Journal of Universal Computer Science, vol. 11, no. 5 (2005), 661-686
submitted: 30/11/04, accepted: 31/1/05, appeared: 28/5/05 © J.UCS

Investigating Atomicity and Observability

Jonathan Burton
(University of Newcastle upon Tyne, UK
j.i.burton@ncl.ac.uk)

Cliff B. Jones
(University of Newcastle upon Tyne, UK
cliff.jones@ncl.ac.uk)

Abstract: Using the fiction of atomicity as a design abstraction and then refining atomicity as we develop an implementation is widely used in areas of concurrent computing such as database systems and transaction processing. In each of these and similar areas, associated notions of correctness are used in order to show that a particular implementation artefact which exhibits concurrency is correct in some sense with respect to a (possibly notional) description which executes with a greater degree of sequentiality. Of crucial importance in the proof and deployment of such notions of correctness is the issue of observability: i.e. in what broad sense do (human or computer) users of a particular implementation artefact observe the effects of its executions. For example, if a human user is allowed to observe directly the execution of a particular concurrent component then he or she will be able to detect the fact of concurrent — and so non-atomic — execution. In general, however, the notion of observability is treated implicitly or not at all. In this paper, we make it explicit and look at the issue of exploring more fully the connections between atomicity and observability. The ultimate aim of this consideration is to work towards constructing a more general framework for (software or hardware) development by refining atomicity.

Key Words: observability, refinement of atomicity, formal development method
Category: F.3.1 Specifying and Verifying and Reasoning About Programs

1 Introduction

Many different areas of computing, such as database systems and transaction processing, use the fiction of atomicity as a design abstraction.[1] Taking such an approach allows developers to design, reason and program in terms of sequential systems and their executions; the complexity inherent in the use of concurrency is masked by the use of notions which relate sequential to concurrent executions and by mechanisms — implicit or otherwise — which guarantee that only concurrent executions which may be so related are allowed. For example, the notion of *serializability* (see [B$^+$87, WV01]) used in database systems allows

[1] In this paper, by the term "atomic" or by the property of "atomicity", we mean the isolation property described by the I of ACID in the database literature (see, for example, [WV01]). We do not mean the all-or-nothing property which is implied by the A of ACID.

one to relate a concurrent execution of various transactions to a sequential execution in some order of the same set of transactions. Similar is the notion of *linearizability* ([HW90]), which lets a programmer work with a system built of a set of concurrent objects as if those objects could only execute sequentially. We shall refer to approaches which relate concurrent implementations to (more) sequential specifications in some way as allowing *refinement* or *relaxation* of atomicity.

It is usually the case that notions of refinement of atomicity are justified — whether formally or informally — in the following way: there does not exist any context[2] in which the (correct) concurrent component can be deployed such that the difference between it and its sequential counterpart can be detected.[3] In other words, any system we can construct using the concurrent component will have the same behaviour in some sense as if the sequential component were being used in its place. Among other things, this means that we may reason about the correctness of the simpler system which uses the sequential component while that reasoning will still be valid for the system built using the concurrent component.

However, it may be possible to construct certain contexts which *are* able to distinguish between the concurrent and sequential components. Hence, it may be necessary to impose certain restrictions, thereby limiting the set of valid contexts: as a result, it is only valid contexts that are unable to distinguish between concurrent and sequential components related using some notion of atomicity refinement, while arbitrary contexts may be able to tell the difference. For example, linearizability imposes the restriction that individual user processes which interact with (possibly concurrent) data objects should be sequential: this means that they are unable to detect the fact of concurrent execution on the part of those data objects. In general, the issue of the restrictions to be placed on contexts is not properly explored: for example, how light in any particular case might those restrictions be made and how exactly do those restrictions connect to the notion of correctness used to relate concurrent to sequential components. Such an exploration is necessary if we are to place atomicity refinement on a more general footing and to apply it more widely to programs and systems.

Because of the usefulness of the abstraction provided by atomicity, we would like to add refinement of atomicity to the collection of formal compositional development methods which can be used to justify the correctness of designs and implementations with respect to abstract specifications. In the sequential domain, VDM [Jon90] uses

– data reification: a program is specified in terms of abstract data objects

[2] A context is simply a process or network with a "hole" into which another process may be placed.

[3] There are, of course, certain notions where this justification is not used. In such cases, the approach is usually not compositional and so less appropriate for use as a development method.

which match the problem; these objects are then reified into representations which match the implementation;

– operation decomposition: sequential program combinators are introduced in stages to satisfy pre/post-condition specifications

Research on rely guarantee-conditions [Jon81, Jon83, Stø90, Col94] extended operation decomposition to cover interfering programs. This research has blossomed in recent years and is seen as a successful way of reasoning about complex uses of interference. But the proofs are — for obvious reasons — more difficult than those for standard sequential programs. Hence, we would like to maximise the use of sequential program development methods (data reification and operation decomposition) and to introduce concurrency late in the development by using some notion of atomicity refinement. As indicated above, such an approach is not new and is practised in a number of different domains but further work is needed to allow its wider application.

As a first step in that further work, we introduce the notion of *observability* as a conceptual framework within which the issue of restrictions on contexts may be discussed. At a fundamental level, observability simply means what one process can observe of (and thereby "know" about) another process, system or its component parts: it implies a capability to garner information from or on those entities. For example, linearizability imposes the (implicit) restriction that user processes which access concurrent data objects are not allowed to communicate directly with each other: that is, all communication must go via the data objects themselves. Such a restriction may be framed in terms of observability: we restrict what each user process can observe of the other user processes and, indeed, forbid all such direct observation. In general, we must disallow our (valid) contexts from observing "directly" the fact of concurrent execution otherwise they will be able to detect the difference between concurrent and sequential component processes (what is meant by "directly" is made clearer in Section 2.5).

The concept of observability also plays a useful role when we focus on the use of refinement of atomicity as a development method. Part of the role of a development method is to take "hard" proofs and do them once-and-for-all, leaving any user of the method to establish only simpler facts. By the discussion in Section 2.6, it may be possible to define the set of valid contexts using certain *syntactic* constraints, the checking of which is easier in general than that of semantic conditions (which may in fact be undecidable in the general case). This issue is also important when we attempt to show the correctness of a particular concurrent component by relating its behaviours to those of its (more) sequential counterpart. In essence, that a particular concurrent component may be shown to be correct in this sense requires that it does not suffer from (intolerable) interference. In practice, certain constraints can be imposed on the component itself which limit or rule out the potential for interference and so make it much easier to show correctness. In particular, Section 4 refers to work which uses an object-oriented language to restrict the interface of component processes — this means that arbitrary and arbitrarily interleaved changes to data variables within those components are ruled out — and thereby to restrict what a context may observe of those components. In addition, restrictions are placed on what constituent entities within the concurrent component itself can observe of each other, by using restrictions on pointers or references: this, too, restricts interference in a way that allows concurrency to be introduced safely — i.e. in a behaviour-preserving way — without requiring extensive proofs to show correctness.

