A Brief Scientific Biography of Robin Milner

Gordon Plotkin, Colin Stirling & Mads Tofte

Robin Milner was born in 1934 to John Theodore Milner and Muriel Emily Milner. His father was an infantry officer and at one time commanded the Worcestershire Regiment. During the second world war the family led a nomadic existence in Scotland and Wales while his father was posted to different parts of the British Isles. In 1942 Robin went to Selwyn House, a boarding Preparatory School which is normally based in Broadstairs, Kent but was evacuated to Wales until the end of the war in 1945.

In 1947 Robin won a scholarship to Eton College, a public school whose fees were a long way beyond the family's means; fortunately scholars only paid what they could afford. While there he learned how to stay awake all night solving mathematics problems. (Scholars who specialised in maths were expected to score 100% on the weekly set of problems, which were tough.)

In 1952 he won a major scholarship to King's College, Cambridge, sitting the exam in the Examinations Hall which is 100 yards from his present office. However, before going to Cambridge he did two years' national military service in the Royal Engineers, gaining a commission as a second lieutenant (which relieved his father, who rightly suspected that Robin might not be cut out to be an army officer).

By the time he went to Cambridge in 1954 Robin had forgotten a lot of mathematics; but nevertheless he gained a first-class degree after two years (by omitting Part I of the Tripos). In 1956 he took a short computing course on the EDSAC; he then deserted numerate study for a while to take Part II of the moral sciences Tripos ("moral sciences" was then Cambridge's name for philosophy).

For some reason, academic life did not appeal and he went to London in 1958, wondering whether to take up music seriously, having done a lot of singing and oboe playing, some cello and some composition while at Cambridge. He decided otherwise, and instead taught mathematics for a year at Marylebone Grammar School.

In 1960 Robin took a programming job at Ferranti, in London. He looked after the program library of a small decimal computer called Sirius, and even helped to sell some of the twenty-odd sold altogether. Then in 1963 he moved to a lectureship in mathematics and computer science at The City University. In the same year he married Lucy. During the next five years while they lived in London their children Gabriel, Barney and Chloë were born.

It was at City that he became interested in artificial intelligence, the semantics of programs and mathematical logic. His interest in the theory of computing was further inspired by Christopher Strachey, Rod Burstall, Peter Landin, David Park, Michael Paterson and Dana Scott.

Moving towards a life in research, he took a position as senior research assistant in the Computer and Logic group of David Cooper at Swansea University in 1968, working on program correctness. He wrote two papers on program schemata (1969, 1970) and one on program simulation (1971). The former was inspired by the work of Michael Paterson; the latter used an algebraic approach, under the influence of Peter Landin. The algebraic orientation continued in later research, providing a valuable means of modelling structure in computing systems and linking up with later interest in process calculi. While at Swansea he learnt of Dana Scott's work with Christopher Strachey on the foundations of programming languages. In particular, in 1969, Dana Scott wrote a celebrated article proposing the use of a hierarchy of continuous partial functions and giving a typed λ-calculus and logic for it; this would prove very influential.

In 1971 Robin moved to Stanford University as a research associate, joining John McCarthy's group at the Artificial Intelligence Project. Robin took up Scott's ideas as the basis of a system for computer-assisted theorem proving, the Stanford LCF system (1972a, b, c)—LCF stands for "Logic of Computable Functions," referring to Scott's logic. He also began his work on concurrency, again in the tradition of Scott and Strachey, formulating a domain-based notion of process to model the behaviour of computing agents.

In 1973, he was appointed to a lectureship at Edinburgh University, and obtained a Personal Chair in 1984. Edinburgh LCF (1979) was a development of the Stanford work, but now with a specially designed programming language, Edinburgh ML, to serve for finding and constructing proofs—ML stands for "Metalanguage." He also worked on the semantic foundations of LCF; see, especially, his (1977). Next, the language ML itself became of independent interest, with its many novel features, such as implicit typing. A new research effort, with Burstall and others, finally led to the development

of Standard ML (1990, 1991).

Perhaps, though, his greatest effort at Edinburgh was devoted to concurrency, starting with the invention of CCS—his Calculus for Communicating Systems (1980, 1989). Its semantics went through an interesting development, beginning with a domain-theoretic approach, but ultimately emphasizing a novel operational approach to the equality of processes, employing the important notion of bisimulation. This in turn led to the development of other calculi, such as the π-calculus for mobile computing and, most recently, to action structures and calculi, intended to provide a framework for comparing process and other calculi, with a view also towards unifying sequential and concurrent computation.

Robin's research forms a very coherent body of work, with one idea or system or calculus, leading to another in what seems to be an entirely natural, if miraculous, progression. His research has had great influence on others, both direct and indirect, and he has received well-deserved recognition. In 1987 he and his collaborators won the British Computer Society Technical Award for the development of Standard ML. In one year, 1988, he became a founder member of Academia Europaea, a Distinguished Fellow of the British Computer Society and a Fellow of the Royal Society. In 1991 he was accorded the ultimate accolade of the Turing Award.

The coherence and strength of Robin's research can in part be attributed to a clear philosophical position: that computer science, like other sciences, has both an experimental and a theoretical aspect. The experimental one resides in computing systems (whether hardware or software, and including applications); these have a rich behaviour, demonstrate interesting and practically important phenomena, and provide the experiments at hand. The theoretical aspect is the provision of mathematical models to ease the construction, and permit the analysis of such systems; the concepts and structures arising in these models originate in the understanding of the systems to which they apply. This view of the subject is exemplified in Robin's work on, say, LCF, ML and CCS: each is rooted in application and all are characterized by an economy of concept that yet permits great elasticity of expression.

