An Abort-Aware Model of Transactional Programming

Kousha Etessami*[1] Patrice Godefroid[2]

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

Abstract. 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. We introduce *Transactional State Machines* (TSMs) as an abstract finite-data model of transactional shared-memory concurrent programs. TSMs are a variant of concurrent boolean programs (or concurrent extended recursive state machines) augmented with additional constructs for specifying potentially nested transactions. Namely, some procedures (or code segments) can be marked as *transactions* and are meant to be executed "atomically", and there are also explicit *commit* and *abort* operations for transactions. 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 transactions which may be aborted explicitly, or aborted automatically by the run-time environment due to memory conflicts.
We show that concurrent executions of TSMs satisfy a correctness criterion closely related to serializability, which we call *stutter-serializability*, with respect to shared memory. We initiate a study of model checking problems for TSMs. Model checking arbitrary TSMs is easily seen to be undecidable, but we show it is decidable in the following case: 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.

1 Introduction

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

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

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., [17] 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, the latter being usually the main concern of the transactional-memory (TM) research community. When formal semantics is discussed, it is usually to offer an abstract characterization of some specific low-level TM implementation details: such semantics are *distinguishing low-level semantics* in the sense that they typically distinguish some newly proposed implementation from all other previous implementations. Even if transactional constructs were themselves given clear semantics, there would remain the important task of verifying specific properties of specific transactional programs.

The aim of this paper is to provide a state-machine based formal model of transactional concurrent programs, and thus to facilitate an abstract framework for reasoning about them. In order for such a model to be useful, firstly, it must be close enough to existing transactional paradigms so that, in principle, such models could be derived from actual transactional programs via a process of abstraction akin to that for ordinary programs. Secondly, the model should be simple enough to enable (automated) reasoning about such programs. Thirdly, the model should be abstract enough to allow verification of properties of transactional programs independently of any specific TM implementation; the model should thus capture a *unifying high-level semantics* formalizing the view of transactional programmers (unlike most distinguishing low-level semantics discussed in the TM research literature, which represent views of TM implementers).

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`. (Simple examples of transactional concurrent shared-memory programs are given in Figure 1. These examples will be discussed later.) But what is the semantics? The most common unifying high-level semantics is the so-called "*single-lock semantics*" (see, e.g., [17]), 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 transactions occurring on 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 by "as if". A semantics which literally assumes that every concurrent execution proceeds via a single lock, rules out any interleaving of transactions on different processes. It also violates the intended

non-blocking nature of the transactional paradigm, and ignores other features, such as the fact that transactions may not terminate, and that typically transactions can be *aborted* either explicitly by the program or automatically by the run-time system due to memory conflicts.

Of course, designers of transactional frameworks would object to this literal interpretation of "as if". Rather, a weaker semantics is intended, but phrasing a simple unifying high-level formal semantics which captures precisely what is desired and leaves sufficient flexibility for an efficient implementation is itself 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. This is because in full-fledged concurrent programming it is no longer the case that every operation on memory is done via a transaction consisting of a block of (necessarily terminating) straight-line code. The "transactional program" running on each process may consist of a mix of transactional and non-transactional code, transactions may be nested, and moreover some transactions (which are programs themselves) may never halt. When adapting correctness criteria to this setting, one needs to take careful account of all these subtle differences.

The key role that aborts play in transactional programming should not be underestimated. Consider a transactional program for reserving a seat on a flight. The program starts a transaction, reads shared memory to see if seats are available and if so, attempts to write in shared memory to reserve a specific seat. If the flight is full or if there is a runtime memory conflict to reserve that specific seat, the transaction must be aborted, and the transactional program must be notified of this abort in order to take appropriate recovery actions. In particular, always forcing each abort to trigger a retry is not a viable option in practice (if the flight is full there is no point retrying forever to book a seat on that flight). So there *must* be some abort mechanism, either through explicit aborts or automatic aborts (or both), which is *not* equivalent to a retry. In other words, those aborts *must be visible* to the transactional programmer and therefore they *must be given a semantics*. As another example, consider transactional programs operating under stringent timing constraints. The programmer may not wish to do arbitrarily many retries after an automatic abort, depending on the current program state. We emphasize these points because earlier feedback we have received suggests that some people in the TM community believe it is adequate to provide the transactional programmer with a high-level semantic model (e.g., single-lock semantics) which does not at all expose them to the possibility of aborts. We believe this is an oversimplification that will only lead to greater confusion for programmers.

In this paper, 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 and transactions which may never terminate.

Using TSMs as a formalization vehicle, we propose a new *abort-aware* unifying high-level semantics which extends the traditional single-lock semantics by allowing the modeling of transactions aborted either explicitly in the program or automatically by the underlying TM implementation. Our abort-aware semantics exposes both explicit and automatic aborts, but it can easily be adjusted to treat automatic aborts as retries.

We define *stutter-serializability*, which we feel captures in a clean and simple way a desired correctness criterion, namely *serializability with respect to committed transactions*, which is (trivially) enjoyed by the single-lock semantics (since no transactions ever abort). We show that our abort-aware TSM semantics preserves this property, while also accommodating aborted transactions.

Finally, 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. This decidability result also holds for several other variants of the abort-aware TSM semantics.

