Executive Summary
Dagstuhl Seminar "Atomicity: a Unifying Concept in Computer Science"

1 Goals of the Seminar

This seminar was based on and continued the interaction of different computer-science communities that was begun in an earlier Dagstuhl seminar in April 2004. Both seminars have aimed at a deeper understanding of the fundamental concept of atomic actions and their roles in system design, execution, modeling, and correctness reasoning, and at fostering collaboration, synergies, and a unified perspective across largely separated research communities. Each of the two seminar brought together about 30 researchers and industrial practitioners from the four areas of database and transaction processing systems, fault tolerance and dependable systems, formal methods, and to 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 postulated property of system components to simplify designs and increase dependability. Nevertheless, all four communities share the hope that it will eventually be possible to unify the different scientific viewpoints into more coherent foundations, system development principles, design methodologies, and usage guidelines.

The 2004 seminar was very successful on connecting the communities. It raised first skepticism and then curiosity about each other's viewpoints and methodologies. As a major achievement, it led to a strategic position paper, entitled "The Atomic Manifesto: a Story in Four Quarks", which appeared, in identical form, in widely read publication venues in the different communities: ACM SIGMOD Record, ACM Operating Systems Review, the Journal of Universal Computer Science, and also within the Dagstuhl Seminar Proceedings. In addition, the seminar produced a special issue of the Journal of Universal Computer Science with 8 full papers that were presented in preliminary form at the seminar and one full paper co-authored by two researchers who had not met before the seminar. The Atomicity seminar in March 2006 was intended to intensify and extend this barely blooming plant of cross-community collaboration, to revisit and refine the technical challenges identified in 2004, and to discuss the progress made in the last two years.

The strategic and timely importance of the atomicity theme has been derived from the following observations and major trends:

1) The world of network-centric computing is rapidly increasing in complexity, with Web service composition, long-running workflows across system boundaries, large-scale peer-to-peer data-sharing and collaboration, ad-hoc networks of mobile devices, large networks of sensors and actuators, and ambient-intelligence environments; all of these critically need support for handling concurrency and component failures but cannot easily use traditional atomicity concepts.
2) There is a proliferation of open systems where applications are constructed from pre-existing components on the fly; it is crucial that properties of components are composable and lead to guarantees about the behavior of the entire system.
3) Application architects will be faced with options and critical choices regarding a wide variety of models for relaxed and extended atomicity; there is a pressing need for an autonomic approach that automatically chooses the most appropriate option and

reconfigures the system as the environment changes.

4) Modern applications and languages like Java lead millions of developers into concurrent programming and advanced exception handling; this is a dramatic explosion in the number of people, many with limited skills or experience, that need to cope with the inherently complex issues of concurrency and failure handling.

2 Results of the Seminar

The presentations and discussions at the seminar reconfirmed that a unified foundation for atomicity is a strategically important and pressing research avenue. Furthermore, the seminar was again successful in spawning new collaborations among participants, some of which span communities. It is planned to prepare another special issue for the Journal of Universal Computer Science, with full papers that hopefully emerge from this ongoing work and the results presented at the seminar.

In terms of specific research issues, the following topics led to intensive discussions and were identified as key directions within the broad theme of atomicity:

1) the integration of open nested transactions into programming languages and their run-time environment,
2) methods for providing strong guarantees about system behavior based on weaker guarantees by the underlying components,
3) handling atomic actions in time-critical environments like operating system kernels.

Open nested transactions have been developed in the database system community; they allow the composition of atomic operations on arbitrary objects with much higher concurrency than the older notion of closed nested transactions would allow. The implementation techniques and the correctness reasoning on run-time traces are reasonably well understood. However, it is unclear how such an advanced notion of atomicity should be integrated with programming language constructs, how to define program correctness, and how to statically analzye programs in terms of behavioral guarantees. The seminar led to lively discussions on specific proposals on embedding open nested transactions in Java, with a notion of atomic blocks that can be defined for methods of arbitrary classes. The run-time environment could be based on an extended form of software transactional memory. The discussion identified various open issues such as: What is the scope of an atomic operation; should it include exception-handling code? What is the best framework for defining the formal semantics of a program with atomic operations and open nesting? How can we help programmers to write robust code with such a powerful construct; are there typical design patterns?