The remainder of the paper is organised as follows. Section 2 sketches informally a framework within which the notion of observation is identified with the concept of communication or interaction between processes; this framework is then used to explore restrictions to be placed on contexts. Section 3 looks at some existing notions which allow refinement of atomicity and considers them in terms of the discussion from Section 2. Section 4 then looks at an example of transforming a sequential into a concurrent process and at the issue of restricting the possibility for interference by restricting observability.

2 A framework for considering observability and refinement of atomicity

In this section, we introduce a useful conceptual framework: this will allow us to unify different notions of observability into the single notion of communication or interaction. Using this framework, we then look at the issue of restrictions to be placed on contexts. Note, however, that the presentation of this framework is informal in nature and it is intended simply as an aid to discussion and as a basis for further work.

In the general case, restrictions on contexts disallow the detection of difference between concurrent and sequential component processes simply on the basis that the former can engage in concurrent executions while the latter cannot. Difference between such components may also be detected due to the communication of data values in the concurrent case which are not possible in the sequential case. That the data values returned are correct will usually be dealt with by the correctness condition which relates concurrent behaviours to sequential behaviours in the component processes under consideration; issues relating to the component itself and the data values it transmits are considered in Section 4.

2.1 Considering observability

At first glance, the computational artefacts which make up the types of system in which we are interested are many and varied: they range from passive data objects to active processes interacting with those objects, from data variables to code of individual programs to whole systems. Moreover, "observation" could be taken to mean reading of a variable, the receipt of a message, receipt of data from a remote procedure call, observation of the execution of an event and so on. In order to manage this complexity, we wish to regard all possible artefacts under consideration as instances of the same type of entity and similarly to regard all observation as instances of the same type of act. Hence, we adopt the approach taken in the process algebraic framework and exemplified by [Mil89] (note that our use of this approach here will be informal in general). In this approach, all computational artefacts are regarded as agents or processes — the notion of process is recursive, meaning that processes may be composed of other processes and so on — and communication is made central to the model of computation employed. Note that (the act of) communication is *synchronous* and so is experienced simultaneously by both participants in that communication. In essence, two processes which are ready to execute the same event may communicate by simultaneously executing that event. As two processes communicate, they are said to *synchronize* on the event involved. Note, of course, that processes may communicate with each other only if they are executing in parallel: thus, communication takes place between concurrently executing processes. Processes compute either by communicating with other processes or by indicating that they are ready to communicate a certain event or events. For example, we could represent as processes both a data variable and a program which read that variable. The act of reading that variable would then be represented as a communication between those two processes, namely the simultaneous execution in both processes of the event representing the transfer of the value stored by the variable.

The notion of observability is central to the idea of communication. For processes can only synchronize on events which are visible and so observable and for one process to observe something of another means that it must communicate with that process. Hence, we may identify the notion of observability with that of communication or interaction. In other words, a process may observe something of another process only if there is a communication link between the two; moreover, the pattern of behaviour offered by both processes over that communication link will determine the nature of the observations which may be made (remember that the two processes have to synchronize if communication is to be effected).

2.2 Contexts, semantics and full abstraction

In general, process algebras take a behavioural approach to semantics and ascribe meaning to processes on the basis of the communications in which they can engage: in other words, the semantics abstracts from internal behaviour and is only interested in externally observable behaviour. In order to make the discussion more concrete in the remainder of this section, we assume that processes are given a semantics in the *traces model*. This is one of the simplest — and least discriminating — semantics which may be given to a process: it regards the meaning of that process to be the set of finite sequences of observable events which it can perform. The way in which process algebraic semantics are usually presented and validated — at an abstract level — will be useful to us in the remainder of this paper as it gives a framework within which may be considered some of the issues raised when looking at atomicity refinement. We identify two entities, a context, C, and a process, P. By context, we simply mean a process which has a "hole" into which another process may be placed and denote by $C[P]$ the process which results from placing the process P in the context C. For our purposes here, we may assume that the context consists of a process which is to communicate with P (i.e. that process and P execute concurrently); moreover, it may be the case that communication between the process and P is made internal by the context, so that it cannot be observed by any further user of $C[P]$. A semantics is then defined which allows us to calculate the meaning of any process: we denote as $[\![P]\!]$ the semantic meaning of P. In order for this semantics to be acceptable, there are two main requirements imposed. Firstly, we require that if $[\![P]\!] = [\![P']\!]$, then $[\![C[P]]\!] = [\![C[P']]\!]$ for any context C which may be constructed in the relevant process language.[4] This is the *congruence* property, which requires that if the semantics cannot distinguish two individual processes in isolation, then it cannot distinguish them when placed in any context. It allows us to calculate the semantics of processes in a compositional manner. Secondly, we require that if $[\![C[P]]\!] = [\![C[P']]\!]$ then $[\![P]\!] = [\![P']\!]$. In other words, if there does not exist any context which allows the semantics to distinguish P and P' when placed in it, then they should be identified by the semantics. A semantics which possesses both of these properties is said to be *fully abstract* with respect to the language in which processes may be described.

2.3 A semantics for atomicity refinement

On the basis of the above discussion, it is clear what the challenges are if one is to achieve a general development method based around atomicity refinement.

[4] We assume that our notion of correctness requires processes to be semantically equivalent; however, the discussion here is still valid if we use instead a notion of *refinement*, where processes are ordered and one process implements another if it is above it in the refinement ordering.

Assume that *Seq* is a sequential process and *Conc* is a process derived from *Seq* but which exhibits a greater degree of concurrency. In other words, we have experienced atomicity refinement in the move from *Seq* to *Conc*. For the purposes of the remainder of this section, we assume that *Seq* and *Conc* are processes with a procedural interface and so communicate with the context C using procedures which are called by C. Moreover, we assume that *Seq* and *Conc* provide exactly a procedure to write data into the relevant process and a procedure to read data from it (in both cases, all internal processing is hidden); it is also implicit that *Seq* and *Conc* are passive entities which respond to requests from other processes. (Note that [Bur05] contains a concrete example of these types of process.) We assume that the context C is fixed and so both *Seq* and *Conc* are to be placed in C. In the general case, it would be possible to explicitly relax atomicity in C and so to introduce a new context; indeed, such a thing can be dealt with using some of the approaches described in Section 3.2. The assumptions detailed here help ease the presentation and make the following discussion more concrete, while not restricting its general relevance.