Robin has also carried these ideas forward in his social contribution to our subject. In 1986 he was one of the founding members and the first director of the Laboratory for Foundations of Computer Science—a happy outcome of the UK Alvey Programme. Robin's philosophy, expounded in his founding lecture (1987b), is pervasive, whether in the title of the Laboratory or in its research. In 1995, he left Edinburgh to take up a Chair at Cam-

bridge (in fact the first established chair in Computer Science at Cambridge) becoming Head of Department a year later. There he continues energetically promoting the integration of advanced application with applicable theory.

Semantics and Domain Theory

Milner's work at Stanford and at Edinburgh (at least, until around 1978) was within the Scott-Strachey paradigm. Here one considers the semantics of programming languages as being given compositionally by denotation functions that ascribe meanings to programs. These meanings are elements of Scott domains or related kinds of complete partial order. The notation used to write the meanings is one or another λ-calculus. Scott's LCF was based on one such, later called PCF; it is simply typed with two base types, natural numbers and booleans, with arithmetic and boolean primitives, and with recursive definitions at all types. Milner's LCF system employed a more elaborate typed calculus, PPλ, with a signature of a finite set of base types and constants, with sum, product and recursively defined types, and with a simple kind of polymorphism.

His other (perhaps less official) strand of work at Stanford was on concurrency, introducing recursively defined domains of processes—later termed *resumptions* by Plotkin—to provide abstract models of the behaviour of transducers or other types of computing agents (1973, 1975a). An example of such a domain equation is:

$$P \cong V \to (L \times V \times P)$$

These resumptions are used to model deterministic agents that input values from V and output them to locations in L. Nondeterminism (used to account for parallelism) is dealt with by means of oracles, raising the issue of an adequate treatment of non-deterministic relations within the domain-theoretic framework; this inspired later work on powerdomains. Algebraic ideas occur here with the presentation of semantics using "combinators," the means for combining programs. The most important one is that for the parallel composition of programs. Milner had an idea of his agents as automata whose only means of communication was via a collection of lines, though that was not strongly emphasized in his writing. Perhaps this and the algebra helped inspire the later invention of CCS.

A notable point was the discussion at a general level of full abstraction. The idea of full abstraction is that two terms are to have the same denotation if—and only if—they are contextually equivalent in a sense determined by an

operational semantics (for example, one given via an abstract machine). It is precisely the difficulty of providing such a fully abstract domain-theoretic treatment of concurrency that led Milner to his later operational treatment. The question also arises as to the full abstraction of Scott's standard model for PCF. After it was shown that this semantics is not fully abstract, Milner gave a fully abstract domain-theoretic model by means of an ingenious syntactic, or "term model," construction (1977).

The subject was developed much further by many people over the following years, searching for the proper notion of model; they were generally looking for an extensional notion of sequentiality at higher types. Notable further contributions included the introduction of stable functions (stability is an approximation to the desired notion of sequentiality) and of game-theoretic models (games provide an intensional characterization of sequentiality). It was recently shown that the operational equivalence relation is undecidable even for PCF restricted so that the only base type is the booleans. It follows that there can be no "finitary" extensional mathematical account of sequentiality, thereby providing a fundamental reason why previous attempts failed.

Beyond PCF, full abstraction studies have been undertaken for a wide variety of languages. However it is fair to say that for the case of concurrency, the one of original interest to Milner, there is as yet no satisfactory widely applicable treatment within the domain-theoretic paradigm.

Computer Assisted Theorem Proving

Milner's LCF system enables one to carry out proofs on a machine, where the structure of the proof is largely known in advance. This was a considerable, if not entirely novel, departure from the contemporary emphasis on theorem proving systems in Artificial Intelligence. Systems such as De Bruijn's Automath and Hewitt's PLANNER were forerunners; the former enabled one to write down and check large proofs and the latter permitted the design of proof strategies.

The need for proof-checking systems resides in their intended application. The proofs needed to show computer systems correct are, in a way, tedious. While their structure is generally fairly clear (say a large induction), they can be very large, not least since computer systems are. One is therefore liable to make mistakes in such proofs when working "by hand," and machine help is much to be desired to provide the security of an assurance that no such mistakes have occurred.

With LCF, Milner, and his colleagues, firmly established the field of large-scale computer-assisted theorem proving. In doing so, they made several important contributions: tactics (subgoaling strategies for finding proofs) and tacticals (for their combination); a specially designed typed programming language for writing tactics and tacticals; and a means to achieve security against faulty deduction.

The programming language was ML, and was designed by Milner and his colleagues at Edinburgh. The features of ML are well-adapted to the application of computer-assisted theorem-proving. Its higher-order and exception features are used to write tactics and tacticals. Its type discipline provides the desired security, achieved through an abstract recursively defined data type of proofs whose elements can only be constructed by means of the inference steps of the underlying logic of LCF. The inclusion of assignment, pattern matching, polymorphism and call-by-value were motivated by related practical concerns.

The article by Gordon in this volume provides much further detail on the development of Stanford and Edinburgh LCF and later projects. In particular, projects such as LCF require the efforts of many people: Gordon gives a good account of their contribution. Milner's own work on computer-assisted theorem proving effectively came to an end in 1978, but the subject had been established. It was pursued further at Cambridge and INRIA with the development of Cambridge LCF, HOL and Isabelle, as well as the further development of ML. Other systems arose: some incorporate various constructive type theories, notable here are Coq and LEGO (both based on the Calculus of Constructions) and NuPrl and ALF (based on Martin-Löf's type theory); others such as Mizar, PVS or (for example) Larch arose from different research traditions.

