

Wanted: a compositional approach to concurrency

C B Jones

School of Computing Science
University of Newcastle
NE1 7RU, UK
e-mail: cliff.jones@ncl.ac.uk

Abstract. A key property for a development method is *compositionality* because it ensures that a method can scale up to cope with large applications. Unfortunately, the inherent *interference* makes it difficult to devise development methods for concurrent programs (or systems). There are a number of proposals such as rely/guarantee conditions but the overall search for a satisfactory *compositional approach to concurrency* is an open problem. This paper identifies some issues including granularity and the problems associated with ghost variables; it also discusses using atomicity as a design abstraction.

Please cite the original source of this article: *Wanted: a compositional approach to concurrency*, C B Jones, pp. 1-15, in *Programming Methodology*, (eds.) Annabelle McIver and Carroll Morgan, Springer Verlag, 2000.

1 Compositionality

Formal specification languages and associated rules for proving that designs satisfy specifications are often called *formal methods*. As well as providing completely formal criteria, it is argued in [Jon00] that formal methods offer thinking tools –such as invariants– which become an integral part of professional practice. The main interest in this paper is on the contribution that formal methods can make to the design process for concurrent systems. Just as with Hoare’s axioms for sequential programs, the sought after gains should come both from (the reference point of) formal rules and from the intuitions they offer to less formal developments.

The development of any large system must be decomposed into manageable steps. This is true both for the construction phase and for subsequent attempts to comprehend a design. For software, understanding after construction is important because of the inevitable maintenance and modification work. But, for the current purposes, it is sufficient to concentrate the argument on the design process.

It is easy to see that it is the design process of large systems which requires support. Regardless of the extent to which techniques for error detection of finished code can be made automatic, there is still the inherent cost of reworking the design when errors are detected and little greater certainty of correctness after modification. The only way to achieve high productivity and correctness is to aim to make designs correct by construction.

What is required therefore is to be able to make and justify one design decision before moving on to further steps of design. To take the design of sequential programs as a reference point, specification by pre and post conditions offers a natural way of recording what is required of any level of component. So, in facing the task of developing some C specified by its pre and post conditions, one might decide that a series of sub-components sc_i are required and record expectations about them by writing their pre and post conditions. The design step must also provide a proposed way of combining the eventual sc_i and this should be one of the constructs of the (sequential) programming language. Each such construct should have an associated proof rule like the Hoare axiom for **while** which can be used to show that any implementations satisfying the specifications of the sc_i will combine with the stated construct into an implementation satisfying the specification of C . This idealised top-down picture requires some qualification below but the essential point remains: pre and post conditions provide an adequate description of the functionality of a system to facilitate the separation of a multi-level design into separate steps. A method which supports such development is classed as *compositional*; one that requires details of the implementations of the sc_i to justify the decomposition is non-compositional.

The above ideal is rarely achieved. The first difficulty is that there is no guarantee against making bad design decisions which result in the need to backtrack in the design process. What a compositional method offers is a way of justifying a design step — not an automatic way of choosing good design decisions. Secondly, there was above a careful restriction to functional properties and performance

considerations, in particular, are commonly excluded. There are also a number of technical points: the case for separating pre from post conditions and the arguments for employing post-conditions of two states (plus the consequent search for apposite proof rules) are explored in [Jon99]. It will also come as no surprise to anyone who has read this author's books on VDM that the method of data reification is considered an essential tool for program design; fortunately there is also a transitivity notion for reification which again facilitates compositional design (see [dRE99] for an excellent survey of data refinement research).

Nothing which has been written above should be seen as assuming that all design has to be undertaken in a top-down order: separate, justification of design steps is necessary in whatever order they are made; a top-down structure of the final documentation might well enhance comprehensibility; and arguments based on specifications rather than on the details of the code have much to commend them however these arguments are discovered.

The key argument of this section is that compositionality is a desirable property of a development method if it is to scale up to large tasks. Subsequent sections explore the difficulties in achieving this property in the presence of concurrency.

