practice, they are no longer viewed as a part of formal methods. This is a pattern which has been repeated many times. Bob Harper's comments on type theory, the widespread use of the once exotic idea of (especially on-the-fly) "garbage collection" and many others were alluded to during the Dagstuhl anniversary.

1.3 Historical comment on semantic description

Beyond syntax, comes semantics. Early work on formally describing the semantics of programming languages dates back to the 1960s. The main purpose of commenting on this history[3] here is to sound a warning note because it must be recognised that this research has had little practical impact in spite of its huge potential.

[3] A proper scientific history of this field of research would be useful.

McCarthy's work in the early 1960s established the approach of writing "abstract interpreters" for programming languages. He and other key researchers met at the 1964 Baden-bei-Wien conference on "Formal Language Description Languages" (see [Ste66] for excellent proceedings which record many fascinating discussions). The initiator of this conference –Professor Heinz Zemanek– was director of the IBM Laboratory in Vienna. Researchers there applied and extended the basic "operational semantics" idea in an attempt to tame the programming language which came to be called PL/I. Apart from including most things which could have been claimed to be comprehended at that time, PL/I embraced the ill-understood topics of exception handling and concurrent tasks. The huge formal definition was a *tour de force* and with hindsight its most important contribution might have been to clean up a language whose ancestors included both FORTRAN and COBOL.

Unfortunately, many aspects of the "Vienna Definition Language" (VDL) operational semantics descriptions made them difficult to use in reasoning about the design of compilers and this fact, more than the desire for more mathematical base, led the Vienna group to embrace –in the 1970s– the research on Denotational Semantics which had been led by Strachey, Landin, Scott and others. The so-called "Vienna Development Method" had its genesis in the denotational description of the ECMA/ANSI version of PL/I. Fortunately for the authors, the standards bodies had rejected the tasking proposals in PL/I and the language to be described was non-deterministic rather than truly parallel. Attempts to provide denotational models for parallelism led to Power Domains which were not for the faint-hearted.

Plotkin's 1981 paper [Plo81] is widely seen as the first to show that those aspects of operational semantics which had made them difficult to understand could be avoided without the need to provide denotational descriptions. It was particularly gratifying that concurrent languages could be described without heavy mathematics. "Structured Operational Semantics" is now an eminently teachable subject which ought to be part of the training of any computer scientist.

That there is a clear need for formal semantic descriptions of significant systems should be clear. I have moved several times between industry and academia: during a recent spell in a software house, a project was built on top of CORBA. Impressive in scope, CORBA's imprecise semantics and poor implementations are squandering the huge potential for object brokers to free information from being locked into propriety interfaces.

In spite of the recognised need for moving beyond the description of syntactic details of interfaces to describing their semantics, it must be conceded that formal semantics is used only in narrow fields. Some of the difficulty is due to the history of the subject and the infatuation of some researchers with mathematical depth without measuring the value of new concepts to practitioners. The lack of well chosen example descriptions must be seen as another tactical mistake. Almost certainly, an opportunity was missed in not providing better tools for experimentation with semantic descriptions. Above all, the desperate need for

thinking tools which would help designers come up with clearer and easier-to-use architectures has only been met to a limited extent by the literature on formal semantic description techniques.

1.4 Historical comment on reasoning about programs

Although the history of work on reasoning about programs in some respects mirrors –and indeed is dependent upon– the work on the semantics of languages, it has had a significantly larger impact on practice. The research on reasoning about programs has provided key "thinking tools" which are in wide use.

Here again there is a rich history to be written and [Jon92] can only be regarded as useful source material.[4] But the essence of the argument is that the idea of reasoning about whether a program could be said to satisfy its specification was present from the early days of programming with both Alan Turing and John von Neumann discussing the idea in papers in the 1940s. The thesis is that what has followed has been a "search for tractability". Seen thus, Floyd, Naur, van Wijngaarden and Hoare were all seeking ways to present formal arguments in an acceptable way. In particular, Hoare's paper [Hoa69] offers a much more telling presentation of Floyd's earlier idea [Flo67]. Hoare generously concedes his debt to Floyd and van Wijngaarden (see historical notes [HJ89, pp45–46]). In mathematics, it is often a key step to propose a tractable notation. Furthermore, it is central to the thesis of the current paper that issues of presentation are often the deciding factor in creating thinking tools.

