Questions to Robin Milner : a responders commentary

Eindhoven University of Technology

Department of Mathematics and Computing Science
Computing Science Section

Computing Science Notes

Questions to Robin Milner
A responders commentary

by

W.P. de Roever

86.11

Questions to Robin Milner
A responders commentary

by

W.P. de Roever

86.11

INFORMATION PROCESSING 86, H.-J. Kugler (ed.)
Elsevier Science Publishers B.V. (North-Holland)
© IFIP, 1986

QUESTIONS TO ROBIN MILNER – A RESPONDER'S COMMENTARY

Willem-P. DE ROEVER
Eindhoven University of Technology
P.B. 513, 5600 MB Eindhoven, The Netherlands

1. Point of Departure

As a responder to Robin Milner's "Process constructors and interpretations" I find myself in a somewhat awkward position. We all love Robin for his essential contributions to concurrency in particular and computer science in general. Personally I regard him as the main researcher responsible for acceptance of the field of semantics, proof theory, and verification of concurrent programs as a scientific discipline in its own right based on techniques derived from logic and mathematics, and introducing new concepts of comparable profundity when required. This implies that I very much would like to voice constructive technical comments on his present contribution to IFIP'86. However, my present orientation in computer science doesn't enable me to contribute to his work along technical lines. On the other hand, criticizing his work is a precarious affair. Because of its rigor and elegance, technically speaking little remains to be criticized. Any criticism on his work tends therefore to be judged from this perspective, and puts the critic in the eyes of many an eminent scholar rather in the position of the accused. The only remaining solution for me is to try to put his work in perspective, as befitting someone who had to agree to respond prior to knowing the technical details of the material he had to respond to. Therefore I've chosen the following procedure: I shall try to view Robin's work through the eyes of some scholars who do eminent yet differently directed research in the field of concurrency, and whose contributions do not tend to occur often among Robin's lists of favorite references.

These scholars do not emphasize the question as to the nature of a concurrent process mathematically speaking, or the question which mathematical principles arise out of such a formal characterization of the true nature of such a process. Rather they address questions such as:

- How does one convince oneself and others that a process does what it is supposed to do according to its specification (a posteriori verification)?
- How does one design processes in order to make them meet their specification (the verify-while-develop paradigm)?
- How does one convince oneself and others that a process is correctly refined and/or implemented (the hierarchical decomposition paradigm)?
- What sense does it make to specify concurrent processes in terms of atomic actions which in any reality of programming aren't atomic at all?
- Do there exist notions of specification based on nonatomic reasoning which can be carried across levels of implementation?

In my opinion it would be a sign of that maturity of the field which is exemplified by Robin Milner's work, when comparison between and integration of different points of view could be freely discussed at this meeting.

As point of departure I quote a remark in a recent keynote address (at the 5th FST and TCS Conference, New Delhi) by one of Robin's compatriots:

> "Theorists should look at a restricted class of problems which professional programmers come face to face with and not work in abstraction and total generality."

And as questions which are not raised in Robin's works, but which are essential from the perspective of the above quotation for the theory of concurrency, I shall pose the following ones:

- The two languages versus a single language question, inspired by Amir Pnueli's work.
- The atomicity versus hierarchical decomposition question, inspired by Leslie Lamport's work.

And, since Robin urgently asked me to do so, I have also tried to formulate some technical questions concerning his approach:

- Some technical questions pertinent to the present paper, inspired by the work of Jan Bergstra & Jan-Willem Klop.

Obviously, neither Pnueli, nor Lamport, nor Bergstra & Klop carry any responsibility for any misconceptions of mine concerning either their own or Robin's work.

2. Questions

2.1 The two languages versus a single language question — Pnueli's point of view

There are two basically different approaches to the formalization of programming systems. The first suggests the use of two languages. One language is the programming language P which characteristically is prescriptive, algorithmic and effective in nature. This is the language in which we program and which should be understandable and executable by a limited intelligence device such as a computer. The second language is the specification language S, which should be descriptive, and powerful enough to express the requirements of a program without yet specifying how they are to be implemented. We may distinguish the roles of the two languages by saying that S should specify the *what* while P should specify the *how*. We refer to this approach as the two languages approach.

Under the two languages approach, the main paradigms for the construction of correct programs, involve relations between the two languages. Thus, the main formal problems studied are the problems of *verification* and *construction*. In the verification problem we are given two objects, a specification $s \in S$, and a program $p \in P$, and asked whether they are consistent, namely, whether the program p satisfies the specification s. In the construction problem we are given a specification $s \in S$, and asked to construct a program $p \in P$ that satisfies s. If the construction activity is expected to be fully automatic, we refer to it as *program synthesis*, while if it is expected to be carried out by disciplined humans, we refer to it as *development*.

