Panel Position statement: Reasoning about the design of programs

Cliff B Jones
University of Newcastle upon Tyne

I have long been involved in using formal notation to explain computer systems and to record our understanding. My views are therefore more concerned with the extent to which what one does when one reasons about software can be compared with normal mathematics than whether or not software theorem provers can help mathematicians.

All scientists and engineers build models which capture essential abstractions of complex systems; at different times we might focus on different facets of a system. Not only does one seek brevity and abstraction, one also seeks a tractable notation which facilitates reasoning about –or calculation of properties of– the subject system. It is often the case that rather deeper results are required to justify the use of a particular reasoning style.

I became involved in what is often termed "formal methods" (for computing) when it became clear that programming languages were becoming too complex to handle via informal methods. (Working on final testing of a major compiler for PL/I in IBM convinced me that quality cannot be achieved by any *post-hoc* technique — even though we designed automatic test tools which were ahead of their time.) The major benefit of writing a formal description of a computer system or programming language is that it helps simplify the design or "architecture": messy interactions of features are spotted long before effort to implement them uncovers the problems (which if detected late are likely to be patched in even more messy ways).

My view of the role of proof (or as I want to propose, "rigorous argument") is similar to this description of the usefulness of abstract specifications. In all but the most trivial cases, whenever I have been faced with a challenge to prove an extant program satisfies its specification, I have failed! What I have been able to do is to start again with a formal specification and systematically develop a new program. The new program might use concepts from the one I had tried to understand; it might also embody new ideas which were prompted by the abstractions. Comparison between the original program and the redeveloped one will often uncover errors in the former while the latter is complete with a design rationale which can help others understand it.

My position is that any process, that starts only after a sloppy design phase, is doomed. This is true if that *post facto* process is testing, model checking or

even proving. It is the "scrap and rework" involved in removing errors which wastes effort in software development. Formalism pays off when it can be used to detect flaws before further work is based on such mistakes. Methods like VDM or B use a "posit and prove" style which fits well with engineering practice; an individual design decision is made and justified before further work is based on it.

Numerous examples could be listed where steps of data reification or operation decomposition provide real insight into the design of a program. (One of my own current research goals is to devise a method of "atomicity refinement".)

Having been one of the first to use the term "rigorous argument" in connection with this sort of development, I'd like to say why I think it closely resembles the sort of outline proofs praised by mathematicians. One can characterize "rigour" as being capable of formalisation. In program design, a step of data reification might be justified by recording a "retrieve function"; if doubt arises, in say a review of a design step, the author can be pressed to add more detail; in the extreme, one might push this to a fully formal proof. Unlike the position of my co-panelist's original paper, I see this as being rather like the social process in mathematics.

There is of course a place for mechanical proof checkers or automatic theorem provers if they can be made sufficiently usable. Jean-Raymond Abrial –for example– has a tool which will automatically discharges the vast majority of "proof obligations" in a careful development using B. But it takes good (mathematical) taste to break the development in just the right way to achieve this. I would also add that changes *are* the norm in software and automatic tools are useful in tracing their impact.

Nor in my opinion does the above exhaust the similarities to mathematical proof. The late Ole-Johan Dahl made the point that program proving would never take off unless we built up bodies of theorems about our basic tools (abstract data objects and frequently used control constructs): this is like the task of developing an interesting mathematical theory. Beyond that, there is the justification of the methods themselves: initial forms of data reification proofs relied on a homomorphism from representation to abstraction; there were interesting cases involving non-determinism where VDM's rules were not complete; finding and justifying complete rules was an important meta result.

Finally, I should like to add a word about "separation of concerns". It is obvious that a proof about a program in a (Baroque) programming language is dubious. But a clearly recorded semantics for a safe subset of such a language can offer a way of dividing a huge task into two more tractable steps: the semantics should be used as a basis for the compiler design/verification *and* used to provide assumptions when reasoning about programs in the language.