The history of reasoning about programs does not stop with Hoare's 1968 paper and one could trace the development of Dijkstra's Predicate Transformers [Dij76,DS90] and subsequent program calculi from Back and Morgan but the main points of the historical note in this section can be understood by anyone who has met Hoare's axiom for the while statement:

$$\frac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ \mathit{while}\ b\ \mathit{do}\ S\ \mathit{od}\ \{p\ \wedge\ \neg b\}}$$

This rule is both elegant and memorable.

Moreover, the invariant p provides a fundamental thinking tool. If one trained programmer wants to convince another that a loop construct serves its intended purpose, the invariant is a key part of the argument. This is true whether the challenge to justify the loop comes in an informal walkthrough or in an attempt to get a formal proof checked by a computer verification system.

Interestingly, another aspect of the correctness of loop constructs is not covered by the rule as shown. Hoare's 1968 paper left the subject of termination to separate argumentation. Here, Dijkstra's *variant function* [Dij76] is most often used and offers another thinking tool for reasoning that programs satisfy their specifications.

[4] The report [Jon92] was commissioned for a scientific encyclopedia which has not appeared; it is now being revised for publication in the IEEE Annals of Computing.

It is instructive to investigate further aspects that are not covered in the above rule. For example, issues of "clean termination" (cf. [Sit74]) are not addressed. The advantages of using post-conditions which are relations (and the search for tractable rules) are discussed in [Jon99]. It might be argued that part of the success of Hoare's axiomatic programme was that it avoided trying to do too much.

There is however an absolutely crucial point about Hoare's rules which has made them essential thinking tools for the progress of computing science and that is that they are *compositional*: the rules can be used to divide a problem into smaller problems whose solutions can be judged solely with respect to their individual specifications.[5] The topic of compositional development is the source of a challenge in Section 3.1 but its practical importance should already be clear: for a large task, one needs ways of creating correct designs; it is not enough to have methods or tools for checking the finished product. Such tools are valuable in their own way but do not help the design process.

One further key contribution in the area of thinking tools for the design process is the story of *data reification* (or refinement). A note on this is included in [Jon99] with further references to earlier papers and a comprehensive review is contained in [dRE99]. Suffice it to say here that brief and perspicuous specification of a system is likely to owe more to well chosen data abstractions than to cute post-conditions.

A word of reservation about the success of the thinking tools for reasoning about programs is in order. It was long my claim that every designer and programmer should be able to read and write formal specifications. To some extent, many researchers felt that the difficulty of teaching VDM or Z might lie in the notation itself. Reluctantly, I at least have accepted that many people find the *process* of abstraction difficult and that it is not learning to write "upside down ∀s" which is the obstacle.

Perhaps the most important observation is that assertions are an orthogonal check on the code of a program. It has often been observed that the attempt to show that a program satisfies a specification uncovers errors in the latter as well as the former. It is however a constructive diversity to have both assertions and algorithm.

In closing this review of a relatively successful deployment of a formal methods idea, it is again encouraging to look at the provision of tools. From complete systems to aid the verification of programs, to the much more approachable *extended static checkers* described in this conference by Rustan Leino, it is clear the presence of underlying formal understanding is essential for meaningful (mechanical) tools.

1.5 Lessons

If the formal methods community is to maximise its future impact on practical computing applications, there are a number of lessons which are worth spelling

[5] Technically, this depends on the programming combinators being monotonic over the specification ordering.

out from the past. It is, for example, difficult to conduct controlled experiments of any useful scale. Small programs are not difficult to get right and a university experiment on 50 line programs is never likely to show the pay off for using careful specification and stepwise design. Unfortunately, huge programs are incredibly expensive to build and no organisation will attempt to build the same product two ways. But it has not been the practice in other engineering disciplines to conduct –for example– controlled experiments of bridge designs with and without stress analysis. The proposal for a new engineering method might require learning some new mathematical tools. Where an engineering culture exists, the burden is readily accepted. Two things work against such acceptance by software engineers: many are untrained in any engineering mathematics; and the consequences of software bugs are either not recognised by liability or they are perceived as easy to correct.

