Splitting atoms safely

C. B. Jones

School of Computing Science, Newcastle University, NE1 7RU, England
cliff.jones@ncl.ac.uk

Abstract

The aim of this paper is to make a contribution to (compositional) development methods for concurrent programs. In particular, it takes a fresh look at a number of familiar ideas including the problem of *interference*. Some subtle issues of observability –including *granularity*– are explored. Based on these points, the paper sets out some requirements for an approach to developing systems by "splitting atoms safely".

1 Introduction

If an action is said to be executed "atomically", it is assumed that it will not be affected by interference and that the environment will not be able to observe intermediate steps of the action in question. The bugbear of concurrency is that interference must be tolerated. With shared state programs, an action must achieve some required result even though its state can be changed by other interfering processes (and interference is also at the heart of communication-based concurrency).

The aim here is to argue that there is a useful method for developing concurrent programs which explicitly uses a "fiction of atomicity" as an abstraction and then allows steps of development which "split atoms safely". This development process might be called "refining atomicity". One of the things which makes the approach interesting is that it is used widely: Edsger Dijkstra's [Dij82] is one of the early examples of making solutions "more and more fine-grained". Much of database implementation (for a recent description see [WV01]) is also about preserving the fiction of atomicity when implementations deliberately overlap the execution of transactions. What is sought here is a *general development method* which has the properties of other formal methods for developing programs.

Before an outline of a development method for refining atomicity can be given in Section 4, it is necessary to look more closely at issues like "granularity". In passing we note how valuable operational semantics is in this study.

John Reynolds recognised role of "interference" in [Rey81] where he focused on the interference caused by the sharing brought about by parameter passing. John's other thoughts on this topic include notions of "separation logic" and "syntactic interference" [Rey78,Rey89,Rey02]. Interestingly, John also made the link with atomicity in a recent talk at POPL'04 (I am grateful to Peter O'Hearn for bringing this to my attention). By coincidence, my thesis [Jon81] was approved in the same year as the publication of John's book [Rey81]. My suggestion there was to use rely/guarantee conditions to specify and reason about the interference which comes from concurrent execution of shared variable imperative programs. Further discussion of rely/guarantee conditions –and citations to developments– are in Section 3. In particular, both the reasons for wanting to limit the use of the rules for rely/guarantee conditions and their subtle interaction with granularity are explored below. Both of these issues are important in understanding the case for "refining atomicity".

2 Interference

It is perhaps useful to say a few words first about "states" *per se*. What classifies a programming language as "imperative" is the ability of its programs to change some form of state. At a macro-level, this might be the content of files or a database; within a program, the state consists of a collection of variables which are affected by assignment statements of some form. The majority of this paper is concerned with imperative programs in which interference is allowed to manifest itself as state changes. While one can try to avoid states, they are an extremely useful abstraction even when specifying a system. (Section 5 indicates how interference reappears with communication-based parallelism and discusses the relevance of the other ideas in this paper to process algebras.)

It has been argued elsewhere (e.g. [Jon90]) that a single state value represents an equivalence class of histories of a system. Consider, for example, the specification given in Figure 1 which indicates how a priority queue might be specified. [1] Starting with an initial state which contains the empty set, operations *ENQ* and *DEQ* (the latter subject to its pre-condition) can be performed in any order; both operations are shown (**ext wr**) as changing the state (*queue*). The operation *ENQ* takes an argument but delivers no result whereas *DEQ* takes no argument and delivers a result. The VDM use –in

[1] The specification here is written in VDM; transliterating into, for example, "B" [Abr96] would change nothing essential.

$ENQ\ (new\colon X)$
ext wr $queue\ :\ X$-**set**
post $queue = \overleftarrow{queue} \cup \{new\}$

$DEQ\ ()\ r\colon X$
ext wr $queue\ :\ X$-**set**
pre $queue \neq \{\,\}$
post $r = mins(\overleftarrow{queue}) \wedge queue = \overleftarrow{queue} - \{r\}$

Fig. 1. A specification for a Priority Queue

post-conditions– of marking the initial value of the state (e.g. $\overleftarrow{queue}$) follows a suggestion of Peter Aczel [Acz82]. The function *mins* is assumed to deliver the minimum value from its (non-empty set) argument

$$mins\colon X\text{-}\mathbf{set} \to X$$

The point about state values representing an equivalence class of histories can now be applied to this example by noting that

$ENQ(x_2)$
$ENQ(x_1); ENQ(x_2); DEQ()$
$ENQ(x_2); ENQ(x_1); DEQ()$
where
$mins(\{x_1, x_2\}) = x_1$

all leave the state value as

$$\{x_2\}$$

It is important to appreciate that the intention of a specification like that in Figure 1 is that the external behaviour is what is defined. It is *not* required that the internal state is implemented with a set data type. In fact, what is called in [Jon90] "data reification" (elsewhere "data refinement") is a development method which has rules for showing that behaviour can be preserved with changed data structures.[2] The key point is that it is assumed that the only way of *observing* the behaviour of the priority queue is with the stated operations.