What started out as a mechanization of one particular logic has evolved into a field of wider scope. Experience shows that one naturally wishes to conduct proofs in a variety of logics, with the particular one being chosen by the user of a system rather than its designer. A subject of logical frameworks has arisen to accommodate such variety, and systems such as Isabelle and ELF are designed for just this task. Finally, the size of the proofs that can be performed has risen substantially. It is now possible to tackle large systems of commercial interest.

Standard ML

As explained above, the design of ML was strongly influenced by the needs of theorem proving. But it was equally shaped by Milner's approach

to language design in general. Two objectives stand out in this approach: first, the language should solve the practical problems that motivated its existence; second, it should do so with an absolute minimum of concepts, all of which should be rigorously defined and analysed. It should also be remarked that ML falls within the wider tradition of functional programming languages. Several such languages were influential in the design of ML: Burstall and Popplestones' POP-2, Evan's PAL, Landin's ISWIM, McCarthy's LISP and Reynold's GEDANKEN.

The detailed story of the evolution of ML is complex; an account of the period from 1974 to 1996 can be found in (1997). Here we rather discuss some highlights of that evolution, with emphasis on Milner's contribution. The first, practical, design objective was addressed through implementation work and experiments. Milner was involved with several people working on early implementations of ML: Malcolm Newey, Lockwood Morris, Michael Gordon, Christopher Wadsworth, Luca Cardelli, Alan Mycroft, Kevin Mitchell and John Scott.

The technical vehicle that guided the design of ML was language semantics. This provided a framework that permitted economy of design, the second objective. In particular Milner's paper on ML's polymorphic type discipline (1978a) was seminal in several ways. Besides presenting the ML type discipline (described below), it provided a strong indication that formal semantics could play a central rôle in the design of non-trivial programming languages.

The ML type discipline possesses several properties which distinguished it from contemporary languages. First, it is provably sound: if an expression of the type system has a type, then its value (the result of its evaluation) has the same type; or, as Milner put it: "well-typed expressions do not go wrong." Second, some expressions are allowed to have more than one type, i.e., to be "polymorphic." For example, the polymorphic list reverse function can reverse all lists, irrespective of the type of the elements of the list. Third, types can be inferred automatically from programs by an algorithm, called W. As shown in the later paper with Damas (1982b), W always terminates, either by failing (when the expression cannot be typed) or with a most general, also called *principal*, type of the expression.

Although discovered independently, Milner's type discipline has much in common with Curry, Hindley and others' earlier work on principal type schemes in Combinatory Logic. In particular, both he and Hindley use Robinson's unification algorithm for type checking. The main difference between the two is that Milner's type discipline allows type schemes with

quantification in the typing of declarations. For example, consider the Standard ML program declaring a function length:

```
fun length [] = 0
  | length (_::xs) = 1 + length xs
val n = length [1,2,3] + length [true, false];
```

The function `length` is given the type scheme

$$\texttt{length} : \forall\alpha.\alpha\,\texttt{list} \rightarrow \texttt{int}$$

which can be instantiated to both $\texttt{int}\,\texttt{list} \rightarrow \texttt{int}$ and $\texttt{bool}\,\texttt{list} \rightarrow \texttt{int}$.

By 1985, several dialects and implementations of ML existed. Further, new ideas had emerged both within the immediate ML community and also from Burstall and his group who had developed influential languages: CLEAR, a specification language designed with Goguen, and HOPE, a functional programming language. It was evident that standardization was necessary. A number of design meetings were held, leading to a technical report consisting of Milner's description of the Core Language, MacQueen's module system and Harper's I/O primitives. Around the same time, operational semantics was gaining momentum, through work by Plotkin, Milner and Kahn. Milner's description of the dynamic semantics of the Core was essentially a natural language formulation of an operational semantics.

In 1985, Milner, Tofte and Harper began working on an operational semantics of full Standard ML, including static and dynamic semantics of both Core and Modules. Every construct of the language was described by a very small number of rules, typically one defining the static semantics (elaboration) and one defining the dynamic semantics (evaluation) of the construct. For example, the elaboration rule for recursive value bindings is

$$\frac{C + VE \vdash \mathit{valbind} \Rightarrow VE}{C \vdash \texttt{rec}\ \ \mathit{valbind} \Rightarrow VE}$$

where C is a context mapping free identifiers of *valbind* to their static meaning, and VE is a static value environment mapping the identifiers defined by *valbind* to their types. The evaluation rule for recursive value bindings is

$$\frac{E \vdash \mathit{valbind} \Rightarrow VE}{E \vdash \texttt{rec}\ \ \mathit{valbind} \Rightarrow \mathrm{Rec}\,VE}$$

where VE is a dynamic value environment (a finite map from variables to values) and $\mathrm{Rec}\,VE$ is a finite value environment which represents one unfolding of the recursively defined values.

More difficult was the handling of Modules, due to novel concepts such as structure sharing and functors. New theory had to be invented for dealing with these constructs (1987c). A key idea was that type checking of structure sharing could be done using a nonstandard form of unification, related to unification in record calculi. The work eventually led to the Definition of Standard ML (1990) and the Commentary on Standard ML (1991). Some years later, in 1996, when a clearer understanding of the semantics and pragmatics of the language had emerged, Milner, Tofte, Harper and MacQueen revised and simplified the language, leading to a new Definition (1997). Structure sharing was abandoned and type abbreviations in signatures were added. The resulting static semantics of Modules is a mere eight pages long and fulfils a longstanding desire to obtain a simple operational semantics for ML Modules.