There are however messages for the formalists as well. The educational load of a new proposal must not be ignored: if the main gain from learning a new branch of mathematics is to cite erudite papers, users will not be persuaded. There has to be a direct value of a new field to make it worth its adoption. Nor is it the case that all computing problems will succumb to an *existing* body of mathematical knowledge. To take one example that I have written about in several places, the assumption that classical logic is appropriate for reasoning about specifications and programs where partial functions abound might be unfortunate.[6] Kline wrote that "More than anything, mathematics is a method" which we could take as a claim that formal methods should offer mathematically sound approaches but not necessarily ones which are parts of recognised mathematics. It is particularly important that the mathematics should fit the real problem rather than the problem be bent to fit the mathematics.

There have, with hindsight, been many missed opportunities in the realm of formal methods. There have been premature attempts to develop new specification languages where old ones were not understood; there have been unnecessary differences between basically similar approaches; and there have been extended developments of closed research schools. There is probably nothing unusual in these aspects of a scientific field. Perhaps more frustrating is the perpetual growth of informal notations which are often diagrammatic and which achieve widespread use in spite of their lack of precise semantics. There is nothing inherently wrong with diagrams; but it is the repeated experience of *post hoc* formalisation that many things are clarified and simplified.

The strongest message from the experience with formal methods is that they have had an impact on practical development although the adoption of formal ideas is often coupled with a denial of their origin. Concepts like data type invariants, compositional development methods, reasoning about termination etc. have become thinking tools of the better educated programmer and designer. Some specification languages such as VDM, Z or B are used in practical projects and commercially available tools support such use. More advanced tools like

[6] The case for a Logic of Partial Functions is set out in [CJ91] where further references can be found.

theorem provers and model checkers[7] are used on critical projects. The wider acceptance of the latter class of tools has interesting lessons and warnings. Although model checking tools are certainly not for the untrained, it is true that they can be deployed both on finished code and without design history. That is attractive especially if the need is to find errors in a product. The thinking tools thesis of this paper would of course argue for ideas which tend to create correct designs rather than for tools –however good– to detect problems at the end of the development phase. It is my experience that trying to prove an extant program satisfies some specification almost invariably results in discarding the program and conducting a new design activity using formal arguments at each stage of design.

2 Some incremental challenges

There are a number of obvious steps which could be taken both in the development and deployment of formal methods. This section mentions a few incremental items so that the more significant challenges in later sections can be seen in context.

There is clearly a need for work on standards for formal methods. There has been for some years an ISO standard for the VDM Specification Language and the Z specification language is nearing that status. More challenging is to look at connections between approaches and [BBD+00] reports on work to link VDM and B. It is of course tempting to say that it might be less work to devise a new language with the best features of both — but the danger here is of an unending proliferation as has been seen in the programming language arena. Standards work might not be exciting to all researchers but it has an effect both on user preparedness to adopt new methods and to the provision of tool support. Both the VDM and Z standardisation activities uncovered significant issues that needed resolution.

The provision of tools is seen by many as necessary for the adoption of formal methods. In many cases, this is actually an excuse since tools do not replace the need for quite fundamental shifts in approach and training. Where tools do become important is in the inevitable maintenance of older formal texts. Links between tools like the IFAD VDM Toolset [WWW00c] and theorem proving systems [WWW00a] are also the subject of ongoing work. Tools for theorem proving will require more research before they are widely used.

A more fundamental missing synergy is between the "correct by construction" camp and the *post hoc* model checking approaches.[8] Saying that the approaches can be viewed as complimentary is a first step but it will require research to devise languages which can carry information about abstractions from the design phase into the state exploration of the final code; linked tool support is a considerable engineering challenge.

[7] See Ed Clarke's chapter of these proceedings.
[8] This was addressed by both Amir Pneuli and John Rushby at the FM'99 conference.