Assume that CW denotes a call to the write procedure, RW a return from the write procedure, CR a call to the read procedure and RR a return from the read procedure. In any sequence of behaviours which is possible for *Seq*, each call other than possibly the last in the sequence must be immediately followed by a matching return. Similarly, a return from a particular procedure must be immediately preceded by the corresponding procedure call. Thus, execution sequences such as $\langle CW, RW, CR, RR, CR\rangle$ or $\langle CW, RW, CW, RW\rangle$ are possible for *Seq*. However, *Conc* may exhibit a wider range of behaviours, since it allows reads and writes to proceed concurrently: for example, $\langle CW, CR, RR, RW\rangle$ or $\langle CW, CR, RW, CW, RW, RR\rangle$ are possible executions of *Conc* which are not possible for *Seq*. Hence, evidence of the atomicity refinement used to derive *Conc* from *Seq* will manifest itself in the visible behaviours which the processes may perform and so the processes will be distinguished by the semantics being used. Moreover, it will be possible to define contexts in general into which *Seq* and *Conc* may be placed such that evidence of that difference is still apparent in the semantics of context plus process. For example, for any particular sequence of actions which *Conc* can perform as evidence of increased concurrency and so which is not possible for *Seq*, the context could perform that sequence followed by some distinguished event. That distinguished event will be possible only for the context plus *Conc* and not for the context plus *Seq*.

However, the use of atomicity refinement in practice shows that, provided the concurrent behaviours of *Conc* may be related to the corresponding sequential behaviours of *Seq* in some way, then there will be a large class of contexts which are *unable* to distinguish between the two processes. Hence, we need a semantic framework which reflects this fact and so — if attempting to define such a semantics in practice — would take the approach which is taken in general by all approaches based around atomicity refinement. Firstly, we would make less discriminating the semantics which is used to assign meaning to *Seq* and *Conc* so that processes exhibiting additional concurrency could be related to sequential processes. Secondly, we would restrict the class of contexts into which processes derived using atomicity refinement could be placed (if it were possible to refine the atomicity of the context, then these restrictions would probably be implicit in the correctness condition to be met by the concurrent context). Finally, we might change the semantics used to assign meaning to context plus process, although we would prefer to reclaim a standard semantics for that composition.[5] In making less discriminating the semantics which is used to relate *Seq* and *Conc*, we would essentially reduce its power of observation: we would let it see less of the difference between the two processes. However, if such a relaxed semantic view is to be useful in practice, it must still be the case that $[\![C[\mathit{Seq}]]\!] = [\![C[\mathit{Conc}]]\!]$ if $[\![\mathit{Seq}]\!] = [\![\mathit{Conc}]\!]$ (note that the semantic function used here and indicated by $[\![.]\!]$ may be different depending on whether it is applied to a component process or to context plus process). In order for this property to hold, any restriction of the observational power of the semantics must be reflected in a restriction on the set of contexts which are to be regarded as valid. At the very least — if we are to retain a standard semantic view of context plus process — the context would have to make internal the communication which occurs between itself and the process *Seq* or *Conc*. In other words, we would restrict what any user of $C[\mathit{Seq}]$ or $C[\mathit{Conc}]$ may observe of that process. If this is not done then it may be possible to observe interleavings of actions in $C[\mathit{Conc}]$ which are not possible for $C[\mathit{Seq}]$ since they are due to the additional concurrency exhibited by *Conc*. There are also further types of restriction which would need to be placed on contexts and which can be framed in terms of the issue of observability. These are discussed in Section 2.6 after introduction of some example processes in Section 2.4 and of another important issue in Section 2.5. (Note that, in Sections 2.4, 2.5 and 2.6, we assume the semantics of context plus process is given using the standard traces model; moreover, we assume that all communication between the context and component process is hidden when the latter is placed in the former.)

2.4 Example processes and contexts

Figure 1 contains some processes to be used to construct example contexts which will be useful in Sections 2.5 and 2.6. Process $P1$ calls the write procedure,

[5] Usually, a process algebraic approach uses a uniform semantics to assign meaning both to component processes and to context plus process. Indeed, they are all processes and not to be distinguished due to the manner of their construction. However, certain existing notions of atomicity refinement use a different means to give a semantics to component processes than they do to give a semantics to context plus component process (under such an approach, the latter would be given a standard semantics). We shall see evidence of this in Section 3.2.

- $P1 = CW; RW; TERM1$
- $P2 = CR; RR; TERM2$
- $P3 = CW; CR; ERROR$
- $P4 = CW; ERROR$
- $P5 = CR; ERROR$

Figure 1: Example processes

receives a return from that procedure and then indicates that it has terminated; $P2$ calls the read procedure, waits for a return and then indicates that it has terminated. $P3$ calls the write procedure, calls the read procedure and then flags an error. $P4$ calls the write procedure and then flags an error; $P5$ calls the read procedure and then flags an error.

If we construct a context by placing $P1$ and $P2$ in parallel then, when *Seq* is placed in that context and thereby all communication hidden between the context and *Seq*, the resulting process will perform (visibly) $TERM1$ and $TERM2$ in either order. If we place *Conc* in the same context then the resulting process will perform exactly the same traces.

If we take $P3$ as a context then, when *Seq* is placed in that context and thereby all communication hidden between the context and *Seq*, no visible events will be possible since *Seq* cannot accept a call to the read procedure while an execution of the write procedure is still ongoing. However, if we place *Conc* in the same context then the resulting process will perform the event $ERROR$, thus indicating the fact that difference has been detected between *Seq* and *Conc*.

Finally, if we construct a context by placing $P4$ and $P5$ in parallel — note that $P4$ and $P5$ would synchronize on the occurrence of $ERROR$ — then, when *Seq* is placed in that context, no visible events will be possible since *Seq* cannot accept a call to one procedure while an execution of the other procedure is still ongoing. However, if we place *Conc* in the same context then the resulting process will perform the event $ERROR$, thus indicating the fact that difference has been detected between *Seq* and *Conc*.

2.5 Computation, state and "knowledge"

Before proceeding, we consider briefly what is meant by the notion of "state" in a process algebraic framework. In such a framework, there is no explicit notion of data and so no conventional notion of state. Instead, by "state" we effectively mean a "point of control" — analogous to a program counter — within the current process definition: such a point of control determines which parts of the relevant process definition are to be executed next and so determines which actions are enabled. In fact, there will usually be multiple points of control within a single process definition, reflecting the fact that we are modelling a concurrent process. Execution of an action then moves us forward to a new point of control and so to a new state.

