Chapter 1
Insight, inspiration and collaboration

C. B. Jones, A. W. Roscoe

Abstract Tony Hoare's many contributions to computing science are marked by insight that was grounded in practical programming. Many of his papers have had a profound impact on the evolution of our field; they have moreover provided a source of inspiration to several generations of researchers. We examine the development of his work through a review of the development of some of his most influential pieces of work such as Hoare logic, CSP and Unifying Theories.

1.1 Introduction

To many who know Tony Hoare only through his publications, they must often look like polished gems that come from a mind that rarely makes false steps, nor even perhaps has to work at their creation. As so often, this impression is a further compliment to someone who actually adds to very hard work and many discarded attempts the final polish that makes complex ideas relatively easy for the reader to comprehend. As indicated on page xi of [HJ89], his ideas typically go through many revisions.

The two authors of the current paper each had the honour of Tony Hoare supervising their doctoral studies in Oxford. They know at first hand his kind and generous style and will count it as an achievement if this paper can convey something of the working style of someone big enough to eschew competition and point scoring. Indeed it will be apparent from the following sections how often, having started some new way of thinking or exciting ideas, he happily leaves their exploration and development to others. We have both benefited personally from this.

Tony retired from Oxford in 1999 and has had, as we write this, 10 extremely active years at Microsoft, improving that company's software development techniques, engaging enthusiastically in the debates of the computer

science and software engineering world, promoting Grand Challenges such as the Verifying Compiler, and taking renewed interests in programming logic thanks to topics such as Separation Logic. We, however, have restricted ourselves to studying his work up to 1999 on the grounds that 10 years is already too short a time to understand the impact of academic work.

In writing about the various phases and topics of Tony's career we have tried to analyse the influences and developing themes that have run through it.

1.2 Education and early career

Charles Antony Richard Hoare, the eldest of five children (he has two brothers and two sisters), was born of British parents on 11 January 1934 in Colombo in what was then called Ceylon (now Sri Lanka). Ceylon was at that time part of the British Empire, and his father and maternal grandfather were both Englishmen engaged in the business of Empire, and from somewhat upper-class backgrounds.[1]

After his family returned to England at the end of World War II, Tony attended the Dragon School, Oxford and King's School, Canterbury before going to Oxford University to study Greats (formally known as *Literae Humaniores*) at Merton College between 1952 and 1956. Greats is Oxford's classics course, in which students study Latin and Greek for the first two years, and concentrate on philosophy, literature and ancient history for the final two. Tony specialised in modern philosophy, being taught by John Lucas, an expert on logic and Gödel in particular, who was then a Junior Research Fellow[2] at Merton. By the time the authors studied at Oxford from the mid 1970s to early 1980s, Greats had gained the reputation of being one of the best courses to do at Oxford if you wanted to become a computer programmer. So perhaps the training Greats offered in systematic thinking, particularly given Lucas' influence, was in fact the ideal education for an early computer scientist. There was no undergraduate course in computer science at Oxford until about a decade after Hoare returned as a professor.

In 1956 Tony was called up into the Royal Navy to do his "National Service", two years' military service that was compulsory for young men in the UK until the early 1960s. Perhaps thanks to his linguistic background, he went on a course on the Russian language while in the Navy.

Tony returned for a further year at Oxford after completing his National Service degree, studying Statistics. During that year he took a course in programming (Mercury Autocode) from Leslie Fox, the founding Director of

[1] See `thepeerage.com`, for example.

[2] This is a a type of position given by Oxford Colleges to allow leading young academics to pursue their research.

Oxford University Computing Laboratory, about two years after the Laboratory was founded. Fox, one of the great figures of Numerical Analysis, remained Director until he retired in 1983, at which point Hoare took over this role.

He then went to Moscow State University as a graduate student and studied Machine Translation, along with probability in the school of the great Russian Mathematician Andrey Kolmogorov. Tony states that it was there, in the context of dictionary processing, that he invented Quicksort while unaware of any sorting algorithms other than bubblesort, which he had rediscovered and decided was too slow. At the same time he began translating Russian literature on computer science into English.

On his return to England he joined the small British computer company Elliott Brothers, by whom he had been recruited while still in Moscow. One of the first tasks he was given there was to implement Shellsort in Elliott 803 Autocode. He remarks in [Hoa81b] that he then bet his manager that he had an algorithm that would usually run faster. He remarks how difficult Quicksort was to explain in the language of the time; but he won his sixpenny (£0.025) bet. He famously led the team that wrote one of the first ALGOL 60 compilers, for the Elliott 503 (the curiously numbered successor to the 803), a computer with 8K of 39-bit words and which was advertised as being able to run "as many as 200 programs per day"[3]. By the time this compiler was released in 1963 Tony had married (in 1962) Jill Pym, a member of his team. The ALGOL compiler was "one pass": in other words it only required a single pass through the source code tree of the object program.

There is no doubt that Tony's work on ALGOL helped to define his understanding of the nature of programming. Indeed, in [Hoa81b], he writes "It was there [an ALGOL 60 course in Brighton by Naur, Dijkstra and Landin which inspired him to choose this language for Elliott] that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining."

In [Hoa81b] Tony goes on to explain how his understanding of programming and the need for clear semantics of programming languages developed as the result of the failure to deliver an operating system for the Elliott 503 Mark II, and how this, in particular, inspired his work on concurrency:

> I did not see why the design and implementation of an operating system should be so much more difficult than that of a compiler. This is the reason why I have devoted my later research to problems of parallel programming and language constructs which would assist in clear structuring of operating systems–constructs such as monitors and communicating processes.

Tony reached the position of Chief Engineer at Elliott Brothers, but decided to leave because of the effects of the company being taken over in 1968. His academic career therefore began after he saw an advertisement for the

[3] According to [27] when the compiler was run on the much slower 803 a typical half-page ALGOL program would take half an hour to compile and execute.

position of Professor at the Queen's University, Belfast. By this time his position in the developing subject of computer science was secure thanks not only to Quicksort but, perhaps more importantly, to the collaborations and contacts he obtained through his ALGOL work and the work he was doing on the ALGOL Working Group (IFIP WG2.1). By the time Tony was recruited to Oxford in 1977 no application was necessary: he was simply contacted and told that he had been elected to the job.

1.3 Programming languages

Hoare's most explicit set of positive rules for designers of programming languages was titled "Hints on Programming Language Design". This was originally written for the first (ACM SIGPLAN) POPL conference held in Boston in October 1973. Sadly, the paper did not appear in the proceedings but has been reprinted several times in slightly different forms — probably the most accessible electronic version is [Hoa73a]. Rather than repeat the points in this important paper, can we encourage our readers to study it? This plea is most strongly addressed to anyone who is thinking of designing a new language.

The importance that Tony Hoare attaches to programming languages is made abundantly clear in his acceptance speech for the ACM Turing Award.[4] This 1980 speech is published as [Hoa81b]. As mentioned in Section 1.2, he remarks there how he could only express Quicksort elegantly after he had seen ALGOL.

Hoare also makes clear that he sees it "as the highest goal of programming language design to enable good ideas to be elegantly expressed". Later in the same paper he observes the importance of "programming notations so as to maximise the number of errors which cannot be made, or if made, can be reliably detected at compile time."

ALGOL 60 had been devised by a committee; but a committee of the highest calibre. Hoare was invited to join IFIP WG 2.1 in August 1962. One of the proposals on which he looked back with pride is the "switch" concept.

His work with Niklaus Wirth to clean up ALGOL 60 led to the elegant ALGOL W proposal in [WH66] which in turn paved the way for Pascal.[5] Sadly, WG 2.1 saw fit to go another way and invent ALGOL 68 [Hoa68]: a language which gives rise in [Hoa81b] to one of Tony's most biting aphorisms

> There are two ways of constructing a software design: one way is to make is so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies.

[4] The Turing Award is often referred to as the "Nobel Prize for computing". It is not clear that the Kyoto Prize committee would concede this — but Tony Hoare has been awarded both.

[5] Probably because of his respect for this language, Hoare outlined its defects in [WSH77].

He recounts the final denouement at which a subset of the members of WG 2.1 submitted a minority report containing the comment "as a tool for *reliable creation* of sophisticated programs, the language was a failure."

The over-ambitiousness of the PL/I project is also described in [Hoa81b] from the standpoint of the ECMA committee of which Tony was initially a member and which he ultimately chaired. After listing an (extremely sobering) litany of failures, Tony writes "I knew that it would be impossible to write a wholly reliable compiler for a language of this complexity and impossible to write a wholly reliable program ...". Again, he ends his observations on this language with the withering observation "The price of reliability is the utmost simplicity. It is the price that the very rich find most hard to pay."

The obvious and then topical reason for selecting the theme of programming language design for his Turing Award lecture was the evolution of the language which became known as Ada. As he observes, Tony had offered advice and judgement that largely went unheeded.

What he did instead was to lead by example. Mastering concurrency is still a major challenge for designers of programming languages. Tony's early work in this area is described in Section 1.6.1 below, but once he saw the depth of the questions surrounding communication, he took the radical step of studying it as "Communicating Sequential Processes": CSP is explored below in Sections 1.6.2 and 1.6.3. This in turn led to his work on occam (see Section 1.7), a language named after an earlier Oxford philosopher whose famous principle *Occam's Razor* was in harmony with Tony's views on programming languages: *entia non sunt multiplicanda praeter necessitatem*, in other words "entities must not be multiplied more than necessary".

1.4 Reasoning about sequential programs

Hoare's "Axiomatic basis" paper [Hoa69] is one of the most influential in the computing literature. It marks a transition from simply adding assertions to programs towards a position that increasingly emphasised reasoning in entirely non-operational terms about the claim that programs match their specifications.

To understand its contribution, it is essential to outline where most researchers in the field stood in the 1960s. There are hints of the need to reason about programs in [12] and a proposal in [35] for an approach that uses a clear notion of assertions being added to a flowchart of a program. The latter paper went unnoticed for decades and had no influence on the development of ideas. Furthermore, as the only mention traced in his writing, one can only guess at the scope of what Turing had in mind. Turing's assertions appear to be limited to relational expressions between (values of) variables of the program. Far more influential than either of these contributions from

the 1940s was Bob Floyd's paper [10]. Floyd again places his assertions on a flowchart but the language in which the assertions are written here is (first-order) predicate calculus. This means that Floyd could be, and was, much more precise than Turing was about the validity of assertions; the fact that he was using a higher-level programming language than his predecessor also helped.