Recent recognition of software architecture (cf. [SG96]) as a topic of research is encouraging because it offers the prospect of re-use of intellectual effort (Software Patterns and Components are other approaches). This could be a key area for the use of formal methods. It is not difficult to argue that the most damaging contribution to the "Total Cost of Ownership" of computer systems is their poor architecture. In all but the most critical applications, the cost of bugs fades into insignificance compared with the time that users waste in not having a model (cf. [Nor88]) of the system (which they had been sold on the basis that it would help them).[9] Formal methods have –since their role in the evolution of PL/I– been thinking tools for cleaning up messy architectures.

3 Conceptual gaps

This section indicates some areas where research is required which is not simply incremental. Any author would list their own set of conceptual gaps and two lists would rarely coincide: the attempt is to indicate the sort of challenge rather than to provide a complete catalogue. Clearly the choice is influenced by my own research and thus largely relates to concurrency.

3.1 Compositional development of concurrent systems

Section 1.4 claims that compositionality is key for the development of sequential programs: one step of design can be justified before effort is put into development of the sub-components. The essence of concurrency is *interference*. With shared variable programs, interference manifests itself by several processes changing the same portion of the state of a computation. Much early research on concurrency was concerned with ways of controlling when processes could access and update the state. Some researchers saw the idea of shared state as the root of the problem and shifted attention to communication based concurrency.[10] But, since one can precisely simulate the notion of a shared-variable as a process, what happens is that the problem shifts to communication interference. The fundamental problem of more than one process generating activity (which can influence other processes) is inherent to concurrency. One can argue that one or another approach facilitates reasoning about interference but does not make it go away.

The sobering realization is that interference makes it difficult to conceive compositional development methods for concurrent systems. Early attempts to present ways of reasoning about concurrent programs barely noticed this problem since the state-of-the-art even for sequential programs was to construct *post-facto* proofs. It was true that those techniques which proposed looking at the

[9] A particularly insidious example of the hidden cost of ownership came up at the conference where two speakers pointed out the difficulty they had experienced in accessing their own 1990 texts which had been created on software which has since been "upgraded".

[10] See CSP [Hoa85], CCS [Mil89] and the π-calculus [MPW92].

cross-product of control points between processes would not scale but this initial objection missed the point that one needed to undertake some proof long before the final code (and its control points) were available.

A widely cited technique for the development of concurrent programs is that in Susan Owicki's thesis [Owi75] and often called the "Owicki/Gries" method. This makes some progress in that part of the proof obligation associated with decomposing a task into sub-tasks is discharged before the sub-tasks are developed but there is still a final stage in which one must check whether the steps of one process can interfere with the proof of others. So one could still decompose a task into sub-tasks; develop them to satisfy their separate specifications and then be forced to discard them and start again when the developed code is shown to interfere with something in the other process. This is clearly non-compositional.

With hindsight the step to specifying interference might seem obvious. The point of this section is to argue that in spite of many theses and other papers, there is still not a completely satisfactory way of reasoning about interference.

The idea of adding rely- and guarantee-conditions to pre- and post-conditions is proposed in [Jon81] and developed in [Stø90,Xu92,Col94,Din00]. Essentially, a rely-condition documents the interference that the (developer must make sure the) program can tolerate and the guarantee-condition records the interference that other processes will have to put up with if they run in parallel with the developed code. These extra predicates are one way of recording interference expectations and they (together with the associated proof rules — see [Jon96a]) regain the key property of compositionality. As with post-conditions in VDM, both rely and guarantee-conditions are relations (predicates over two states). The proof rules are more complicated than the rules for sequential programming constructs but that is almost certainly inevitable. It is worth pointing out that complexity in formal proofs often comes from the design: the effect of formalisation is to expose rather than cause difficulty. A poor interface will make justification difficult whether formal or informal. Ignoring a difficulty at one step of development can result in major costs in subsequent correction; worst of all is repeated patching of code to work around an early design flaw.

Although the eventual form of the proof rule for parallel composition with rely/guarantee-conditions is reasonably memorable, it is complex enough that it is worth avoiding its use where possible. (Indeed, juggling clauses between pre/rely and post/guarantee conditions is non-trivial as shown in [CJ00].) For this reason, it is useful to look at ways of localising areas of potential interference. This of course echoes early concurrency research from semaphores to modules. A series of papers (see [Jon96a] for references) show the role of a parallel object-based language in fixing such limitations.