2 Overview of the Abort-Aware TSM Semantics

Our abort-aware TSM semantics 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 an atomic (hardware or software implemented) 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 two assumptions, we can now give an informal description of the abort-aware TSM semantics. 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

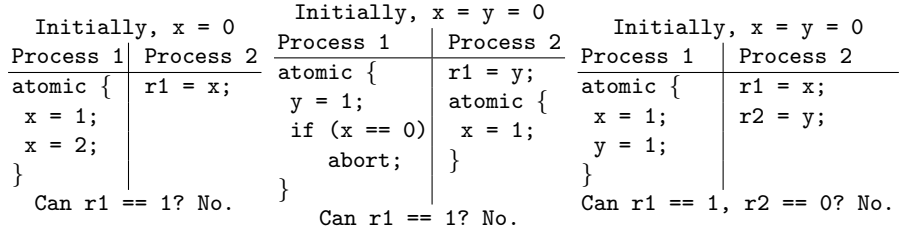

Fig. 1. Examples

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 *stutter-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. We describe the full model in detail in Section 3 and the appendix (due to space constraints). We show that TSMs are stutter-serializable in Section 4. We study model checking for TSMs in Section 5, and show that, although model checking for general TSMs is undecidable, there is an interesting fragment for which it remains decidable.

Examples. Figure 1 contains simple example transactional programs (adapted from [12]). Transactions are syntactically defined using the keyword `atomic`. With each example, we describe the possible effect, in our TSM model, on the variables r_1 (and r_2) at the end of the example's execution. As mentioned, in the TSM model the execution of transactions on multiple processes can interleave, and moreover the execution of transactional and non-transactional code can also interleave. So, in the leftmost example, what happens if the non-transactional code executed by Process 2 executes before the transaction on Process 1 has

completed? In the TSM model, Process 2 would read the value of the shared variable `x` from a *universal copy* of shared memory which has not yet been touched by the executing transaction on Process 1. If Process 1 completes its transaction and commits successfully, then the final value 2 is written to this universal copy of x, and thereafter Process 2 could read this copy and thus it is possible that $r1 == 2$ after this program has finished. However, $r1 == 1$ is not possible. We note that $r1 == 1$ would be possible at the end under forms of *weak atomicity*, e.g., if `atomic` was implemented as a *synchronized* block in Java (see [12]). The middle example in Figure 1 contains an explicit abort. In the TSM model, all write operations on shared variables performed by a transaction only have an effect on (the universal copy of) shared memory if the transaction successfully commits. Otherwise they have no effect, and are not visible to anyone after the transaction has been aborted. Thus $r1 == 1$ is not possible at the end of this program. This is a form of *deferred update* as opposed to *direct update* ([17]), where writes in an aborted transaction do take effect, but the abort overwrites them with the original values. In that case, such a write might be visible to non-transactional code and `r1` might have the value 1 at the end of execution of this example. Note that our semantics for TSMs does not take into account possible re-orderings that may be performed by standard compilers or architectures. For instance, compilers are usually allowed to re-order read operations, such as those performed by Process 2 in the rightmost example in Figure 1. Such reordering issues [12] are not addressed in this paper. One could extend TSMs to incorporate notions of reordering in the model, but we feel that would complicate the model too much and detract from our main goal of having a clean abstract reference model which brings to light the salient aspects of transactional concurrent programs.

3 Definition of Transactional State Machines

In this section we define *Transactional State Machines* (TSMs). The definition resembles that of (concurrent) boolean programs and (concurrent) extended recursive state machines (see, e.g., [5, 2, 3]), but with additional constructs for transactions. Our definition will use some standard notions (e.g., *valuations* of variables, expressions, types, etc.) which are defined formally in appendix A.1.

3.1 Syntax of TSMs

A *Transactional State Machine* $\mathcal{A}$ is a tuple $\mathcal{A} = \langle S, \sigma_{init}, (P_r)_{r=1}^{n} \rangle$, where S is a set of *shared* variables, σ_{init} is an initial valuation of S, and $P_1, \ldots, P_n$, are processes. Each process is given by $P_r = (L_r, \gamma^r_{init}, p_r, (A^r_i)_{i=1}^{k_r})$ where L_r is a finite set of (non-shared) *thread-local*[1] variables for process r, γ^r_{init} is an initial valuation of L_r, $p_r \in \{1, \ldots, k_r\}$ specifies the index of the *initial (main) procedure*,

[1] We note that these thread-local variables are used by all procedures running on the process. For simplicity, we do not include procedure-local variables, and we assume procedures take no parameter values and pass no return values. This is done only for clarity, and we lose nothing essential by making this simplification.