An alternative approach recommends the usage of a single language. The language should be expressive enough in order to specify computational tasks implicitly, i.e., without actually programming them, and should have a well identified fragment that can be effectively and efficiently executed. Under this approach, the process of program construction starts by specifying the required task in the single language. The initial specification should emphasize the desired results or behaviour and pay little attention to the question of how they are to be algorithmically implemented. Then, by a sequence of transformations, the implicit definition of the required program or package is gradually transformed into an explicit implementation of the program.

Not too surprisingly, the two languages approach is compatible with *logic* as the main formalization tool, and the single language approach is usually supported by an *algebraic* framework. This is appropriate because the basic structure of logic if founded on the distinction between syntax and semantics, and the main relation is the heterogeneous *satisfiability* relation holding between objects of different kinds, between a *model* which belongs to the semantic domain and a *formula* which belongs to the syntactic domain. Analogously, the two languages approach to specification and development of programs, is based on the *satisfaction* relation holding between programs (semantics objects in some sense) and specifications, which typically to this approach are expressed in some logic-of-programs language. Examples of the two languages approach are Floyd's and Hoare's systems, the weakest precondition calculus, the μ-calculus for sequential programs and their corresponding extensions including temporal logic for concurrent programs.

The algebraic framework, on the other hand, is based on homogenous relations such as equality or inclusion between objects of the same kind. Consequently, it is an appropriate vehicle to support meaning — preserving or equivalence transformations. Successful examples of the single language approach, which are usually referred to as *algebraic*, are the abstract data type theory, LCF, and the program transformation approach (Burstall, Darlington, etc.) for sequential programs, and the algebraic approach to concurrency represented by the work of Milner, Hennessy, Bergstra & Klop, etc.

Milner's work has been consistently algebraic, i.e., strongly a single language approach. This extends to his LCF work which is based on the thesis that verification of recursively defined functions is mainly the establishment of equivalence or inclusion between an obviously correct but highly inefficient version of the function (the specification) and a more complex but also more efficient version which is the one to be finally used (the implementation). His work on concurrency to which I will summarily refer to as CCS, contains several very important ideas. Two of these ideas which are typically algebraic are his notion of semantics and the implied notions of specification and development.

Traditionally, the non-operational semantics of programming languages had two main roles. The first was explanatory, that is, explain the meaning of the new language by mapping it into a better understood and a more familiar mathematical structure. The second role, usually regarded as secondary, was to determine which programs should be regarded as equivalent. This second role is, for example, very important for an optimizing compiler because it defines the degree of freedom it has in rearranging the program without changing its declared meaning. Obviously, the explanatory component suggests a mapping between the programming language and another language, and is therefore inconsistent with the single language approach. Indeed, one of the novel ideas introduced by Milner in CCS was emphasizing the role of semantics in identification of equivalent programs, and actually basing his semantics on the definition of the equivalence relation between programs, completely discarding the explanatory component. I find it difficult to argue with this approach, since it is questionable what degree of clarity is gained by mapping recursive equations of programs into recursive domain equations.

The second typically algebraic cornerstone of CCS is the premise that the same language, namely CCS, should be used for both specification and implementation of computational tasks. The difference between these two uses of the same language lies only in the degree of effectiveness and efficiency. As in the LCF case, specifications are distinguished by being short and obviously correct, while implementations are expected to be efficient.

Points of Debate:

- The one language approach tends to produce overly restrictive specifications that describe how the program should be implemented rather than what it should do. If one chooses an implementation different from the one envisioned when writing the specification, then verifying its correctness becomes a difficult problem of proving the equivalence of two concurrent programs, and requires complex reasoning.

  This argument applies only if the property at hand can be specified, indeed, within that one language. But what to do if this cannot be done, such as specifying mutual exclusion? Mutual exclusion protocols represent one of the fundamental problems of concurrent processing. It seems to me they are discarded in your set-up.
- An advantage of the two level approach is that a certain amount of decidability can be preserved in the assertion language which is absent in the single language approach. This aspect of decidability is especially useful when developing machine-support systems for program development. How do you, an expert on such systems, respond to this aspect?
- A specification should in general specify a SET of processes rather than a single one. For example to specify a process that will do either an 'a' or a 'b' operation we write in CCS '$a + b$' which specifies the process that does both. There is no way to specify the set $\{a, b\}$. This has been recently corrected in Sifakis' work. Relevant to this is also the fact that Milner's CCS does not have a natural definition of INCLUSION relation that subsumes the equivalence relatin. Again Sifakis' recent work introduced an inclusion between his objects in his extended language which are really sets of processes. Hennessy's CCS, which is based on linear semantics, does have a general notion of inclusion (or ordering).
- CCS that was suggested as a formalism for dealing with concurrency translates concurrency away into nondeterminism. The equation $a \parallel b = ab + ba$ is typical to this translation. This is different from the approach in Petri nets, or from maximal parallelism as advocated by Salwicki & Müldner.

2.2 The atomicity versus hierarchical decomposition question — Lamport's point of view

One of the paradigms of computer science is *hierarchical decomposition*. Indeed, as some authors argue, every program of any complexity at all should be developed using this strategy.