The most significant issue with regard to the capabilities of the context is the following. Since to observe means to communicate and the context communicates with the concurrent component process, *Conc*, then the context *as a whole* will observe and so "know about" those executions of *Conc* which are not possible for the sequential *Seq*. However, it is not entirely straightforward for the context to unify knowledge regarding the concurrent executions of *Conc* in a single location so that action can be taken on its basis. For example, consider the case that *Conc* is placed in the context from Section 2.4 where $P1$ and $P2$ are executing in parallel. In this case, $P1$ and $P2$ may both have executed a call without receiving a corresponding return — i.e. $P1$ knows a write procedure is executing and $P2$ knows a read procedure is executing — but they are unable to bring that knowledge together and so the context as a whole is unable to conclude that both procedures are executing simultaneously.

In the framework we are considering, that the context, C, could detect that *Conc* was concurrent would be indicated by the occurrence of an execution sequence (of visible actions) for $C[\mathit{Conc}]$ which was not possible for $C[\mathit{Seq}]$. In other words, the two compositions would perform some common sequence and, after that sequence had been executed, we would be able to perform a new event in $C[\mathit{Conc}]$: this is illustrated by the second and third example contexts given in Section 2.4.[6] Thus, the detection of difference equates to the enabling of events and the subsequent performance of those events. In our framework, the performance of an event is ultimately undertaken by a single sequential process — i.e. one defined without any parallel composition — or by a number of sequential processes synchronizing in parallel.[7] Hence, the detection of difference

[6] In general, looking only at traces may not allow us to detect any difference between $C[\mathit{Seq}]$ and $C[\mathit{Conc}]$ even though a semantics incorporating some sort of notion of liveness *would* allow us to detect a difference. For the purposes of the discussion here, we are interested in broad classes of context which allow us to detect the fact of concurrent execution in *Conc*. In general, those types of context will have a property which allows us, on the observation of concurrent behaviour in *Conc*, to move to some state in the context which it is not possible to reach when it is composed with *Seq*. Having reached that new state, the context must be able to do something which was not previously possible and which can be detected by the semantics. What that thing actually is — for example, whether it is the performance of an event not previously possible or the refusal to perform any event — is not of itself important provided that it is visible to the semantic notion being used.

[7] We may think of sequential processes as described here as being like a single thread of control. In other words, they need not be a distinct logical entity — such as a user process accessing a database, for example — perhaps being part of a much larger entity, and may be ephemeral.

means either that the information regarding this difference has been accumulated in a particular sequential process within the context — see, for example, the context given by $P3$ in Section 2.4 — and so is encoded in its state (i.e. in the events which are enabled in that process at that state); or that a number of different sequential processes within the context are simultaneously in a state where they are able to execute the same action, thereby indicating the fact of concurrent execution, while it was not previously possible for them all to be in those states simultaneously. (The simultaneous execution by different processes of this same action constitutes a bringing together of the knowledge possessed by those different processes.) The context given by $P4$ in parallel with $P5$ gives an example of this latter way in which difference may be detected.

Thus, if difference is to be detected then knowledge in the context that procedures are executing concurrently must be lodged in a single sequential process or must be brought together by communication. Conversely, if we can ensure that such knowledge is *not* unified in these ways then it will not be possible for the context under consideration to detect the difference between sequential and concurrent component processes: i.e. the context will be unable to detect the fact of concurrent execution. (Note we assume that the concurrent component process, *Conc*, has already been shown to be "correct" in relation to the sequential *Seq* and so the only way in which difference between the two may be detected is by the context detecting the fact of concurrent execution on the part of *Conc*.) This idea is illustrated further by the discussion in the following section.

2.6 Restrictions on contexts

One possible way in which the context could detect the difference between *Seq* and *Conc* is the following. In *Conc*, that concurrency is in evidence will usually be a result of the fact that multiple processes are executing in parallel: for example, each concurrent procedure execution would be provided by a separate process. If a single sequential process in the context can communicate with — i.e. observe — multiple individual processes in *Conc*, then it may be able to detect the fact of concurrent execution. For example, if we let a single sequential process in the context do a call to one procedure and then do a call to another procedure before receiving the return from the first, then that process can discover that *Conc* is executing concurrently ($P3$ from Section 2.4 constitutes a simple example of such a context). The execution of these two call events in the sequential process in the context will lead to a state not reached when interacting with the sequential *Seq* and so events may be enabled which were not enabled in the sequential case: hence, the fact of difference may be detected. In order to avoid this problem, we can restrict what individual sequential processes within the context can observe of the component process: in particular, we could require that each such process should not be able to observe multiple concurrent processes in the component.

Even if a restriction such as this is enforced, it might be possible for different sequential processes within the context to communicate with each other and so to detect a difference. For example, consider the case that two sequential processes within the context are each calling a procedure in *Conc*. If we allow each of those processes to do a call to the procedure and then, before receiving the corresponding return to both do the same distinguished event, they could synchronize on this event and so both of those processes could identify the fact that multiple procedures were executing concurrently (the context from Section 2.4 which is given by $P4$ in parallel with $P5$ is a very simple example of this type of context, although it does not actually engage in return events at all). Hence, it may be that we have to prevent processes within the context from communicating with — i.e. observing — each other while they are in the middle of co-operating with the component process regarding some extended operation: in this case, they should not be able to communicate with or observe each other while in the middle of executing a particular procedure. Alternatively, we could require that any communication possible during the execution of a particular procedure should also be possible both before and after that procedure execution: hence, its occurrence could not be used to transfer information regarding the execution of the procedure.

That the problem described in the last paragraph could arise depends on the fact that communication is synchronous and so processes may communicate directly with each other. If processes were forced to communicate asynchronously, then it could not arise as such. However, asynchronous communication may be modelled in our framework by requiring that processes communicate via a third-party process, perhaps representing a buffer, and then this type of problem can be replicated. In particular, having called a certain procedure in *Conc* and not yet received a return, process A could send a message via the buffer to inform process B of that fact. Process B could then return a message via another buffer indicating that it, too, had called a procedure but not yet received a return. Process A could then reply that, when it received B's message, it still had not received a return. Hence, since A had not yet received a return on the receipt of B's message then it had not received a return when B sent the message. This means B can conclude that both it and A were simultaneously engaged in procedure calls to *Conc* and so that *Conc* is executing concurrently. This sort of problem could be avoided by restricting what third-party processes and processes which call procedures may observe of each other, at least when procedures have been called and no return yet received.

Even if we avoid all of the problems considered so far, it may still be possible for the context to detect difference. For example, information regarding concurrent executions could be brought together in the context *after* those concurrent executions have ended. One possible way to do this is to use a timestamp facility, which would let us attach to each procedure call or return event the time at which it occurred. This information would let us reconstruct the executions of *Conc* and so would let us identify the fact of concurrent execution. In order to provide such a timestamp in the framework we are considering here, we would assume that processes engaging in call and return events could simultaneously observe a process representing a global clock. Thus, in order to avoid this problem we could simply forbid the observation of such a clock process should it exist.