$A^r_{p_r}$, for process r (where runs of that process begin). The A^r_i's are the *procedures* (or *components* in the RSM terminology) for process r. We assume that the first d_r of these are *ordinary* and the remaining $k_r - d_r$ are *transactional* procedures. The two types of procedures have a very similar syntax, with the slight difference that transactional procedures have access to an additional *abort* node, ab_i. Specifically, each procedure A^r_i is formally given by: $\langle N^r_i, en^r_i, ex^r_i, ab^r_i, \delta^r_i \rangle$, whose parts we now describe (for less cluttered notation, we omit the process superscript, r, when it is clear from the context):

- a finite set N_i of nodes (which are control locations in the procedure)
- special nodes: $en_i, ex_i \in N_i$, known respectively as the *entry node* and *exit node*, and (only for transactional components) also an *abort node* $ab_i \in N_i$.
- A set δ_i of *edges*, where an edge can be one of two forms:
  - *Internal edge:* A tuple (u, v, g, α). Here u and v are nodes in N_i, $g \in \mathit{BoolExp}(S \cup L_r)$ is a *guard*, given by a boolean expression over variables from $S \cup L_r$ (see appendix A.1). $\alpha \in \mathit{Assign}(S \cup L_r)$ is a (possibly simultaneous) *assignment* over these variables (again, see appendix A.1 for definition of assignments). We assume that u is neither ex_i nor ab_i (because there are no edges out of the exit or abort nodes), and that v is not the entry node en_i. Intuitively, the above edge defines a possible transition that can be applied if the guard g is true, and if it is applied the simultaneous assignments are applied to all variables (all done atomically), and the local control node (i.e., program counter) changed from u to v. The set of internal edges in procedure A_i is denoted by δ^I_i.
  - *Call edge:* A tuple (u, v, g, c). u and v are nodes in N_i, $g \in \mathit{BoolExp}(S \cup L_r)$ is a guard, $c \in \{1, 2, \ldots, k\}$ is the index of the procedure being *called*. Again, we assume $u \notin \{ex_i \cup ab_i\}$, and $v \neq en_i$. Calls are either *transactional* or *ordinary*, depending on whether the component A_c that is called is transactional or not (i.e., whether $c > d_r$ or $c \leq d_r$). Intuitively, a call edge defines a possible transition that can be taken when its guard g is true, and the transition involved calling procedure A_c (which of course involves appropriate call stack manipulation, as we'll see). Upon returning (if at all) from the call to A_c, control resumes at control node v. The set of call edges in component i is denoted by δ^C_i.

3.2 Abort-Aware Semantics of TSMs

A full formal semantics of TSMs is given in Appendix A.2 (due to space constraints). Here we give an informal description to facilitate intuition and describe salient features. TSMs model concurrent shared memory imperative procedural programs with bounded data and transactions. A *configuration* of an TSM consists of a call stack for each of the r processes, a current node (the program counter) for each process, as well as a *universal valuation* (or *universal copy*), $\mathcal{U}$, of shared variables. Crucially, during execution the "view" of shared variables may be different for different processes that are inside transactions. In particular, different processes, when executing inside transactions, will have their own

local copies (valuations) of shared variables on their call stack, and will evaluate and manipulate those valuations in the middle of transactions, rather than the single universal copy.

A transaction keeps track (on the stack) of what shared variables have been *used* and *written*. If a shared variable is written by one of the processes inside the scope of one of the transactions, the universal copy $\mathcal{U}$ is not modified. Instead, an in-scope *mutable* copy of that shared variable is modified. The *mutable* copy resides on the stack frame of the innermost transaction on the call stack for that process. The first time a shared variable is read (i.e., used) inside a transaction, unless it was already written to in the transaction, its value is copied from $\mathcal{U}$ to a *fixed* local copy for that transaction and also to a separate *mutable* local copy, both on the stack. Thereafter, both reads and writes inside the transaction will be to this mutable copy.

At the end of a transaction, the written values will either be committed or aborted. The transaction is automatically aborted if a shared memory conflict has arisen, which is checked using an atomic compare-and-swap operation as follows. We compare the values of variables in the fixed copy of shared memory with the values of those same shared variables in the universal copy $\mathcal{U}$, and if these are all equal, then for the *written* variables, we copy their valuations in the mutable copy on the stack frame to the universal copy $\mathcal{U}$. If, on the other hand, the *compare-and-swap* fails, i.e., the compared values are not all equal then we *abort* the transaction, discard any updates to shared variables, pop the transactional stack frame and restore the calling context. (How this all works is described in detail in appendix A.2).

If we have nested transactions, and values are committed inside an inner nested transaction, then their effect will only be immediately visible in the next outer nested transaction (i.e., this follows the semantics of *closed* nested transactions), and the committed values will only be placed in the mutable copy of shared variables of the next outer transaction. Otherwise, if the inner transaction aborts, then its effect on shared variables is discarded before control returns to the calling context.

Again, see appendix A.2 for detailed semantics. We highlight here some other salient features of the semantics which will be pertinent in other discussions:

- The universal valuation $\mathcal{U}$ is only updated upon a successful commit of outermost (non-nested) transactions, not of inner (nested) transactions.
- There are two distinct ways in which an abort can occur. One is an explicit abort, which occurs if a transaction reaches a designated abort node. The other is an automatic abort, carried out by the memory system due to conflicts with universal memory. (For nested transactions, the only possible abort is an explicit one, because no conflict is possible.)