Hierarchical decomposition occurs in essentially two different kinds.

One kind I shall call *refinement*, and may be characterized as decomposition in which the construct to be composed (= the composee) and the decomposed construct (= the composant) are expressed within the same semantic framework. This case applies, e.g., when one is programming using one fixed (wide-spectrum) programming language.

The other kind of hierarchical decomposition, which I call *implementation*, is the one which I shall focus on in the remainder of this section. Implementation in my sense is to be distinguished from refinement in that composee and composant are expressed in *different* semantic frameworks. E.g., the case applies when one is programming an abstract tree manipulation algorithm in a language without pointers, and this program has to run on a computer with fixed-size words. In case the abstract trees can grow arbitrarily large, pointers must be used in some form to implement them, and hence a different formalism is required for implementation.

Nearer to Robin's subject of research is the case that one is presented with a program written in CCS or CSP. Now communication in CCS is synchronous. Yet in networks communication is not synchronous. So if one wants to consider this CCS program as a high-level description of some network algorithm, it has to be implemented within an asynchronous context implying a formalism different from CCS. A more practical example of hierarchical decomposition of the implementation kind is given by the ISO reference model for networks.

A third example concerns real time. In which sense is a real time process a process in Robin Milner's sense? One might wish to simulate clocks within CCS, communicating their ticks via shared channels to all processes concerned. To what extent does this help a programmer having to deal with the protocols of the already partly standardized ISO-layers just referred to?

The introduced notion of implementation is closely related to that of *atomicity*. What is specified as a concurrent process on one level of atomicity is a different object once the atomic actions are implemented using another level of atomicity within a different semantic framework.

Point of debate:

- A general notion of concurrent process should be insensitive to the level of atomicity chosen within a framework. It should not matter whether operations are atomic or not, because a specification on one level of atomicity should remain relevant to a deeper, more implementation oriented, level on which the atoms of the previous level are decomposed and new, more realistic, semantic notions have been introduced. That is, my ideal notion of process should remain invariant under different implementations of increasing complexity and detail, and should not depend on an a priori fixed level of atomicity.

2.3 Technical questions — Bergstra & Klop's point of view

Points of debate:

In Robin's present paper prefixing is suggested as additional construction (Principle 8), instead of the more general operation of sequential composition. Now the advantage of general sequential composition is its greater expressiveness. There are programs which can be defined using a finite number of recursion equations in case general composition is available, but which would require an infinite number of such equations had merely prefixing been available (– a result of Hoare's). Now I seem to recall that general sequential composition can be simulated within CCS, and therefore the above remark only applies if no CCS-type concurrency is present. Yet Hoare's observation does point to the fact that general sequential composition is of great help when specifying programs, and that on the level of specification one does not want to be faced with cumbersome terms simulating general sequential composition. So why not introduce general sequential composition, as a first class citizen in its own right, as a primitive operation, and later on point to some refinement relationships?

- As Bergstra & Klop have argued, one shouldn't introduce τ immediately as the result of a communication ($c/\bar{c} = \tau$), but one should rather introduce an intermediate step i, which should be renamed later as τ: $c/\bar{c} = i$ and later $\tau_{\{i\}}(i) = \tau$.

  Their motivation is that i is sometimes needed as guard in recursion equations in order to obtain unique solutions of these equations. The fact that equations such as $X = \tau X$ have no unique solutions complicates their understanding and ease of manipulation.

- How do you distinguish between deadlock and divergence, and between deadlock and termination?

- (Axiomatic approach) A characteristic feature of Bergstra & Klop's algebra of concurrent processes (ACP) is their (algebraic) axiomatic basis. This contrasts with the usual way of characterizing concurrent processes prior to Bergstra & Klop's work (to the best of my knowledge), which uses (some abstraction of the notion of) trees, and then "discovers" that laws such as $x + x = x$ apply.

  An axiomatic methodology inverts this style of characterization. First axioms are given, then their models are studied (cfr. the theory of groups, cited on pg. 2 of Robin's book). The first advantage of the axiomatic approach is its model theory in which the possible variations between models of the same theory are established, and relationships between these models derived. A second advantage is that axiom systems can be manipulated in modular fashion, such as is, e.g., the case in the theory of abstract data types.

Should one interpret your current lecture as positive indication of the fact that you have also been converted to this viewpoint?

3. References

{1} J.A. Bergstra and J.W. Klop, Process Algebra for Synchronous Communication. *Information and Control*, vol. 60, nos. 1–3, January/February/March 1984.

{2} L. Lamport, Specifying concurrent program modules, *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 2, April 1983.

{3} H. Barringer, R. Kuiper, A. Pnueli, And now you may compose temporal logic specifications, *Proc. of the 16th ACM Symposium on the Theory of Computing*, 1984.

Acknowledgements

I wish to thank the Weizman Institute of Science for its hospitality, and Carol Weintraub for typing the manuscript upon late notice.