2.7 General comments on observability

In the above discussion, we have sketched informally a possible framework within which observability may be considered and have shown how it can be used to consider and describe restrictions to be placed on contexts when carrying out atomicity refinement. Before proceeding, it is worth making some further points. The list of types of "problem" which contexts may exhibit is not intended to be exhaustive, nor are the possible restrictions described intended to be anything other than a basis for further work and further discussion. In particular, it is not claimed that they are necessary if atomicity refinement is to be used successfully.

It is perhaps inevitable that a discussion of observability should be essentially syntactic in nature since most of the information in which we are interested is lost on the calculation of process semantics. However, semantic considerations will certainly play a role in future work which explores how light restrictions on contexts may be made; in particular, semantic conditions are useful in general since they may unify a set of diverse syntactic restrictions and give a better insight into what those restrictions actually mean. Of course, syntactic restrictions are still useful since they are more easily checkable than semantic conditions, the latter not even being decidable in the general case.

Finally, note that we have made no reference to the issue of mobility in our consideration of restrictions on contexts. Considering mobility explicitly would not change significantly the flavour of the above discussion. Rather, it simply adds a new set of ways in which undesirable observations may be made and so necessitates an additional class of (syntactic) restrictions to be imposed.

3 Atomicity refinement in the literature

There are a number of notions of correctness which allow relaxation of atomicity and we look here at some of them in terms of the framework presented in Section 2. This consideration is not intended to be an exhaustive survey of the literature, rather it has two main purposes. The first is to illustrate the usefulness of the concept of observability when looking at existing approaches. The second is to highlight the way in which certain existing approaches — some of which are well-known and widely used — are flawed if our goal is to construct a more general framework for (software or hardware) development by refining atomicity. In particular, a number of existing approaches either do not treat at all the question of contexts or treat them in a limited and perhaps overly restrictive way. Hence, we may draw the conclusion that further work is needed if atomicity refinement is to be added to the set of formal, compositional development methods.

3.1 Serializability, linearizability and atomicity

Serializability (see, for example, [B^+87, WV01]) is used in relation to database systems and allows us to relate the concurrent execution of a set of transactions performed by the system to some sequential execution of the same set of transactions. The formal models of serializability presented in [B^+87, WV01] are interested only in the transactions which may be requested of and performed by the database system. Hence, they do not concern themselves with the issue of the wider context in which any particular database system might be deployed: there is no formal notion of user process or of what user processes might do when not requesting transactions of the database system. Thus, we do not know — formally at least — what user processes can observe of each other or what patterns of observation exist between the constituent entities of user processes. Nor do we know what further users of the system consisting of database plus immediate user processes might be able to observe of that composite system. Albeit informally, serializability is often justified as ensuring that user processes think they are interacting with a database system where transactions execute sequentially: i.e. users should not be able to tell the difference between a sequential and a concurrent system. However, the discussion in Section 2 indicates that a consideration of the issue of context is crucial to ensuring that this goal is met. One could argue that serializability is most interested in the data values returned from transactions, specifically the fact that the same data values would be returned by a sequential execution of the relevant transactions: on this reading of the situation, it is not so important that a user should be unaware of concurrent activity *per se* — which is what the consideration of context in Section 2 is primarily concerned with — as long as the data returned is acceptable. Nonetheless, if we are to give atomicity refinement the status of a formal compositional development method then we must be absolutely precise regarding not only the semantics of components where additional concurrency may be introduced but also regarding the semantics of wider systems where those components may be deployed. After all, a *compositional* method requires an explicitly defined way of composing systems and a way of giving a formal semantics to such compositions. Even on the basis that the formal models of serializability from [B^+87, WV01] focus on the interface between the scheduler and the database itself rather than on that between user programs and the scheduler/database system as a whole, the formal model says nothing regarding the behaviour of the scheduler.

Linearizability ([HW90]) is used to show the correctness of systems composed of concurrent data objects: in this framework, user processes interact with those objects using a method-based interface and correctness means they should not be able to detect the difference between the concurrent objects and their sequential counterparts. In this case, we *do* have an explicit (semantic) notion of user process. However, there are still certain problems which limit the wider applicability of linearizability as presented in [HW90]. In essence, there is a lack of fundamental detail on issues of observability and of computation in general, as well as at least one restriction on observability that is perhaps too harsh. In this framework, user processes are sequential — they are essentially individual threads of control — and are disallowed from observing concurrent executions on the part of the data objects in the sense that, once a process has executed the call of a particular method, it is not allowed to execute the call of another method until the first method has returned. In addition, the only events which user processes perform are interactions with the data objects.

It is not clear to what extent more general types of system could be encoded in the framework given and so if the broad "architecture" described actually imposes a limitation on those systems which may be reasoned about. In particular, the computations of user processes independently of their interactions with data objects are not described and there is no explicit notion of combining multiple threads of control into larger logical entities. There is no notion of making certain data items local to certain processes (indeed, it is not clear what determines which data objects a particular process can or will interact with). In addition, there is no clear notion of how a further system may be composed with the one under consideration or whether we are simply interested in closed systems (it is implicit that the latter is the case, since meaningful composition would require processes to be ready to communicate with objects in different systems and this idea does not seem to be captured).

There are also problems with respect to observability. Firstly, it is not clear what those observations might be which are being forbidden by the restrictions imposed on user processes and making this clear depends on the model of communication being used. In other words, the only way in which the impact of the restrictions placed on observability can be properly assessed is by knowing what would be possible if they were lifted. If user processes could communicate synchronously with each other in the general case, then the power of observation of those processes would be restricted in two main ways in the linearizability framework: other user processes may not be observed directly and individual user processes are not endowed with the ability to observe concurrent executions of *any* data objects. These restrictions may be related directly to the general discussion in Section 2: they mean that distinct user processes may not share information on the methods which are currently executing and so may not detect the fact that multiple such methods are executing concurrently (in the same object); moreover, individual user processes are denied the capability to detect directly the fact of concurrent execution on the part of a particular object. However, by the discussion in Section 2, the restriction on inter-process communication is perhaps too harsh. It may not be necessary that user processes should never communicate directly with each other; rather, it may be sufficient to restrict communication between any two threads of control so that it occurs only when no methods requested by those threads of the same object are currently executing. (Of course, in the linearizability framework we have no information on what user processes do between method calls.) Moreover, the restriction that processes may not make *any* method call while waiting for a return could also be eased (this restriction means that, as soon as a method call is carried out by a process, it is unable to observe anything of any data object until it receives the corresponding return). It is certainly necessary that no process should be able to make a method call to a particular object while it is waiting for a return from that *same* object, since such a possibility would allow it to detect the fact of concurrent execution on the part of that object. However, even when dealing only with sequential objects, methods from *different* objects can execute in parallel. Thus, it seems acceptable that user processes should be able to observe and interact with other objects even while waiting for a method return from this one. Of course, as discussed in Section 2 in relation to communication between two processes via a third-party process, certain restrictions may still need to be placed on such interactions if evidence of newly-concurrent execution is not to be detected. If we assume only asynchronous communication between processes — i.e. that processes communicate only via shared data objects — then the restrictions imposed on observability are reduced somewhat. In this case, direct inter-process communication is not possible anyway and so the only restriction imposed is that individual threads of control are not able to be engaged simultaneously in the execution of multiple methods. As we have shown above, this is a restriction which could possibly be eased.