4 Correctness: Stutter-Serializability

In this section we discuss a correctness property that TSMs possess. Informally, the correctness property relates to "atomicity" and serializability of transactions,

but such notions have to be defined carefully with respect to the model. What we wish to establish is the following fact: if there exists any run π of a TSM which witnesses a (possibly infinite) sequence of changes to the universal copy (valuation) of memory, there must also exist a run π' which witnesses exactly the same sequence of changes to the universal copy, but such that all transactions which start *and which do not abort and do terminate* in π' must execute entirely serially without any interleaving of steps on other processes in the execution of the terminating transaction. Formally, this requires us to consider *stutter-invariant* temporal properties over atomic predicates that depend only on the universal valuation of shared variables in a state of the TSM, and stutter-equivalence (see appendix section A.3 for definitions).

We say a run ρ of a TSM contains *only serialized successful transactions* if every transaction on any process in the run ρ that starts and successfully commits, executes serially without any interleaving of steps by other processes. In other words, the entire execution of each successful transaction occupies some contiguous sequence $\rho_i\rho_{i+1}\ldots\rho_{i+m}$ in the run. For a run ρ of $\mathcal{A}$, let $\rho[\mathcal{U}]$ denote a new word, over the alphabet of shared variable valuations, such that $\rho[\mathcal{U}]$ is obtained from ρ by retaining only the universal valuation of shared variables at every position of the run (i.e., replacing each state ρ by the universal valuation in that state). We say that a TSM, $\mathcal{A}$, is *stutter-serializable* if: for every run ρ of $\mathcal{A}$, there exists a (possibly different) run ρ' of $\mathcal{A}$ such that $\rho[\mathcal{U}]$ is stutter-equivalent to $\rho'[\mathcal{U}]$, and such that ρ' contains only serialized successful transactions.

Theorem 1. *All TSMs are stutter-serializable.*

Proof. (*Sketch)* A proof is in the appendix. Here we sketch the basic intuition. If at the end of a non-nested transaction which is about to attempt to commit, the atomic compare-and-swap operation succeeds, then at exactly the point in "time" when the compare-and-swap operation executed, the values in the universal copy of shared variables used inside the transaction are exactly the same as the values that were read from the universal copy the first time these variables were encountered in the transaction. Each shared variable is read from the universal copy at most once inside any transaction. All subsequent accesses to shared variables are to the local mutable copy on the transactional stack frame. Consequently, since the values of shared variables are the only input to the transaction from its "environment" (i.e., from other processes), the entire execution of that transaction can be "delayed" and "rescheduled" in the same "infinitesimal time slot" just before the atomic compare-and-swap operation occurred, and the resulting effect of the transaction on the universal copy of memory after it commits would be identical (because it would have identical input). The only visible effect on the universal copy of memory during the run that this rescheduling has is that of adding or removing "stuttering" steps, because the rescheduled steps do not change values in the universal copy of shared memory. □

Note that TSMs can reach new states due to transactions being aborted by the run-time environment due to memory conflicts. In other words, even aborted transactions have side effects. For instance, a TSM can use a thread-local

variable to test/detect that its last (possibly nested) transaction was aborted, and take appropriate measures accordingly, including reaching new states that are reachable only following such an abort. This fact does not contradict the above correctness assertion about TSMs, because the correctness assertion does not rule out the possibility that in order for a certain feasible sequence of changes to universal memory $\mathcal{U}$ to be realized some transactions might necessarily have to abort during the run. In general, it does not seem possible to devise a reasonable model of imperative-style transactional programs where transactions that are aborted will have no side effects. Anyway, there are good reasons not to want this. One useful consequence of side effects is that one can easily implement a "retry" mechanism in TSMs which repeatedly tries to execute the transaction until it succeeds. Some transactional memory implementations offer "retry" as a separate construct (see [17]).

5 Model Checking

It can be easily observed (via arguments similar to, e.g., [22]) that model checking for general TSMs, even with 2 processes, is at least as hard as checking whether the intersection of two context-free languages is empty. We thus have:

Proposition 1. *Model checking arbitrary TSMs, even those with 2 processes, even against stutter-invariant LTL properties of shared memory is undecidable.*

On the other hand, we show next that there is an interesting class of TSMs for which model checking remains decidable. Let the class of *top-transactional* TSMs be those TSMs with the property that the *initial* (*main*) procedure for every process makes only transactional calls (but inside transactions we can execute arbitrary recursive procedures). Let us call a TSM *almost-top-transactional* if one process is entirely unrestricted, but all other processes must have main procedures which make only transactional calls, just as in the prior definition.

Theorem 2. *The model checking problem for almost-top-transactional TSMs against all stutter-invariant linear-time (LTL or ω-regular) properties of (universal) shared memory is decidable.*

Proof. Given a TSM, $\mathcal{A}$, our first task will be to compute the following information. For each process r (other than the one possible process, r', which does not have the property that all calls in its main procedure are transactional) we will compute, for every transactional procedure, A_c on process r, certain *generalized summary paths*. A *generalized summary path* (GSP) for a transactional procedure A_c is a tuple $G = (\gamma_{start}, R, \gamma_{finish}, status, \sigma)$. γ_{start} and γ_{finish} are valuations of the thread-local variables L_c. *status* is a flag that can have either the value `commit` or `abort`. σ is a *partial valuation* of shared variables, meaning it is a set of pair (x, w) where x is a shared variable and w is a value in x's domain (and there is at most one such pair in σ for every shared variable x). $R = R_1, \ldots, R_d$ is a sequence of distinct partial valuations of shared variables,

where furthermore, different R_i's do not evaluate the same variable. In other words, for each shared variable x, there is at most one pair of the form (x, w) in the entire sequence R. Such a sequence R yields a partial valuation $\sigma_R = \cup_{i=1}^{d} R_i$ (and we shall need to refer both to the sequence R and to σ_R).