2 The essence of concurrency is interference

The easiest way to illustrate interference is with parallel processes which can read and write variables in the same state space. Simple examples can be constructed with parallel execution of assignment statements but to avoid an obvious riposte it is necessary to resolve an issue about granularity. Some development methods assume that assignment statements are executed atomically in the sense that no parallel process can interfere with the state from the beginning of evaluation of the right hand side of the assignment until the variable on the left hand side has been updated. The rule is reciprocal in the sense that the assignment in question must not interfere with the atomic execution of one in any other process. Essentially, assignments in all processes are non-deterministically merged in all processes but never allowed to overlap. A few moments thought makes it clear that such a notion of granularity would be extremely expensive to implement because of the setting and testing of something equivalent to semaphores. There is a suggestion to remove the need for semaphores: sometimes referred to as "Reynold's rule", the idea is to require no more than one reference (on the left or right hand sides) in any assignment to potentially shared variables. Section 6 argues that not even variable access or change are necessarily atomic; but even without opening this facet to investigation, one can observe that Reynold's rule is also arbitrary and prohibits many completely safe programs.

Thus, for the purposes of this section, assignment statements are not assumed to be executed in an atomic step. If then a variable x has the value 0 before two assignment statements

$$x \leftarrow x + 1 \parallel x \leftarrow x + 2$$

are executed in parallel, what can be said of the final value of x ? In the simplest case, where one parallel assignment happens to complete before the other begins, the result is $x = 3$; but if both parallel assignments have their right hand sides evaluated in the same state ($x = 0$) then the resulting value of x could be 1 or 2 depending on the order of the state changes.¹

Some computer scientists recoiled at the difficulty of such shared state concurrency and their idea of stateless communicating processes might appear to finesse the problem illustrated above. Unfortunately, escaping the general notion of interference is not so easy. In fact, since processes can be used to model variables, it is obvious that interference is still an issue. The shared variable problem above can be precisely mirrored in, for example, the π -calculus [MPW92] as follows

$$(\bar{x}0 \mid !x(v).(\bar{r}_x v.\bar{x}v + w_x(n).\bar{x}n)) \mid r_x(v).\bar{w}_x v + 1 \mid r_x(v).\bar{w}_x v + 2$$

One might argue that assertions over communication histories are easier to write and reason about than those over state evolutions but the issue of interference has clearly not been avoided. Furthermore, interference affects liveness arguments as well as safety reasoning.

3 Reasoning about interference

Before coming to explicit reasoning about interference, it is instructive to review some of the early attempts to prove that shared-variable concurrent programs satisfy specifications. One way of proving that two concurrent programs are correct with respect to an overall specification is to consider their respective flow diagrams and to associate an assertion with every pair of arcs (i.e. quiescent points). So with sc_1 having n steps and sc_2 having m , it is necessary to consider $n \times m$ steps of proof. This is clearly wasteful and does not scale at all to cases where there are more than two processes. There is also here an assumption about granularity which is dangerous: are the steps in the flow diagram to be whole assignments? For the current purposes, however, the more fundamental objection is that the approach is non-compositional: proofs about the two processes can only be initiated once their final code is present; nothing can be proved at the point in time where the developer chooses to split the overall task into two parallel processes; there is no separate and complete statement of what is required of each of the sc_i .

Susan Owicki's thesis [Owi75] proposes a method which offers some progress. Normally referred to as the Owicki-Gries method because of the paper she wrote [OG76] with her supervisor David Gries, the idea is to write normal pre/post condition specifications of each of the sc_i and develop their implementations separately with normal sequential proof rules. Essentially, this first step can be thought of as considering the implementation as though it is a non-deterministic choice between one of two sequential implementations: sc_1 ; sc_2 or

¹ Atomicity of update of scalar values is assumed – for now!

$sc_2; sc_1$. Having completed the top level decomposition, developments of the separate sc_i can be undertaken to obtain code which satisfies their specifications. So far, so good — but then the Owicki-Gries method requires that each program step in sc_i must be shown not to interfere with any proof step in sc_j . With careful design, many of these checks will be trivial so the worrying product of $n \times m$ checks is not as daunting. It is however again clear that this method is non-compositional in that a problem located in the final proof of “interference freedom” could force a development of sc_i to be discarded because of a decision in the design of sc_j . In other words, the specification of sc_i was incomplete in that a development which satisfied its pre and post condition has to be reworked at the end because it fails some criteria not present in its specification.