Somewhat before 1969 –in 1964 to be precise– there was an important meeting in Baden-bei-Wien (Austria) organised by Heinz Zemanek of the IBM Vienna Laboratory. This was, in fact, the first of many highly influential IFIP working Conferences and led to the creation if IFIP Working Group 2.2. The proceedings took some time to be published but [33] is invaluable in understanding scientific opinion of the time and, specifically, in charting the development of Hoare's thinking. From the conference proceedings, it is clear that considerable attention was given to the need for, and challenges of, formally defining the semantics of programming languages. McCarthy's clarion call of [21] to define semantics formally is backed up in [22] by an operational semantics of "Micro-ALGOL". Both Strachey and Landin discuss the connections between programming languages and Church's Lambda Calculus. On the other hand, Jan Garwick's paper opens with the provocative sentence: "No programming language for a given computer can be better defined than by its compiler."

Hoare did not present a formal contribution, but one of the helpfully recorded discussion items [33, pp. 142–143] indicates his perception of "the need to leave [aspects of] languages undefined". At the following meeting of IFIP WG 2.1, Hoare gave the example of fixing the meaning of functions like *mod* (modulus) by stating their required properties.

The IBM Vienna group borrowed concepts from McCarthy, Landin and Cal Elgot as the basis for the first version of the huge operational semantics for the language PL/I. This approach was to be named "Vienna Definition Language" (VDL) — see [20]. In 1965, Hoare attended a course in Vienna on VDL. The Vienna Lab at that time tended to do things in style and the ECMA TC10 guests were booked into the Hotel Imperial (where the British Queen stayed on her visit a few years later). On paper of that imperious hotel, Tony Hoare wrote a sketch of his first attempt at an axiomatic treatment of languages. Of the two-part draft dated December 1967, the first axiomatised execution traces as a partial order on states. It is probably fair to say that the objectives are clearer in the 1967 draft than the outcome. Hoare sent these notes to (at least) Peter Lucas of the Vienna group.

Tony recalled years after the event that, on his arrival to take up his chair in Belfast in October 1968, he "stumbled upon" the mimeographed draft (dated 20 May 1966) of Floyd's paper [10]. Peter Lucas had sent this partly as a response to Hoare's 1967 draft. Floyd's ideas on predicate calculus assertions had a major impact on Hoare's thinking and the debt is clearly acknowledged in [Hoa69]. Hoare produced in December 1968 a further two-part draft that strongly resembles the final Communications of the ACM paper.

The first part addresses the thorny issue that numbers stored in computers are not quite the same as those of mathematics — in this Hoare was following van Wijngaarden's lead in [36], which is again gratefully acknowledged in the CACM paper. The second part of the 1968 draft contains the core of what is today called "Hoare axioms".

In this draft, Hoare followed Floyd's original "forwards" assignment axiom, which requires an existential quantifier in the post condition of any assignment statement. The now common "backwards" rule that only needs substitution was first published in Jim King's thesis [17], where he attributes it to his supervisor Bob Floyd. Hoare uses this in the published version of "axiomatic basis" [Hoa69]; he was made aware of the idea by David Cooper who gave a seminar at Belfast on his return from a sabbatical in Pittsburgh. Hoare's decision to use this version possibly sparked the later development of "weakest pre condition" thinking.

Hoare's paper was quickly accepted by CACM and was far more approachable than Floyd's earlier paper. Where Floyd had bundled together many ideas including early hints of what would later become known as "healthiness conditions" for proof rules, Hoare limited what he covered even to the point of not handling termination.[6]

Much the most important step was the move away from Floyd's flowcharts to a view of program texts decorated with axioms as part of a unified formal system. It was this point that changed the way whole generations of researchers have been persuaded to approach programming. Hoare's decision to use post conditions of the final state alone led to concise axioms, but it is fair to say that later he conceded the value of using relational post conditions that link to the initial state as well.

As soon as a decade after the first appearance of the "axiomatic basis" paper, Krzysztof Apt published a summary of its already significant impact in [2].[7] Hoare's language had only sequential composition, conditional and a "while" repetitive construct; attention soon turned to tackling other features commonly found in high-level programming languages and relevant papers include [Hoa71a, Hoa72a, ACH76]. An attempt at the whole of Pascal [HW73] was however incomplete but this is indicative of the fact that formalism makes its largest contribution if used during –rather than after– design. In fact, the only language with a complete Hoare axiom system is probably "Turing" [13].

Given the unbridled enthusiasm of researchers to propose new languages, a far more productive avenue was probably that of showing where clean axiomatisations were consistent with subsets of languages. Peter Lauer did part of his PhD under Tony Hoare in Belfast and Lauer's thesis [18] is clearly summarised in their joint paper [HL74]. A fuller discussion of the history and impact of research on reasoning about programs can be found in [15].

---

[6] In terminology that some find unfortunate –but that has become ubiquitous– he limited himself to "partial correctness" whereas Floyd treated "total correctness".

[7] The slightly enigmatic "Part I" sub-title indicates Apt's strong interest in nondeterminism which he covered in [3].

1.5 Formal program development

Hoare's axioms in [Hoa69] possessed a crucial property that was not exploited within that paper: the given axioms are "compositional" in a sense that made it possible to employ them to reason about combinations of yet-to-be-developed code (e.g. one can prove that a while construct satisfies a specification even where its body is so far only a specification). Technically, each axiom is monotonic in the satisfaction ordering; practically, this opens the door to their use is stepwise development. Hoare first wrote his "Proof of a Program: *Find*" as a *post facto* proof of correctness but revised it before publication as [Hoa71b] to describe a stepwise development. The omission in not revising the title is surprising from a writer who takes such care with the prose of each revision.

The final text in [Hoa71b] is far more readable than the first version, more importantly, it is also much more convincing. Recall that there were almost no programs available in the early 1970s to check (let alone help construct) such predicate calculus proofs so the move to a top-down development of *Find* decomposed the proof into more manageable steps. Given the comments in Section 1.4 above, it will come as no surprise that the termination proof (that Tony conceded "was more than usually complex") had to be handled separately from that for correctness (see [Hoa71b, §4]). Furthermore, the decision to use post conditions of only the final state left the need for a section entitled "Reservation" ([Hoa71b, §5]) that concedes "one very important aspect of correctness has not been treated, namely that the algorithm merely rearranges the elements of array A without changing any of their values"; post conditions of two states would have allowed the "permutation" property to have been handled within the main proof.

One extremely important and far sighted point was the recognition of the way that programs can be designed via their loop invariants.

Having developed the method, it was possible in fairly short order to apply it to a range of problems:

- [FH71] returns to the *Quicksort* algorithm discussed in Section 1.2 above and presents its stepwise development.
- [Hoa72c] tackles finding primes using the "sieve of Eratosthenes" (a problem which is used to illustrate the development of concurrent implementations in [Hoa75] and by other subsequent authors).
- [Hoa73b] nicely links to Hoare's work on operating systems by tackling a structured paging system.

"Structured Programming"

Many who know the literature on "Structured Programming" might be surprised that the important book [Hoa72b] has not yet been mentioned. The book is widely cited and the topic of its title has had major impact on software design. Like many successful ideas, however, the term was abused to cover a range of things from a narrow message of "avoiding goto statements" through to systematic, justified design processes. Hoare's solo chapter [DDH72][8] starts with the ringing

> "In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction."

and goes on to provide a masterly description of concepts of data structuring for programming languages.

Data refinement

Another important contribution to the formal, stepwise, development of programs was the recognition of the importance of using, in specifications, objects that are abstract in the sense that they match the problem being described. Development by data refinement can then bring in representations on which efficient programs can be based as part of the design process. Hoare's paper [Hoa72d] is widely cited as one that recognised this aspect of formal program development. In common with other authors, Hoare recognised later that the neat homomorphism rule does not cover all situations and was a coauthor of [HHS86] which presents a more general rule.

An interesting success of Tony's ability to inspire other scientific activity was the way he brought Jean-Raymond Abrial and the first author together in Oxford. In 1979, Abrial was working on ideas that were eventually developed into the "Z" specification language. Jones had been a key member of the Vienna work on denotational semantics which had been published in [4, 5] and had developed his earlier ideas on program development to provide the other part of VDM: [14] was printed in the famous "red and white" Prentice-Hall series edited by Hoare.

Tony thought it would be interesting for both Abrial and Jones to share an office when they both arrived in Oxford in 1979 — at that time the "Programming Research Group" (PRG) had rather cramped quarters in 45 Banbury Road. This was certainly an inspired and inspiring idea. Often a discussion would result in a blackboard containing a mixture of notations

[8] The material had been presented in his Marktoberdorf lectures of 1970.

but the fact that the basic ideas of abstraction were shared meant that the focus was on the underlying issues.[9]

The language known as "Z" continued to develop after Abrial and Jones left Oxford and has become a widely used specification language with tool support. Its use on IBM's "CICS" system led to a Queen's Award for Technological Achievement to the Oxford PRG group in 1992.

1.6 Concurrency

As shown by the quotation in Section 1.2, Hoare's work on concurrency was inspired by the problems of operating system design. This influence is still very much apparent in the text and examples in his 1985 book on CSP (Chapter 6, for example).

The driving theme in his work is the need to keep separate threads from interfering with each other in ways that are undesired or hard to understand. We can see this in the evolution from work based on shared memory to CSP, in which all interaction is via explicit communication over channels.

1.6.1 Concurrency with shared variables

Before turning, as we do in the next section, to Tony's most radical and influential suggestion of communication-based concurrency, it is important to understand his earlier attempts to tame its shared-variable cousin. It is easy to decry the use of variables that can be changed by more than one thread, but at the machine interface there is little else. There were various suggestions for programming constructs to make this troublesome fact tolerable: the hope to extend the axiomatic approach to concurrency was already there in [Hoa69]; in [Hoa72e], Hoare had tackled the sort of disjoint parallelism that could be controlled by conditional critical sections. In [Hoa75] he moves on to more general concurrency governed by his "monitor" proposal [Hoa74].