We now define what it means for a GSP, G, to be *valid* for the transactional procedure A_c. Informally, this means that G summarizes one possible *terminating* behavior of the transaction A_c if it is run in sequential isolation (with no other process running). More formally, we call a GSP, $G = (\gamma_{start}, R, \gamma_{finish}, status, \sigma)$, *valid* for the transactional procedure A_c, if it satisfies the following property. Suppose a call to A_c is executed in sequential isolation (i.e., with no other process running). Suppose, furthermore that in the starting state ψ_0 in which this call is made γ_{start} is the valuation of thread-local variables L_r on process r, and that the universal copy of shared memory $\mathcal{U}$ is *consistent* with the partial valuation σ_R (in other words it agrees with σ_R on all variables evaluated in σ_R). Then there exists some sequential run of A_c from such a start state ψ_0 where during this run:

1. The sequence of reads of the universal copy of shared memory variables executed during the run corresponds precisely to the d partial valuations $R_1, \ldots, R_d$. For example, if $R_3 = \{(x_1, w_1), (x_2, w_2)\}$, then the third time during the run in which the universal copy of shared memory is accessed (i.e., third time when shared variables are used that have not been used or written before) requires a simultaneous read[2] of shared variables x_1 and x_2 from the universal copy $\mathcal{U}$, and clearly the values read will be w_1 and w_2, because $\mathcal{U}$ is by definition consistent with σ_R. (Note that $\mathcal{U}$ does not change in the middle of the sequential execution of A_c, because it is run in sequential isolation, with no other process running.)
2. After these sequences of reads, the run of A_c terminates in a state where the valuation of local variables is γ_{finish} and either commits or aborts, consistent with the value of *status*.
3. Moreover, if it does commit, then the partial valuation of shared variables that it writes to the universal copy $\mathcal{U}$ (via *compare-and-swap*) at the commit point is σ. (And otherwise, σ is by default the empty valuation.)

Let $\mathcal{G}_c$ denote the set of all valid GSPs for transactional procedure A_c. It is clear that for any transactional procedure, every GSP G is a finite piece of data, and furthermore that there are only finitely many GSPs. This is because the universal valuation of every shared variable can be read at most once during the life of the transaction, and of course there are only finitely many variables, and each variable can have only finitely many distinct values.

Lemma 1. *The set $\mathcal{G}_c$ is computable for every transactional procedure A_c.*

[2] Again, recall that the reason there may be simultaneous reads from universal shared variables is an artifact of the strong isolation assumption combined with our formulation of (potentially simultaneous) assignment statements.

See appendix A.5 for a proof of the Lemma. We shall compute the set $\mathcal{G}_c$ for every transactional procedure A_c and use this information to construct a finite-state summary state-machine B_r, for every process r, which summarized that process's behavior. We will also describe the behavior of the single unrestricted process r' using a a Recursive State Machine (RSM), $B'_{r'}$. We shall then use these B_r's and $B'_{r'}$ to construct a new RSM $B = (\otimes_{r \neq r'} B_r) \otimes B'_{r'}$ which is an appropriate *asynchronous product* of all the B_r's and $B'_{r'}$. The RSM B essentially summarizes (up to stutter-equivalence) the behavior of the entire TSM with respect to shared memory. The construction of the B_r's, $B'_{r'}$, and B is described in appendix A.6.

It follows from the construction that B has the following properties. For every run ρ of the entire TSM, $\mathcal{A}$, there is a a run π of B such that π is stutter-equivalent to the restriction $\rho[\mathcal{U}]$ of the run ρ to its sequence of universal shared memory valuations. And likewise, for every run π of B, there is a run ρ of $\mathcal{A}$ such that $\rho[\mathcal{U}]$ is stutter-equivalent to π. (Again, see A.3 for definitions pertaining to stutter-equivalence.) Thus, once the RSM B is constructed, we can use the model checking algorithm for RSMs ([2]) on B to check any given stutter-invariant LTL, or stutter-invariant ω-regular, property of universal shared memory of $\mathcal{A}$. □

We remark that the complexity of model checking can be shown to be singly-exponential in the encoding size of the TSM, under a natural encoding of TSMs. (Note that TSMs are compactly encoded: they are *extended* concurrent recursive state machines, with variables that range over bounded domains.)

Finally, we note that a similar decidability result can be obtained with other variant semantics where (1) automatic aborts are systematically considered as retries, (2) terminating transactions nondeterministically commit or abort, or (3) never more than one transaction executes concurrently (this is equivalent to the single-lock semantics). Indeed, those variant semantics are simpler to define and can be viewed as particular cases of the abort-aware TSM semantics.

6 Related Work

There is an extensive literature on Transactional Memory and there are already many prototype implementations (see the online bibliography [8], and see the recent book by Larus and Rajwar [17]). Most of this work discusses how to implement transactional memory either in hardware or software, from a systems point of view with the main emphasis on performance. Some researchers have formalized and studied the semantics of transactional memory implementations, in order to clarify subtle semantics distinctions between various implementations and the interface between these implementations and higher-level "transactional programs" running on top of them. Such distinguishing low-level semantics are quite complicated, and are not suitable for higher-level transactional program verification.