Rely/guarantee-conditions are one form of assumption-commitment specification and there are several others. One that is less well-known than it deserves is Kees Middelburg's use of Temporal Logic to capture interference [Mid93]. An excellent survey of existing compositional approaches to concurrency is being published by de Roever and colleagues.

There is then work required in evaluating different approaches; but there are also gaps that need addressing in all approaches. In known approaches there is –for some applications– a need to include some form of "ghost variable" which keeps track of the point in control flow where assertions hold. A particularly trivial and annoying case is where two parallel processes are each tasked with incrementing a variable, say x, by 1. This class of limitations prompts the idea that there ought to be a totally different way of discussing the combined effects of operations which are in the same algebraic group: where such operators commute, their combined effect on x is the same independent of their order. So one shortcoming of the current rely/guarantee reasoning about interference is that it tries too hard to do everything with predicates.[11] There is here an indication of a much deeper question. In some sense, a development method is made valuable by what it avoids making explicit. So the Hoare rule for assignment expresses neatly what does change without making explicit what stays constant. The trade off between ghost variables and the power of proof rules is an intriguing subject which deserves more explicit research. Other ways in which interference arguments need to be extended are in the handling of real-time and probabilistic specifications.

3.2 Design by atomicity refinement

It is clear from both the work on specification by post-conditions (say "what", rather than "how") and the use of abstract objects in specifications that abstraction is the main tool to tackle complexity. If a specification is to be at least one order of magnitude shorter than an eventual implementation, knowing what to omit is essential. One powerful abstraction is that of pretending atomicity. I first encountered the idea in reasoning about concurrent object-oriented programs. Essentially, it was easy to show that an abstract program achieved a desired effect by assuming that certain operations –although executed in a non-deterministic order– were atomic. A subsequent stage of development showed that although the allegedly atomic operations were composed of smaller steps, these steps could not influence other processes and that steps of separate processes could overlap. What is more exciting is that the idea of refining atomic operations occurs in many areas. The basic correctness notion for database transactions is that any implementation should give a result that could have been obtained from some order of executing transactions atomically. Any reasonable implementation will actually overlap transactions and operate a variety of clash avoidance or detection algorithms to simulate the atomicity abstraction. Similar comments can be made about distributed caches and the same issue comes up again in the design of asynchronous circuits such as AMULET.

Several researchers have considered aspects of this problem but I am not aware of any general treatment of design by atomicity refinement. Such ubiquitous problems are usually challenging and their solutions powerful.

[11] There are other difficulties but these are being covered in a longer and more detailed paper on compositionality for a volume of contributions by members of IFIP's Working Group 2.3; the editors are McIver and Morgan.

3.3 Unification of state and communication

The discussion of compositional development methods for concurrency alluded to the distinction between state-based and communication-based concurrency. At some level, this distinction appears to be spurious in that either can be simulated in the other. Indeed, looking down from the abstraction of a programming language to the physics of the hardware devices, one appears to switch to and fro between these views. It would appear therefore to be unfortunate that the (generally predicate-based) methods for reasoning about states and the (generally trace-based) ways of documenting communication are separate. One specific example is the view taken of specifying the circumstances under which something should terminate. State-based specification languages like VDM and B[12] separate out a pre-condition over which the specified operation must terminate. In contrast, trace-based assertions adopt divergences and refusals that are linked to the evolving computation. There is a suspicion that there are two notions here which could each be applied in the other domain. There are earlier authors who sensed this challenge: Hewitt's Actors being the earliest example.

Another related point came up when attempting to verify some equivalences of the parallel object-based language referred to in Section 3.1. The proofs in [Jon94] were based on a translation into the π-calculus. As Robin Milner had prophesied, this turned out to be a rather low-level language and coding everything as communication clouded the reasoning. This prompted the suspicion that some halfway house of what might be called "stateful processes" would be more tractable. Both David Walker and Davide Sangiorgi have published papers on the required equivalence results and particularly the notion of "uniformly receptive processes" in [San99] would appear to be a step in this direction. The challenge is to find a useful notion of stateful process that has a nice algebra and clearly links state and communication.