Finally, due to the way in which user processes are defined — and because the issue is not made explicit at all — we have no information on what further users may observe of the system comprising (immediate) user processes plus data objects. Although we may give a semantics to that system, it is still not possible to give a semantics to any wider system in which it may be deployed. As stated with respect to serializability, a compositional development method needs a general means of composition and a general means of assigning semantics to compositions.

We also comment briefly on the approach described in [L$^+$94] and which uses the *i/o automaton* model from [LT87] as a means of representing systems (both context and component processes are described using the same automaton model and so contexts are described explicitly.) The approach is concerned with the correctness of transaction-processing systems and ultimately considers this correctness at the interface between user programs and the transaction scheduler. More specifically, a particular (concurrent) transaction-processing system is regarded as correct if the system plus scheduler has the same visible behaviours as a counterpart sequential system plus the scheduler, where only behaviours at the scheduler/user-program interface are regarded as visible. The scheduler therefore functions as a wrapper to hide direct evidence of additional concurrent behaviour and so we disallow user processes from observing directly the behaviours of the concurrent transaction-processing system. This use of the scheduler is necessary since the i/o automaton model cannot relate concurrent to sequential behaviours when those behaviours are to be regarded as visible in the components under consideration: in essence, it suffers from the problem described at the beginning of Section 2.3. In the general case, it may not be possible to hide concurrent behaviours from user processes using such a wrapper process and it is this fact which limits the usefulness of the i/o automaton model as we work towards a general development method based around atomicity refinement.

3.2 Atomicity refinement within a process algebraic framework

We also consider three approaches which have been presented in the process algebraic framework and which allow atomicity refinement. By their nature, they are presented in a fully formal framework and are endowed with explicit means for process or system composition and for deriving the semantics of such compositions.

The approaches in [Bur04, RG01] are conceptually similar although different in technical execution. In both of these approaches, the behaviours of the concurrent process *Conc* are related to those of *Seq* using an interpretive mapping. It is in this sense that the semantics used to interpret these component processes relaxes its power of observation. In addition, *Conc* may be placed in a context C' rather than in C, and C' is then related to C using an interpretive mapping. In these approaches, the contexts C and C' must be such that *Seq* and *Conc* respectively will be placed into them within the scope of a hiding operator: this hiding operator makes internal all direct evidence of communication between context and component and means that it cannot be observed by any user of the composition. Hence, the power of observation of such a user will be restricted. This allows us to reclaim standard process algebraic notions of semantics when considering context plus component. Beyond this restriction regarding hiding, the set of valid contexts is restricted implicitly by the requirement that it must be possible to verify the correctness of each context using an appropriate interpretive mapping. As a result, it is not clear in practical terms what exactly are the restrictions imposed on contexts, nor indeed how those restrictions relate to questions of observability and the use of atomicity refinement. This is therefore an area in which further work is needed.

In [MS92], *barbed congruence* is introduced as a notion of behavioural equivalence. The idea behind this is to equip an observer with a *minimal* ability to observe actions and/or process states, which ability induces an equivalence relation between processes. This equivalence relation induces in turn a congruence, namely equivalence in all contexts (this follows the discussion in Section 2). In the event that no restrictions are placed on those contexts which may be regarded as valid — i.e. valid contexts are simply all those which may be defined in the language under consideration — then barbed congruence is equivalent to a standard process algebraic notion of equivalence. However, it becomes strictly weaker than that existing notion if the set of contexts into which a process may be placed is restricted (this is the benefit of essentially parameterising the notion of correctness by the set of valid contexts). In [San99a], Sangiorgi presents a notion of typing called *uniform receptiveness* and this notion is used to restrict the set of contexts which are regarded as valid. Under this typing, barbed congruence can be used to show the correctness of processes derived using atomicity refinement (see, for example, [San99b] which considers the verification of a process similar to the example given in Section 4).

There are two important features here. Firstly, the observational power of the semantics used to relate component processes has been reduced so that processes exhibiting additional concurrency can be related to their sequential counterparts. Moreover, that same semantic notion is used to assign a meaning to context plus component. Though having a uniform notion of correctness across all processes gives a cleaner treatment, there is no explicit sense in which a standard notion of correctness may be reclaimed. Hence, there is perhaps lacking an intuitive sense of what it means to be correct according to this notion. Most significant, however, is the notion of uniform receptiveness, which is used to restrict both the set of contexts and the set of possible component processes (it is used to describe the communication or observational capabilities of processes). If a particular communication link is receptive in a particular process, then it will *always* be ready to receive data over that link. The property of uniformity requires that data received over the communication link is processed in the same way whenever it is received. It seems that uniform receptiveness would regard as invalid the problem contexts described in Section 2.6 and it is likely that it will play an important role in future work on atomicity refinement. However, it was not developed specifically to deal with the problem of atomicity refinement and so the question remains of whether it constitutes the best or most suitable restriction to be imposed.

4 An extended example

This section gives an example of "atomicity refinement". The idea of the research described in [Jon96] (see there for other references) is to maximise the use of sequential program development methods (data reification and operation decomposition) and to introduce concurrency late in the development by using equivalence transformations. Most interestingly for the current paper, studying some of these equivalences throws considerable light on the conditions under which it is possible to refine atomicity.

Transformations which are claimed (under reasonable assumptions) to preserve observational equivalence are outlined in Section 4.2. The setting is the use of a simple concurrent object-oriented language; Section 4.1 outlines the language and argues for the use of OO languages for concurrency. Section 4.3 contains a discussion of some delicate issues of observation, while two approaches to giving semantics are outlined in Section 4.4.