On the theoretical side, ML's type discipline gave rise to a considerable body of theoretical work. Particularly interesting was the result that deciding ML typability is complete for deterministic exponential time, contrasting with the observation that ML type inference works well in practice. Another important result is that type checking polymorphic recursion is equivalent to the (undecidable) semi-unification problem. There has also been work extending the ML type discipline with higher ranks of polymorphism, subtyping, object types and overloading—much more than can be described here. The ML type system appears to be a local optimum in the design space. Subtyping is a case in point: while there has been some success in extending the notion of principal type to a notion of principal solution of constraint sets, it seems to be very hard to find a notion of "most general type" which coincides with "most readable type."

Standard ML has developed in other ways, for example through work on implementations. Implementation technology was developed in the context of Standard ML of New Jersey, Edinburgh ML and Poly/ML. Other Standard ML implementations have emerged: Moscow ML, The ML Kit and, most recently, MLWorks, a commercial implementation developed by Harlequin. Good textbooks on programming with Standard ML have been written and the language has become quite widely used for teaching and research.

Variants of ML have emerged, notably Caml Light and Objective Caml, both developed at INRIA. Objective Caml extends ML with facilities for object-oriented programming. Several researchers, including Berry, Milner and Turner (1992b), have explored the combination of ML and concurrency, leading to Concurrent ML, FACILE, Distributed Poly/ML and LCS. Finally,

there is a new design effort underway, known as the ML2000 project; this involves researchers from France and several sites in the United States.

Concurrency

As outlined above, Milner's initial work (1973, 1975a) on concurrency was carried out within the Scott-Strachey approach to semantics. The intention was to extend the scope of denotational semantics beyond sequential input/output programs. Difficulties arise from non-terminating programs with side-effects, non-deterministic programs and parallel programs. Milner showed that these computational features can be modelled operationally using (deterministic) transducers. The notion of name or location was important here. It was given as an address at which communication takes place (which later turned into the notion of a port): in a given state with a given input value the output function of a transducer determines both the output communication line and the value to be transmitted on that line.

Transducers are intensional objects whose extensional behaviour as processes he wished to capture mathematically. Such processes were modelled by means of the domain of resumptions given by the recursive domain equation presented above. The domain was intended to play the same rôle for non-sequential programs as the domain of continuous functions for sequential programs. This was before the invention of powerdomains, and so for nondeterminism oracles were employed. A semantics for a programming language with these non-sequential features was presented in (1973). Notable here was the global recursive definition of a combinator for the parallel composition of processes, a definition made possible by the use of the domain of resumptions. The analysis of assignment as a complex action involving communication with registers was also important.

In (1975a) he discussed this model further, including a general discussion of criteria for denotational semantics. Compositionality can be achieved by regarding syntax as a word algebra and semantics as the (unique) homomorphism to an algebra of operators on processes (or other suitable mathematical entities). The semantics should be justified by its relationship to an operational semantics for the language; in particular it should be *adequate* or, and much better, *fully abstract*, as described above.

The work of the later 1970s has a stronger algebraic flavour. Flowgraphs were introduced in the two papers (1979b, d), both written in 1977, the second jointly with Milne. Communication plays a central rôle, and is to be understood as exchange of values at explicitly named ports. A flowgraph

is a (pictorial) representation of the potential flow of information through ports in a system. Just as with Scott's flow diagrams, flowgraphs provide an intermediary between a program and its meaning. Combinators for combining flowgraphs were introduced; these became the static operators of CCS: binary parallel composition, renaming and restriction. Various laws of flow were presented (such as the commutativity and associativity of parallel composition), and in (1979b) Milner showed that flowgraphs form the free such algebra, thereby justifying the laws. Flowgraphs can be viewed as an expression of *heterarchy*, where one and the same system can be viewed as built up from subsystems in distinct ways. The contrast is with a *hierarchical* view, where systems can be uniquely analysed into subsystems and so have a tree-like form rather than a graphical one.

Meanings of concurrent programs, the processes, are elements of a powerdomain of resumptions: this is where the dynamics of a system are described. The domain of processes is also a flow algebra. The domain equation for processes is

$$P_L \;\cong\; \mathcal{P}(\sum_{\mu\in L}(U_\mu \times (V_\mu \to P_L)))$$

where L is the set of ports, U_μ and V_μ are, respectively, domains of input and output values at port μ, and $\mathcal{P}$ is a powerdomain operator (for nondeterminism). Milner had in mind the Smyth powerdomain with an added empty set (for termination), although he was unhappy with this account as it identifies too many processes.

Further developments culminated in CCS. In (1978b) Milner recounted the definition of a flow algebra, and introduced, as one particular instance, synchronization trees (without the presence of value-passing). The dynamic operators of CCS, prefixing and summation, were then introduced as combinators on these trees. The prefixing operator provides a facility for value-passing *actions*, whether for input or output. The silent τ action also appeared as the result of synchronization. Later that year in (1978c), written while visiting Aarhus, these dynamic operators were explicitly included alongside the static operators as part of the definition of an algebra for modelling communicating systems. The notion of a single observer was used to justify interleaving (instead of a partial order approach as in the case of Petri nets). This was exemplified in the equational law relating parallel composition with nondeterminism, which later became the expansion theorem. With these two papers the general conception of CCS was in place: concurrent systems can be modelled by behaviour expressions, and equational reasoning provides a mechanism for showing properties of sys-

tems. By way of an example, Milner showed the possibility of deadlock for a description of the dining philosophers. However, one ingredient was still missing: a justification for the equational laws.

