Journal of Universal Computer Science, vol. 11, no. 5 (2005), 636-650
submitted: 30/11/04, accepted: 31/1/05, appeared: 28/5/05 © J.UCS

The Atomic Manifesto

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

David Lomet
(Microsoft Research, USA
lomet@microsoft.com)

Alexander Romanovsky
(University of Newcastle upon Tyne, UK
alexander.romanovsky@ncl.ac.uk)

Gerhard Weikum
(MPI Saarbruecken, Germany
weikum@mpi-sb.mpg.de)

Dagstuhl Seminar (Organizer Authors)

and Alan Fekete, Marie-Claude Gaudel, Henry F. Korth, Rogerio de Lemos,
Eliot Moss, Ravi Rajwar, Krithi Ramamritham, Brian Randell, Luis Rodrigues
Dagstuhl Seminar (Participant Authors).

Abstract: This paper is a manifesto for future research on “atomicity” in its many guises and is based on a five-day workshop on “Atomicity in System Design and Execution” that took place in Schloss Dagstuhl in Germany in April 2004.

Key Words: atomicity, transactions, dependability, formal methods, hardware, programming languages
Category: A.0 General Literature

1 Introduction

This paper is based on a five-day workshop on “Atomicity in System Design and Execution” that took place in Schloss Dagstuhl in Germany [5] in April 2004 and was attended by 32 people from different scientific communities. [1] The participants included researchers from the four areas of:

– *database and transaction processing systems*;

[1] The full list of participants is given at [5].

- *fault tolerance and dependable systems*;
- *formal methods for system design and correctness reasoning* and,
- to a smaller extent, *hardware architecture and programming languages*.

The interpretations and roles of the atomicity concept(s) vary substantially across these communities. For example, the emphasis in database systems is on algorithms and implementation techniques for atomic transactions, whereas in dependable systems and formal methods atomicity is viewed as an intentionally imposed (or sometimes postulated) property of system components to simplify designs and increase dependability. On the other hand, all communities agree on the importance of gaining a deeper understanding of composite and relaxed notions of atomicity. Moreover, the hope is that it will eventually be possible to unify the different scientific viewpoints into more coherent foundations, system-development principles, design methodologies, and usage guidelines.

Atomicity is, of course, an old concept; in particular, transaction technology is considered as very mature. So why would there be a need for reconsidering it, and why now? There are several compelling reasons for reviving and intensifying the topic at this point:

- The world of network-centric computing is changing. For example, consider:
  - web services;
  - long-running workflows across organizational boundaries;
  - large scale peer-to-peer publish-subscribe and collaboration platforms;
  - ambient-intelligence environments with huge numbers of mobile and embedded sensor/actor devices.

  These all critically need support for handling or even masking concurrency and component failures, but cannot use traditional atomicity concepts.
- There is a proliferation of open systems where applications are constructed from pre-existing components. The components and their configurations are not known in advance and they can change on the fly. Thus, it is crucial that atomicity properties of components are composable and that we can predict and reason about the behavior of the composite system.
- Even if we can successfully develop adequate notions of relaxed atomicity, it is unlikely that one particular solution can handle all cases across the wide spectrum of application needs. So, application designers and programmers will be faced with several options and critical choices. Since humans are the bottleneck in terms of cost, time and errors, it would be optimal to have an autonomic approach [3] that automatically chooses the most appropriate option and reconfigures the system as the environment changes.

– Modern applications and languages like Java lead millions of developers into concurrent programming ("synchronized classes"). This is a drastic change from the classical situation where only a few hundred "five-star wizard" system programmers and a few thousand programmers working in scientific computing on parallel supercomputers would have to cope with the inherently complex issues of concurrency (and advanced failure handling as well).

– On an even broader scale, the drastically increasing complexity of the new and anticipated applications is likely to lead to a general "dependability crisis" in the not-too-distant future. The multi-technology nature of these applications strongly suggests that a multi-disciplinary approach is essential if researchers are to find ways to avert such a crisis.