In "Parallel Programming: An Axiomatic Approach", Hoare carefully distinguishes:

- disjoint processes: [Hoa75, §3] essentially reproduces the earlier "symmetric parallel rule", but the rule is still limited to partial correctness. In view of the way parallelism is often used, this is perhaps more reasonable here.

[9] The development of Abrial's ideas through to "B" [1] (and beyond) deserves separate discussion elsewhere.

- competition ([Hoa75, §4]) clearly establishes the notion of ownership[10].
- cooperation ([Hoa75, §5]) recognises the importance of a commutativity requirement between operations in the cooperating processes. It is here that Hoare returns to the "sieve of Eratosthenes" from [Hoa72c]: he also concedes that "when a variable is a large data structure ... the apparently atomic operations upon it may in practice require many actual atomic machine instructions". The atomicity issue is extremely important and has been pursued by other researchers (e.g. [16]).
- the section on communication gives an insight into the way Hoare develops ideas: ([Hoa75, §6]) recognises that communication does not fit the commutativity property above; he introduces a notion of "semi-commutativity" that clearly only handles uni-directional communication. One can see here the seeds of CSP (see Section 1.6.2) whose realisation took several years of further hard struggle, before it could be published.

1.6.2 Imperative CSP

Tony published two works entitled "Communicating Sequential Processes", the CACM paper from 1978 and the 1985 book. The languages in these publications are very different from each other. In this section and the next we discuss the development of the two versions, and try to understand why they are as they are. The first version of the language is essentially Dijkstra's language of guarded commands [8] (a simple imperative language) with point-to-point communication added, so we have termed it *Imperative* CSP.

Hoare states[11] that the move from studying concurrency via shared variables and monitors to explicit communication over channels was inspired by the advent of the microprocessor and the thought (later realised in the transputer) of these "communicating with other microprocessors of a similar nature along wires".

An Imperative CSP program is a parallel composition of named sequential processes: the only parallelism is at the highest level. Thus this language fits the name *Communicating Sequential Processes* much better than Algebraic CSP, where the parallel operator would be on a par with all others, or even occam where the same is true.

Hoare was clearly inspired by examples such as the sieve of Erastothenes, where it was natural to create an array of processes with closely related structure, and included explicit notation for addressing members of an indexed array of processes in the language.

He specifies that communication between processes is *synchronised* (only taking place when both outputter and inputter are ready) but gives little ex-

[10] At the April 2009 event to celebrate Tony's 75th birthday, Peter O'Hearn linked this to his own research on Separation Logic.

[11] In the interview with Bowen cited in Sources.

planation of this decision, which was to prove so important in the structure of the many languages and theories that would be inspired by CSP. This clearly fits well with the intuition of communication being direct from one process to another along a piece of wire, and Hoare implies that where buffering is wanted it can be introduced explicitly via buffer processes.

It is natural in contemplating communications between two processes to think of one as outputting and one as inputting, and in this first version of CSP Hoare makes this an important distinction. For example, as would later be the case in occam, only the input end of a communication may appear with alternatives. He does, however, discuss allowing outputs in guarded alternatives, and in doing so raises the possibility of proving the parallel program $[X!2 \parallel Y!3]$ equivalent to a sequential one, while commenting that this is not achieved by the program

$$[true \rightarrow X!2;\ Y!3 \ \Box\ true \rightarrow Y!3;\ X!2]$$

in which the implementation is permitted to resolve the choice, thereby flagging the importance of nondeterminism in reasoning about CSP. This problem would be solved by using outputs directly in the guards:

$$[iX!2 \rightarrow Y!3 \ \Box\ Y!3 \rightarrow X!2]$$

This one example and the motivation behind it is a powerful indication of an inevitable move towards an algebra of communication, concurrency and nondeterminism.

1.6.3 Algebraic CSP

The language in the 1985 book started to develop even before the 1978 paper was published. Indeed, by the time the second author joined Tony as a research student in October 1978 the *process algebra* CSP (i.e. the one in the 1985 book) was almost completely formed as a notation and Tony was working on his traces model.

The most obvious difference between the old and new CSPs is that the former is a conventional programming language with point-to-point communication added in a natural way, whereas the new one looks like some sort of abstract algebra, hence our name *Algebraic CSP*. Indeed it is one of the first two developed examples (the other being CCS) of what rapidly became known as a *process algebra*: a notation for creating algebraic terms representing the interacting behaviour of a number of distinct *processes*. These processes are themselves patterns of communication: it would be wrong to call them *threads* since there is no guarantee that the processes are sequential. Indeed, in both CSP and CCS and most subsequent process algebras, there is

no semantic distinction between sequential and parallel processes, and every parallel process is equivalent to a sequential one,

¿From this discussion alone the reader will appreciate that the creation of these first process algebras represented a huge intellectual step: that from a programming language to calculi that attempt to attribute meaning to patterns of communication.

The need for such a meaning was clear from the fact that concurrent programs behave so differently from sequential ones, with phenomena such as deadlock and livelock to worry about, as well as the nondeterminism caused by resource contention (common in operating systems) and similar situations that are intrinsic to concurrency. This led both Hoare and Milner down remarkably similar paths: discarding almost all the things that programs do between communications, and developing notations that allowed them to concentrate purely on the synchronised communications between processes and the way in which patterns of these arise.

Whether CSP and CCS seem similar to a reader will depend on his or her viewpoint, but certainly they are very similar when viewed from the standpoint of Imperative CSP. Both Hoare and Milner were working on this convergent course before either had a clear idea of what the other was doing. Milner, indeed, had been looking at the semantics of interaction since the early 1970s. He had started off [23] working on the semantics of shared-variable "transducers" and by 1977 was working on "flow graphs", a partly graphical notation of parallel composition that was not specific about the protocol used on channels. They only got a clear vision of each other's work at a meeting in Aarhus in June 1977, by which time Hoare had already put considerable efforts into understanding the algebra of CSP. Hoare's paper at that workshop (which we have unfortunately been unable to locate) was entitled "A relational trace-oriented semantics for Communicating Sequential Processes". W.P. de Roever worked with Hoare in Belfast in 1977 on the semantics of CSP, the results of which were reported in [FHLdR79]. Both this work and Milner's initial thoughts on concurrent semantics centred on domain theory. This was very natural in the context of the times, given the success of Strachey, Scott and others in developing and applying domain theory to the semantics of a wide range of programming language constructs in the preceding years. The main challenge to domain theory inherent in giving semantics to communicating processes was the need to handle an interleaving sequence of external choices and nondeterministic choice. External choices could be handled with function spaces, which work extremely elegantly in domain theory. But nondeterministic choice seemed to require *powerdomains* [26], in other words a domain-theoretic analogue of the powerset operator.

There are two major problems with powerdomains. The first is that it proves very difficult to combine the natural set theoretic order structure with the order of the underlying domain in a satisfactory way, and none of the available orders (including the strong Egli-Milner order of the Plotkin pow-

erdomain and the angelic order of the Hoare powerdomain[12]) produce equivalences between processes at a persuasive level of abstraction. The second is that the powerdomains are in themselves difficult to understand, meaning that any semantics based on them is unlikely to be useful in explaining concurrency beyond a select group of researchers.

Of course the reason for needing domain theory for other "interesting" languages is the quality of self-referentiality they have, in particular programs that accept other programs as functional arguments, as exemplified by the λ-calculus. Dijkstra's guarded command language, without such constructs, can easily be given a semantics as a relation on $S \times (S \cup \{\bot\})$, where S is the set of states and $\bot$ represents non-termination or *divergence*. So while concurrency is itself certainly "interesting", there is nothing in Imperative CSP that implies the need for domain theory. And both Milner, and Hoare in turn, reacted against powerdomains. Indeed Willem-Paul de Roever tells us this this was evident during his 1977 visit: the powerdomain models were "not to Tony's taste", and Tony would regularly propose models that were converging on the traces model.

Milner chose an operationally based theory in which equivalences are developed between processes described as labelled transition systems. Hoare has stated that his approach to process equivalence, based on algebraic laws relating processes and behaviourally-based models, was a reaction against Milner's operational approach. Both their philosophies, quite clearly, have been extremely successful. Of course they have long since reconciled, for example by the development of operational semantics for CSP and the testing equivalences for CCS. It was these constrasting decisions, of course, which led to the different choice operators of CCS [25] and CSP: the CCS "+" being the obvious operational version.

There are two other interesting contrasts between CCS and (Algebraic) CSP. The first is that CSP contains a great many more operators than CCS, such as sequential composition, very general renaming and interrupt. Here, Hoare seems to have been driven by the types of system he wished to model, for example Imperative CSP (with sequential composition, of course) where variables are modelled as parallel processes, and operating systems where processes are interrupted and checkpointed.

The second is the very different factorisation of the "natural" parallel construct in which multiple processes communicate point-to-point over channels with these communications internalised, or hidden. Here Hoare seems to have been inspired by the algebra of synchronisation, interleaving and hiding, which required process *alphabets* to determine which events are synchronised. Milner, on the other hand, has stated that he decided to avoid using alphabets and was able to do so by devising an extremely clever device whereby (i) events synchronise not with themselves but with duals (ii) dual events

[12] The powerdomain of downward-closed sets was not developed by Hoare, but named after him by Plotkin, because of its close relationship to Tony's important work on partial correctness.

in parallel processes can either synchronise to become τ or happen independently, and (iii) such "free radical" events are restricted outside the syntactic level where synchronisation can occur. This approach is already evident in his flow-graph work, for example [24]. Milner's trick makes multi-way synchronisation unnatural (because of the duality), and in any case requires events to be hidden once synchronised. So CSP ended up with much more flexible parallel and hiding operators, at the expense of the need to declare (in one way or another) the interfaces that parallel processes use to communicate with each other.

It is hard to overstate the importance of algebraic laws in guiding Hoare's intuition about what were the "right" models of CSP. In particular, his belief in the laws of distribution over non-deterministic choice corresponds almost exactly, theoretically, to the decision to model processes as sets of *linearly observable* behaviours. This means that each behaviour may be observed as time progresses forward on a single execution of the process: in particular no branching behaviour is recorded.