4 A grand challenge: dependability

Contributors to the Dagstuhl conference were encouraged to identify grand challenges in their research areas: I am now the Project Director of a six-year research project on *Dependability of Computer-Based Systems* (DIRC) [WWW00b], so it is natural to view this as a grand challenge. The term dependability is preferred to reliability because of the wish to encompass topics such as availability, security etc. The phrase "computer-based" systems is chosen to indicate that the collaboration will consider broad user and society issues as well as technical computer questions. This section points to material on the overall project and pinpoints one example of a topic for future research.

Researchers in the consortium have worked for many years on aspects of the dependability of computer systems themselves. Earlier research on computer

[12] Many users of Z also distinguish the pre-condition from the post-condition but the language itself merges them.

fault-tolerance and avoidance has led to concepts like recovery blocks and Coordinated Atomic Actions (see [Ran00] for references). There is no sense in which the computer scientists in the new collaboration regard all of the computer research issues as resolved and –for example– researchers at Newcastle are involved in two significant European projects on "Dependable Systems of Systems" and tolerating malicious attacks. But there is a clear realisation that, as computers enter ever more into everyday life, many system failures occur at the link between humans and computers. Nor is this problem confined to HCI questions, important though they are. The DIRC collaboration includes researchers from psychology, sociology and management science as well as computer scientists. An example of an application to which they would like their research to contribute is the proposed introduction of a Electronic Health Record in the UK. Obvious dependability issues include availability, security and reliability. Without an understanding of the notions of trust and responsibility that are current in the medical profession, a computer system could be devised which is a complete disaster (even were it to completely satisfy its formal specification!).

The general area then is *Dependability of Computer-Based Systems*; specific research themes cover risk, diversity, timeliness, trust and structure. As an example of an idea which I wish to study under the "structure" research theme, the topic of *faults as interference* is described. In order to discuss fault-tolerance, one needs to document the sort of faults that are to be tolerated. This observation was made when consulting on what was known as the Inherently Safe Automatic [Reactor] Trip; ISAT is described in [SW89]. In that project it was shown that rely-conditions (see Section 3.1 above) could be used to specify faults as though they were interference and then to show that the system developed could tolerate that level of interference.[13] There are many aspects of this idea which need study even as they relate to computer systems alone. In particular, it was clear from the ISAT work that the understanding of "faults" is often recorded at a much lower level of abstraction than one wants to reason about with a top-level specification. (One might then have to construct fault-containment abstraction levels.) Replication for fault-tolerance and data fragmentation for security are other interesting aspects to be investigated as is the idea of proof carrying guarantee-conditions for code which is imported. There are many examples in the literature such as the Karlsruhe Production cell and the Gas burner system (this originated in a Dagstuhl seminar and is described in [GNRR93]). Michael Jackson's new book [Jac00] also shows how to address a system wider than the machine and provides the traffic light example which was used as an illustration at the conference. Examples with timing assumptions and constraints become particularly interesting. But linking back to the broader view of Computer-Based systems, it is intriguing to study the extent to which the idea of faults as interference could cope with human errors. It is inevitable that humans make mistakes. One of the many contributions that psychologists bring to our understanding of computer-based systems is their attempts to characterise human mistakes —

[13] At the Dagstuhl meeting, Fred Schneider pointed out that he had written about a related idea in [SS83].

see for example [Rea90]. Such a categorisation could be linked with the idea of faults as interference and make it possible to record which sorts of human errors a system is claimed to tolerate.

As indicated in [Nor88], many so-called operator errors result from a false model of what is going on within a system. The interesting approach of John Rushby in [Rus99] is to attempt to record both an actual system model and a model of how a user perceives that system. Rushby shows how his approach can explain –and even mechanically find– some errors which are made at a pilot/control interface but concedes that it is difficult to extract the model of the user's view of a system. It might be easier to express assertions about the user's view (than elicit a finite state model) and "faults as interference" reasoning could be employed. There is anyway much research in this specific area and even more in broader issues of the dependability of computer-based systems.