[2] Here again, the specifics of the VDM approach ("retrieve functions", "adequacy", "implementation bias") are unimportant; for a broad comparison of approaches to "data refinement" see [dRE99].

The post-conditions of Figure 1 define acceptable final results and, for sequential programs, the issue of atomicity does not arise. Again, the key point of "observability" is that no process can see intermediate states of the operations. VDM uses the phrase "operation decomposition" for the use of proof rules for introducing programming language constructs like **while**. These rules are like "Hoare axioms" except that they cope with VDM's insistence on post-conditions of two states (initial and final). A post-condition does not require execution in a single step, such execution can only be thought of as atomic in the sense of no interference on –and no visibility of– intermediate states. Development is expected to create a program which executes in many steps; but for sequential programs, we ignore interference during their execution.

Since the priority queue example is used below, it is worth highlighting the point about the range of implementations: an implementer could, for example, arrange for quick response to *ENQ* by adding new elements to an unordered state — *DEQ* would then need to determine the minimum element; at the other extreme, there are implementations in which *ENQ* takes more time to place new elements in an ordered data structure so that *DEQ* responds quickly. Section 4 shows how concurrency can be used to make both operations respond quickly.

Experience with the use of formal development methods like VDM or B suggests a number of properties that are desirable:

(1) designers are good at making design decisions and a method should support the stepwise introduction of detail from an abstract specification through to an executable implementation;
(2) design steps must support –not constrain– a designer's intuition;
(3) recording the steps ought to yield a useful design history of the artefact;
(4) the advantages of redundancy and diversity can be enjoyed in the development process if the designer can *posit* a design step and the rules of the method can generate *proof obligations* whose discharge ensures the step is correct (with respect to previous specifications);[3]
(5) the amount of proof work to discharge the proof obligations must bear a reasonable relation to the size of the step;
(6) in particular, while the justification of the proof obligations themselves might prove challenging, their *use* should be within the grasp of software engineers;
(7) one step of development ought not to be invalidated by later decisions.

The last property (7) warrants expansion since it is important to carry forward from methods which handle sequential programs to the objectives for

[3] The phrase "posit and prove" has been used to describe the way in which a designer can make a natural design step and justify that it satisfies a prior specification.

methods that cope with concurrency. Both operation decomposition and data reification are "compositional" in the sense that the specifications define all that is required of an implementation. In other words, one can prove that one step of design is correct and know that –if sub-components are developed according to their specifications– there will not be a need to reject them because they do not fit their context. Understanding why compositionality is difficult to achieve for concurrent programs is a prerequisite for devising useful development methods.

Interference is the essence –and bugbear– of concurrent execution. There are two ways in which the sort of uninterrupted operations of the standard (sequential) interpretation of VDM specifications are too limiting. In many important computer systems, the notion of overlapping updates is inherent in their intended function. An obvious example is a banking system which has to deal with concurrent transfer transactions. (The links to database transaction ideas are picked up at the end of the next section.) The other reason for wanting development methods which do not assume that operations like *ENQ* and *DEQ* of Figure 1 are executed atomically is that the implementation itself might use concurrent operations to increase performance. (This is of course making assumptions about multi-processor implementations but the concern here is on specifying and reasoning about concurrency not on utilizing hardware.) The example which is used below –see Figure 2 and the discussion around it– to draw out more subtle points of atomicity is determining the prime numbers up to some stated value by "sieving" out all of the composites: specifying separate processes which each remove multiples of their own index is an example of the sort of interference which must be handled.

The "Owicki/Gries method" [Owi75,OG76] extended Hoare-like approaches to handle concurrency but the resulting method is non-compositional in the sense that proven developments of independent processes might have to be discarded if they fail a final "interference freedom" property *of their proofs*. The fact that this test cannot be applied until *after the complete development of each process*, means that their rejection would require much rework.