The following few paragraphs describe the development, during 1979, of the failures model. It was clear that the traces model was too weak: a model was required that distinguished nondeterministic from external choice, and which captured the phenomena of deadlock and livelock accurately.

The failures model started its life as the *acceptances* model, in which a process was modelled as the set of pairs (s, A) where s is a trace and A is a set of events from which the process accepts. The meaning of this phrase is deliberately vague, since HBR (Hoare, Brookes and Roscoe) spent some time experimenting with it. After a few weeks working on this, they came to the conclusion that a good way to interpret this was "the process can choose to restrict its next actions to being within A". In other words, P *actually* offers some subset of A. HBR realised the interesting fact that this interpretation fails to distinguish between the processes[13]

$$STOP \sqcap (?x : \{a, b\} \to STOP) \quad \text{and}$$
$$STOP \sqcap (a \to STOP) \sqcap (?x : \{a, b\} \to STOP)$$

even though the first (unlike the second) has no state from which it *actually* accepts $\{a\}$. Nevertheless, the interpretation in which these two are equated was found to be a congruence which, if one interprets livelock as the most nondeterministic process $CHAOS$, seemed to satisfy the better set of algebraic laws than did what is now known as the *acceptances*, or *ready sets* congruence, which distinguishes these two processes. In the latter, the alge-

[13] $P \sqcap Q$ is a *nondeterministic* process which is itself allowed to decide which of P and Q to run. Thus the second of these two processes can opt to offer just the event a, while the first has to offer nothing at all ($STOP$) or $\{a, b\}$.

braic law $P = P \,\Box\, P$ does not hold, for example[14]. That congruence was later developed by Olderog and Hoare [HO83].

Quickly after this, HBR decided to turn their somewhat convoluted acceptance sets into refusal sets by the simple device of complementation: where A was an acceptance set, $\Sigma \setminus A$ (Σ being the set of all visible events) was a *refusal set*, which could be understood as being a set of events that the process might not accept a member from, even if offered it for ever. Since HBR's acceptance sets were upward closed (if a process can accept from A, it can also accept from $A' \supseteq A$), refusal sets became downward closed, but somehow this seemed much more natural.

The result, a pair (s, X) where s is a trace and X is one of these refusal sets was called a *failure* because it represents an experiment to which the process fails to respond. The model of [HBR81, BHR84] is, of course, excellent at representing nondeterminism, and makes it very clear that $P \sqsubseteq Q$ (refinement modelled by reverse containment) corresponds to P being more nondeterministic than Q. It is worth noting that the healthiness conditions of this model were in effect a statement of what a process *should* look like, rather than being derived from an operational semantics, since none of these then existed and LTS's were not considered!

It was immediately apparent that the refinement maximal elements of this model were in natural 1–1 correspondence with the traces model and were exactly the deterministic processes, judged extensionally. Its refinement-minimum element was $CHAOS$, the most nondeterministic process which contained every failure imaginable.

The obvious choice for a least fixed point based semantics for recursion was therefore based on the refinement order, but this was in any case appealing since it meant iteration corresponded to reduction of nondeterminism and identified the undefined recursion $\mu\, p.p$ with the sort of divergence produced by hiding $(\mu\, p.a \rightarrow p) \setminus \{a\}$.

Unsurprisingly, given the heritage we have described, the second author discovered the fatal flaw in the original failures model by observing that a self-evident law failed, namely

$$(P \parallel Q) \setminus A = (P \setminus A) \parallel (Q \setminus A) \quad \text{if } A \cap \alpha P \cap \alpha Q = \emptyset$$

This is the principle that, provided no synchronised events are hidden, one can distribute hiding over parallel. The fatal flaw is that the identification of divergence with $CHAOS$ is not robust enough to survive some of the operator definitions in CSP, so the expression on the right above might have some of the behaviours introduced from divergence removed by the parallel composition.

[14] $P \,\Box\, Q$ means that the environment has the choice of the initial events offered by P and Q. The counter-intuitive failure of this law comes about because in $P \,\Box\, P$ the two copies of P might, because they resolve nondeterminism differently, choose to offer different sets, which the operator combines into a single offer that P alone cannot make. This distinction is made in the acceptances or ready-sets congruence, but not using the upwards-closed version of acceptances.

This flaw was also discovered by de Nicola and Hennessy [9]. Their work was communicated to Brookes, who had recently moved to Pittsburgh with Dana Scott. The fix to this flaw, the *failures-divergences* model in which the failure set was augmented was thus discovered separately by Brookes and the second author, appearing in [28, 7, 6]. This model, consistently with the intuition about divergence in [BHR84], has explicitly *strict* divergence: no attempt is made to see what goes on after a process might have diverged[15]

This brought the theory of CSP to the level that was presented in the 1985 book, and since Hoare's involvement in the development of its core theory since then has been relatively small we will leave it here[16].

This book was developed by Tony over a period of several years, parts of it having appeared as a technical report in 1983, together with a separate set of exercises. Tony was able to try it on numerous groups of students, for example those studying the MSc in Computation which Tony had helped set up in Oxford in 1979.

In the early 1980s, Hoare developed techniques [Hoa81a, Hoa85a] both for specifying and giving semantics to CSP processes in the predicate calculus: a program or specification is described as a predicate calculus formula over formal variables representing a typical trace, a typical refusal set coupled with that trace, a divergence etc. By describing not the whole process, but a typical individual behaviour, in this way, the resulting semantics – albeit just a recoding of the set-theory based ones discussed above – gained much in elegance. For example, the specification that a process P whose alphabet is $in.T \cup out.T$ is a buffer in terms of traces is just written P **sat** $tr \downarrow out \leq tr \downarrow in$, with the quantification of tr over all traces of P being implicit. This work was not restricted to CSP, as shown by the paper "Programs are Predicates" [Hoa85b], which clearly links it with the earlier work on Hoare logic.

As part of this project he developed some new logical notations, such as $x \lhd b \rhd y$, the infix version of *if b then x else y*. The point of the operator $\lhd b \rhd$, of course, is that it puts conditional choice at the same linguistic level as the other choice operators $\sqcap$ and $\Box$, allowing it to be compared with these and reasoned about with similar laws.

By incorporating ideas such as these he made the 1985 book a masterpiece of presentation: it succeeded in making material which in truth is really quite difficult seem accessible, natural and elegant.

---

[15] The intuition that divergence should be disasterous, derived from the first failures model, was very strong at that time. It is interesting that neither Brookes nor the second author then discovered the "stable failures model", in which divergence is not recorded, so (as in the traces model) the simply divergent process is top of the refinement order. The existence of that model was conjectured by Albert Meyer and Lalita Jagadeesan in the late 1980s, and developed by the second author following a conversation with them.

[16] The second author's paper elsewhere in this volume illustrates how well CSP has stood the test of time. His 1997 book [31] and forthcoming book [32] both give extensive updates on CSP, its theory, tools and applications.

1.7 occam and the Transputer

In 1978 the UK government sponsored the creation of a company called inmos, led by Iann Barron, part of whose vision was to create components for parallel processing systems. This would be a microchip company to rival the foreign giants, it was hoped. Its design operations were based in Bristol and it had a fabrication plant in Newport, South Wales. Barron's vision of a network of components interacting via serial links emerged from his work for the Science Research Council on its Distributed Computing Program. This brought him into contact with the ideas of Hoare and Milner, and the more practically-based work of David May at Warwick on the design and implementation of distributed systems and languages to program them.

inmos's early products were memory chips, but it was always anticipated by Barron that its flagship product would be the *transputer*, a single chip that contained a processor, cache memory and communications hardware. Barron hired Hoare and May as consultants in 1978, and the latter joined inmos as full-time "Chief Technologist" in mid 1979. This team refined the concept of a transputer.

May, with input from Hoare, designed the *occam* programming language, which is a low-level imperative language based on CSP, inheriting features both of Imperative CSP and Algebraic CSP (the latter including the idea of parallel as a first-class language construct). May has told us various respects, such as having the `ALT` construct analogous to external choice $\Box$ rather than explicit channel polling, in which occam moved closer to CSP during its design process. This particular change – lobbied for by Hoare – was doubtless in pursuit of the stated goal of giving occam the cleanest possible semantics. The fact that occam was so cleanly defined and so close to CSP meant that it had clean *formal* semantics. The second author and Hoare each played a large role in defining these through papers such as [29] (denotational semantics), [HR84] (logical semantics in the sense discussed above) [RH86] (algebraic semantics).

It was, of course, extremely bold (and some might put it stronger than that) of inmos to base itself on such a novel product and a completely novel language. The transputer's primary market was intended, from a fairly early stage, to be in embedded applications, but naturally it was the prospect of large parallel systems composed of many transputers that caught the imagination. The first (16-bit) transputers were delivered in 1985, with 32-bit ones following soon afterwards.

May and Hoare had extraordinary vision when it came to the use of occam, and the language quickly became the main medium by which hardware was specified within inmos. They realised that the clean semantics of occam made this an excellent vehicle for formal verification work. The first major exercise in this did not involve occam's parallel capabilities at all, but solely involved reasoning about sequential occam programs: the microcoded instructions for the FPU of the T800 transputer. This project, which was conceived

by Hoare and May, involved translating the IEEE 754 floating-point number standard into Z, developing correct occam programs from that for its various operations, and proving these equivalent to highly stylised occam programs representing the microcode programs designed for a special data path. The translation of the specification was done by Geoff Barrett, as was the derivation of the top level programs from these. The proofs of equivalence were performed by David Shepherd using the *occam transformation system* [11], a tool that implemented the algebraic semantics for occam developed by Hoare and the second author in [RH86].

The error-free FPU was developed at a considerable saving to what would have been achieved with a traditional testing regime.

The use of occam in hardware design at inmos was taken to entirely new levels in the design of the T9000: a pipelined processor, executing RISC-style instructions that were automatically grouped into compound instructions, and with far more advanced communications hardware. Associated formal methods work was successful [30], but no longer involved Hoare closely.