5 A grand challenge: programming paradigms

The procedural programming paradigm has had a remarkably long innings. Many different languages have come and gone; a few genuinely new concepts like objects have been adopted; somehow, natural selection seems to favour the Baroque over the simple and elegant. But most programming is still done in an imperative style. Improvements in both machine performance and compiler technology have given new impetus to functional programming; logic programming has its adherents; it could even be argued that people using spreadsheets are programming non-declaratively. But the need in so many applications to access and change something like a database gives imperative programs a head start. Unfortunately, it is all too clear that constructing such programs is expensive and creates error-laden implementations which are extremely fragile when modifications become necessary. What can be done? Is a complete change of paradigm thinkable? I have no instant proposal but I believe that something different will arise. In lieu of a concrete proposal, I'd like to make some observations that might suggest that programming could be different.

Careful programmers do write assertions and I am aware of large systems where such assertions are compiled into running code.[14] The argument for doing this is that failing an assertion often detects a bug before other data is corrupted and the origin of the error obscured. Assertions then can be a form of redundancy not just in the design process but in the executing system. Human systems rely on redundancy and often detect errors and even correct them by using redundant information. There are also claims that the overall "mean time between failure" of telephone switch software has been dramatically extended by evaluating assertions and selectively dropping suspect calls. Obviously one can now dive into technical questions: how rich is the assertion language? are the predicates over one state or can they refer to older states (the recovery block idea permitted a limited form of relational check)? what action is prescribed when an

[14] This idea is at least as old as [Sat75].

assertion is false? could one find an efficient way of periodically "retrieving" the abstractions used in the abstract design? But for now, all that is important is the idea of redundancy and some coherence check. Suppose the assertions were given priority over the algorithm. Furthermore, suppose they were not just validity assertions but they described "desirable states" which actions perturbed. As an example, a desirable state for a hospital is that it has no sick patients; the arrival of a sick person is a perturbation; a better state than one where someone has undiagnosed symptoms is that they have been diagnosed and are undergoing treatment.[15] Would it be easy to write routines which moved from less to more desirable states? could they be redundant? would such systems be easier to maintain?[16]

I might of course be on totally the wrong tack. Perhaps the real villain is the assignment statement and we should revisit Wadge's Lucid or McCarthy's Elephant languages. Estimates were given at the Dagstuhl anniversary of millions of people who are employed in something approximating to programming. It is clear that this growth cannot go on (anymore than the early extrapolations of the number of people required to operate manual telephone exchanges were sustainable). Moreover, it is fairly clear from the quality of current software, that a reduction not an increase is highly desirable. Either (good) programmers have to start producing ubiquitous artifacts which avoid the wasteful recreation of close approximations to other systems and/or the task of programming has to be made more resilient.

6 A grand challenge: information access

This final offering has little to do with formal methods except in so far as they might be used –as always– to clarify thinking. Computing as a way of providing computation power has been the most dramatic success of the century just finished; but for providing information, we haven't even started. The WWW is certainly not a counter argument: even if the information one wants is available, it is almost impossible to find it in what is becoming the "World Wide Wastebasket". Not only is it difficult to locate information structured and stored by others, one can get locked out of one's own information by "upgrades" to software. This moved John Gurd and I to decline to prophesy the future of our own research specialties in an earlier crystal ball activity and to write instead [GJ96]. The "Global yet Personal Information System" was a dream whose availability was not offered at the Dagstuhl conference. Others at this conference are more versed than us in the current state of the art but the challenge of providing (controlled) access to both free-form and structured information has to be one of the key tests for computing science in the new century.

[15] Notice that this contrasts with *process modelling* in which the patient approximates to an instruction counter in a more-or-less fixed procedure.

[16] There were discussions at the Dagstuhl meeting about how to deploy parallel processes in novel ways: some approach like that above appears to have more inherent parallelism than could ever be detected in FORTAN code!

7 Conclusions

The thesis of this paper must be clear: formal methods can and must offer the thinking tools which will both ensure progress in computing and elucidate fundamental concepts. To achieve this, one cannot ignore the real problems which exist. One can decry the rush to get systems to market; one can warn that we are "headed for a spill"; but we must at least see in which direction the main line is moving.

The need for thinking tools cannot be clearer than in the early stages of system development. Tools to help simplify and generalise designs are necessary. Ways of getting the design right from the beginning are a means to make a dramatic improvement in productivity.