2 The Views of Four Communities

2.1 Database and TP Perspective

2.1.1 Position

Database transaction concepts have been driven by traditional business applications and a style of software called OLTP (On-Line Transaction Processing) where fast-executing, independently coded application programs run against data stored in some general purpose DBMSs (Data Base Management Systems), which provide a mechanism called ACID transactions to support correct operation of the combined system [4, 23]. ACID stands for "atomicity, consistency, isolation and durability". In the OLTP approach, the application programmer delegates to the DBMS software responsibility for preventing damage to the data from threats such as concurrent execution, partial execution or system crashes, while each application programmer retains the obligation to think about the impact on data consistency of the code they are writing, when executed alone and without failures.

There are many threats to the overall dependability of the combined system formed from the databases and the application programs. The focus of database transactions is on dealing with threats from concurrent execution, from incomplete execution (e.g., due to client crash or user-initiated cancellation) and from system crashes that lose up-to-date information from volatile buffers. The traditional DBMS solution is to provide "ACID transactions". There are two ways a transaction can finish: it can commit, or it can abort. If it commits, all its changes to the database are installed, and they will remain in the database until some other application makes further changes. Furthermore, the changes will seem to other programs to take place together. If the transaction aborts, none of its changes will take effect, and the DBMS will "rollback" by restoring previous values to all the data that was updated by the application program.

From a programmer's perspective, the power of the transaction paradigm is that it reduces the task of concurrent failure-aware programming of the whole system to that of correct sequential programming of each application program separately. It is worth pointing out that while other fields describe the concept of apparently indivisible, point-like behavior as “atomicity”, in the database community “atomic” means that all the changes happen, or none do. The appearance of happening at a point is referred to as “isolated” (or serializable) behavior.

Internally, the DBMS uses a variety of mechanisms, including locking, logging and two-phase commit, to ensure that the application programs get the ACID transactional behavior they expect. The basic algorithms are fairly straightforward, but they interact in subtle ways, and have serious performance impacts, so the actual implementation of these facilities is very complicated [8].

2.1.2 Challenges Ahead

One major theme that came up during the workshop is the need to provide support for application domains that need different design points than the very short, completely independent, programs typical of OLTP, but where there is still the goal to help avoid problems from interleaving, system crashes, etc. For example, design applications were studied extensively in the 1980s; in the late 1990s workflows (or business processes) became important, and the latest domain of this type is composite web services where several business processes interact across organizational trust boundaries. Key features in these domains include the expectation for cooperation between programs rather than complete independence; the long duration (hours or even weeks) of an activity; and the desire to move forward even when something goes wrong, rather than throwing away all the work and returning to a previous state (so, we really want “exactly once” or “run then compensate” rather than “all or nothing”). Another very different class of domain occurs in security work, e.g., identifying attacks, where immediate results are more important than precise ones, and where the activity taking place against the database is itself data of importance (and should be recorded and preserved even if the activity fails).

In all these domains, it seems impossible to have each application program written in complete ignorance of the other applications, and to have the infrastructure work no matter what the application programs do; however, one would wish to limit the cross-component dependencies in some way, so that it is possible to reason about the combined effect of applications in the presence of concurrency, partial execution and system crashes. The database community has already proposed a range of extended transaction models [12] (often based on some form of nesting of scopes). There have even been designs for a broad framework within which one can describe multiple extended transaction models. However few of the extended transaction models have seen wide use by application programmers so far, and there remain two open questions: what transaction-like (unbundled, relaxed, or extended) features the infrastructure should provide and how to reason about application programs that use these features.

Two other important workshop themes connect the database community with others. One needs close cooperation with both formal methods and hardware people; this concerns the implementation of transactional mechanisms inside the DBMS. As noted above, the internals are very complex, and their design is sometimes based on principles such as internal support for atomicity through layered notions of transactions. Indeed many of the early proposals for richer models of transactions which did not get taken up by application programmers can today be found in DBMS implementations, where small groups of sophisticated programmers can work with them. It is still unclear how to best reason about the full complexity of a DBMS implementation of transactions in ways that take account of the interactions between aspects like buffer management, fancy synchronisation properties of the hardware disk controller and OS, and multiple threads running in the DBMS code. These low-level internals are likely to be the cause of occasional (albeit very infrequent) “Heisenbugs” [8], and recovery code is the last resort to avoid damage by such software failures. So it would be highly desirable to verify mathematically the correctness of this transactional core of the mission-critical DBMS software.

