An Abort-Aware Model of Transactional Programming (Position Paper)

Kousha Etessami*[1] Patrice Godefroid[2]

[1] University of Edinburgh, kousha@inf.ed.ac.uk
[2] Microsoft Research, pg@microsoft.com

There has been a lot of recent research on transaction-based concurrent programming, aimed at offering an easier concurrent programming paradigm that enables programmers to better exploit the parallelism of modern multi-processor machines, such as multi-core microprocessors. Roughly speaking, transactions are marked code segments that are to be executed "atomically". The goal of such research is to use transactions as the main enabling construct for shared-memory concurrent programming, replacing more conventional but low-level constructs such as locks, which have proven to be hard to use and highly error prone. High-level transactional code could in principle then be compiled down to machine code for the shared memory-machine, as long as the machine provides certain needed low-level atomic operations (such as atomic *compare-and-swap*). Already, a number of languages and libraries for transactions have been implemented (see, e.g., [2] which surveys many implementations).

Much of this work however lacks precise formal semantics specifying exactly what correctness guarantees are provided by the transactional framework. Indeed, there often appears to be a tension between providing strong formal correctness guarantees and providing an implementation flexible and efficient enough to be deemed useful. Even if transactional constructs were themselves given clear semantics, there would remain the important task of verifying specific desired properties of specific transactional programs.

The aim of our work is to provide a state-machine based formal model of transactional concurrent programs, and thus to facilitate an abstract framework for reasoning about them.

So, what is a "transaction"? Syntactically, transactions are marked code segments, e.g., demarcated by "`atomic` {...}", or, more generally, they are certain procedures which are marked as `transactional`. But what is their semantics? An informal and naive view of their semantics is given by the so-called "*single-lock semantics*" (see, e.g., [2]), which says that during concurrent execution each executed transaction should appear "as if" it is executing serially without any interleaving of the operations of that transaction with other concurrent transactions executed by other processes. In other words, it should appear "as if" executing each transaction requires every process to acquire a single global transaction lock and to release that lock only when the transaction has completed. The problem with this informal semantics has to do with precisely what is meant

* The work of this author was done partly while visiting Microsoft Research.

by "as if". A semantics which literally assumes that every concurrent execution proceeds via a single lock, literally rules out any interleaving of transactions on different processes. This is too restrictive and disallows a lot of useful concurrency. It also violates the intended non-blocking nature of the transactional paradigm, and it completely ignores another key feature. Namely, transactions can typically be *aborted*, either *explicitly* by the programmer, or *automatically* by the run-time system due to memory conflicts. Aborted transactions can even have side effects (see the examples below), yet single-lock semantics ignores them.

We argue that these deficiencies make the "single-lock" semantic model unacceptable as a basic programmer's model of the semantics of transactional programs, even when one assumes *strong isolation* or *strong atomicity*, where one can essentially view all computation as taking place "atomically" at some level of granularity.[1] We argue in particular that for a programmer to use the explicit abort operations offered by the programming environment, he/she must know what those operations mean. A semantics which says nothing about what aborts mean can thus not be considered adequate.

Of course, single-lock semantics is not literally what is intended by Transactional Memory (TM) designers. Rather, a weaker semantics is intended, but phrasing a simple formal semantics which captures precisely what is desired and leaves sufficient flexibility for an efficient implementation is a non-trivial task. Standard correctness notions such as *serializability*, which are used in database concurrency control, are not directly applicable to this setting without some modification, in part because, in full-fledged transactional programming, it is no longer the case that every operation on shared memory is done via a necessarily terminating transaction: transactions may never halt, or may be aborted.

We propose *Transactional State Machines* (TSMs) as an abstract finite-data model of transactional shared-memory concurrent programs. The TSM model is non-blocking and allows interleaved executions where multiple processes can simultaneously be executing inside transactions. It also allows nested transactions, transactions which may never terminate and, importantly, transactions which may be aborted explicitly, or aborted automatically due to memory conflicts. We define the concept of *Abort-Aware Stutter- (AAS-) serializability*, which we feel captures in a clean and simple way a desired correctness criterion, and we show that TSMs satisfy this condition. We also study model checking of TSMs. We show that, although model checking for general TSMs is easily seen to be undecidable, it is decidable for an interesting fragment. Namely, when recursion is exclusively used inside transactions in all (but one) of the processes, we show that model checking such TSMs against all stutter-invariant ω-regular properties of shared memory is decidable.