Recent work [19, 1] discuss transaction semantics in the difficult setting of *weak* isolation/atomicity, where implementations do not detect conflicting accesses to shared memory between non-transactional and transactional code, and

thus these may interfere unpredictably. By contrast, we assume a form of strong isolation, as described earlier. We aim for a clean model that can highlight the issues which are specific to transactions, and we do not want to obfuscate them with difficult issues that arise by introducing weak memory models, weak consistency, out-of-order execution, and weak isolation. Such notions are somewhat orthogonal, and are problematic semantically even in settings without transactions. Our goal is to define an abstract, idealized, yet relevant, model of transactional programming that could in principle serve as a foundation for verification. There are various design choices in the implementation of a transactional memory framework (see [17] for a taxonomy of choices), and our TSM model reflects several such choices. For instance, our definition of nested transactions is a form of *closed* nested transactions. We do not consider so-called *open* nested transactions, where an inner transaction may commit while an outer one aborts (because we can not see any sensible semantics for them, even in the single-process purely sequential setting). Some of these choices are adjustable in the model, as discussed in the previous section.

Independently, [13] has recently proposed the notion of "opacity" as an alternative semantics criterion for transactions. Loosely speaking, opacity also requires serializability of aborted transactions in addition to serializability of committed transactions, with the goal of preventing aborted transactions from reading "inconsistent" values. In contrast, our abort-aware semantics does not require the stronger opacity criterion. Instead, it assumes that programmers can deal with automatically aborted transactions as they currently handle runtime exceptions in other programming languages. Of course, opacity could be formalized using an alternate TSM semantics.

Mechanisms other than transactions, such as locks, have been proposed to enforce "atomicity" and have been studied from a verification point of view. For instance, concurrent reactive programs where processes synchronize with locks were studied in [21] where a custom procedure exploiting "atomicity" (based on Lipton's reduction) is used to simplify the computation of "summaries" for such programs. Also, several verification problems are shown to be decidable in [15] for a restricted class of programs where locks are nested. Several other restrictions of concurrent pushdown processes for which verification problems are decidable have also been identified (e.g., [9], among others). There are some high-level similarity between these prior results and our results in Section 5, but the details are substantially different due to the specifics of the TSM model.

Other related work discusses how to check the correctness of implementations of transactional memory, based on lower level constructs, using testing ([18]) or model checking ([10]). By contrast, we do not address the problem of analyzing the correctness of implementations of transactional memory, but rather the correctness of transactional programs running on top of (correct) implementations.

Notions of serializability have been studied in database concurrency control for decades ([6]). However, there are subtle distinctions between the semantics of serializability in different setting. [4] systematically studied automata-based formalization of serializability and other related concepts. We formulate a clean

and natural notion of *stutter-serializability* for TSMs, and show it is satisfied by them. The notion arose from our considerations of the abort-aware TSM model, and does not appear to have been studied before in the literature.

7 Conclusions

This work initiates a study of transactional programming from a program analysis and verification point of view. Our goal is to provide a formal foundation for high-level reasoning about transactional programs, which nevertheless does not ignore the meaning of manual aborts nor automatic aborts in such programs, and facilitates building program analysis and verification tools for transactional programs. In contrast with prior semantics work on transactional memory systems, we do not consider the (lower-level) verification of transactional-memory implementations but instead focus on the (higher-level) abstract semantics of transactional programs running on top of those implementations. The paper makes two main contributions.

- We propose Transactional State Machines as an abstract finite-data model for transactional programs. TSMs are essentially concurrent extended recursive state machines augmented with constructs to specify transactions. Their significant expressiveness allows the modeling of interleaved executions of concurrent and potentially nested and/or non-terminating transactions. However, we show that, provided recursion is confined to occurring inside transactions, the expressiveness of TSMs is reduced and model checking of a large class of properties becomes decidable.
- We offer a critique of the current dominant high-level semantics for transactional programming, namely the single-lock semantics, and extend it with an alternative abort-aware semantics which captures important features of real transactional programs such as explicit and automatic aborts. We identify stutter-serializability as a key formal property (enjoyed, e.g., under single-lock semantics), and we show that our abort-aware semantics still enjoys this property and provides a clean and precise high-level semantics also for explicit and automatic aborts.

TSMs are concurrent state machines so it is natural to study them under fairness assumptions that insure progress on all processes. Note that for model checking, such fairness assumptions can be specified within LTL specifications.

Acknowledgements: We thank Jim Larus for several helpful discussions and encouragements, and Martin Abadi, Tom Ball, Sebastian Burckhardt, Dave Detlefs, Tim Harris, Madan Musuvathi, Shaz Qadeer and Serdar Tasiran for helpful comments.