The next crucial step was the paper (1980a) written with Hennessy. This paper isolated basic CCS where there is no value-passing, only synchronization. Basic CCS bears much the same relation to full CCS as propositional logic does to predicate logic. Observational equivalence of synchronization trees was introduced both in a strong form, and in a weak form where τ actions are abstracted away. At this stage equivalences were defined iteratively instead of using greatest fixed points; they arose from the simple idea that an observer can repeatedly interact with an agent by choosing an available transition from it. The equivalence of processes was then defined in terms of the ability of these observers to continually match each other's choices of transition by one labelled with the same action. Hennessy-Milner logic (with strong and weak modalities) was also introduced in order to provide a logical account of the equivalences. Equational axiomatizations of the associated congruences were presented for finitary terms (here, those built from prefixing, binary summation and nil). As a result there was an intended model for process expressions, given by synchronization trees quotiented by observational congruence.

One additional ingredient needed to define observational equivalence directly on process expressions was their structural operational semantics using transition systems labelled by actions. The combination of structure and labelling has since proved a very adaptable specification method and is now standard. These ideas were presented in the influential CCS book (1980) written in Aarhus, and presented as lectures. The book inspired the whole field of process calculus in the same way that Milner's paper (1978a) inspired that of (polymorphic) type theoretic work in programming languages. The two paradigms reflect his principled approach to theoretical computer science, with its concern for applicability and the use of a small number of primitives. As Milner says (1979a) about his approach to concurrency

> The broad aim of the approach is to be able to write and manipulate expressions which not only denote ... the behaviour of composite communicating systems, but may reveal in their form the physical structure of such systems.

The accessibility of the material is also most important. Parts of the book (1980) can be taught as an undergraduate course. Indeed Milner's later

more polished book (1989) is a distillation of the ideas arising from teaching CCS to final year undergraduates at Edinburgh. Both books give interesting accounts of the evolution of CCS, as does (1990b).

Further developments of the theory of CCS occurred through the 1980s. In 1981 Park gave a notion of bisimulation leading to a somewhat different, but more satisfactory notion of observational equivalence (for both the strong and weak forms); it also has an interesting and useful characterization as a greatest fixed point. Milner used bisimulations in the paper (1983a) which, further, introduced SCCS, a synchronous version of CCS (and, in a way, more basic). Mathematically SCCS is very elegant, and its model turns out to provide a canonical model for non-well founded set theory. Variations on the notion of observational equivalence were considered. For example, in (1981) Milner defined an observational preorder for processes which is sensitive to divergence. Again, an alternative framework for defining equivalences using testing was introduced: the resulting equivalences are very closely related to the failures model for Hoare's CSP.

Milner extended the finitary axiomatizations of strong and weak bisimulation to finite terms which permit guarded recursion (1984a, 1989a). The axiomatization in the first of these papers was based on Salomaa's axiomatization of language equivalence for regular expressions, except for the axiom $a(X + Y) = aX + aY$. However the theory is subtly different. Indeed automata theory from the perspective of bismulation equivalence, as opposed to language equivalence, contains surprises. One example is that bisimulation equivalence is decidable for context-free grammars. In another direction, some recent work has concentrated on value-passing, providing complete equational theories for regular value-passing process expressions, for both testing and bisimulation congruences. Again, Hennessy-Milner logic is not very expressive, and is unable to capture liveness and safety properties of processes. Various extensions have been proposed, such as modal μ-calculus, for describing temporal properties of processes: these extensions have the feature that two bisimulation equivalent systems have the same temporal properties.

The theory of CCS has inspired tools such as AUTO/GRAPH and the Concurrency Workbench for analysing concurrent systems. These tools allow automatic checking of equivalences and preorders between finite-state processes. The Concurrency Workbench, written in ML and developed jointly in Edinburgh and Sussex, also permits model checking temporal properties of processes. Notions of simulation and bisimulation have also found their way into model-checking tools.

Since the mid-80s various extensions to process calculi have been presented, for example for modelling time, probability, location and priority. The CCS paradigm has motivated various results about these extensions, including definitions of equivalence, characteristic modal logics and temporal extensions, and connections with automata theory.

Operational semantics are paramount in the theory of CCS and related calculi; indeed Milner has never returned to a denotational theory of processes. However, a denotational account of strong bisimulation is possible. Semantics fully abstract with respect to strong bisimulation have been given within a variety of mathematical frameworks: non-well founded sets, domains and metric spaces. In all cases appropriate "domain equations" are employed, giving a suitable notion of resumption. It should be emphasized that, to date, no corresponding treatment of weak bisimulation is available.

Milner was dissatisfied with CCS for two reasons. The first originated in a particular practical problem presented in the CCS book (1980). There the meanings of parallel imperative programs were given by translation into value-passing CCS. Program variables were modelled using explicit registers which appeared in parallel with the translation of the programs themselves. However when the programming language permits recursive procedures, the modelling suffered because concurrent calls of the same procedure were interleaved. Milner remarked that a more natural modelling would include a return link for each call of a procedure, and this would require passing ports as values, which is impossible in CCS. The second concern resulted from the notable success of process calculi, as inspired by Milner's work. Numerous process calculi have flourished over the years, and many different equivalences have been defined. But there are too many apparently different calculi, with none accepted as canonical, and too many equivalences. Recent work on rule formats for defining process operators has offered some insights into the dynamics of some classes of calculi, but that can only be part of the story.

Both these concerns have underpinned Milner's later work in concurrency. An important development was a tractable calculus for passing ports, thereby allowing dynamic communication topologies to be modelled. Early discussions with Nielsen in 1981, while in Aarhus, had failed to produce such a calculus. Then, in 1986, Engberg and Nielsen made an important breakthrough, finding such a calculus. Following this, Milner jointly with Parrow and Walker, beginning in 1987, produced a simpler approach, the π-calculus (1992d, e; see also the paper by Engberg and Nielsen in this volume). This calculus contains fewer primitives than value-passing CCS, be-

ing based only on names. There is also a subtle mechanism for dynamically changing the scope of static name bindings. A fundamental point is that the λ-calculus can be encoded within π-calculus.