The third theme that came up consistently at Dagstuhl connects the database community to formal methods work. It was the need to reason about applications that do not use ACID transactions. In commercial reality, the performance impact of the ACID mechanisms is so high that most application programs actually do not use the full functionality. While the applications do want “all or nothing” and “committed state persists despite crashes”, they are usually willing to give up on ”the activity appears like a point”, by using weaker isolation levels than serializability. Indeed, some vendors do not implement serializability exactly, but rather use a “snapshot isolation” approach which avoids many but not all cases of data interference in concurrent execution. Since weak isolation is widely used, researchers need to offer help for the application developer to use it correctly.

2.2 Dependable Systems Perspective

2.2.1 Position

The dependability of a computing system is its ability to deliver service that can justifiably be trusted [2]. The major activities associated with the means of achieving dependability are fault tolerance, prevention, removal and forecasting. Atomicity plays an important role in designing and analysing dependable systems. As the fundamental approach assisting abstraction and system structuring, it is crucial in attempts to prevent the occurrence and introduction of faults since it allows the complexity of a design to be reduced. Use of abstraction and structuring in system development facilitates fault tolerance (by confining error) and fault removal (by allowing component validation and verification). Atomicity often makes fault forecasting simpler as it makes it easier to reason about likely consequences of faults.

Fault tolerance is a means for achieving dependability despite the likelihood that a system still contains faults and aiming to provide the required services in spite of them. Fault tolerance is achieved either by fault masking, or by error processing, which is aimed at removing errors from the system state before failures happen, and fault treatment, which is aimed at preventing faults from being activated again [2]. Atomic actions can be used as the basis of error confinement strategies — these play a central role in the design and justification of both error masking and error recovery policies.

The development of atomic action techniques supporting the structured design of fault tolerant distributed and concurrent applications is an important strand of dependability research. The work effectively started with the paper [20] where the concept of a conversation was introduced. An atomic action (conversation) consists of a number of concurrent cooperating participants entering and leaving it at the same time (i.e. concurrently). Here the word atomic also refers to the property that the changes made by an operation are only visible when it completes. When an error is detected in a conversation all participants are involved in cooperative recovery. Backward error recovery (rollback, retry, etc.) and forward error recovery (exception handling) are allowed. Actions can be nested and when recovery is not possible the responsibility for recovery is passed to the containing action. Action *isolation* makes the actions into error confinement areas and allows recovery to be localised, at the same time making reasoning about the system simpler.

2.2.2 Challenges Ahead

Atomic actions, initially introduced for systems consisting of cooperating activities, were later extended to allow actions to compete for shared resources (e.g. data, objects, devices). By this means the work was brought together with that on database transactions, which concerned systems of independent processes that simply competed for shared resources, i.e. the database. Coordinated atomic actions [24] thus can be used to structure distributed and concurrent systems in which participants both cooperate and compete, and allow a wide range of faults to be tolerated by using backward and forward recovery. These actions can have multiple outcomes, extending the traditional all-or-nothing semantics to make it possible to deal with those environments that do not roll back or for which backward recovery is too expensive (web services, external devices, human beings, external organisations, etc.). The challenge here is to work closely with the formal method group on developing rigorous design methods and tools supporting atomic actions and error recovery. More effort needs to be invested into developing advanced atomic actions techniques for emerging application domains and architectures, such as mobile and pervasive systems, ambient intelligence applications, and service-oriented architecture.