Unfortunately it became apparent that a company the size of inmos (by 1990 a branch SGS Thomson Microelectronics) could no longer compete with the investment put in by the giants in leading-edge microprocessors, so the transputer concept and with it its implementation of CSP ceased to be developed in the early 1990s.

It is of course interesting that the lesson of how valuable formal methods are to microprocessors was not learned by these giants until one of them had a problem with a floating point unit some years after the successful Oxford/inmos collaboration. These companies are now by far the biggest users of formal verification.

1.8 Unified Theories of Programming

He Jifeng first came to work in Oxford in 1984, and he remained there, either full or part time, until 1998: one year before Tony's retirement. For most of this time, he and Tony were a close working partnership. As Tony's ideas stretched beyond CSP to the idea of correctness in a completely general setting, Jifeng provided him with mathematical support in a similar sense that Brookes and the second author had done on CSP and occam.

One early project was "The laws of programming" [HHH$^+$87], a project that drew on earlier work on algebra for the more complex language occam [19] and set out a programme for using algebraic laws for language definition and formal methods. Part of this programme was the use of *weakest prespecifications*, as defined in [HH86], the paper which began serious Hoare/He effort to understand programming and specification via the relational calculus.

They used the relational calculus to discover much about the nature of specification and implementation, using such tools as Galois connections to relate programming and specification constructs. One important idea that emerged from this work was that of an operational semantics defined in terms of algebraic transformation towards normal form: see [HHS93], for example.

Hoare and He took on the extremely ambitious project of creating a framework in which one could give semantics, make specifications, reason about and relate a wide range of programming languages such as imperative, logical and concurrent languages. This led to the book *Unifying Theories of Programming* [HH98], and, though the book makes few explicit references to the relational calculus, it is there throughout as the mathemtical foundation upon which this work is built.

In the book, we can see direct influences from all the previous joint work of Hoare and He that we have discussed in this section as well as Hoare's earlier work on algebraic laws, coding semantics in predicate calculus, and logical notation. Certainly it is easy to see the roots of it in Hoare's intuitions about CSP and its presentation in the late 1970s and early 1980s. In this respect we are thinking not only of the importance of algebraic laws and of the logical representation of observable behaviour, but also about the problem of how to construct denotational semantics without domain theory.

Sources

The main source we used on Tony's research is, of course, his published work of which we give a bibliography below. He has also written several articles containing reminiscences, most notably [Hoa81b], and there are several published interviews with him, of which we have used the following:

```
www.simple-talk.com/opinion/geek-of-the-week/
sir-tony-hoare-geek-of-the-week

archive.computerhistory.org/resources/text/
Oral_History/Hoare_Sir_Antony/102658017.05.01.pdf
```

This paper has two bibliographies. The first lists all of Hoare's papers that we have either cited above or which do not appear in the bibliography of his papers to 1987 that appeared in *Essays in computing science* [HJ89], a book that arose from discussions between the first author and Tony in Austin Texas. The second bibliography consists of all those papers we have referred to that do not have Tony as an author. To help the reader distinguish between the two, citations in the Hoare bibliography (which is sorted into date order) are given thus [Hoa81b] while those in the second (sorted alphabetically) are numerically labelled [7].

Acknowledgements

The authors are grateful to Tony Hoare and Robin Milner for their recollections about the development of process algebra, to David May, Gordon Plotkin, Brian Randell, Willem-Paul de Roever and others for their memories and comments, and to many of Tony's academic descendants for contributing to the family tree below.

We are extremely grateful to Lucy Li of Oxford University Computing Laboratory who undertook the monumental task of assembling the family tree as well as helping us to put together the extended bibliography.

The first author's research is supported by the EPSRC Platform Grant on "Trustworthy Ambient Systems" and EU FP7 "DEPLOY project". The second author's is supported by the EPSRC Grant "CSP Model Checking: New Technologies and Techniques" and by grants from the US ONR.

Bibliography 1: Papers by Hoare

[Hoa68] C.A.R. Hoare. Critique of ALGOL 68. *ALGOL Bulletin*, 29:27–29, November 1968.

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

[FH71] M. Foley and C.A.R. Hoare. Proof of a recursive program: Quicksort. *BCS, Computer Journal*, 14(4):391–395, November 1971.

[Hoa71a] C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium On Semantics of Algorithmic Languages — Lecture Notes in Mathematics 188*, pages 102–116. Springer-Verlag, 1971.

[Hoa71b] C.A.R. Hoare. Proof of a program: Find. *Communications of the ACM*, 14(1):39–45, January 1971.

[DDH72] O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors. *Structured Programming*. Academic Press, 1972.

[Hoa72a] C.A.R. Hoare. A Note on the FOR Statement. *BIT*, 12(3):334–341, 1972.

[Hoa72b] C.A.R. Hoare. Notes on data structuring. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 83–174. Academic Press, 1972.

[Hoa72c] C.A.R. Hoare. Proof of a structured program: 'The Sieve of Eratosthenes'. *BCS, Computer Journal*, 15(4):321–325, November 1972.

[Hoa72d] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[Hoa72e] C.A.R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[Hoa73a] C.A.R. Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Stanford, October 1973.

[Hoa73b] C.A.R. Hoare. A structured paging system. *BCS, Computer Journal*, 16(3):209–215, August 1973.

[HW73] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[HL74] C.A.R. Hoare and P.E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3(2):135–153, 1974.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa75] C.A.R. Hoare. Parallel programming: An axiomatic approach. *Computer Languages*, 1(2):151–160, June 1975.

[ACH76] E.A. Ashcroft, M. Clint, and C.A.R. Hoare. Remarks on "program proving: Jumps and functions". *Acta Informatica*, 6(3):317–318, 1976.

[WSH77] J. Welsh, W.J. Sneeringer, and C.A.R. Hoare. Ambiguities and insecurities in PASCAL. *Software Practice and Experience*, 7(6):685–96, November – December 1977.

[FHLdR79] N. Francez, C.A.R. Hoare, D.J. Lehmann, and W.P. de Roever. Semantics of nondeterminism, concurrency and communication. *Journal of Computer and System Sciences*, 19(3):290–308, December 1979.

[HBR81] C.A.R. Hoare, S.D. Brookes, and A.W. Roscoe. A theory of communicating sequential processes. Technical Report PRG 16, Oxford University Computing Laboratory, Programming Research Group, 1981.

[Hoa81a] C.A.R. Hoare. A calculus of total correctness for communicating processes. *The Science of Computer Programming*, 1(1–2):49–72, October 1981.

[Hoa81b] C.A.R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, February 1981.

[HO83] C.A.R. Hoare and E.R. Olderog. Specification-oriented semantics for communicating processes. In *Automata Languages and Programming 10th Colloquium*, volume 154 of *Lecture Notes in Computer Science*, pages 561–572. Springer-Verlag, 1983.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[HR84] C.A.R. Hoare and A.W. Roscoe. Programs as executable predicates. In *Proceedings of the International Conference on Fifth Generation Computer Systems, November 6–9 1984, Tokyo, Japan*, pages 220–228. ICOT, 1984.

[Hoa85a] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 256 pages, ISBN 0-13-153271-5.

[Hoa85b] C.A.R. Hoare. Programs are predicates. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 141–154. Prentice-Hall, 1985.

[HH86] C.A.R. Hoare and J. He. The weakest prespecification I. *Fundamenta Informaticae*, 9(1):51–84, March 1986.

[HHS86] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *ESOP '86: Proceedings of the European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[RH86] A.W. Roscoe and C.A.R. Hoare. Laws of occam programming. Monograph PRG-53, Oxford University Computing Laboratory, Programming Research Group, February 1986.

[HHH+87] C.A.R. Hoare, I.J. Hayes, Jifeng He, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sørensen, J.M. Spivey, and B.A. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987. see Corrigenda in Communications of the ACM, 30(9): 770.

* *

The following is a list of all Hoare's papers since 1988, complementing the list published in [HJ89].

[HG88] C. A. R. Hoare and Michael J. C. Gordon. Partial correctness of CMOS switching circuits: An exercise in applied logic. In *LICS*, pages 28–36, 1988.

[RH88] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theor. Comput. Sci.*, 60:177–229, 1988.

[HH89] Jifeng He and C. A. R. Hoare. Categorical semantics for programming languages. In *Mathematical Foundations of Programming Semantics*, pages 402–417, 1989.

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

[Hoa89] C. A. R. Hoare. The varieties of programming language. In *TAPSOFT, Vol.1*, pages 1–18, 1989.

[BHL90] Dines Bjørner, C. A. R. Hoare, and Hans Langmaack, editors. *VDM '90, VDM and Z - Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, Proceedings*, volume 428 of *Lecture Notes in Computer Science*. Springer, 1990.

[Hoa90a] C. A. R. Hoare. Fixed points of increasing functions. *Inf. Process. Lett.*, 34(3):111–112, 1990.

[Hoa90b] C. A. R. Hoare. Let's make models (abstract). In *CONCUR*, page 32, 1990.

[Hoa90c] C. A. R. Hoare. A theory of conjunction and concurrency. In *PARBASE / Architectures*, pages 18–30, 1990.

[Hoa91a] C. A. R. Hoare. A theory for the derivation of combinational CMOS circuit designs. *Theor. Comput. Sci.*, 90(1):235–251, 1991.

[Hoa91b] C. A. R. Hoare. The transputer and occam: A personal story. *Concurrency - Practice and Experience*, 3(4):249–264, 1991.

[MHH91] C. E. Martin, C. A. R. Hoare, and Jifeng He. Pre-adjunctions in order enriched categories. *Mathematical Structures in Computer Science*, 1(2):141–158, 1991.

[ZHR91] Chaochen Zhou, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276, 1991.

[HG] C.A.R. Hoare and M. J. C. Gordon, editors. *Mechanised Reasoning and Hardware Design*. Prentice Hall International Series in Computer Science. ISBN 0-13-572405-8, 1992.

[Hoa92] C. A. R. Hoare. Programs are predicates. In *FGCS*, pages 211–218, 1992.

[ZH92] Chaochen Zhou and C. A. R. Hoare. A model for synchronous switching circuits and its theory of correctness. *Formal Methods in System Design*, 1(1):7–28, 1992.