The combination of names and binding causes difficulties when giving an operational semantics for the calculus. Transition system models are not entirely natural as actions are unexpectedly complex: both bound and free output actions are required. This induces corresponding complexity in the definition of both strong and weak bisimulation and their characteristic Hennessy-Milner logics (1992d, e, 1993f). The calculus does, however, highlight an interesting difference between early and late bismulation, which also applies to value-passing CCS (1993f).

In order to resolve these problems, Milner introduced a change in the style of semantics. Instead of transition systems, he used reductions, based on Berry and Boudol's elegant Chemical Abstract Machine approach. With Sangiorgi, Milner could then define bisimulation congruence using reductions and barbs (1993e). The calculus presented there was also more refined than the original one, incorporating sorts (analogous to types) and the communication of tuples. An interesting question for reduction-based approaches is how to define temporal logics for π-calculus agents.

The π-calculus has had a strong impact. In part this is because process calculi, λ-calculi and concurrent object-oriented programs can all be modelled within it. This yields a relationship with functional programming and a fundamental model of "mobile" objects providing a framework for understanding contemporary programming disciplines. Other developments include higher-order process calculi, and experimental programming languages such as Pict and the join calculus.

Milner's most recent work in concurrency is on action structures and calculi, and is intended to address the second concern. The aim is to find mathematical structures which can underlie concrete models of computation and which are free from ad hoc details. Again the motivations reflect basic concerns (1994a), for we

> lack a canonical structure which is combinational i.e. which explains how processes are synthesized, and which embodies the dynamics of interaction among processes.

Action structures are categories with extra structure, including certain "controls"; actions are modelled as morphisms and the controls allow complex actions to be built from simple ones. They also possess an ordering on the actions, used to specify (reduction) dynamics. The categorical structure has

been shown to link up with (categorical models of) Girard's linear logic, a topic of independent computational interest. The controls allow an analysis of the structural aspects of such process calculi as the π-calculus; however it is still not clear how to give a uniform analysis of such aspects of their dynamics as observational equivalence. These issues remain an active concern of Milner's.

Doctoral Students

Before 1980 R. Aubin (1976); G. Milne (1978).

From 1980 to 1989 A. Cohn (1980); A. Mycroft, M. Sanderson (1982); L. Damas, B. Monahan (1985); K. Larsen, K. Mitchell (1986); K. V. S. Prasad, M. Tofte (1988); F. Moller (1989).

After 1990 D. Berry, C. Tofts (1990); D. Sangiorgi (1993); P. E. Sewell (1995); D. N. Turner (1996); A. Mifsud (1996);

Current J. Leifer, M. Sawle, A. Unyapoth

Publications

Books

1976 (Edited with S. Michaelson) *Proc. 3rd. Int. Coll. on Automata, Languages and Programming*, Edinburgh, Edinburgh University Press.

1979 (With M. J. Gordon & C. P. Wadsworth) *Edinburgh LCF; a Mechanized Logic of Computation*, Lecture Notes in Computer Science, Vol. 78, Berlin, Springer-Verlag.

1980 *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol. 92, Berlin, Springer-Verlag.

1989 *Communication and Concurrency*, New York, Prentice-Hall.

1990 (With M. Tofte & R. Harper) *The Definition of Standard ML*, Cambridge, MIT Press.

1991 (With M. Tofte) *Commentary on Standard ML*, Cambridge, MIT Press.

1996 (Edited with I. Wand) *Computing Tomorrow*, Cambridge, Cambridge University Press.

1997 (With M. Tofte & R. Harper & D. MacQueen) *The Definition of Standard ML (Revised)*, Cambridge, MIT Press.

1998 *Communicating and Mobile Systems: the Pi Calculus*, Cambridge, Cambridge University Press.

Articles

1968 *String handling in ALGOL*, British Computer Journal, Vol. 10, pp. 321–324.

1969 *Program schemes and recursive function theory.* In Machine Intelligence 5 (eds. B. Meltzer & D. Michie), pp. 39–58, Edinburgh, Edinburgh University Press.

1970 *Equivalences on program schemes*, Journal of Computer and Systems Sciences, Vol. 4, No. 2, pp. 205–219.

1971 *An algebraic notion of simulation between programs.* In Proc. 2nd Int. Joint Conf. on Artificial Intelligence, London, pp. 481–49, London, British Computer Society.

1972a *Implementation and applications of Scott's logic for computable functions.* In Proc. ACM Conf. on Proving Assertions about Programs, New Mexico State University, pp. 1–6, New York, ACM.

b (With R. W. Weyhrauch) *Program semantics and correctness in a mechanized logic.* In Proc. USA–Japan Computer Conf., Tokyo, pp. 384–392.

c (With R. W. Weyhrauch) *Proving compiler correctness in a mechanized logic.* In Machine Intelligence 7 (eds. B. Meltzer & D. Michie), pp. 51–70, Edinburgh, Edinburgh University Press.

1973 *An approach to the semantics of parallel programs.* In Proc. Convegno di Informatica Teoretica, pp. 285–301. Pisa, Instituto di Elaborazione della Informazione.

1974 *A calculus for the mathematical theory of computation*, Int. Symp. on Theoretical Programming (eds. A. Ershov and V. A. Nepomniaschy), Novosibirsk, USSR, August 1972, Lecture Notes in Computer Science, Vol. 5, pp. 332–343, Berlin, Springer-Verlag.