Several authors took up the challenge of recording assumptions and commitments which include a characterisation of interference. In [FP78], an interference constraint has to be found which is common to all processes. In [Jon81], pre/post conditions specifications for such processes are extended with rely and guarantee conditions. The subsequent description here is in terms of the rely/guarantee proposal.

The basic idea is very simple. Just as a pre-condition records assumptions the developer can make about the initial state when designing an implementation, a rely condition records assumptions that can be made about interference from other processes: few programs can work in an arbitrary initial condition; only vacuous specifications can be met in the presence of arbitrary interference. Thus pre and rely conditions record assumptions that the developer can make.

Just as post-conditions document commitments which must be (shown to be) fulfilled by the implementation, the interference which can be generated by the implementation is captured by writing a guarantee condition.

A specification of a component C then is written $\{p, r\} C \{g, q\}$ for a pre-condition p , a rely condition r , a guarantee condition g , and a post-condition q . It has always been the case in VDM that post-conditions were predicates of the initial and final states ²:

$$q: \Sigma \times \Sigma \rightarrow \mathbb{B}$$

Since they record (potential) state changes, it is natural that rely and guarantee conditions are both relations:

$$r: \Sigma \times \Sigma \rightarrow \mathbb{B}$$

$$g: \Sigma \times \Sigma \rightarrow \mathbb{B}$$

Pre-conditions indicate if an initial state is acceptable and are thus predicates of a single state:

$$p: \Sigma \rightarrow \mathbb{B}$$

² See [Jon99] for discussion

The compositional proof rule for decomposing a component into two parallel components is presented in Figure 1. It is more complicated than rules for sequential constructs but is not difficult to understand. If $S_1 \parallel S_2$ has to tolerate interference r , the component S_1 can only assume the bound on interference to be $r \vee g_2$ because steps of S_2 also interfere with S_1 . The guarantee condition g of the parallel construct cannot be stronger than the disjunction of the guarantee conditions of the components. Finally, the post condition of the overall construct can be derived from the conjunction of the individual post-conditions, conjoined with the transitive closure of the rely and guarantee conditions, and further conjoined with any information that can be brought forward from the pre-condition \overleftarrow{p} .

$$\boxed{\text{III}} \frac{\begin{array}{l} \left\{ p, r \vee g_2 \right\} S_1 \left\{ g_1, q_1 \right\} \\ \left\{ p, r \vee g_1 \right\} S_2 \left\{ g_2, q_2 \right\} \\ g_1 \vee g_2 \Rightarrow g \end{array}}{\frac{\neg p \wedge q_1 \wedge q_2 \wedge (r \vee g_1 \vee g_2)^* \Rightarrow q}{\left\{ p, r \right\} (S_1 \parallel S_2) \left\{ g, q \right\}}}$$

Fig. 1. A proof rule for rely/guarantee conditions

There are more degrees of freedom in the presentation of such a complex rule than those for sequential constructs and papers listed below experiment with various presentations. It was however recognised early that there were useful generic thinking tools for reasoning about concurrent systems. “Dynamic invariants” are the best example of a concept which is useful in formal and informal developments alike. A dynamic invariant is a relation which holds between the initial state and any which can arise. It is thus reflexive and composes with the guarantee conditions of all processes. It is accepted by many who have adopted methods like VDM that standard data type invariants are a valuable design aid and their discussion even in informal reviews often uncovers design errors. There is some initial evidence that similar design pay off comes from dynamic invariants. In fact, they have even been seen as beneficial in the design of sequential systems (e.g. [FJ98]).

There have been many excellent contributions to the rely/guarantee idea in the twenty years since it was published ([Jon83] is a more accessible source than [Jon81]). Ketil Stølen tackled the problem of progress arguments in his thesis [Stø90]. Xu Quiwen [Xu92] in his Oxford thesis covers some of the same ground but also looks at the use of equivalence proofs. Pierre Collette’s thesis was done under the supervision of Michel Sintzoff: [Col94] makes the crucial link to Misra and Chandy’s Unity language (see [CM88]). Colin Stirling tackles the issue of Cook completeness in [Sti88] and in [Sti86] shows that the same

broad form of thinking can be applied to process algebras. Recent contributions include [Din00]³ and [BB99].