Our definition of TSMs is based on two natural assumptions which are close in spirit to assumptions used in transactional memory systems. First, we implicitly assume the availability of a atomic (hardware or software implemented)

[1] More elaborate semantic models have been proposed recently in settings with weak isolation and weak memory models, but these typically build on top of semantics which in the basic setting with strong isolation are variants of single-lock semantics.

multi-word *compare-and-swap* operation, $CAS(\bar{x}, \bar{x'}, \bar{y}, \bar{y'})$, which compares the contents of the vector of memory locations $\bar{x}$ to the contents of the vector of memory locations $\bar{x'}$, and if they are the same, it assigns the contents of the vector of memory locations $\bar{y'}$ to the vector of memory locations $\bar{y}$. How such an atomic CAS operation is implemented is irrelevant to the semantics. (It can, for instance, be implemented in software using lower-level constructs such as locks blocking other processes.) Second, we assume a form of *strong isolation* (*strong atomicity*). Specifically, there must be minimal atomic operation units on all processes, such that these atomic units are indivisible in a concurrent execution, meaning that a concurrent execution must consist precisely of some interleaved execution of these atomic operations from each process. Thus "atomicity" of operations must hold at some level of granularity, however small. Without such an assumption, it is impossible to reason about asynchronous concurrent computation via an interleaving semantics, which is what we wish to do.

Based on these assumptions, we can now give an informal description of the TSM model. TSMs are concurrent boolean programs with procedures, except that some procedure calls may be *transactional* (and such calls may also be nested arbitrarily). Transactional calls are treated differently at run time. After a transactional call is made, the first time any part of shared memory is used in that transaction, it is copied into a *fixed* local copy on the stack frame for that transaction. A separate, *mutable*, copy (valuation) of shared variables is also kept on the transactional stack frame. All read/write accesses (after the first use) of shared memory inside the transaction are made to the mutable copy on the stack, rather than to the universal copy. Each transaction keeps track (on its stack frame) of those shared memory variables that have been *used* or *written* during the execution of the transaction. Finally, if and when the transaction terminates, we use an atomic *compare-and-swap* operation to check that the current values in (the used part of) the universal copy of shared memory are exactly the same as their fixed copy on the stack frame, and if so, we copy the new values of *written* parts of shared memory from the mutable copy on the stack frame to their universal copy. Otherwise, i.e., if the universal copy of used shared memory is inconsistent with the fixed copy for that transaction, we have detected a memory conflict and we abort that transaction.

The key point is this: if the compare-and-swap operation at the end of a transaction succeeds and the transaction is not aborted, then we can in fact "schedule" the entire activity of that transaction inside the "infinitesimal time slot" during which the atomic compare-and-swap operation was scheduled. In other words, there exists a serial schedule for non-aborted transactions, which does not interleave the operations of distinct non-aborted transactions with each other. This allows us to establish the *AAS-serializability* property for TSMs. The above description is over-simplified because, e.g., TSMs also allow nested transactions and there are other technicalities, but it does describe some key aspects of the model. Please see our full paper [1] for details.

Examples. Figure 1 contains two example transactional programs which illustrate basic points. The example on the left illustrates how "single-lock" (SL)

Initially, x == 0

Process 1	Process 2
atomic { x := 1; }	atomic { while (true) do { nothing} }

Can (x==0) always hold true? No.

Initially, x == y == 0

Process 1	Process 2
r := 1; atomic { x := 1; if (y == 0) {abort;} r := 0; } atomic { if (r == 1) { x := 2;} }	atomic { y := x; }

(y == 2) is possible. Not (y==1)!

Fig. 1. Examples

semantics can not capture the non-blocking nature of transactional programming. Process 2 in this example is a non-terminating transaction which does nothing. Thus, under SL semantics, this process could obtain the lock and hold it forever, preventing Process 1 from executing. Thus, under SL semantics variable x may never attain value 1, even when the two processes are fairly scheduled, because the first process remains blocked forever. However, this directly violates the intended non-blocking nature of the transactional paradigm. A large amount of work in software transactional memory (STM) has been about assuring that such blocking can not occur. If the semantics fails to capture this, then reasoning about blocking aspects of the higher-level transactional program becomes difficult. The TSM model is non-blocking and will allow execution of Process 1. Therefore, as long as the two processes are fairly scheduled (a standard assumption for a concurrent system) `x` $==$ 1 will eventually hold, unlike under SL semantics. The example on the right illustrates that even aborted transactions have visible "side effects", and that therefore the meaning of aborts (both explicit and automatic) has to be specified in the programmer's view of semantics. In this example `r` is a local variable of process 1. Regardless of how these transactions are executed, the first transaction on Process 1 will abort. Nevertheless, it will have a "side effect", because the value of the local variable r will remain 1, whereas if the first transaction had committed it would be set to 0. Consequently, the second transaction of Process 1 sets $x := 2;$. Thus, when process 2 reads `x`, it can read 2. In SL semantics, aborts have no meaning: once the lock is obtained the transaction executes until completion, or executes forever. It is not even clear what meaning SL semantics would ascribe to such a program. For instance, if an "abort" is considered a "retry", Process 1 would block forever attempting to execute its first transactions, again a violation of the non-blocking nature of transactions.

References

1. K. Etessami and P. Godefroid. An abort-aware model of transactional programming. Technical report, Microsoft Research, 2008.
2. J. Larus and R. Rajwar. *Transactional Memory.* Morgan & Claypool, 2007.