It was striving for compositionality which led this author to look for ways of documenting interference in rely/guarantee conditions.[4] Essentially, a *rely condition* records the interference which an implementation must tolerate on its state and a *guarantee condition* documents a limit on the interference the component can generate. Both rely and guarantee conditions are –like VDM's post-conditions– relations over pairs of states. Like pre-conditions, rely conditions can be thought of as giving permission for the implementer to ignore

[4] In addition to the thesis cited above [Jon81], more readily obtained publications are [Jon83a,Jon83b,Jon96]. Further research contributions to this approach to interference include[Stø90,Col94,Xu92,Bue00,Din00,BS01].

certain potential deployments of the code to be created (few systems work in an arbitrary starting state; even fewer can tolerate arbitrary state changes during their execution). On the other hand, guarantee conditions are like post-conditions in that they record an obligation on the created program. The referenced papers contain proof rules for showing that a decomposition into parallel processes will be correct if the components are developed so as to satisfy their specifications.

The details of particular methods vary but are not important here; [dR01] compares different compositional and non-compositional approaches.

What *is* of interest in this paper is the connection between interference and atomicity. This is explored in the next section after some insight is sought from operational semantics. The immediate reason for looking at techniques for describing language semantics is to indicate that SOS provides a straightforward way to formalize many of the points on granularity etc. This material is only sketched here because it is relatively routine. The real justification for exploring SOS can be seen in Section 5 when the issue of justifying design methods is addressed.

McCarthy's original view [McC66] of an "abstract interpreter" recorded operational semantics by a (recursive) function $exec$: $Statement \times \Sigma \to \Sigma$, (where Σ is the set of machine states) which, for a given program and a given starting state, computes the final state. These were interpreters, but made abstract by the use of abstract objects such as sets and maps (especially the use of the latter for environments and stores).

The generalization to define non-deterministic constructs requires a function which computes the set of possible final states. But this idea does not generalise easily to cover concurrency because of the need to define all possible merges of order to get the set of possible resulting states $exec$: $Statement \times Statement \times \Sigma \to \mathcal{P}(\Sigma)$. One could define a predicate which characterized the valid state pairs but it is actually far easier to move to an SOS frame.

It is observed in [Jon03] that a crucial contribution of "Structural Operational Semantics" (as in [Plo81] [5]) is that the presentation as rules shifts the non-determinism to a meta-level: "Plotkin rules" appear to show single transitions and it is the fact that more than one rule matches a given situation that expresses the non-determinism.

One can think of an SOS description as defining a relation over pairs of program texts and states; thus

$$\xrightarrow{s}: \mathcal{P}((Statement \times \Sigma) \times (Statement \times \Sigma))$$

[5] Now reprinted as [Plo04b] with an accompanying note [Plo04a].

What is going on with shared variable concurrency is that interference occurs when more than one thread of control can read and/or write to the same portion of a state.

As a simple illustration, consider parallel execution of two sequences of assignments. Ignoring for now the possibility of non-determinism in expression evaluation, use

$$\mathit{eval}\colon \mathit{Expression} \times \Sigma \to \mathit{Value}$$

The (atomic) execution of the assignment statement from the head of the left sequence is expressed as

$$\frac{v = \mathit{eval}(e, \sigma)}{([x \leftarrow e] \overset{\curvearrowright}{;} \mathit{restleft} \mid\mid \mathit{right}, \sigma) \xrightarrow{s} (\mathit{restleft} \mid\mid \mathit{right}, \sigma \dagger \{x \mapsto v\})}$$

With the obvious symmetric rule for the right sequence,

$$\frac{v = \mathit{eval}(e, \sigma)}{(\mathit{left} \mid\mid [x \leftarrow e] \overset{\curvearrowright}{;} \mathit{restright}, \sigma) \xrightarrow{s} (\mathit{left} \mid\mid \mathit{restright}, \sigma \dagger \{x \mapsto v\})}$$

This shows how non-determinism can arise depending on the order in which statements are executed from the two parallel streams. Thus

$$(x \leftarrow x * 2; x \leftarrow x * 3) \mid\mid (x \leftarrow x * 4; x \leftarrow x * 5)$$

will, if the initial value of x is 1, set the final value of x to factorial 5. Whereas, when x starts at 1

$$(x \leftarrow x + 1) \mid\mid (x \leftarrow x * 2)$$

can leave x as 3 or 4.

Although the basic points are illustrated here with assignment statements, this should not disguise the fact that the same issues arise at different language levels. Interference could be shown with separate programs accessing shared files or separate "transactions" changing a database.

Of course, there are extensive areas of programming language design devoted to *limiting interference* (e.g. semaphores, conditional critical regions, monitors) and thus making it safer to write programs in such languages. But the semantics of a language must illustrate the most general case and our purpose below is to find safe ways of developing interfering programs.

3 Granularity

It should already be clear that "observability" has a key connection with atomicity: it is, in fact, not too strong to say that it *defines* the notion of sequential (non-interfering) programs. This section explores more subtle issue of "granularity".