Returning to the fact that there have been other assumption-commitment approaches which record interference in ways different from the specific rely-guarantee conditions used here, the reader is referred to the forthcoming book from de Roever and colleagues for a review of many approaches. As far as this author is aware, none of the recorded approaches avoids the difficulties discussed in the next sections.

4 Some problems with assumption/commitment reasoning

In spite of the progress with rely-guarantee specifications and development, much remains to be done. It should not be surprising that reasoning about intimate interference between two processes can be tricky. An illustration of how delicate the placing of clauses in assumptions and commitments is given in [CJ00]. Perhaps much of what is required here is experience and classification of types of interference.

One obvious conclusion is to limit interference in a way which makes it possible to undertake much program development with sequential rules. This echoes the message that Dijkstra et al. were giving over the whole early period of writing concurrent programs. One avenue of research in this direction has been to deploy object-based techniques to provide a way of controlling interference; this work is outlined –and additional references are given– in [Jon96].

Turning to the rules for the parallel constructs, that given in Figure 1 is only one with which various authors who are cited above have experimented. There more degrees of freedom than with rules for sequential constructs.⁴ Again, experiments should indicate the most usable rules.

There are some general developments to be looked at in combination with any form of assumption-commitment approach. One is the need to look at their use in real-time programs. Intuitively, the same idea should work but what the most convenient logic is in which to record assumptions and commitments might take considerable experimentation. Another extension which would require careful integration is that to handle probabilistic issues. This is of particular interest to the current author because –as described in [Jon00]– of the desire to cover “faults as interference”.

³ Note [Sti88,Din00] employ unary predicates and experience the problems that are familiar from unary post-conditions when wanting to state requirements like variables not changing.

⁴ There is of course some flexibility with sequential constructs such as whether to fold the consequence rule into those for each programming construct.

5 Role of ghost variables

A specific problem which arises in several approaches to proofs about concurrency is finding some way of referring to points in a computation. A frustratingly simple example is the parallel execution of two assignment statements which are, for this section, assumed to be atomic.

$$\langle x \leftarrow x + 1 \rangle || \langle x \leftarrow x + 2 \rangle$$

The subtlety here is that because both increments are by the same amount one cannot use the value to determine which arm has been executed. A common solution to such issues is to introduce some form of “ghost variable” which can be modified so as to track execution. There are a number of unresolved questions around ghost variables including exactly when they are required; what is the increase in expressivity and their relationship to compositionality.

For the specific example above, this author has suggested that it might be better to avoid state predicates altogether and recognise that the important fact is that the assignments commute. Of course, if one branch incremented x and the other multiplied it by some value, then they would not commute; but it would also be difficult to envisage what useful purpose such a program would have. So the proposal is that reasoning about concurrency should not rely solely on assertions about states; other –perhaps more algebraic techniques– can also be used to reason about the joint effect of actions in parallel processes.

6 Granularity concerns

Issues relating to granularity have figured in the discussion above and they would appear to pose serious difficulties for many methods. The problems with assuming that assignment statements can be executed atomically are reviewed in Section 2 but the general issue is much more difficult. For example, it is not necessarily true that variables can be read and changed without interference. This should be obvious in the case of say arrays but it is also unlikely that hardware will guarantee that long strings are accessed in an uninterrupted way. There is even the danger that scalar arithmetic value access can be interrupted.

Having ramified the problem, what can be done about it? Any approach which requires recognising the complete proof of one process to see whether another process can interfere with proof steps appears to be committed to low level details of the implementation language. To some extent, a rely-guarantee approach puts the decision about granularity in the hands of the designer. In particular, assertions carried down as relations between states can be reified later in design. This works well in the object-based approach described in [Jon96]. But granularity is a topic which deserves more research rather than the regrettable tendency to ignore the issue.