As seen above, cooperation and coordination are essential for the kind of atomicity required for the structured design of distributed fault-tolerant systems. When building such systems, one is often faced with the necessity of ensuring that different processes obtain a consistent view of the system evolution. This requirement may be expressed in different ways, for instance:

– A set of processes involved in a distributed transaction may need to agree on its outcome: if a transaction is aborted at some process it should not be committed at some other processes. This is known as the distributed atomic commitment problem.

– Replicas of a component, when applying non-commutative updates, must agree not only on the set of updates to apply but also on the order in which these updates are applied. This is known as the atomic multicast problem.

Many of the challenges that are involved in solving these agreement problems in fault-tolerant distributed systems are captured by the consensus problem, defined in the following way: each process proposes an initial value to the others, and, despite failures, all correct processes have to agree on a common value (called a decision value), which has to be one of the proposed values. Unfortunately, this apparently simple problem has no deterministic solution in asynchronous distributed systems that are subject to even a single process crash failure: this is the so-called Fischer-Lynch-Paterson's impossibility result [7]. This impossibility result does not apply to synchronous systems but, on the other hand, fully synchronous systems are hard to build in practice.

A significant amount of research has been devoted to defining models that have practical relevance (because they capture properties of existing systems) and allow for consensus to be solvable in a deterministic way. Such models include partial synchronous, quasi-synchronous and asynchronous models augmented with failure detectors, among others [22]. At the workshop, there was some confusion among the participants from the database community as to how these various models relate to each other, what (realistic as well as unrealistic) assumptions they make, and what properties and limitations they have. A unifying framework would be highly desirable, and this should include also the database-style (2PC-based) distributed commitment.

In component-based development, atomicity, seen as guaranteeing hermetic interfaces of components, is a key element of the so-called orthogonality property of system designs. The aim of an orthogonal design is to ensure that a component of the system does not create side effects on other components. The global properties of a system consisting of components can then be stated strictly from the definition of the components and the way they are composed.

Some extended notions of atomicity and orthogonality could be used as a mechanism for composing services by incorporating the interactions between components. This would be feasible if it was possible to abstract the actual component behaviour from the well-defined interfaces that allow expression of the different roles which a component might play. However, for this to happen it is necessary to replace the traditional notion of atomicity with a more relaxed one where, for example, the components taking part in a transaction are not fully tied up for the whole length of the transaction. Although different applications might require different forms of such quasi-atomicity, it might be possible to identify useful design patterns specific for the application domain. Even assuming that a useful relaxed notion of atomicity could be defined and implemented, the task of incorporating this concept into a development process is still not a straightforward one. For example, the transformation of a business dataflow into an implementation based on the synchronization of components cannot be captured by a simple top-down process consisting of refinement rules if system decomposition leads to the identification of new behaviours (including new failure behaviours). Instead, this essentially top-down process should be modified by allowing bottom-up revisions.

2.3 Hardware and Language Perspective

2.3.1 Position

Explicit hardware support for multithreaded software, either in the form of shared-memory-chip multiprocessors or hardware multithreaded architectures, is becoming increasingly common. As such support becomes available, application developers are expected to exploit these developments by employing multithreaded programming. But although threads simplify the program's conceptual design, they also increase programming complexity. In writing shared memory multithreaded applications, programmers must ensure that threads interact correctly, and this requires care and expertise. Errors in accessing shared data objects can cause incorrect program execution and can be extremely subtle. This is expected to become an even greater problem as we go towards heavily threaded systems where their programmability, debuggability, reliability and performance become major issues.

Explicitly using atomicity for reasoning about and writing multithreaded programs becomes attractive since stronger invariants may be assumed and guaranteed. For example, consider a linked list data structure and two operations upon the list: insertion and deletion. Today, the programmer would have to ensure the appropriate lock is acquired by any thread operating upon the linked list. However, an attractive approach would be to declare all operations upon the linked list as "atomic". How the atomicity is provided is abstracted away for the programmer and the underlying system (hardware or software) guarantees the contract of atomicity.