The operational semantics in the previous section shows assignment statements being executed atomically. That is, if the head of *left* (say $x \leftarrow e$) is being executed, there are no state changes made between the beginning of the evaluation of e and the change to x; nor can *right* be observing x whilst it is being changed. For a useful programming language, such an assumption of atomicity is unrealistic in that it would be extremely expensive to implement (in terms of say semaphore setting). [6]

One attempt to avoid the problems posed by two threads referring to shared variables is to say that any assignment statement can use (in either left or right-hand contexts) at most one shared variable. This is sometimes referred to as "Reynolds' rule". [7] It has its own obvious disadvantage in that *any* statement of the form

$$x \leftarrow e(x)$$

has to be rewritten as

$$local \leftarrow e(x); x \leftarrow local$$

even where the logic of the program shows that in this context, no other thread could change x. Moreover, the expansion does nothing to avoid "lost updates".

More seriously, this idea gives no clue as to how one might handle variables which cannot be accessed and/or changed atomically: consider for example array or record assignments (but it is not necessarily safe to assume that numbers can be changed by an indivisible machine operation).

The preceding section explains how rely and guarantee conditions are assertions about interference but the cited papers on this way of developing con-

[6] If assignment statements are not to be executed atomically, interference can occur during the evaluation of expressions (and the last example of the previous section could also result in x having the value 2). It is worth noting that an operational semantics for this "finer granularity" needs to show values being slotted into expressions (see discussion in [Jon03] as to why this is troublesome and what might be done to circumvent it).

[7] Attributed to Reynolds in [Owi75] but I have heard John disown it!

current programs only hint at the question of "atomicity". In cases like simple (Boolean) switches, a rely condition which states that the environment will never set *myswitch* to **false**

$$\overleftarrow{myswitch} \;\Rightarrow\; myswitch$$

is safe. Conditions that, say, state a variable is monotonically decreasing are quite often needed in program developments using rely/guarantee conditions and can be more delicate in the sense that realising changes atomically can be difficult in most programming languages.

In many such cases, there is a fascinating interplay with data reification.[8] There is an example in [Owi75] where two (or more) parallel processes are searching for the least index (i) to an array A for which a predicate $p(A(i))$ holds. A compositional development of the same example is given in [Jon81]. A key point in the development there states that both processes rely on, and guarantee, that the lowest index t where such a value has been found monotonically decreases ($t \leq \overleftarrow{t}$). If t were itself a shared variable, even an assignment like $t \leftarrow v$ would not safely decrease t in the case where $v \leq t$ at the start of execution because interference could reduce t to a value less than v. What is actually done in [Jon81] is to reify t into two variables v_1 and v_2 and use the retrieve function

$$t = min(v_1, v_2)$$

(The ith process can write v_i; either process can read both v_j.) The ith process can now reduce t by changing v_i without fear of interference and *without atomicity assumptions* on things like assignment statements.

This observation appears to be important because it is echoed in other examples — some of which are considerably more complicated. In fact, the extremely subtle "Asynchronous Communication Mechanisms" (ACM) like Simpson's "four slot" algorithm can be understood in this way. The essence of the problem with ACMs is to have a shared variable into which processes can both read and write without ever waiting on any locks. If it is not assumed that read and write of whole variables is atomic, this becomes very difficult. Hugo Simpson has shown that –under the assumption only of atomic update of some 1-bit indicators– this can be achieved with 4 slots for values (see [Sim97] and back references therein). The paper [HP02] develops this example using data reification (the question of "metastability" of the bit variables is tackled in [PHA04]).

[8] Although the examples are from this author's own papers, the link with atomicity has only recently become evident.

$Rem(i)$
ext $s\colon \mathbb{N}$-**set**
rely $s \subseteq \overleftarrow{s}$
guar $\overleftarrow{s} - s \subseteq mults(i) \wedge s \subseteq \overleftarrow{s}$
post $mults(i) \cap s = \{\,\}$

Fig. 2. Parallel implementation of *Sieve*

The same link between data reification and interference can be seen in the development of a concurrent "Sieve of Eratosthenes". The aim is to arrange that some shared set s is set to exactly the set of primes up to some value n. A sequential sieve would first remove all multiples (2 and above) of 2, then find the next lowest value in s and remove its multiples and so on up to the square root of n. The idea of a parallel version is to use a family of parallel processes $Rem(i)$ –one for each index between 2 and the square root of n– and make them responsible for removing the multiples

$$mults\colon \mathbb{N} \to \mathbb{N}\text{-}\mathbf{set}$$

of their index. The specification of *Rem* is as in Figure 2. Full developments of this example can be found in several of the cited papers. [9] Here, the interest is on the reification of the set s. Even if one had a programming language which supported variables of type set, an assignment like

$$s := s - \{i * n\}$$

would not satisfy the guarantee condition in the presence of interference (e.g. having accessed s to compute the set difference, another process could remove some composite j which would then be re-inserted by the assignment to s). Again, it is the choice of a data representation for s which makes things work: s is represented by a vector of bits; an element of the set can be removed by setting the corresponding bit to **false**; this operation is assumed to be atomic.