4.1 A simple OOL

In standard imperative languages, the interference which arises from concurrency is difficult to control. A long sequence of innovative ideas including semaphores, (conditional) critical regions, and monitors have been proposed.[8] Actually the question of granularity in imperative programs is even more vexed. In many papers on concurrency, assignment statements are assumed to execute atomically but this is clearly an unrealistic assumption for any reasonable implementation.

A key advantage of Object-Oriented Languages (OOLs) is that they put granularity control in the hands of the program designer. One can note that:

- local (instance) variables can only be referenced by methods of their class;
- if (as below) only one method at a time is active in any one object, there is no local interference;
- controlled visibility of references (see below) limits interference between objects; but
- shared references let designers permit interference where needed.

These graded ways of permitting interference result from the fact that methods provide the only access to instance variables. In other words, we restrict

```
class $Tab$
vars $mk$: $Key \leftarrow$ nil; $md$: $Data \leftarrow$ nil;
       $l$: private reference $Tab \leftarrow$ nil;
       $r$: private reference $Tab \leftarrow$ nil
$insert(k\colon Key,\ d\colon Data)$ method
  begin
    if $mk =$ nil then $(mk \leftarrow k;\ md \leftarrow d)$
    else if $mk = k$ then $md \leftarrow d$
    else if $k < mk$ then
        (if $l =$ nil then $l \leftarrow$ new $Tab$ fi ;
        $l.insert(k, d)$)
    else
        (if $r =$ nil then $r \leftarrow$ new $Tab$ fi ;
        $r.insert(k, d)$)
    fi;
    release
  end
$search(k\colon Key)$ method $res\colon Data$
  if $k = mk$ then return($md$)
  else if $k < mk$ then return($l.search(k)$)
  else return($r.search(k)$)
  fi
end $Tab$
```

Figure 2: Symbol table program (sequential)

the manner in which other processes may observe and so interact with these variables.

Here, a simple OOL which enjoys these properties and can be used to illustrate transformations (and discuss the extent to which they are equivalences) is used. The language is known from [Jon96] and earlier papers cited there as $\pi o \beta \lambda$.[9] The main novelty in $\pi o \beta \lambda$ is the option to mark references as private

[8] Hoare's contributions (collected in [HJ89]) can be said to have led him away from the imperative paradigm to communication-based concurrency; it has been argued elsewhere [Jon05] that the problems of interference are present there as well.

[9] Note that $\pi o \beta \lambda$ is not fully object oriented in the way Smalltalk is because of the issue that losing the Boolean type would necessitate the passing of something like "closures"; it is not possible to "call back" because of single method active rule (one could introduce pure functions and allow for them); the reason that the $\pi o \beta \lambda$ acronym implies that the language is "object based" is that it offers no inheritance (there are problems with inheritance and concurrency referred to as the "inheritance anomaly" by Kim Bruce).

(the restrictions relating to private references are defined in [Jon94]). Consider the program in Figure 2: it was developed from a specification written in terms of abstract objects (mappings) which were reified into the chosen tree representation; the post-conditions on that implementation representation were then decomposed into operations of the (so far) sequential language. Thus, for example, the code for the *insert* method has been shown to satisfy a specification of an earlier step of design.

The *insert* method in Figure 2 is sequential in the sense that its invocation holds its client in a *rendez-vous* until calls ripple down the tree to the appropriate insertion point and returns ripple all the way back up to the first server object. A similar description can be given for *search* which has to return a value up the tree and ultimately to the client. (A decision in the design of $\pi o\beta\lambda$ is that only one method is active per object; a method is active until it completes.) Thus both the *insert* and *search* methods can be viewed as "atomic" in the sense that nothing else (in the program) is active while they are executing.

The next section shows how this atomicity can be relaxed but real interest is in deeper questions of observability (and how one reasons about it): see Section 4.3.

4.2 Equivalences

Figure 2 contains a sequential program and the idea is at this late stage in design to "split the atoms" in the sense that the sub-steps of previously atomic operations will run interleaved with sub-steps of other objects. Obviously, such splitting will not in general preserve any useful notion of equivalence. It is the aim of a development method for refining atomicity to identify conditions under which equivalence will be preserved. Moreover, just as with other formal development methods like data reification, the *use* of the development method should be straightforward (whereas its *justification* might require more technical reasoning).

Compare the modified program in Figure 3 with the original in Figure 2. In the second version, *insert* holds its client up only while the parameters are transferred to the method invocation; after this, the client is free to execute in parallel with the server. Furthermore, once the top of the tree passes on the *insert* task to the next layer down the tree, the server is also free to accept other calls; thus multiple *insert* methods can be active within the tree at the same time.[10] Notice however that the "single method active rule" prevents overtaking.

The story for *search* is similar but more subtle because a value has to be returned. It is therefore necessary to hold the client until the required value is

[10] In a queue, the order of returns from *insert* is governed by position; in the tree here, the order is not even constrained by depth because parallel processes can use different sub-trees and respond earlier from greater depth.

```
class $Tab$
vars $mk$: $Key \leftarrow$ nil; $md$: $Data \leftarrow$ nil;
        $l$: private reference $Tab \leftarrow$ nil;
        $r$: private reference $Tab \leftarrow$ nil
$insert(k$: $Key$, $d$: $Data)$ method
  begin
    release;
    if $mk =$ nil then $(mk \leftarrow k;\ md \leftarrow d)$
    else if $mk = k$ then $md \leftarrow d$
    else if $k < mk$ then
        (if $l =$ nil then $l \leftarrow$ new $Tab$ fi ;
        $l.insert(k, d)$)
    else
        (if $r =$ nil then $r \leftarrow$ new $Tab$ fi ;
        $r.insert(k, d)$)
    fi
  end
$search(k$: $Key)$ method  $res$: $Data$
  if $k = mk$ then release$(md)$
  else if $k < mk$ then delegate$(l.search(k))$
  else delegate$(r.search(k))$
  fi
end $Tab$
```

Figure 3: Symbol table program (concurrent)

available; but a higher node in the tree can delegate the task of returning that value to a lower node. This has the effect of opening up the server to accept other invocations. Not only does this mean that multiple searches can run in parallel, but many *insert* and *search* methods can all be active in a tree in parallel. As above, no overtaking can occur.

Informally it is clear that the programs in Figures 2 and 3 can be viewed as equivalent but one would like formal correctness arguments for such "equivalences": these are discussed after some subtle questions about observability are explained.

4.3 A closer look at observability

Picking up on the notion of context described in Section 2, one might say a user context would not detect the difference between the *Tab* in Figures 2 and 3: the client ought not be able to detect which is used.[11] But hopefully a user with a stopwatch could detect difference, since using concurrency should improve performance (of course, there is an assumption here about multiple processors). For example, it should be clear that the statement after $insert(x)$ begins executing earlier in the parallel case because the return from *insert* is immediate. So it is already clear that one only means (in some sense) "functionally" equivalent, *not* equal timing. In order to be sure that the user cannot detect this difference *within* a program, there is no time stamp facility in $\pi o\beta\lambda$.