7 Atomicity as an abstraction, and its refinement

As well as rely and guarantee conditions, the object-based design approach put forward in [Jon96] employs equivalence transformations. The idea is that a relatively simple process could be used to develop a sequential program which can be transformed into an equivalent concurrent program. The task of providing a semantic underpinning in terms of which the claimed equivalences could be proved to preserve observational equivalence proved difficult (see for example [PW98,San99]).

The key to the equivalences is to observe that under strict conditions, islands of computation exist and interference never crosses the perimeter of the island. One of the reasons that these equivalences are interesting is because their essence –which is the decomposition of things which it is easy to see possess some property when executed atomically– occurs in several other areas. In particular, “atomicity” is a useful design abstraction in discussing database transactions and cache coherence: showing how these “atoms” can overlap is an essential part of justifying a useful implementation. There are other approaches to this problem such as [JPZ91,Coh00]; but the ubiquity of atomicity refinement as a way of reasoning about some concurrency problems suggests that there is a rich idea lurking here.

8 Conclusion

The general idea behind assumption/commitment specifications and proof rules would appear to be a useful way of designing concurrent systems. Much detailed research and experimentation on practical problems is still required to come up with some sort of agreed approach. Even as a proponent of one of the assumption (rely) commitment (guarantee) approaches, the current author recognises that there are also quite general problems to be faced before a satisfactory compositional approach to the development of concurrent programs can be claimed. One area of extension is to look for more expressiveness whether to merge with real-time logics or to cope with probabilities. Another issue is that arguments which do not appear to be dealt with well by assertions about states. In all of this search for formal rules, one should continue to strive for things which can be adopted also informally as thinking tools by engineers.

Acknowledgements

I gratefully acknowledge the financial support of the UK EPSRC for the Interdisciplinary Research Collaboration “Dependability of Computer-Based Systems”. Most of my research is influenced by, and has been discussed with, members of IFIP’s Working Group 2.3.

References

- [BB99] Martin Buechi and Ralph Back. Compositional symmetric sharing in B. In *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 431–451. Springer-Verlag, 1999.
- [BG91] J. C. M. Baeten and J. F. Groote, editors. *CONCUR'91 – Proceedings of the 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
- [CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction*, chapter 10, pages 275–305. MIT Press, 2000.
- [CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
- [Coh00] Ernie Cohen. Separation and reduction. In *Mathematics of Program Construction, 5th International Conference, Portugal, July 2000*, pages 45–59. Springer-Verlag, 2000.
- [Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY*. PhD thesis, Louvain-la-Neuve, June 1994.
- [Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.
- [dRE99] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, 1999.
- [FJ98] John Fitzgerald and Cliff Jones. A tracking system. In J. C. Bicarregui, editor, *Proof in VDM: Case Studies*, FACIT, pages 1–30. Springer-Verlag, 1998.
- [FP78] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.
- [Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.
- [Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.
- [Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.
- [Jon99] C. B. Jones. Scientific decisions which characterise VDM. In *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 28–47. Springer-Verlag, 1999.
- [Jon00] C. B. Jones. Thinking tools for the future of computing science. In *Dagstuhl*, volume 2000 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
- [JPZ91] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In [BG91], pages 298–316, 1991.
- [MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
- [OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.
- [Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975. 75-251.

- [PW98] Anna Philippou and David Walker. On transformations of concurrent-object programs. *Theoretical Computer Science*, 195:259–289, 1998.
- [San99] Davide Sangiorgi. Typed π -calculus at work: a correctness proof of Jones’s parallelisation transformation on concurrent objects. *Theory and Practice of Object Systems*, 5(1):25–34, 1999.
- [Sti86] C. Stirling. A compositional reformulation of Owicki-Gries’ partial correctness logic for a concurrent while language. In *ICALP’86*. Springer-Verlag, 1986. LNCS 226.
- [Sti88] C. Stirling. A generalisation of Owicki-Gries’s Hoare logic for a concurrent while language. *TCS*, 58:347–359, 1988.
- [Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. available as UMCS-91-1-1.
- [Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming*. PhD thesis, Oxford University, 1992.