It appears then that compositional development of concurrent programs using rely/guarantee conditions –at least sometimes– depends on the choice of representations where an appropriate atomic operation is available. So, while the rely/guarantee idea does not commit to a specific level of granularity (nor does it need "Reynolds' rule"), there is certainly a danger that a development could hit a dead end if appropriate atomic operators cannot be found. This

[9] It is a super illustration of the usefulness of rely/guarantee conditions and has become our equivalent of the *Stack* example for property-oriented specifications of abstract data types.

is not a reason to doubt the usefulness of a development method: no development method can be expected to avoid all false moves; the formalism is intended to check individual steps rather than guarantee that there will never be dead ends. In the *Sieve* example, the trick with a "characteristic function" of a set is well known and the development is well studied. This leaves open the question of whether dead ends are harder to spot in this sort of development of concurrent programs. A more "experimental" exploration of a novel algorithm using rely/guarantee conditions is given in [CJ00].

4 Splitting atoms safely

If a development method is to be found for concurrent programs, it is worth looking back at data reification and operation decomposition (under which heading, for now, development of concurrent programs using rely/guarantee conditions is included). Both of these approaches are compositional in a useful sense. From a short specification, key early design decisions can be recorded and justified in the knowledge that further, more detailed, steps will not invalidate the early decisions. Development processes, however formal, which create a whole program which is then subjected to some final "check" leave the danger of massive "scrap and rework". Notice that this applies whether the *post facto* check is testing, model-checking or even proof. This is precisely the problem with Owicki's final "interference freedom" proof: having completely developed separate programs and shown they satisfy their individual post-conditions, they might have to be discarded because a statement in one interferes with a proof step in another.

A further bonus of using both data reification and operation decomposition is that they leave behind useful design abstractions as documentation. So we would hope to find methods for developing concurrent programs which are also compositional.

What is being suggested here is to design as though things will be atomic and then to allow steps of the processes to overlap. This approach might be called "atomicity refinement". There are obviously cases where it is trivial and cases where it is invalid. Almost all of our programs are now run on operating systems which share the resources of the hardware; even the physical memory itself is shared by paging schemes; but it is the responsibility of the operating system to keep such programs completely independent from interfering with each other. Mostly, they do this successfully.

Atomicity can be relaxed where there is no danger of interference; the design has been clarified by using the "fiction of atomicity".

On the other hand, it is not valid to relax the (unrealistic) assumption of atomicity on assignments in the sequence of statements in Section 2 for computing factorial 5. So another way of describing the sought development approach is "splitting atoms safely".

The idea of "atomicity refinement" underlies the notion of database transactions. It is straightforward to write an SOS for transactions being executed in arbitrary order but separately (i.e. atomically). Any database system actually executes overlapped transactions because the amount of computation is small relative to the overall duration of a transaction. The whole literature on optimistic/pessimistic approaches and the various schemes like two-phase locking are about ensuring that no transaction can detect the concurrent transactions.

Once one is aware of the power of the "fiction of atomicity", it becomes clear that it is a very useful way of understanding the intention of a large range of concurrency ideas. Pipelining is one hardware example but, more generally, the design of asynchronous circuits is about (safely) overlapping sub-operations. Another example is (distributed) caching. In spite of the fact that this abstraction appears so useful, there is no general way of arguing that the subsequent splitting of atoms is safe. The database literature [B$^+$87,L$^+$94,GR93,WV01] is interesting but the methods exhibited there are honed to their specific application.

A worked example from another domain is worth considering as an existence proof for a general method. The example of atomicity refinement in [Jon96] satisfies the desiderata listed in Section 2. It can be applied to the development of the specification given in Figure 1. This example is simpler than examples in earlier papers; the point here is to draw out the lessons. In particular some new comments about observability give pointers to the further work sketched in the last section of this paper.

Before tackling the example itself, it is worth sketching the genesis of the move to concurrent object-based languages because it again links to the properties desired for methods. After Ketil Stølen had completed his PhD [Stø90], he and the current author worked on a joint paper which would have demonstrated the extended rely/guarantee method that he had developed to handle progress arguments (an issue which earlier work had failed to address). The paper was never finished because of the realization that the rely/guarantee proofs became too heavy if one was forced to use their full weight for every programming construct. What was needed was a clear way to show where interference could not occur and to limit the use of interference proofs to the remaining [10] portions of the program.