References

1. M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *Proceedings of POPL'08*, 2008.

2. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.
3. R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. In *TACAS*, pages 61–76, 2005.
4. R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.
5. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN'2000*, volume 1885 of *LNCS*, pages 113–130, 2000.
6. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and Recovery in Database Systems*. Addison-Wesley, 1987.
7. C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.
8. J. Bobba, R. Rajwar, and M. Hill (editors). Transactional memory bibliography (online). `http://www.cs.wisc.edu/trans-memory/biblio/index.html`.
9. A. Bouajjani, J. Esparza, and T. Touili. A Generic Approach to the Static Analysis of Concurrent Programs with Procedures. In *Proceedings of POPL'03*, 2003.
10. A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying Correctness of Transactional Memories. In *Proceedings of FMCAD'2007 (Formal Methods in Computer-Aided Design)*, 2007.
11. K. Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In *Proceedings of the 11th International Conference on Computer Aided Verification*, pages 236–248, 1999.
12. D. Grossman, J. Manson, and W. Pugh. What Do High-Level Memory Models Mean for Transactions? In *Memory System Performance and Correctness (MSPC'06)*, pages 62–69, 2006.
13. R. Guerraoui and M. Kapalka, On the correctness of transactional memory., In *Proc. 13th ACM PPoPP*, pages 175–184, 2008.
14. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. of 22nd Symp. on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.
15. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning About Threads Communicating via Locks. In *Proceedings of CAV'05*, 2005.
16. L. Lamport. What good is temporal logic. In R. E. A. Mason, editor, *Information Processing '83: Proc. IFIP 9th World Computer Congress*, pages 657–668, 1983.
17. J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2007.
18. C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, and K. Olukotun. Testing Implementations of Transactional Memory. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, 2007.
19. K. F. Moore and D. Grossman. High-Level Small-Step Operational Semantics for Transactions. In *Proceedings of POPL'08*, 2008.
20. D. Peled and Th. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63:243–246, 1997.
21. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing Procedures in Concurrent Programs. In *Proceedings of POPL'04*, 2004.
22. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.
23. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

A Appendix

A.1 Basic definitions used in the TSM model

We require some preliminary definitions. Let V denote a set of variables. Each variable, $v \in V$ has a type, $T(v) \in Types$. Each type, $t \in Types$, is associated with a domain, $D(t)$ (which may in general be infinite, but for analysis purposes we will assume it to be finite for all variables). In particular, we allow a *boolean* type, $Bool$, with domain $D(Bool) = \{T, F\}$. Let $Expr$ denote a set of *expressions*, which we assume are built up from variables, using operators and combinators. In particular, all variables are expressions. Each expression $expr \in Expr$ also has a type $T(expr)$. For example, a boolean expressions, $bexpr$, is any expression which has type $T(bexpr) = Bool$. We let $BoolExp(V)$ denotes the set of boolean expressions over variables V. The expressions of various types are typically given by grammars which describe how well-formed expressions are built up from variables of various types and using appropriate operators and constants. But we leave these details unspecified. We shall call the variables that appear in an expression its *free variables*. For a set of expressions $Expr'$, and a set of variables V', we let $Expr'(V') \subseteq Expr'$ denote those expressions whose free variables are a subset of V'. For example, for an integer variable x, the following is a possible boolean expression: $(x \geq 5) \vee (x < -3)$. It has one free integer variable, x.

A *valuation* (or *interpretation*) of V is a map, $\sigma : v \in V \mapsto d \in D(T(v))$. In other words, σ maps every variable $v \in V$ of type t to an element of the domain $D(t)$. It is assumed that every valuation σ of the free variables V' in a set of expressions $Expr(V')$ uniquely extends to a valuation for the expressions, i.e., to a map $\sigma : expr \in Expr(V') \mapsto d \in D(T(expr))$.

An *assignment* over V has the form $[x_1, x_2, \ldots, x_k] := [exp_1, exp_2, \ldots, exp_k]$, where $x_j \in V$ are distinct variables, and for all j, $exp_j \in Expr(T(x_j))$. The semantics of assignments are defined over pairs (σ_1, σ_2) of interpretations of V. Given an assignment α of the above form, we use the notation $\sigma_2 := \alpha(\sigma_1)$ to denote that (1) $\sigma_2(x_j) = \sigma_1(exp_j)$ for all x_j, and (2) $\sigma_1(y) = \sigma_2(y)$ for all other variables $y \in V \setminus \{x_1, x_2, \ldots, x_k\}$. In other words, apply assignment α to valuation σ_1 yields the new valuation σ_2.

A.2 Semantics of TSMs

A *configuration* (i.e., state), ψ, of a TSM, $\mathcal{A}$, is a tuple $\psi = \langle \mathcal{U}, (stack_j, u_j, \gamma_j)_{j=1}^n \rangle$, where $\mathcal{U}$, called the *universal copy* of shared memory, is a valuation of the set S of shared variables. For each j, $stack_j$ is a stack, i.e., a (possibly empty) list of "stack frames". The contents of stack frames is described in detail later. u_j is a node in some procedure of process j, which denotes the current control location (i.e., the program counter). We shall call this the *current node* of process j in configuration ψ. γ_j is a valuation of the thread-local variables L_j on process j. The *initial configuration* of $\mathcal{A}$ is $\psi_0 = \langle \sigma_{init}, (\emptyset, en^r_{p_r}, \gamma^r_{init})_{r=1}^n \rangle$. This starts each process at the entry point, $en^r_{p_r}$ of its initial ("main") procedure, $A^r_{p_r}$, with an

empty call stack, and initializes the (universal) values of shared variables, and initializes the thread-local variables of each process. We shall now describe the global transition relation of a TSM by describing the effect of each kind of possible transition on the current configuration, including the stacks. When the TSM is in a given configuration (i.e., at a given state), a number of possible transitions can be *enabled*. In particular, a transition can be enabled because the *guard*, g, of some (internal or call) edge, e, of the TSM holds true in that configuration. We can categorize transitions basically into various different types: *Calls*, *Internal transitions*, (successful) *Returns* (which are *Commits*, in case the procedure is transactional), and (explicit or automatic) *Aborts*. We will consider each possible type of transition separately.