The hardware notion of atomicity involves performing a sequence of memory operations atomically. The identification of the sequence is, of course, best left to the programmer. However, the provision and guarantee of atomicity comes from the hardware. The core algorithm of atomically performing a sequence of memory operations involves obtaining the ownership of appropriate locations in hardware, performing temporary updates to the locations, and then releasing these locations and making the updates permanent instantaneously. In the event of failures, any temporary updates are discarded, thus leaving all critical state consistent. Hardware has become exceedingly proficient in optimistically executing operations, performing updates temporarily and then making them permanent instantaneously if necessary.

Transactional Memory [10] was an initial proposal for employing hardware support for developing lock-free programs where applications did not suffer from the drawbacks of locking. It advocated a new programming model replacing locks. Recently, Transactional Lock-Free Execution [18, 19] has been proposed, where the hardware can dynamically identify and elide synchronization operations, and transparently execute lock-based critical sections as lock-free optimistic transactions while still providing the correct semantics. The hardware identifies, at run time, lock-protected critical sections in the program and executes these sections without acquiring the lock. The hardware mechanism maintains correct semantics of the program in the absence of locks by executing and committing all operations in the now lock-free critical section “atomically”. Any updates performed during the critical section execution are locally buffered in processor caches. They are made visible to other threads instantaneously at the end of the critical section. By not acquiring locks, the hardware can extract inherent parallelism in the program independent of locking granularity.

While the mechanism sounds complex, much of the hardware required to implement it is already present in systems today. The ability to recover to an earlier point in an execution and re-execute is used in modern processors and can be performed very quickly. Caches retain local copies of memory blocks for fast access and thus can be used to buffer local updates. Cache coherence protocols allow threads to obtain cache blocks containing data in either shared state for reading or exclusive state for writing. They also have the ability to upgrade the cache block from a shared state to an exclusive state if the thread intends to write into the block. The protocol also ensures all shared copies of a block are kept consistent. A write on a block by any processor is broadcast to other processors with cached copies of the block. Similarly, a processor with an exclusive copy of the block responds to any future requests from other processors for the block. The coherence protocols serve as a distributed conflict detection and resolution mechanism and can be viewed as a giant distributed conflict manager. Coherence protocols also provide the ability for processors to retain exclusive ownership of cache blocks for some time until the critical section completes. A deadlock avoidance protocol in hardware prevents various threads from deadlocking while accessing these various cache blocks.

2.3.2 Challenges Ahead

Crucial work remains both in hardware and software systems. The classic chicken-and-egg problem persists. On one hand, existing software-only implementations of atomicity and transactions for general use suffer from poor performance, and on the other hand, no hardware systems today provides the notion of generalized atomic transactions. A major hurdle for hardware transactions remains in their specification. Importantly, what hardware transaction abstraction should be provided to the software? How is the limitation of finite hardware resources for temporarily buffering transactions handled? A tension will always exist between power users who would like all the flexibility available from the hardware and the users who would prefer a hardware abstraction where they do not worry about underlying implementations. These are some of the questions that must be addressed even though many of the core mechanisms in hardware required for atomic transactions, such as speculatively updating memory and subsequently committing updates, are well understood and have been proposed for other reasons, including speculatively parallelizing sequential programs [21].

The software area requires significant work. The first question remains the language support. Harris and Fraser [9] provided a simple yet powerful language construct employing conditional critical regions. In simple form, it is as follows:

$$\textbf{atomic}\ (p)\ \{\ S\ \}$$

Semantically this means S executes if p is true. If p is false, it needs to wait for some other process to change some variable on which p depends.

However, more rigorous constructions are required for specification of such language constructs. At least from the formal methods community perspective, specifying a concise formal description of the above constructs as a semantic inference rule in the operational semantics style would be necessary.

A first pass at such a declaration would be as follows:

$$s[p](s', true) \wedge s'[S]s'' \vdash s[\textbf{atomic}\ (p)\ \{\ S\ \}]s''$$

In English: If we start in state s and the guard predicate p evaluates to *true*, then we make the atomic state transition that evaluates p and then S. No other process will be able to observe or affect the intermediate state s' or any other intermediate state.