[10] It should be observed that a bad decomposition will always be very hard to prove correct; this is certainly true for overly cunning concurrent splits; one is reminded of Plato's note (in Phaedrus) "The separation of the Idea into parts, by dividing it at

```
Priq class
vars v: ℕ, next: private ref(Priq) ← nil;
enq(x: ℕ) method
  begin
    return;
    if next = nil then (v ← x; next ← new(Priq))
    else if v < x then enq(x)
    else (enq(v); v ← x)
    end
...
end Priq
```

Fig. 3. Priority queue program

Object-oriented languages in general –and POOL [Ame89] in particular– provided the inspiration for the next step. Class-based OOLs offer the program designer ways to control the degree of interference: local "instance" variables are only accessible via local methods; the degree to which references (i.e. pointers to instances) are shared between objects governs the degree of interference. In [Jon93a,Jon93b,Jon96], a language ($\pi o \beta \lambda$) was explored which required that only one method was active in any object instance at one point in time. Coupled with the identification of **private** references which (among other restrictions) could not be copied, an intermediate class of interference control was described where "islands" were immune from interference and could execute in parallel with other processes.

For the priority queue specified in Figure 1, it is a straightforward step of data reification to represent the set by an ordered sequence of values. It is also not difficult to develop (by sequential operation decomposition) a ($\pi o \beta \lambda$) program which sequentially inserts an *enq*ued value into its correct position.

The idea now is to introduce concurrency into such a sequential program. Figure 3 shows the **return** statement at the head of the *enq* method. In contrast to the sequential program, this version of *enq* would release its client from the *rendezvous* immediately and the client could proceed in parallel with the activity within the chained sequence of *Priq* elements. Furthermore, as soon as a call is made to *enq* in the *next* element, activity can ripple down the sequence in parallel. (It is also possible to make *deq* calls interleave.)

Unfortunately, it proved non-trivial to justify the equivalence rules of $\pi o \beta \lambda$. (In addition to commuting **return** statements as above, there was a rule for "delegating" return values.) Attempts include [Wal93,Wal94,LW95,Jon94,HJ96]. Davide Sangiorgi [San99] used "barbed bi-simulation" and introduced the idea of "uniformly receptive processes". With hindsight, it can be seen that what

---

the joints, as nature directs, not breaking any limb in half as a bad carver might".

dogged these proofs was an extremely subtle question of observability (or context). It is *not* true that the behaviour of the sequential and concurrent *enq* operations is identical. Of course, we have already observed that operation decomposition introduces extra steps and that data reification changes internal (state) representations. But in both of these cases, there is a clear notion of what is observable "at the interface" (via external types).[11] The situation with *enq* is more complicated. A sufficiently rich observation language *could* observe that in

$$enq(2); enq(3); enq(1)$$

the *completion* of $enq(1)$ can precede that of $enq(3)$ in the concurrent queue but not in the sequential queue. The point is that $\pi o\beta\lambda$ is (deliberately) expressively too weak to be able to detect such differences.

The transformational introduction of concurrency by the use of $\pi o\beta\lambda$'s equivalence rules comes close to meeting the desiderata of development methods given in Section 2. The issue might be whether the $\pi o\beta\lambda$ approach is sufficiently general. One way to move forward would be to embrace (concurrent) object-oriented languages and search for more equivalences. The final section of this paper draws more general conclusions from this specific approach in the acknowledgement that a set of such transformations might not be general enough.

5 Further work

A beginning has been made on formalising notions of "atomicity refinement"; but much remains to be done.

The ubiquity of informal "atomicity refinement" is both challenging and a pointer to considerable intellectual leverage. An interesting test case is to take Dijkstra's example of on-the-fly garbage collection from [Dij82] and tackle it as a refinement from an atomic GC algorithm: the challenge would be to use only rules from a general method.

There are of course, many concurrent programs which do not fit the mould of atomicity refinement. In programs which use locking, the termination of one process might rely on progress in the other.

One non-issue should be recognised: just switching to a Process Algebra is

[11] Even here, there is delicacy in the details of the observation language — cf. [Nip86,Nip87].

not a solution to "interference". It is argued above that "the essence of concurrency is interference": communication based concurrency just has to deal with interfering communication. This is obvious once one has programmed a shared variable in the pi-calculus: one can then program up examples which precisely mimic the interference on shared variables just by having two or more processes interact with the process which models the variable.