[HH93] Jifeng He and C. A. R. Hoare. From algebra to operational semantics. *Inf. Process. Lett.*, 45(2):75–80, 1993.

[HHS93] C. A. R. Hoare, Jifeng He, and A. Sampaio. Normal form approach to compiler design. *Acta informatica*, 30(8):701–739, 1993.

[Hoa93] C. A. R. Hoare. Algebra and models. In *SIGSOFT FSE*, pages 1–8, 1993.

[HHF+94] Jifeng He, C. A. R. Hoare, Martin Fränzle, Markus Müller-Olm, Ernst-Rüdiger Olderog, Michael Schenke, Michael R. Hansen, Anders P. Ravn, and Hans Rischel. Provably correct systems. In *FTRTFT*, pages 288–335, 1994.

[Hoa94] C. A. R. Hoare. Editorial. *J. Log. Comput.*, 4(3):215–216, 1994.

[HP94] C. A. R. Hoare and Innes Page. Hardware and software: The closing gap. In *Programming Languages and System Architectures*, pages 49–68, 1994.

[Hoa95] C. A. R. Hoare. Unification of theories: A challenge for computing science. In *COMPASS/ADT*, pages 49–57, 1995.

[vKH95] Burghard von Karger and C. A. R. Hoare. Sequential calculus. *Inf. Process. Lett.*, 53(3):123–130, 1995.

[Hoa96a] C. A. R. Hoare. How did software get so reliable without proof? In *FME*, pages 1–17, 1996.

[Hoa96b] C. A. R. Hoare. The logic of engineering design. *Microprocessing and Microprogramming*, 41(8-9):525–539, 1996.

[Hoa96c] C. A. R. Hoare. Mathematical models for computing science. In *NATO ASI DPD*, pages 115–164, 1996.

[Hoa96d] C. A. R. Hoare. The role of formal techniques: Past, current and future or how did software get so reliable without proof? (extended abstract). In *ICSE*, pages 233–234, 1996.

[Hoa96e] C. A. R. Hoare. Unifying theories : A personal statement. *ACM Comput. Surv.*, 28(4es):46, 1996.

[WH66] N. Wirth and C.A.R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9(6):413–432, June 1966.

[HH97] C. A. R. Hoare and Jifeng He. Unifying theories for parallel programming. In *Euro-Par*, pages 15–30, 1997.

[HH98] C.A.R. Hoare and Jifeng He. *Unifying theories of programming.* Prentice Hall, 1998.

[HH99a] Jifeng He and C. A. R. Hoare. Linking theories in probabilistic programming. *Inf. Sci.*, 119(3-4):205–218, 1999.

[HH99b] C. A. R. Hoare and Jifeng He. A trace model for pointers and objects. In *ECOOP*, pages 1–17, 1999.

[Hoa99a] C. A. R. Hoare. Theories of programming: Top-down and bottom-up and meeting in the middle. In *Correct System Design*, pages 3–28, 1999.

[Hoa99b] C. A. R. Hoare. Theories of programming: Top-down and bottom-up and meeting in the middle. In *World Congress on Formal Methods*, pages 1–27, 1999.

[JRH+99] Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *PLDI*, pages 25–36, 1999.

[SSH99] Silvija Seres, J. Michael Spivey, and C. A. R. Hoare. Algebra of logic programming. In *ICLP*, pages 184–199, 1999.

[HH00] Jifeng He and C. A. R. Hoare. Unifying theories of healthiness condition. In *APSEC*, pages 70–, 2000.

[HHS00] C. A. R. Hoare, Jifeng He, and Augusto Sampaio. Algebraic derivation of an operational semantics. In *Proof, Language, and Interaction*, pages 77–98, 2000.

[Hoa00a] C. A. R. Hoare. Assertions. In *IFM*, pages 1–2, 2000.

[Hoa00b] C. A. R. Hoare. A hard act to follow. *Higher-Order and Symbolic Computation*, 13(1/2):71–72, 2000.

[Hoa00c] C. A. R. Hoare. Legacy code. In *ICFEM*, page 75, 2000.

[Hoa01a] C. A. R. Hoare. Growing use of assertions. In *TOOLS (38)*, page 3, 2001.

[Hoa01b] C. A. R. Hoare. Legacy. *Inf. Process. Lett.*, 77(2-4):123–129, 2001.

[BFG+02] Robert S. Boyer, W. H. J. Feijen, David Gries, C. A. R. Hoare, Jayadev Misra, J. Moore, and H. Richards. In memoriam: Edsger w. Dijkstra 1930-2002. *Commun. ACM*, 45(10):21–22, 2002.

[Hoa02a] C. A. R. Hoare. Assertions in modern software engineering practice. In *COMPSAC*, pages 459–462, 2002.

[Hoa02b] C. A. R. Hoare. Assertions in programming: From scientific theory to engineering practice. In *Soft-Ware*, pages 350–351, 2002.

[Hoa02c] C. A. R. Hoare. Towards the verifying compiler. In *10th Anniversary Colloquium of UNU/IIST*, pages 151–160, 2002.

[Hoa03a] C. A. R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

[Hoa03f] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003. (This paper also appeared in a number of other publications.)

[BHF04] Michael J. Butler, C. A. R. Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, pages 133–150, 2004.

[FHRR04] Cédric Fournet, C. A. R. Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-free conformance. In *CAV*, pages 242–254, 2004.

[Hoa04a] C. A. R. Hoare. Process algebra: A unifying approach. In *25 Years Communicating Sequential Processes*, pages 36–60, 2004.

[Hoa04b] C. A. R. Hoare. Towards the verifying compiler. In *Essays in Memory of Ole-Johan Dahl*, pages 124–136, 2004.

[BBF+05] Roberto Bruni, Michael J. Butler, Carla Ferreira, C. A. R. Hoare, Hernán C. Melgratti, and Ugo Montanari. Comparing two approaches to compensable flow composition. In *CONCUR*, pages 383–397, 2005.

[HH05] Jifeng He and C. A. R. Hoare. Linking theories of concurrency. In *ICTAC*, pages 303–317, 2005.

[HM05a] C. A. R. Hoare and Robin Milner. Grand challenges for computing research. *Comput. J.*, 48(1):49–52, 2005.

[HM05b] C. A. R. Hoare and Jayadev Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. In *VSTTE*, pages 1–18, 2005.

[BHH+06] Bernhard Beckert, C. A. R. Hoare, Reiner Hähnle, Douglas R. Smith, Cordell Green, Silvio Ranise, Cesare Tinelli, Thomas Ball, and Sriram K. Rajamani. Intelligent systems and formal methods in software engineering. *IEEE Intelligent Systems*, 21(6):71–81, 2006.

[BHW06] Juan Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: a step towards the verifying compiler. *Formal Asp. Comput.*, 18(2):143–151, 2006.

[HH06] Jifeng He and C. A. R. Hoare. CSP is a retract of CCS. In *UTP*, pages 38–62, 2006.

[Hoa06b] C. A. R. Hoare. The ideal of verified software. In *CAV*, pages 5–16, 2006.

[Hoa06c] C. A. R. Hoare. Why ever CSP? *Electr. Notes Theor. Comput. Sci.*, 162:209–215, 2006.

[VHHS06] Viktor Vafeiadis, Maurice Herlihy, C. A. R. Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPOPP*, pages 129–136, 2006.

[Hoa07a] C. A. R. Hoare. Fine-grain concurrency. In *CPA*, pages 1–19, 2007.

[Hoa07b] C. A. R. Hoare. The ideal of program correctness: *hird Computer Journal* lecture. *Comput. J.*, 50(3):254–260, 2007.

[Hoa07c] C. A. R. Hoare. Science and engineering: A collusion of cultures. In *DSN*, pages 2–9, 2007.

[HO08] C. A. R. Hoare and Peter W. O'Hearn. Separation logic semantics for communicating processes. *Electr. Notes Theor. Comput. Sci.*, 212:3–25, 2008.

[Hoa08a] C. A. R. Hoare. Keynote: A vision for the science of computing. In *BCS Int. Acad. Conf.*, pages 1–29, 2008.

[Hoa08b] C. A. R. Hoare. Verification of fine-grain concurrent programs. *Electr. Notes Theor. Comput. Sci.*, 209:165–171, 2008.

[Hoa08c] C. A. R. Hoare. Verified software: Theories, tools, experiments. In *ICECCS*, page 3, 2008.

[HM09] C. A. R. Hoare and Jayadev Misra. Preface to special issue on software verification. *ACM Comput. Surv.*, 41(4), 2009.

[HMLS09] C. A. R. Hoare, Jayadev Misra, Gary T. Leavens, and Natarajan Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41(4), 2009.

[HMSW09a] C. A. R. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene algebra. In *CONCUR*, pages 399–414, 2009.

[HMSW09b] C. A. R. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Foundations of concurrent Kleene algebra. In *RelMiCS*, pages 166–186, 2009.

[Hoa09] C. A. R. Hoare. Viewpoint - retrospective: an axiomatic basis for computer programming. *Commun. ACM*, 52(10):30–32, 2009.

[WHO09] Ian Wehrman, C. A. R. Hoare, and Peter W. O'Hearn. Graphical models of separation logic. *Inf. Process. Lett.*, 109(17):1001–1004, 2009.