The adoption of formal methods will be gradual and be influenced more by good engineering training than any single idea.

Acknowledgements

I am grateful to Reinhard Wilhelm and his colleagues both for the invitation to take part in the Dagstuhl event and for their generous hospitality. To the other participants, I should like to express my thanks for many stimulating discussions. I gratefully acknowledge the financial support of the UK EPSRC for the Interdisciplinary Research Collaboration "Dependability of Computer-Based Systems". Most of my research is influenced by, and has been discussed with, members of IFIP's Working Group 2.3.

References

[BBD+00] J. Biccaregui, Matthew Bishop, Theo Dimitakos, Kevin Lano, Brian Matthews, and Brian Ritchie. Supporting co-use of VDM and B by translation. In J. S. Fitzgerald, editor, *VDM in 2000*, 2000.

[CJ91] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction*, chapter 10, pages 275–305. MIT Press, 2000.

[Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY*. PhD thesis, Louvain-la-Neuve, June 1994.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.

[dRE99] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, 1999.

[DS90] Edsger W Dijkstra and Carel S Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990. ISBN 0-387-96957-8, 3-540-96957-8.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[GJ96] J. R. Gurd and C. B. Jones. The global-yet-personal information system. In Ian Wand and Robin Milner, editors, *Computing Tomorrow*, pages 127–157. Cambridge University Press, 1996.

[GNRR93] R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors. *Hybrid Systems VIII*, volume 736 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[HJ89] C. A. R. Hoare and C. B. Jones. *Essays in Computing Science*. Prentice Hall International, 1989.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Jac83] Michael Jackson. *System Design*. Prentice-Hall International, 1983.

[Jac00] Michael Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2000.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon92] C. B. Jones. The search for tractable ways of reasoning about programs. Technical Report UMCS-92-4-4, Manchester University, 1992.

[Jon94] C. B. Jones. Process algebra arguments about an object-based design notation. In A. W. Roscoe, editor, *A Classical Mind*, chapter 14, pages 231–246. Prentice-Hall, 1994.

[Jon96a] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon96b] C. B. Jones. A rigorous approach to formal methods. *IEEE, Computer*, 29(4):20–21, 1996.

[Jon99] C. B. Jones. Scientific decisions which characterise VDM. In *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 28–47. Springer-Verlag, 1999.

[Mid93] Cornelius A. Middelburg. *Logic and Specification: Extending VDM-SL for advanced formal specification*. Chapman and Hall, 1993.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[Nor88] Donald A Norman. *The Psychology of Everyday Things*. Basic Books, 1988.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975. 75-251.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[Ran00] B. Randell. Facing up to faults. *The Computer Journal*, 43(2):95–106, 2000.

[Rea90] James Reason. *Human Error*. Cambridge University Press, 1990.

[Rus99] John Rushby. Using model checking to help discover mode confusions and other automation surprises. In *Proceedings of 3rd Workshop on Human Error*, pages 1–18. HESSD'99, 1999.

[San99] Davide Sangiorgi. Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. *Theory and Practice of Object Systems*, 5(1):25–34, 1999.

[Sat75] Edwin H. Satterthwaite. *Source Language Debugging Tools*. PhD thesis, Stanford University, 1975.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[Sit74] R.L. Sites. Some thoughts on proving clean termination of programs. Technical Report STAN-CS-74-417, Computer Science Department, Stanford University, May 1974.

[SS83] F. B. Schneider and R. D. Schlichting. Fail-stop processors: an approach to designing fault-tolerant computing systems. *TOCS*, 1(3):222–238, 1983.

[Ste66] T. B. Steel. *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. available as UMCS-91-1-1.

[SW89] I. C. Smith and D. N. Wall. Programmamable electronic systems for reactor safety. *Atom*, (395), 1989.

[WWW00a] WWW. www.dcs.gla.ac.uk/prosper/, 2000.

[WWW00b] WWW. www.dirc.org.uk, 2000.

[WWW00c] WWW. www.ifad.dk/products/vdmtools.htm, 2000.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming*. PhD thesis, Oxford University, 1992.