1975a *Processes: a mathematical model of computing agents*, In Proc. Logic Colloquium (eds. H. E. Rose & J. C. Shepherdson), Bristol, July 1973, Studies in Logic and the Foundations of Mathematics, Vol. 80, pp. 157–174, Amsterdam, North-Holland.

b (With L. Morris & M. Newey) *A logic for computable functions with reflexive and polymorphic types*, Conf. on Proving and Improving Programs, Arc-et-Senans, July 1975, pp. 371–394, Colloques IRIA, Rocquencourt, IRIA-Laboria.

1976a *LCF: A methodology for performing rigorous proofs about programs.* In Proc. 1st. IBM Symp. on Mathematical Foundations of Computer Science, Amagi, Japan.

b *Models of LCF.* In Foundations of Computer Science II, Part 2 (eds. K. Apt & J. W. de Bakker), Mathematical Centre Tracts 82, pp. 49–63, Amsterdam, Mathematisch Centrum.

c *Program semantics and mechanized proof.* In Foundations of Computer Science II, Part 2 (eds. K. Apt & J. W. de Bakker), Mathematical Centre Tracts 82, pp. 3–44, Amsterdam, Mathematisch Centrum.

1977 *Fully abstract models of typed λ-calculi*, Theoretical Computer Science, Vol. 4, pp. 1–22.

1978a *A theory of type polymorphism in programming*, Journal of Computer and Systems Sciences, Vol. 17, No. 3, pp. 348–375.

b *Algebras for communicating systems.* In Proc. AFCET/S.M.F. joint colloquium in Applied Mathematics, Paris.

c *Synthesis of communicating behaviour.* In Proc. 7th. Int. Symp. on Foundations of Computer Science (ed. J. Winkowski), Zakopane, Lecture Notes in Computer Science, Vol. 64, pp. 71–83, Berlin, Springer-Verlag.

d (With M. Gordon, L. Morris, M. Newey & C. Wadsworth) *A metalanguage for interactive proof in LCF.* In Proc. 5th. Annual ACM Symp. on Principles of Programming Languages, New York, ACM.

1979a *An algebraic theory for synchronization.* In Proc. 4th. G.I. Conf. on Theoretical Computer Science (ed. K. Weihrauch), Aachen, Lecture Notes in Computer Science, Vol. 67, pp. 27–35, Berlin, Springer-Verlag.

b *Flowgraphs and Flow Algebras*, Journal of the ACM, Vol. 26, No. 4, pp. 794–818.

c *LCF: a way of doing proofs with a machine.* In Proc. 8th. Int. Symp. on Foundations of Computer Science (ed. J. Bečvář), Olomouc, Lecture Notes in Computer Science, Vol. 74, pp. 146–159, Berlin, Springer-Verlag.

d (With G. Milne) *Concurrent processes and their syntax*, Journal of the ACM, Vol. 26, No. 2, pp. 302–321.

1980a (With M. Hennessy) *On observing nondeterminism and concurrency.* In Proc. 7th. Coll. on Automata Languages and Programming (eds. J. de Bakker & J. van Leeuwen), Lecture Notes in Computer Science, Vol. 85, pp. 299–309, Berlin, Springer-Verlag.

1981 *A modal characterisation of observable machine-behaviour.* In Proc. 6th. Colloquium on Trees in Algebra and Programming (eds. E. Astesiano & C. Böhm), Genoa, Lecture Notes in Computer Science, Vol. 112, pp. 25–34, Berlin, Springer-Verlag.

1982a *Four combinators for concurrency.* In Proc. 9th. ACM Symp. on Principles of Distributed Computing, Ottawa, pp. 104–110, New York, ACM.

b (With L. Damas) *Principal type schemes for functional programs.* In Proc. 9th. Annual ACM Symp. on Principles of Programming Languages, Albuquerque, pp. 207–212, New York, ACM.

1983a *Calculi for synchrony and asynchrony*, Journal of Theoretical Computer Science, Vol. 25, pp. 267–310.

b *How ML Evolved*, Polymorphism—The ML/LCF/Hope Newsletter, Vol. 1, No. 1.

1984a *A complete inference system for a class of regular behaviours*, Journal of Computer and Systems Sciences, Vol. 28, No. 2, pp. 439–466.

b *The use of machines to assist in rigorous proof*, Phil. Trans. Roy. Soc. London, Ser. A, Vol. 312, pp. 411–422.

c *Using Algebra for Concurrency.* In Chapter 21, Distributed Computing (eds. F. B. Chambers, D. A. Duce & G. P. Jones), pp. 291–305, London, Academic Press.

1985 (With M. Hennessy) *Algebraic laws for nondeterminism and concurrency*, Journal of the ACM, Vol. 32, No. 1, pp. 137–161.

1986a *Lectures on a calculus for communicating systems.* In Control Flow and Data Flow: Concepts of Distributed Programming (ed. M. Broy), Proc. Int. Summer School at Marktoberdorf, pp. 205–228, Springer Study Edition, Berlin, Springer-Verlag.

b *Process constructors and interpretations.* In Proc. 10th. IFIP World Computer Congress (ed. H.–J. Kugler), Dublin, pp. 507–514, Amsterdam, North-Holland.