Bibliography 2: papers by other authors

1. J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.
2. K. R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3:431–483, 1981.
3. K. R. Apt. Ten years of Hoare's logic: A survey – part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.
4. H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A formal definition of a PL/I subset. Technical Report 25.139, IBM Laboratory Vienna, December 1974.
5. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.
6. S.D. Brookes and A.W. Roscoe. An improved failures model for communicating processes. 1985.
7. S.D. Brookes. *A mathematical theory of communicating processes*. PhD thesis, University of Oxford, 1983.
8. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
9. R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. 1983.
10. 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.
11. M.H. Goldsmith and A.W. Roscoe. Transformation of occam programs. In *Design and Application of Parallel Digital Processors, 1988*, pages 180–188, 1988.
12. H. H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument, 1947. Part II, Vol. 1 of a Report prepared for U.S. Army Ord. Dept.; republished as pages 80–151 of [34].
13. Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Programming Language: Design and Defintion*. Prentice Hall International, 1988.
14. C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980.
15. Cliff B. Jones. The early search for tractable ways of reasonning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.
16. C. B. Jones. Splitting atoms safely. *Theoretical Computer Science*, 357:109–119, 2007.
17. J. C. King. *A Program Verifier*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1969.
18. P. E. Lauer. *Consistent Formal Theories of the Semantics of Programming Languages*. PhD thesis, Queen's University of Belfast, 1971. Printed as TR 25.121, IBM Lab. Vienna.
19. INMOS Ltd. *Occam programming manual*. Prentice Hall, 1984.
20. P. Lucas and K. Walk. *On The Formal Description of PL/I*, volume 6, Part 3 of *Annual Review in Automatic Programming*. Pergamon Press, 1969.
21. J. McCarthy. A basis for a mathematical theory for computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland Publishing Company, 1963. (A slightly extended and corrected version of a talk given at the May 1961 Western Joint Computer Conference).
22. J. McCarthy. A formal description of a subset of ALGOL. In *[33]*, pages 1–12, 1966.

23. R. Milner. Processes: a mathematical model of computing agents. In *Logic Colloquium.73*. North Holland, 1973.
24. R. Milner. Flowgraphs and flow algebras. *Journal of the ACM*, 26(4):794–818, 1979.
25. R. Milner. *A calculus of communicating systems*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1982.
26. GD Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5:452, 1976.
27. E.D. Reilly. *Milestones in computer science and information technology*. Greenwood Pub Group, 2003.
28. A.W. Roscoe. *A mathematical theory of communicating processes*. PhD thesis, University of Oxford, 1982.
29. A.W. Roscoe. Denotational Semantics for occam. 1985.
30. A.W. Roscoe. Occam in the specification and verification of microprocessors. *Philosophical Transactions: Physical Sciences and Engineering*, pages 137–151, 1992.
31. A.W. Roscoe. *The theory and practice of concurrency*. Prentice-Hall, 1997.
32. A.W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.
33. T. B. Steel. *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.
34. A. H. Taub, editor. *John von Neumann: Collected Works*, volume V: Design of Computers, Theory of Automata and Numerical Analysis. Pergamon Press, 1963.
35. A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, June 1949.
36. A. van Wijngaarden. Numerical analysis as an independent science. *BIT*, 6:66–81, 1966. (Text of 1964 talk).

The academic family tree of C.A.R. Hoare

We began this article with some remarks about Tony's family background. We conclude it with as much as we have been able to piece together about his *academic* family, of which we are both proud members, namely his doctoral students, their doctoral students and so on. To become a member of the family below we asked that a student had successfully completed their doctorate by the time this article was finalised. This family is not entirely without incest, in that some students were jointly supervised by two other members of the tree (sometimes including Tony himself). We have organised the tree below so that each student is given as short a route to Tony as possible. Entries in *italics* (e.g. Roscoe's supervision of Kong) indicate that the student's main entry is elsewhere. A joint supervisor (JS) in *italics* is elsewhere in the tree.

In most cases, for brevity, we give here the *topic* or *area* of the thesis, rather than its title.

We intend to supply our information, including titles where we have them, to the Mathematics Genealogy Project[17], from where, in turn, some of this

[17] http://genealogy.math.ndsu.nodak.edu/

information was gathered. Therefore if you have any corrections or additions to this tree, we encourage you to upload the details there.

C.A.R. Hoare

1 Peter Lauer, Belfast 1971, *Axiomatic semantics*

1.1 Eike Best, Newcastle[18] 1981, *Concurrency* (JS Brian Randell)
1.1.1 Lucia Pomello, Milano and Torino 1988, *Petri nets* (JS Giorgio De Michelis and Mariangiola Dezani-Ciancaglini)
1.1.1.1 Stefania Rombolà, Milano-Bicocca 2009, *Concurrency theory* (JS Luca Bernardinello)
1.1.2 Javier Esparza, Hildesheim 1993, *Model checking*
1.1.2.1 Richard Mayr, Munich 1998, *Infinite state systems*
1.1.2.1.1 Noomene Ben Henda, Uppsala 2008, *Infinite-state systems*
1.1.2.2 Stephan Melzer, Munich 1998, *Verifying distributed systems*
1.1.2.3 Stefan Römer, Munich 2000, *Verification using unfoldings*
1.1.2.4 Christine Röckl, Munich 2001, *Validation of reactive and mobile systems* (JS Davide Sangiorgi)
1.1.2.5 Leonor Prensa-Nieto, Munich 2002, *Verification of parallel programs in Isabelle/HOL* (JS *Nipkow*)
1.1.2.6 Stefan Schwoon, Munich 2002, *Checking pushdown systems*
1.1.2.7 Barbara König, Stuttgart 2004, *Analysing dynamic systems*
1.1.2.8 Alin Stefanescu, Stuttgart 2006, *Distributed systems*
1.1.2.9 Claus Schröter, Stuttgart 2006, *Partial-order verification*
1.1.2.10 Dejvuth Suwimonteerabuth, Munich 2009, *Pushdown systems*
1.1.2.11 Stefan Kiefer, Munich 2009, *Positive polynomial equations*
1.1.3 Hans-Günther Linde-Göers, Hildesheim 1994, *Partial order semantics*
1.1.4 Hanna Klaudel, Paris 1995, *Algebraic Petri net models* (JS Elisabeth Pelz)
1.1.4.1 *Robert-Christoph Riemann*, (see 1.1.6)
1.1.4.2 Franck Pommereau, Paris 2002, *Petri nets & real-time*(JS Pelz)
1.1.4.3 Cécile Bui-Thanh, Paris 2004, *OO concurrent programs*
1.1.4.4 Cédric Siléo, Evry 2007, *Adaptive systems* (JS Hutzler)
1.1.4.5 Benoît Calvez, Evry 2007, *Agent-based Simulations* (JS Hutzler)
1.1.4.6 Anastassia Yartseva, Evry 2007, *Modelling of biological networks* (JS Franois Képès and Raymond Devillers)
1.1.5 Jon Hall, Newcastle 1996, *An Algebra of high-level Petri nets* (JS Maciej Koutny and Richard Hopkins)
1.1.5.1 Adrian Hilton, OU[19] 2004, *Hardware-software co-design*
1.1.5.2 Zhi Li, OU 2007, *Deriving specifications* (JS Lucia Rapanotti)

[18] University of Newcastle upon Tyne

[19] Open University

[20] Technical University of Munich

7.4.6 David von Oheimb, TUM 2001, *Analyzing Java*
7.4.7 Leonor Prensa Nieto, TUM 2002, *Parallel program verification*
7.4.8 Markus Wenzel, TUM 2002, *Human-readable formal proofs*
7.4.9 Gerwin Klein, TUM 2003, *Verifying a bytecode verifier*
7.4.9.1 Harvey Tuch, UNSW 2008, *Formal memory models*
7.4.10 Stefan Berghofer, TUM 2003, *Executable specifications*
7.4.11 Gertrud Bauer, TUM 2006, *Plane graph theory*
7.4.12 Norbert Schirmer, TUM 2006, *Verifying sequential code*
7.4.13 Martin Wildmoser, TUM 2006, *Proof carrying code*
7.4.14 Amine Chaieb, TUM 2008, *Automated formal proofs*
7.4.15 Tjark Weber, TUM 2008, *SAT-based finite model generation*
7.4.16 Steven Obua, TUM 2008, *Flyspeck II: The basic linear programs*
7.4.17 Alexander Krauss, TUM 2009, *Recursion in higher-order logic*
7.4.18 Florian Haftmann, TUM 2009, *Code generation*
7.5 Mario Wolczko, Manchester 1988, *Semantics of OO languages*
7.6 Ralf Kneuper, Manchester 1989, *Symbolic execution*
7.7 Jean Alain Ah-Kee, Manchester 1989, *Operation decomposition*
7.8 Ketil Stølen, Manchester 1990, *Parallel programs*
7.8.1 Ida Hogganvik, Oslo 2007, *Security risk analysis*
7.8.2 Ragnhild Kobro Rund, Oslo 2007, *UML interaction diagrams*
7.8.3 Mass Soldal Lund, Oslo 2008, *Sequence diagram specifications*
7.8.4 Atle Refsdal, Oslo 2008, *Probabilistic sequence diagrams*
7.8.5 Bjørnar Solhaug, Bergen 2009, *Sequence diagrams*
7.9 John Fitzgerald, Manchester 1991, *Formal specification*
7.9.1 Andrej Pietschker, Newcastle 2001, *Automated Test Generation*
7.9.2 Daniel Owen, Newcastle 2005, *Real-time networks*
7.9.3 *Neil Henderson, Newcastle 2006, Asynch. communications* (see 7.14)
7.10 Alan Wills, Manchester 1993, *Formal methods in OO*
7.11 Carlos de Figueiredo, Manchester 1994, *Object-based languages*
7.11.1 Elaine Pimentel, UFMG[21] 2001, *Sequent calculus* (JS Dale Miller)
7.11.2 Cristiano Vasconcellos, UFMG 2004, *Type inference*
7.12 Juan Bicarregui, Manchester 1995, *Model-oriented specification*
7.13 Steve Hodges, Manchester 1996, *Development of object-based programs*
7.14 Neil Henderson, Newcastle 2006, *Asynch. communications* (JS *Fitzgerald*)
7.15 Tony Lawrie, Newcastle 2006, *System dependability*
7.16 Martin Ellis, Newcastle 2008, *Compiler-generated function units*
7.17 Joseph Coleman, Newcastle 2008, *Operational semantics*
7.18 David Greathead, Newcastle 2008, *Code comprehension* (JS George Erdos and Joan Harvey)
7.19 Ken Pierce, Newcastle 2009, *Atomicity in refinement*