In fact, reverting to the topic of what is meant by "state", one could say that the most abstract evidence of state is that a system will behave differently at two different times when presented with the same inputs. One realisation of this abstract notion of state as affecting behaviour is to view a process algebraic term as having an associated "program counter"; changes in behaviour come about because the process is at a different point in its execution. This is not, of course, the normal terminology of the process algebra community but it is important to understand what is –and is not– really different from shared variable concurrency. This does not, of course, argue against the insight that can be gained from process algebra.[12] Indeed, it could be that the expressive power of trace assertions is weak enough to win decidability (and thus move from proof to "types").

The $\pi o \beta \lambda$ example provides two interesting pointers to future work.

- Arguments about "refining atomicity" are likely to depend rather intimately on notions of "context".
- The use of equivalences in a design language will have to be carefully checked against the eventual implementation language.

Obviously, it is important to recognise other related contributions and to understand the extent to which they might have already solved sub-problems that relate to atomicity refinement. The work on refinement calculi [CM88,Mor90,BvW98] has led to the notion of "Action Systems". Furthermore, the rich sets of database transaction implementation techniques need to be investigated: $\pi o \beta \lambda$'s existing transformations show how closely these connect. It also appears likely that in some cases rely/guarantee conditions will be needed to show valid "atom splitting". In spite of what is said above about process algebras not of themselves solving the problem of interference, there are related methods being investigated there: for example, [Bur04] takes a careful look at contexts for refinements.

[12] The study of the fundamental nature of communication is one of the most important on-going challenges in Computing Science and can be compared to the early clarification of the the notion of computation. The largest contribution to our understanding of communication has come from the study of process algebras or calculi where the notion of shared state is deprecated.

The second example of threads of concurrent assignments in Section 2

$$(x \leftarrow x + 1) \,||\, (x \leftarrow x * 2)$$

was not randomly chosen: it is interesting that the non-determinacy (even with the assumption of assignment atomicity) comes from the fact that the operators in the two assignments do not commute. Some authors have tried to base methods broadly like "atomicity refinement" on commutative sub-operations. This author does not see how this can be made to yield a compositional method.

It will be interesting to see whether embedding SOS definitions in "logical frames" offers any purchase on the proof of the methods themselves. If one views $\xrightarrow{s}$ as a relation, it can be embedded directly into a proof tool. The "Plotkin rules" are used directly as the inference rules about programs and the logical frame of the proof tool is enriched to reason about the logic of $\xrightarrow{s}$. (Tobias Nipkow and colleagues have done this for definitions of significant subsets of Java [Nip04]; the idea is also discussed in [Jon03] but appears to originate with [CM92].) It is also possible to extend this to stepwise development by introducing assumptions about part of a program which is yet to be developed. [13] More importantly for the purposes here, the observation that one can easily reason directly about the SOS rules of a language removes from us any temptation to search for complete sets of rules which have anyway proved elusive for most languages. [14] In particular, if there is a need to talk about non-interference conditions like Reynolds' $S\#T$, one can do this directly in terms of the semantics where the "environment" is a direct coding of variable sharing.

[13] The question can then be asked as to the role of an "axiomatic semantics". Is the proof rule for, say, **while** statements to be viewed as a proof heuristic for the extended logical frame? It must firstly be observed that axiomatic descriptions of languages are pedagogically invaluable. Where they can be used on idealised design languages, the idea of "verification condition generators" is useful in formal development support systems.

[14] An understanding of this link could have avoided the opaque soundness proofs in [Jon81].

Acknowledgements

This paper –like the others in this volume– is dedicated to Prof John Reynolds. [15] It has been a pleasure to write in honour of someone I respect so much and I was delighted to have been asked to make a contribution. Apart from having known John for many years (and having met often particularly through IFIP's WG 2.3), our interests in languages and their semantics overlap. Add to this the synchronicity of our birthdays being on the same day of the year, and it was just too tempting an invitation.

My evolving thoughts on semantics were influenced both by the fact that I am teaching a course on SOS and an exchange of communications with Gordon Plotkin when he was writing [Plo04b] and I [Jon03]. Turning to this paper, it has benefited from the discussion with Jon Burton whose recent thesis [Bur04] tackles the questions reviewed above from a process algebra viewpoint. The Schloß Dagstuhl event on "Atomicity" organised with David Lomet, Sascha Romanovsky and Gerhard Weikum provided further insight (see [JLRW05] and other papers in the same edition of JUCS). The range of backgrounds of the participants made the discussions both challenging and stimulating.

The author is extremely grateful for the comments from the referees who helped make the objectives of this paper clearer. Similarly, much clarification resulted from discussions in London with Peter O'Hearn and Steve Brookes.

For funding to support my research, I am grateful to the (UK) EPSRC funding for the Dependability IRC (see www.dirc.org.uk).

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[Acz82] P. Aczel. A note on program verification. (private communication) Manuscript, Manchester, January 1982.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4), 1989.