1987a *Dialogue with a proof system.* In Proc. TAPSOFT '87, Vol. 1 (eds. H. Ehrig, R. Kowalski, G. Levi & U. Montanari), Pisa, Lecture Notes in Computer Science, Vol. 249, pp. 271–275, Berlin, Springer-Verlag.

b *Is computing an experimental science?*, Journal of Information Technology, Vol. 2, No. 2, pp. 60–66.

c (With R. Harper & M. Tofte) *A type discipline for program modules.* In Proc. TAPSOFT '87, Vol. 2 (eds. H. Ehrig, R. Kowalski, G. Levi & U. Montanari), Pisa, Lecture Notes in Computer Science, Vol. 250, pp. 308–319, Berlin, Springer-Verlag.

d (With K. G. Larsen) *Verifying a protocol using relativized bisimulation.* In Proc. 14th. ICALP (ed. Th. Ottman), Lecture Notes in Computer Science, Vol. 267, pp. 126–135, Berlin, Springer-Verlag.

1988a *Interpreting one concurrent calculus in another.* In Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Vol. 2, pp. 321–326.

b *Some directions in concurrency theory*, Statement for panel on "Theory and Practice in Concurrency." In Proc. Int. Conf. on Fifth Generation Computer Systems (edited by the Institute for New Generation Computer Technology), Tokyo, Vol. 1, pp. 163–164.

1989a *A complete axiomatisation for observational congruence of finite-state behaviours*, Journal of Information and Computation, Vol. 81, No. 2, pp. 227–247.

1990a *Functions as processes.* In Proc. 17th. Int. Conf. on Automata, Languages and Programming (ed. M. S. Paterson), University of Warwick, Lecture Notes in Computer Science, Vol. 443, pp. 167–180, Berlin, Springer-Verlag.

b *Operational and algebraic semantics of concurrent processes.* In Handbook of Theoretical Computer Science (ed. J. van Leeuwen), Vol. B: Formal Models and Semantics, Chapter 19, pp. 1201–1242, Amsterdam, Elsevier.

1991a (With M. Tofte) *Co-induction in relational semantics*, Theoretical Computer Science, Vol. 87, No. 1, pp. 209–220.

1992a *Functions as processes*, Mathematical Structures in Computer Science, Vol. 2, No. 2, pp. 119–141.

b (With D. Berry & D. N. Turner) *A semantics for Standard ML concurrency primitives.* In Proc. 17th. Annual ACM Symposium on Principles of Programming Languages, San Francisco, pp. 119–129, New York, ACM.

c (With K. G. Larsen) *A compositional protocol verification using relativized bisimulation*, Information and Computation, Vol. 99, No. 1, pp. 80–108.

d (With J. Parrow & D. Walker) *A calculus of mobile processes, I*, Information and Computation, Vol. 100, No. 1, pp. 1–40.

e (With J. Parrow & D. Walker) *A calculus of mobile processes, II*, Information and Computation, Vol. 100, No. 1, pp. 41–77.

f (With D. Sangiorgi) *Barbed bisimulation.* In Proc. 19th. Int. Conf. on Automata, Languages and Programming (ed. W. Kuich), Wien,

Lecture Notes in Computer Science, Vol. 623, pp. 685–695, Berlin, Springer-Verlag.

g (With D. Sangiorgi) *The problem of "weak bisimulation up to."* In Proc. CONCUR'92: Third International Conference on Concurrency Theory (ed. W. R. Cleveland), Stony Brook, Lecture Notes in Computer Science, Vol. 630, pp. 32–46, Berlin, Springer-Verlag.

1993a *Action calculi, or syntactic action structures.* In Proc. 18th. MFCS Conf. (eds. A. M. Borzyszkowski & S. Sokolowski), Gdańsk, Lecture Notes in Computer Science, Vol. 711, pp. 105–121, Berlin, Springer-Verlag.

b *An action structure for synchronous π-calculus.* In Proc. 9th. FCT Conf. (ed. Z. Ésic), Szeged, Lecture Notes in Computer Science, Vol. 710, pp. 87–105, Berlin, Springer-Verlag.

c *Elements of interaction*, Communications of the ACM, Vol. 36, No. 1, pp. 78–89.

d *Higher-order action calculi.* In Proc. 9th. CSL Conf. (eds. E. Börger, Y. Gurevich & K. Meinke), Swansea, Lecture Notes in Computer Science, Vol. 832, pp. 238–260, Berlin, Springer-Verlag.

e *The polyadic π-calculus: a tutorial.* In Logic and Algebra of Specification (eds. F. L. Bauer, W. Brauer & H. Schwichtenberg), pp. 203–246, Berlin, Springer-Verlag.

f (With J. Parrow & D. Walker) *Modal logics for mobile processes*, Theoretical Computer Science, Vol. 114, pp. 149–171.

1994a *Action structures and the π-calculus.* In Proof and Computation (ed. H. Schwichtenberg), pp. 317–378, Series F: Computer and Systems Sciences, Vol. 139, NATO Advanced Study Institute, Proc. Int. Summer School held in Marktoberdorf, Germany, 1993, Berlin, Springer-Verlag.

b *Computing is interaction* (abstract). In Proc. 13th. IFIP World Computer Congress (eds. B. Pehrson & I. Simon), Hamburg, Vol. 1, pp. 232–233, Amsterdam, North-Holland.

1995 (With A. Mifsud & J. Power) *Control structures.* In Proc. Tenth Symposium on Logic in Computer Science, San Diego, pp. 188–198, Washington, IEEE Computer Press.

1996a *Calculi for interaction.* In Acta Informatica, Vol. 33, No. 8, pp. 707–737.

b *Semantic ideas in computing.* In Computing Tomorrow (eds. I. Wand & R. Milner), Cambridge, pp. 246–283, Cambridge University Press.

1997a *Strong normalisation in higher-order action calculi.* In Proc. 3rd. TACS Symp. (eds. M. Abadi and T. Ito), Lecture Notes in Computer Science, Vol. 1281, pp. 1–19, Berlin, Springer Verlag.