8 Bill Roscoe, Oxford 1982, *Concurrency*

[21] Federal University of Minas Gerais, Brazil

8.1 *Tat Yung Kong, Oxford 1986, Digital topology* (see 14)
8.2 Andrew Boucher, Oxford 1986, *Real-time semantics for occam*
8.3 Mike Reed[22], Oxford 1988, *Timed CSP*
8.3.1 Steve Schneider, Oxford 1989, *Timed CSP*
8.3.1.1 Carl Adekunle, RHUL 2000, *Feature interaction*
8.3.1.2 Helen Treharne, RHUL 2000, *Controlling specifications*
8.3.1.2.1 Damien Karkinsky, Surrey 2007, *Mobile B*
8.3.1.2.2 Chris Culnane, Surrey 2009, *Digital watermarking*
8.3.1.3 James Heather, RHUL 2000, *Security protocols*
8.3.1.3.1 Kun Wei, Surrey 2006, *Automated proofs and CSP*
8.3.1.3.2 James Salter, Surrey 2007, *P2P networks* (JS Antonopoulos)
8.3.1.3.3 *Zhe Xia, Surrey 2009, Electronic voting* (see 8.3.1.8)
8.3.1.4 Neil Evans, RHUL 2003, *Security*
8.3.1.5 Roberto Delicata, Surrey 2006, *Security*
8.3.1.6 Wilson Ifill, Surrey 2008, *CSPB*
8.3.1.7 Siraj Shaikh, University of Gloucestershire 2008, *Security*
8.3.1.8 Zhe Xia, Surrey 2009, *Electronic voting* (JS *Heather*)
8.3.2 David Jackson, Oxford 1992, *Verification of timed CSP*
8.3.3 Abida Mukkaram, Oxford 1993, *Refusal testing in CSP*
8.3.4 Joel Ouaknine, Oxford 2001, *Timed CSP*
8.4 Naiem Dathi, Oxford 1989, *Deadlock avoidance*
8.5 Martin Ward, Oxford 1989, *Program transformation*
8.5.1 Matthias Ladkau, De Montfort 2008, *Program transformation*
8.5.2 Shaoyun Li, De Montfort 2008, *Program transformation*
8.5.3 Stefan Natelberg, De Montfort 2009, *Prog. transf.* (JS Zedan)
8.6 Geoff Barrett, Oxford 1989, *Operational semantics of the transputer*
8.7 Alan Jeffrey, Oxford 1989, *Timed concurrency*
8.7.1 Ralf Schweimeier, Sussex 2000, *Categorical models*
8.8 Gavin Lowe, Oxford 1993, *Priority and probability in timed CSP*
8.8.1 Mei Lin Hui, Leicester 2001, *Security protocols*
8.8.2 Gordon Rohrmair, Oxford 2005, *Security*
8.8.3 Chris Dilloway, Oxford 2008, *Security protocols*
8.8.4 Allaa Kamil, Oxford 2009, *Security protocols*
8.9 Brian Scott, Oxford 1995, *Semantics of occam II*
8.10 Lars Wulf, Oxford 1996, *Noninterference security*
8.11 Bryan Scattergood, Oxford 1998, *Tools for CSP*
8.12 Ranko Lazić, Oxford 1999, *Data independence*
8.12.1 *Tom Newcomb, Oxford 2003, Data independence* (see 8.16)
8.12.2 Aleksandar Dimovski, Warwick 2007, *Game semantics*
8.13 Richard Forster, Oxford 1999, *Non-interference*
8.14 Sadie Creese, Oxford 2001, *Inductive proofs of networks*

[22] Mike Reed has two doctorates, one from Auburn in 1970 under Ben Fitzpatrick on Set-theoretic Topology, and the one listed above. In addition to Reed's Computer Science students listed above, he has had a number in topology, but we have decided not to list those here.

8.15 Philippa Broadfoot, Oxford 2002, *Cryptographic protocols*
8.16 Tom Newcomb, Oxford 2003 *Data independence* (JS *Lazić*)
8.17 Richard Tolcher, Oxford 2006, *Ethnomethodology*
8.18 Lee Momtahan, Oxford 2007, *Data independence*
8.19 Eldar Kleiner, Oxford 2008, *Security protocols*
8.20 Long Nguyen, Oxford 2009, *Pervasive computing security*

9 Geraint Jones, Oxford 1983, *Timed concurrency*
10 Stephen Brookes, Oxford 1983, *Concurrency*

10.1 Michel Schellekens, CMU 1995, *Semantics and complexity*
10.1.1 David Hickey, UCC[23] 2008, *Static analysis*
10.1.2 Maria O'Keeffe, UCC 2008, *Analysis of algorithms*
10.2 Shai Geva, CMU 1996, *Higher-order sequential computation*
10.3 Susan Older, CMU 1996, *Fair communicating processes*
10.3.1 Dan Zhou, Syracuse 1999, *Secure email*
10.3.2 Jang Dae Kim, Syracuse 2002, *Instruction-set calculus*
10.3.3 Byoung Woo Min, Syracuse 2005, *Instruction-set calculus*
10.3.4 Polar Humenn, Syracuse 2008, *The authorisation calculus*
10.4 Denis Dancanet, CMU 1998, *Intensional investigations*
10.5 Jüergen Dingel, CMU 1999, *Parallel programs*
10.5.1 Jeremy Bradbury, Queen's University[24] 2007, *Mutation testing* (JS Jim Cordy)
10.5.2 Michelle Crane, Queen's University 2008, *UML modeling*
10.5.3 Hongzhi Liang, Queen's University 2009, *Sequence diagrams*
10.6 Kathryn Van Stone, CMU 2003, *Denotational complexity*

11 Andrew Black, Oxford 1984, *Exception handling*

11.1 Leif Nielsen, UW[25] 1986, *Concurrent program development*
11.2 Norman Hutchinson, UW 1987, *Distributed systems* (JS Hank Levy)
11.2.1 Clair Bowman, Arizona 1990, *Descriptive naming*
11.2.2 Michael Coffin, Arizona 1990, *Parallel programming*
11.2.3 Sean O'Malley, Arizona 1990, *Programming with protocols*
11.2.4 Siu Ling Lo, UBC[26] 1995, *Real-Time tasking*
11.2.5 Peter Smith, UBC 1998, *Process migration*
11.2.6 Alistair Veitch, UBC 1998, *Operating systems*
11.2.7 Dwight Makaroff, UBC 1998, *Media file-servers*
11.2.8 Dima Brodsky, UBC 2005, *Policy driven replication*
11.2.9 Chamath Keppitiyagama, UBC 2005, *Multiparty communication*
11.2.10 Lechang Cheng, UBC 2009, *Distributed systems*
11.3 Eric Jul, UW 1988, *Object mobility* (JS Hank Levy)

[23] University College Cork
[24] Queen's University, Kingston, Ontario
[25] University of Washington
[26] University of British Columbia

- 11.3.1 Charlotte Lunau, Copenhagen 1989, *OO programing*
- 11.3.2 Birger Andersen, Copenhagen 1991, *OO parallelism*
- 11.3.3 Niels Juul, Copenhagen 1993, *Concurrent garbage collection*
- 11.3.4 Povl Koch, Copenhagen 1996, *Distributed systems*
- 11.3.5 Jørgen Hansen[27], Copenhagen 2000, *Distributed systems*
- 11.3.6 Czeslaw Kazimierczak, Copenhagen 2001, *Distributed systems*

11.4 Emerson Murphy-Hill, Portland State 2009, *Refactoring tools*

12 Christopher Dollin, Oxford 1984, *Language definition*
13 Alex Teruel[28], Oxford 1985, *Formal specification*
14 Tat Yung Kong, Oxford 1986, *Digital topology* (JS *Roscoe*)

- 14.1 Cherng-Min Ma, CUNY[29] 1994, *Digital topology*
- 14.2 Chyi-Jou Gau, CUNY 2005, *Digital topology*

15 Bryan Todd, Oxford 1988, *Formally based diagnostics*
16 Stephen Page, Oxford 1988, *Music information retrieval*
17 Jeremy Jacob, Oxford 1989, *Shared systems*

- 17.1 Phillip Brooke, York 1999, *Timed semantics for DORIS*
  - 17.1.1 Shukor Razak, Plymouth 2007, *Intrusion detection*
  - 17.1.2 Paul Hunton, Teesside 2007, *Performance management*
  - 17.1.3 Graham Evans, Teesside 2009, *Technology enabled change*
- 17.2 John Clark, York 2001, *Metaheuristic search in cryptology*
  - 17.2.1 Thitima Srivatanakul, York 2005, *Security analysis*
  - 17.2.2 Howard Chivers, York 2006, *Security analysis*
  - 17.2.3 Yuan Zhan, York 2006, *Test-set generation*
  - 17.2.4 Paul Massey, York 2007, *Quantum Software*
  - 17.2.5 *Hao Chen, York 2007, Security protocol synthesis* (see 17.5)
- 17.3 Sun Woo Kim, York 2001, *OO mutation testing*
- 17.4 Nathalie L Foster, York 2003, *Security protocol development*
- 17.5 Hao Chen, York 2007, *Security protocol synthesis* (JS *Clark)*

18 Clare Martin, Oxford 1991, *Predicate transformers*
19 Ken Wood, Oxford 1992, *Parallel logic simulation*
20 Augusto Sampaio, Oxford 1993, *Compiling using normal forms*

- 20.1 Leila de Almeida e Silva, UFPe[30] 2000, *HW/SW partitioning*
- 20.2 Alexandre Cabral, Mota UFPe 2001, *Model checking CSPZ*

---

[27] Hansen supervised a number of students in Health-related topics at Groningen before undertaking his doctoral studies with Jul.

[28] Alex Teruel set up the Parallel and Distributed Research Group at Simon Bolivar University, Venezuela. He supervised numerous MSc students there but advised them to do their PhD's abroad. He is happy to report that since a number of these people did this successfully and now have completed PhD students of their own, the fruits of Tony's advisorship are thriving in Venezuela.

[29] City University of New York

[30] Federal University of Pernambuco, Brazil

20.2.1 Adalberto Cajueiro de Farias, UFPe 2009, *Abstraction in CSPZ*
20.3 Roberta Lopes, UFPe 2003, *Formal methods for genetic algorithms*
20.4 Adolfo de Almeida Duran, UFPe2005, *Algebraic design of compilers*
20.5 Adnan El Sherif, UFPe 2006, *Formal methods in timed systems*

21 Stephen Brien, Oxford 1995, *Generically typed set theory*
22 Paul Rudin, Oxford 2001, *Diagrammatic reasoning*