[15] There has been a rather long gap between the writing and printing of this paper. Many developments have occurred including new work with Joey Coleman on rely/guarantee conditions and explorations in Cambridge, London and Newcastle universities on combining rely/guarantee thinking with separation logic. Reflecting all of this would have resulted in a different paper (which might be written after some separate specific publications). Again, John Reynolds has been a catalyst.

[B+87] P. A. Bernstein et al. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.

[Bue00] Martin Buechi. *Safe Language Mechanisms for Modularization and Concurrency*. PhD thesis, Turku, 2000.

[Bur04] J. Burton. *The Theory and Practice of Refinement-After-Hiding*. PhD thesis, University of Newcastle upon Tyne, 2004.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A systematic Introduction*. Springer Verlag, 1998.

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In G. Plotkin, C. Stirling, and M. 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.

[CM92] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, August 1992.

[Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY*. PhD thesis, Louvain-la-Neuve, June 1994.

[Dij82] Edsger W Dijkstra. On making solutions more and more fine-grained. In *Selected Writings on Computing: A Personal Perspective*, pages 292–307. Springer-Verlag, 1982. (Orignially EWD622, written 26 May 1977).

[Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.

[dR01] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[dRE99] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, 1999.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[HJ96] Steve J. Hodges and Cliff B. Jones. Non-interference properties of a concurrent object-based language: Proofs based on an operational semantics. In Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jörg Schek, editors, *Oject Orientation with Parallelism and Persistence*, pages 1–22. Kluwer Academic Publishers, 1996.

[HP02] N. Henderson and S. E. Paynter. The formal classification and verification of Simpson's 4-slot asynchronous communication mechanism. In L.-H. Eriksson and P.A Lindsay, editors, *FME 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 350–369. Springer Verlag, 2002.

[JLRW05] C. B. Jones, D. Lomet, A. Romanovsky, and G. Weikum. The atomicity manifesto, 2005.

[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.

[Jon83a] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon83b] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.

[Jon93a] C. B. Jones. Constraining interference in an object-based design method. In M-C. Gaudel and J-P. Jouannaud, editors, *TAPSOFT'93*, volume 668 of *Lecture Notes in Computer Science*, pages 136–150. Springer-Verlag, 1993.

[Jon93b] C. B. Jones. Reasoning about interference in an object-based design method. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93*, volume 670 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1993.

[Jon94] C. B. Jones. Process algebra arguments about an object-based design notation. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, chapter 14. Prentice-Hall, 1994.

[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.

[Jon03] Cliff B. Jones. Operational semantics: concepts and their expression. *Information Processing Letters*, 88(1-2):27–32, 2003.

[L+94] Nancy Lynch et al. *Atomic Transactions*. MIT Press, 1994.

[LW95] X. Liu and D. Walker. Confluence of processes and systems of objects. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzback, editors, *TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*, pages 217–231. Springer-Verlag, 1995.

[McC66] J. McCarthy. A formal description of a subset of ALGOL. In *[Ste66]*, pages 1–12, 1966.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[Nip86] T. Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22:629–661, 1986.

[Nip87] T. Nipkow. *Behavioural Implementation Concepts for Nondeterministic Data Types*. PhD thesis, University of Manchester, May 1987.

[Nip04] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. Manuscript, Munich, 2004.

[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.

[PHA04] S. E. Paynter, N. Henderson, and J. M. Armstrong. Ramifications of meta-stability in bit variables explored via Simpson's 4-slot mechanism. *Formal Aspects of Computing*, 16(4):332–351, 2004.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[Plo04a] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July–December 2004.

[Plo04b] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July–December 2004.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Proceedings of fifth POPL*, pages 39–46. ACM, 1978.

[Rey81] J. C. Reynolds. *The Craft of Programming*. Prentice Hall International, 1981.

[Rey89] John C. Reynolds. Syntactic control of interference: Part 2. In *Proceedings of 16th ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722. Springer-Verlag, 1989.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th LICS*, pages 55–74. IEEE, 2002.

[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.

[Sim97] H. R. Simpson. New algorithms for asynchronous communication. *IEE, Proceedings of Computer Digital Technology*, 144(4):227–231, 1997.

[Ste66] T. B. Steel. *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures.* PhD thesis, Manchester University, 1990. Available as UMCS-91-1-1.

[Wal93] D. Walker. Process calculus and parallel object-oriented programming languages. In *In T. Casavant (ed), Parallel Computers: Theory and Practice.* Computer Society Press, 1993.

[Wal94] D. Walker. Algebraic proofs of properties of objects, 1994. Proceedings of ESOP'94.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., 2001.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming.* PhD thesis, Oxford University, 1992.