A characteristic example for strong guarantees on top of weaker ones is the issue of providing serializable schedules of transactions (i.e., schedules that are equivalent to sequential ones) even if the underlying component ensures only snapshot isolation using a multiversion concurrency control protocol. This is a practically very important question, as some of the most widely used commercial database systems use snapshot isolation as their strongest or default model. Snapshot isolation seems to work very well in real applications, but is has a small probability of creating inconsistencies; this is unsatisfactory from a conceptual viewpoint and it may be intolerable for very critical applications. The classes of serializable and snapshot-isolated schedules are incomparable; neither of the two properties implies the other. To ensure serializability over a run-

time system with snapshot isolation, additional mechanisms are needed, the goal obviously being that these mechanisms are as light-weight as possible. The seminar discussed a specific approach that maintains, at run-time, an additional graph with transactions as nodes and captures various kinds of interference edges. An elegant theorem characterizes when there are one or more transactions that potentially violate serializability, and these transactiions can be forced to abort in order to ensure the stronger guarantee. The seminar discussed how such specific solutions can be generalized into fundamental methodologies and framework for dealing with relaxed notions of atomicity. Furthermore, the issue of design patterns to ease system design and provide guidelines to architects and programmers was identified as a crucial research topic.

Operating system kernels are highly concurrent and are vulnerable to all kinds of racing conditions. Even mature systems such as certain Linux variants are known to have windows of vulnerability; one particular pattern are TOCTTOU intervals: time-of-check to time-of-use. A typical example of this kind is editing a file under the root user (e.g., as part of some sophisticated scripting) and subsequently changing its ownership to a non-priviledged user. Here the TOCTTOU pair of actions is an open call, to create the file under the root user, followed later by a chown, to change ownership. If the file itself is a symbolic link to /etc/passwd, an attacker that repeatedly issues chown calls has a high probability of obtaining ownership of the password file. A seemingly straightforward solution would be to encapsulate the entire interval between the first and the second action of the TOCTTOU pair into a single uninterruptable atomic unit. However, using standard techniques for atomic transactions seems to incur unacceptable overhead and is thus considered non-viable for the time-critical code in the OS kernel. The solution presented at the seminar thus used a relaxed notion of atomicity with event-driven guarding of invariants. This technique can be seen as introducing semantic locks for TOCTTOU-pair-specific invariants, but it is implemented in very light-weight manner that incurs less than 3 percent run-time overhead. The broader questions that were discussed at the seminar and left as challenges are: Can we generalize this approach of weaker-than-atomicity properties; can we formally specify and prove these properties in a principled manner beyond the specific case of TOCTTOU pairs? How can we efficiently implement these properties; for example, can we make transactional semantic locking more light-weight, or can we apply predicate-oriented optimistic concurrency control methods?

The above three specific topics are a good sample of the presentations and discussions at the seminar; it should be emphasized, however, that this is not an exhaustive summary. As the result of a wrap-up discussion, the seminar participants compiled a list of the 10 most interesting research topics within the atomicity theme:

1) specifying, proving, and providing execution guarantees,
2) giving objects control over which other objects can observe which state, and providing means for composability,
3) transactional guarantees for service composition,
4) defining different notions of relaxed or extended isolation and atomicity, and developing design patterns for them,
5) automatically generating skeletons for compensation activities,
6) leveraging the log for fault containment and system-level debugging,
7) exception handling beyond atomic actions,

8) synchronization models for long-lived applications such as workflows,
9) embedding composable atomic transactions in programming languages,
10) correctness proofs for atomicity implementations.