Looking forward, we suggest language designs will need to go beyond such simple constructs. Some of the issues designs might want to handle include: connecting with durability somehow, perhaps through providing special durable memory regions; expressing relative ordering constraints (or lack thereof) for transactions issued conceptually concurrently (e.g. iterations of counted loops, as typical of scientific programs operating on numerical arrays); supporting *closed* nesting [16] and the bounded rollback that it implies on failure; supporting *open* nesting where commitment of a nested transaction releases basic resources (e.g. cache lines) but implies retention of semantic locks and building a list of undo routines to invoke if the higher level transaction fails; providing for lists of actions to perform only if the top-level enclosing transaction commits; supporting the leap-frogging style of locking along a path one is accessing in data structures like linked lists and trees.

2.4 Formal Methods Perspective

2.4.1 Position

Formal methods [13] offer rigorous and tractable ways of describing systems. This is nowhere more necessary than with subtle aspects of concurrency: being precise about atomicity, granularity and observability is crucial.

The concept of *atomicity* –which is central to this manifesto– can easily be described using "operational semantics". McCarthy's seminal contribution on operational semantics [15] presented an "abstract interpreter" as a recursive function $exec : Program \times \Sigma \to \Sigma$ where Σ is the domain of possible states of a running program. As an interpreter, *exec* computes the final state (if any) which results from running a program from a given starting state; such descriptions are abstract in the sense that they use sets, maps, sequences, etc. rather than the actual representations on a real computer.

The obvious generalisation of McCarthy's idea to cope with concurrency turns out not to provide perspicuous descriptions because functions which yield sets of possible final states have to compound each other's non-determinism. In 1981 Ploktin [17] proposed presenting "structural operational semantics" (SOS) descriptions as inference rules. Essentially, rather than the function above, a relation can be defined

$$\mathcal{P}((Program \times \Sigma) \times \Sigma)$$

where, if $((p,\sigma),\sigma')$ is in that relation, σ' is a possible final state of executing p in a starting state σ.

Since the origin of these ideas is with programming language semantics, the description begins there; but the relevance to the fields above is easily demonstrated. Consider a simple language with two threads each containing sequences of assignment statements. It is assumed initially that assignment statements execute atomically. Two simple symmetric rules show that a statement from the head of either sequence can execute atomically

$$\frac{\begin{array}{l} hd(s1) = x \leftarrow e \\ (e,\sigma) \xrightarrow{e} v \end{array}}{(s1||s2,\sigma) \xrightarrow{s} (tl(s1)||s2, \sigma \dagger \{x \to v\})}$$

$$\frac{\begin{array}{l} hd(s2) = x \leftarrow e \\ (e,\sigma) \xrightarrow{e} v \end{array}}{(s1||s2,\sigma) \xrightarrow{s} (s1||tl(s2), \sigma \dagger \{x \to v\})}$$

In these rules: hd and tl stand for the head and tail of a list; $x \leftarrow e$ denotes the assignment of expression e to variable x; $(e,\sigma) \xrightarrow{e} v$ denotes that in program state σ expression e can evaluate to value v; $||$ denotes parallel execution of two statements; $\sigma \dagger \{x \to v\}$ is the state that is identical to σ except for the fact that variable x is now mapped to value v; and $(l,\sigma) \xrightarrow{s} (l',\sigma')$ means that the execution of statement list l in state σ leads to state σ' with the statement list l' left to be executed (the overall relation $\mathcal{P}((Program \times \Sigma) \times \Sigma)$ is derived when the statement list l' is empty).

From the above rules, it is easy to show that

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

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

$$(x \leftarrow x+1; x \leftarrow x-1)||(x \leftarrow x*2; x \leftarrow x/2)$$

can leave x as 1 or a range of other values.

This second example begins to form the bridge to transactions but, before taking that step, it is worth thinking a little more about atomicity. It would be trivial to extend the programming language to fix the second example so that it always left x at 1. One way to do this would be to add some sort of atomic brackets so that $s_1; < s_2; s_3 >; s_4$ executes as three (rather than four) atomic transitions. The changes to the SOS rules are simple. Moving atomicity in the other direction, it would actually be extremely expensive in terms of dynamic locking to implement assignments atomically. Showing all of the places where another thread can intervene is also possible in the SOS rules. Programming language designers have spent a lot of effort on developing features to control concurrency; see [11] for a discussion of "(conditional) critical sections", "modules" etc.

