Formal Methods in Software Engineering

C. A. R. Hoare Oct 1986.

sent to Hopcroft 20 Oct 1986

The general goal of an engineer is to design and construct a range of products to satisfy a range of human requirements, possibly evolving over time. The goal of engineering science is to formalise the design process, so as to permit collaboration of large teams of engineers, and to provide scientific and mathematical justification for confidence in the reliability and cost-effectiveness of the resulting designs.

The first task in formal design is elucidate a complete and precise specification of the full range of relevant requirements, including variations over time. Here, the overriding goal is that of clarity; there can never be a formal check on the suitability of an initial specification, so it is only the ready understanding of the engineer and of his client that protects them from the futility of solving the wrong problem.

The second task is to translate the formal statement of requirements into a formal description of a range of products, together with a plan for their implementation over a period of time. This is where formal methods are applied to split large tasks into smaller components, and large projects into smaller stages, and to define the

interfaces between them. Here, the judgement, experience, insight and inventiveness of the engineers are supplemented by a variety of symbolic calculation and proof techniques, to guarantee correctness, ~~and aid in~~ improve efficiency, and reduce the cost of the eventual product.

The third and final task is to carry out the plans and manufacture the product formally described in the design stage. If the previous stage has been successful, the design documentation is sufficiently precise and complete that its detailed implementation is entirely routine. In the case of software engineering, the final design document is a computer program — a text so formal, so precise, and so complete that no further human intervention is required: even a computer can input, translate, and follow the instructions.

These general principles of engineering design are not yet universal in computer programming projects. Nevertheless, they are taken as the basis for research in software engineering, and provide useful headings under which research results can be classified.

Research into capture of requirements has revealed the great power of Boolean logic for splitting complex descriptions into manageable components, whose details can be independently elaborated. The most natural way of splitting and combining requirements is by the humble conjunction "and". In maintaining a high level of abstraction, in surveying a wide range of requirements, and in preserving freedom for subsequent design decisions, the disjunction "or" is a most useful connective. Finally, it is often easier and clearer to describe what the product must not do, and this requires the negation operator. These connectives are generalised to quantification in the predicate calculus, which appears to be the most general basis for formal methods in software engineering.

General methods must be supplemented by more elaborate theories specialised to particular applications. For applications of computers to science and mathematics, the relevant theories are those already available within these disciplines. For other applications, the theory has been discovered, developed and applied by computer scientists. Michael Jackson, for example, has formalised capture of requirements in commercial data processing; Codd has applied relation theory to the design of data bases; and more recently, Hoare's theory of Communicating Sequential processes has been put forward as a framework for specification of

interactive and concurrent systems. The tools of the software engineer themselves require careful specification. Hoare Abrial and his followers at Oxford have applied the concepts ~~of~~ in the foundations ~~studies on~~ of mathematics to the specification of general purpose software and operating systems; and Dana Scott has laid ~~the foundations~~ a basis in domain theory for the denotational description of programming languages.

Apart from its contribution to software engineering, ~~requir~~ research into the capture of requirements promises ~~a contribution to~~ assistance in the design of simple, regular and controllable interfaces for human-computer interaction, accompanied by user manuals ~~an order of magnitude~~ shorter, clearer, and more complete than those generally prevalent today; and eventually these benefits should extend even to the standardisation of software at ~~this~~ national and international levels.

It is in the following design stage of software engineering that research into formal methods should eventually provide the most significant benefits. The reason is that both the initial specification of requirements and the final computer program are mathematical formulae, and their proper relationship can be explored by mathematical theory-building, without recourse to physical or psychological experimentation. But this is also a reason why it is difficult to obtain objective and impartial evaluation of the results of the research, which remain ~~for considerable periods~~ a subject of [illegible]

An obvious method of formalising design is to split it into a number of stages. The interface between each stage is expressed in some formal notation intermediate between the expressive generality of the mathematical concepts used in requirement specification, and the more restrictive and cumbersome notations of an efficiently implementable ~~procedural~~ programming language. The goals of research into these intermediate notations are:

1. to provide calculation and proof techniques relating it to the notations which precede and follow it in the design process
2. to provide ~~transformation~~ mathematical techniques for reasoning within the notation; and in particular for transforming a clear and efficient description (close to specification) into an efficient algorithm (close to runnable code).
3. to implement computer assistance ~~to aid in~~ for the formal manipulations described above, particularly automatic construction or checking of proofs.
4. to implement an automatic translator to enable a computer to execute the description with an efficiency adequate at least for early prototyping and other experimentation on the design.

in particular, for construction and checking of proofs

A notation close to the general purpose predicate calculus used in requirements specification is that of logic programming, recommended by Kowalski and Robinson. This allows conjunction and disjunction, but places important restrictions on negation. The language can be implemented by the tree-searching technique used in PROLOG. ~~The~~ Algebraic notation, ~~is rather more restrictive~~ exemplified by the OBJ language of Goguen and LARCH of Guttag and Horning, is more restrictive, in that disjunction is prohibited (together with existential quantification). As a result it can be implemented much more efficiently without ~~the~~ the exponential inefficiency of tree-searching.

By observing some further restrictions on the left hand sides of the algebraic equations, they can be executed as programs in a functional programming language like Turner's MIRANDA or Milner's ML. Such programs can be translated from clear ~~forms~~ descriptions to efficient algorithms by ~~used~~ mathematically sound transformations, as shown in original work by Burstall and Darlington, pursued more deeply by the CIP project in Munich. A promising new direction in research into functional programming is provided by Martin-Löf's type theory, which simultaneously defines a specification language, a programming language, and a technique for proving that programs meet their specifications. A mechanisation of this logic has been undertaken by Constable.

Programs expressed in a conventional procedural programming language are inherently more subtle and correspondingly more efficient than those expressed in a functional or logic programming language. The reason is that the programmer has more responsibility and opportunity to control the acquisition, use, reuse, and release of computing resources, particularly storage and input/output capacity. Nevertheless, one of the first successes of research in formal methods was to reveal a direct link between procedural programs and their specifications expressed in the abstract language of mathematics, without passing through any intermediate formal notation. The assertional method proposed by Floyd was developed by Hoare and Wirth to apply to programs expressed in PASCAL. It has been taken as the basis of an effective procedure for design and implementation of algorithms by Dijkstra and Gries; and it has been adopted in the Vienna Development Method for general systems programming. Mechanical assistance in construction of programs together with their proofs has been provided by Don Good's GYPSY project. These methods are being extended by Roscoe to the design of multiprocessor systems in the new programming language occam. Research and education in these areas has reached a stage at which the results should now be applied in the programming of safety-critical systems.