This naturally leads to the question of whether there are other assumptions about $\pi o\beta\lambda$ and its ability to observe. There are such assumptions and they do relate to atomicity.

Perhaps the most informative way of looking at the problem is to try to detect the differences between Figures 2 and 3. Assuming that a *Table* contains an association of Key 1 with the value a and 2 with the value b and running $insert(2, bb); insert(1, aa)$ in parallel with $print(search(1)); print(search(2))$ is non-deterministic. With either the sequential or the parallel versions of *Tab* the only answer we can *not* get is aa, b. In the parallel case, the "single method" rule plays a key role because it prevents overtaking.

In *Tab*, it is key that $\pi o\beta\lambda$ has no way to detect when things finish in the sense of their *order*. If $\pi o\beta\lambda$ had such a language feature, it could detect differences between the sequential and parallel versions.

It is clear that programs with more pointers (references) into the trees or linked queues *would* be able to detect with which version they were working. This is the precise role of the restrictions on private references.

4.4 Semantics

The claims above about (non)observability require an underlying semantics for their justification. Papers on $\pi o\beta\lambda$ have employed both operational semantics and mappings to other (tractable) languages. Each of these approaches has advantages and drawbacks; they are outlined here because more work is required to underpin the method of refining atomicity.

One way to give the semantics of a language is to map it to a known language. It is desirable that it is relatively easy to prove results in this "known" language. This is the essence of "denotational semantics" (see [Sto77]): imperative (sequential) languages are mapped to the Lambda calculus which has a sound mathematical basis in terms of which proofs can be conducted. There is a rather natural mapping from $\pi o\beta\lambda$ to the π-calculus [MPW92, SW01] in the sense that the expansion is linear and there are concepts in this process algebra

[11] Note that here the set of contexts is immediately limited to those which may be expressed using $\pi o\beta\lambda$.

which nicely capture key facets of concurrent OOLs. Given that there is an algebra of π-calculus, this presents a good starting point for proofs. Such a mapping is described in [Jon94].

The influence of the π-calculus on the design of $\pi o\beta\lambda$ is acknowledged in the choice of alphabet for its name (the other major influence was POOL [Ame89]). After running into difficulties with proofs in terms of the π-calculus mapping, attempts were made to provide a direct operational semantics for $\pi o\beta\lambda$. Although the second author of the current paper is planning another publication on this approach, it is worth making a few comments.

It is a straightforward task to construct an SOS of a language like $\pi o\beta\lambda$: in fact the second author uses it in teaching. The essential fact is that the non-determinacy of progressing any object can be captured by a mapping

$$\mathit{ObjMap} = \mathit{Handle} \xrightarrow{m} \mathit{Oinfo}$$

So it is again not difficult to record the semantics for notions related to atomicity and observability. But as has been remarked above, the key test is whether the semantics facilitates proofs of the needed results. Recent progress is encouraging. The proof attempts in [HJ96] led to the pessimistic comment that "no natural algebra is available for SOS"! But the second author began to think of ways of adding SOS rules to a generic logical frame like that in HOL or Isabelle. This is what Nipkow and his colleagues have already done for Java/Isabelle in [Nip04] (an earlier reference to this idea is [CM92]).

5 Conclusions

In this paper, we have identified the notion of observability with that of communication or interaction and have highlighted the connections which exist between observability and atomicity refinement. In particular, we have shown the need to consider fully and explicitly the question of restrictions to be imposed on contexts if concurrent and sequential components are to be used interchangeably in those contexts. Moreover, we have argued for the use of object-oriented languages (under certain restrictions on pointers and on the ability of such a language to observe temporal ordering of events) as a means of limiting observations to be made of and within concurrent components: this has the effect of preventing interference within those components and of easing the proof required in order to show correctness.

The ultimate aim of these considerations is to work towards a formal compositional development method based around the refinement of atomicity. In order to proceed further towards this goal, we can identify a number of necessary steps. Firstly, we aim to develop a significant set of examples where atomicity refinement is needed to show correctness and then to assess against these examples a number of existing approaches from the literature. This will allow us to develop a taxonomy both of types of problem and of approaches to atomicity refinement, as well as to identify connections between the two. On the basis of such work, it should be possible to develop a suitable general semantic framework within which questions relating to atomicity refinement might be considered and in terms of which a general development method might be built.

Acknowledgments

The Dagstuhl meeting of April 2004 (see [J+05] for details) was a major stimulus to writing the current paper whose authors acknowledge the open and productive discussions.

For funding of their research, the current authors are grateful to (UK) EPSRC for the "Dependability IRC" (DIRC — see www.dirc.org.uk) and for "Splitting Atoms Safely"; and for EU-IST funding for "Rodin" (see rodin.cs.ncl.ac.uk).

We also thank the referees for their valuable comments on this paper.

References

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

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

[Bur04] J. Burton. *The Theory and Practice of Refinement-After-Hiding.* PhD thesis, University of Newcastle upon Tyne, 2004. Available as University of Newcastle upon Tyne Technical Report CS-TR-904.

[Bur05] J. Burton. Relaxing atomicity and verifying correctness: considering the case of an asynchronous communication mechanism. *Journal of Universal Computer Science*, 11(5):771 – 802, 2005.

[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.

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

[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.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[J+05] Jones et al. The atomic manifesto. *Journal of Universal Computer Science*, 11(5):636 – 650, 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.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

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

[Jon94] C. B. Jones. Process algebra arguments about an object-based design notation. In A. W. Roscoe, editor, *A Classical Mind*, chapter 14, pages 231–246. 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.

[Jon05] C. B. Jones. Splitting atoms safely. *(accepted for publication in) Theoretical Computer Science*, 2005.

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

[LT87] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, 1987.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[MS92] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *Proceedings ICALP '92*, volume 623, pages 685–695, Vienna, 1992. Springer-Verlag.

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

[RG01] Arend Rensink and Roberto Gorrieri. Vertical implementation. *Information and Computation*, 170:95–133, 2001. Extended version of "Vertical Bisimulation" (TAPSOFT '97). Full report version: Hildesheimer Informatik-Bericht 9/98, University of Hildesheim.

[San99a] Davide Sangiorgi. The name discipline of uniform receptiveness. *Theor. Comput. Sci.*, 221(1-2):457–493, 1999.

[San99b] Davide Sangiorgi. Typed pi-calculus at work: A correctness proof of Jones's parallelisation transformation on concurrent objects. *TAPOS*, 5(1):25–33, 1999.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

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

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.

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