SOS rules can be read in different ways and each provides insight. As indicated above, they can be viewed as inductively defining a relation between initial and final states. They can also be viewed as defining a logical frame for reasoning about constructs in a language. Implications of this point are explored in [14].

As might be guessed by now, it is easy to define the basic notion of database "serializability" by fixing the meaning of a collection of transactions as non-deterministically selecting them one at a time for atomic execution. Of course, this overall specification gives no clues to the invention of the clever implementation algorithms studied in the database community. As with programming language descriptions, the relation defined by the SOS rules should be thought of as a specification of allowed behaviour.

2.4.2 Challenges Ahead

It is not only in the database world that "pretending atomicity" is a powerful abstraction idea. It was argued at the workshop that a systematic way of "splitting atoms safely" could be a useful development technique with applicability to a wide range of computing problems. Essentially, given a required overall relation defined by an SOS description, one needs to show that an implementation in which sub-steps can overlap in time exhibits no new behaviours at the external level.

An interesting debate at the workshop was what one might learn from trying to merge programming and database languages. Despite considerable research efforts in this direction [6], no convincing solution seems to have emerged yet. The most important challenge would be to look at how the two communities handle concurrency control.

SOS rules are certainly not the only branch of formal methods which could help record, reason about and understand concurrency notions in, for example, databases.

For example, we emphasize the insight which can be derived from process algebras and the distinction between interleaving and "true concurrency" as explored by the Petri Net community.

Finally, the intriguing notion of refinements that are accompanied by rigorous correctness reasoning has been successfully applied in the small [1], for example, to derive highly concurrent and provably correct data structures, e.g. priority queues, but it is unclear to what extent it can cope with the complexity of large software pieces like the full lock manager or recovery code of a DBMS or the dynamic replication protocol of a peer-to-peer file-sharing system. Tackling the latter kinds of problems requires teaming up expertise in formal methods with system-building know-how.

3 Lessons and Challenges

There was clear consensus across all four participating communities that atomicity concepts, if defined and used appropriately, can lead to simpler and better programming, system description, reasoning, and possibly even better performance. Some of the technical challenges that emerged as common themes across all communities are the following:

– A widely arising issue in complex systems is how to build strong guarantees on top of weaker ones, or global guarantees at the system and application level on top of local ones provided by components. Examples of this theme are how to ensure global serializability on top of components that use snapshot isolation or how to efficiently implement lazy replication on top of order-preserving messages.

– There was consensus that we still lack a deep understanding of the many forms of relaxed atomicity, their mutual relationships, prerequisites, applicability, implications, and limitations. For example, what are the benefits and costs of serializability vs. relaxed isolation, lazy vs. eager replication, or distributed commit in the database world vs. weaker forms of distributed consensus in peer-to-peer systems?

– Given the variety of unbundled, relaxed and extended atomicity concepts, there is a high demand for design patterns and usage methodology that helps systems designers to choose the appropriate techniques for their applications and make judicious tradeoff decisions. For example, when exactly is it safe to use snapshot isolation so that serializability is not needed; and under which conditions is it desirable to trade some degree of reliability for better performance?

– Along more scholarly but nevertheless practically important lines, we should aim to develop a unified catalog of failure models, cost models and formal properties of all variations of atomicity and consensus concepts, as a basis for improving the transfer of results across communities and for easier comprehension, appreciation and acceptance of the existing variety of techniques by practitioners.

– A long-term issue that deserves high attention is the verification of critical code that handles concurrency and failures, for example, the recovery manager of a DBMS. Which high-level structuring ideas from the dependability community and which formal reasoning and automated verification techniques from the formal-methods world can be leveraged to this end and how should they be used and interplay with each other when tackling the highly sophisticated software that we have in the kernels of DBMSs, middleware systems and workflow management systems?

For compelling reasons pointed out in the introduction, now is the right time for the different research communities to jointly tackle the technical challenges that impede the turning of atomicity concepts into best-practice engineering for more dependable next generation software systems. With rapidly evolving and anticipated new applications in networked and embedded environments that comprise many complex components, we face another quantum leap in software systems complexity. We are likely to run into a major dependability crisis unless research can come up with rigorous, well-founded, and at the same time practically significant and easy-to-use concepts for guaranteeing correct system behavior in the presence of concurrency, failures and complex cross-component interactions. The atomicity theme is a very promising starting point with great hopes for clear foundations, practical impact and synergies across different scientific communities.

Observations on Sociology: It was both ambitious and interesting to run a workshop with participants from four different communities. Many of the discussions led to misunderstandings because of different terminologies and implicit assumptions in the underlying computation models (failure models, cost models, etc.). A not quite serious but somewhat typical spontaneous interruption of a presentation was the remark “What were you guys smoking?”. The wonderful atmosphere at the Dagstuhl seminar site, the excellent Bordeaux, and a six-mile hike on the only day of the week with rain were extremely helpful in overcoming these difficulties. In the end there were still misunderstandings, but the curiosity about the applicability of the other communities’ results outweighed the skepticism, and a few potentially fruitful point-to-point collaborations were spawned. We plan to hold a second Dagstuhl seminar on this theme, again with participation from multiple research communities, in spring 2006.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. A Avizienis, J-C Laprie, C. Landwehr, B. Randell. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing* 1(1):11-13, 2004.
3. *1st Int’l. Conference on Autonomic Computing*. New York, 2004. http://www.caip.rutgers.edu/˜parashar/ac2004/
4. P. Bernstein and E. Newcomer. *Principles of Transaction Processing for the Systems Professional*. Morgan Kaufmann, 1997.

5. Dagstuhl Seminar 04181. *Atomicity in System Design and Execution*. Organized by C. Jones, D. Lomet, A. Romanovsky, G. Weikum. http://www.dagstuhl.de/04181/
6. *Int'l. Workshops on Database Programming Languages*, http://www.cs.toronto.edu/ mendel/dbpl.html
7. M. Fischer, N. Lynch, M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM* 32(2):374-382, 1985.
8. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
9. T. Harris, K. Fraser. Language support for lightweight transactions. In *Proc. of the Int'l. Conference on Object-Oriented Progamming Systems, Languages, and Applications*, 2003.
10. M. Herlihy and J.E.B. Moss. Transactional Memory: Architectural support for lock-free data structures. In *Proc. of the Int'l. Symposium on Computer Architecture*, 1993.
11. C.A.R. Hoare, C.B. Jones. *Essays in Computing Science*. Prentice Hall International, 1989.
12. S. Jajodia and L. Kerschberg (Editors). *Advanced Transaction Models and Architectures*. Kluwer, 1997.
13. C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.
14. C.B. Jones. Operational semantics: concepts and their expression. *Information Processing Letters* 88:27–32, 2003.
15. J. McCarthy. A formal description of a subset of ALGOL. In *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.
16. J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.
17. G.D. Plotkin. A structural approach to operational semantics. *Journal of Functional and Logic Programming*, forthcoming.
18. R. Rajwar and J.R. Goodman. Transactional Execution: Toward Reliable, High-Performance Multithreading. In *IEEE Micro* 23(6):117–125, 2003.
19. R. Rajwar and J.R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. of the Int'l. Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.
20. B. Randell. System Structure for Software Fault- Tolerance. *IEEE Trans. on Software Engineering* SE-1(2):220-232, 1975.
21. G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Int'l. Symposium on Computer Architecture*, 1995.
22. P. Verissimo, L. Rodrigues. *Distributed Systems for System Architects*. Kluwer, 2000.
23. G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.
24. J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *Proc. of the 25th Int'l. Symposium on Fault-Tolerant Computing Systems*, 1995.