In order to simplify our notation when describing these, we have to make a natural adjustment to the interpretation of guards g and assignments α. Namely, guards (and assignments) define functions of valuations of variables, but which copies of those variables? Recall that there are potentially multiple copies of the same shared variable, x, including the universal copy as well as fixed/mutable copies on different stack frames. Different copies will be *in-scope* at different times during the execution. A guard $g(\bar{x})$ is only a function of the *in-scope* valuation of relevant variables, $\bar{x}$, and likewise an action α evaluates expressions over the in-scope copies of variables and assigns values to in-scope copies of variables. What copies of variables are in scope depends on the stack,, as will become clear from the descriptions we shall give of different transitions.

Therefore, for simplifying notation, when the guard, g, of a (internal or call) edge on process j is evaluated, we write g as a boolean-valued function $g(\mathcal{U}, stack_j, \gamma_j)$ of both the stack at process j and the current universal valuation, $\mathcal{U}$ of shared variables, as well as the current valuation γ_j of thread-local variables. Likewise, we abuse notation somewhat and write the effect of an assignment α as $(\mathcal{U}', stack'_j, \gamma'_j) := \alpha(\mathcal{U}, stack_j, \gamma_j)$. Here the effect is again a mapping from the in-scope valuation of variables to a new in-scope valuation. The primed tuple $(\mathcal{U}', stack'_j, \gamma'_j)$ on the left hand side reflects the effect that this mapping has on $(\mathcal{U}, stack_j, \gamma_j)$.

We now consider, case by case, every possible type of transition. In every configuration, the stack, $stack_j$, of each process j, is a (possibly empty) sequence $SF_1, SF_2, \ldots, SF_m$ of stack frames. The stack grows from left to right, so SF_m is the top-most stack frame.

- **[Call]:** If the current configuration is $\psi = \langle \mathcal{U}, (stack_j, u_j, \gamma_j)_{j=1}^n \rangle$, and for some process j, which is currently at node u_j, there exists a call edge $e = (u_j, v, g, c)$, and the guard g holds true in the current configuration ψ, i.e., $g(\mathcal{U}, stack_j, \gamma_j) = \texttt{True}$, then an associated call transition is enabled in the current configuration. Assuming that $stack_j = SF_1, SF_2, \ldots, SF_m$, this transition will result in a new configuration ψ', in which a new stack frame SF_{m+1} will be pushed on to $stack_j$. Furthermore, in ψ' the new node on process j shall be updated to the entry node en_c^j of component A_c^j (i.e., the entry of the procedure that is called). The new stack frame, SF_{m+1}, will be as follows. If the procedure A_c called by e is *transactional*, then

$SF_{m+1} := (e_{m+1}, TP_{m+1}, \texttt{Fixed}_{m+1}, \texttt{Mutable}_{m+1}, \texttt{Used}_{m+1}, \texttt{Written}_{m+1})$, otherwise, if it is *non-transactional (ordinary)*, $SF_{m+1} := (e_{m+1}, TP_{m+1})$. The contents of these tuples are as follows:

- $e_{m+1} = e = (u_j, v, g, c)$, is the call edge itself, which is stored on the stack frame for being able to restore the calling context upon returning/aborting.
- TP_{m+1} is a *transaction pointer*, which points to the most recent transactional stack frame, if one exists, and is otherwise NULL. Specifically, if e is a *transactional call*, i.e., if component A_c^j is a transactional component, then $TP_{m+1} := m + 1$ (in other words, the transaction pointer points to its own stack frame as the latest *transactional frame*). If e is not a transactional call, then: if the the stack $stack_j$ is not empty, then $TP_{m+1} := TP_m$, and if the call stack is empty then $TP_m := NULL$.[3]

Furthermore, if the call e is a transactional call, i.e., if $TP_{m+1} = m+1$, then the stack frame SF_{m+1} additionally contains, the following things:

- a set $\texttt{Fixed}_{m+1}$ of pairs of the form (x, w), where x is shared variable name and w is a value in the domain of x, $D(T(x))$. The set $\texttt{Fixed}_{m+1}$ will hold a copy of those variables x of universal shared memory whose values are used during the life of the outer-most transaction on the stack. This is the transaction which has an associated stack frame SF_d which is transactional and such that there does not exist a smaller $d' < d$ such that $SF_{d'}$ is transactional. The valuation in $\texttt{Fixed}_{m+1}$ will hold the values of these shared variables at precisely the time when they were first *used* during this outer-most transaction.
  When the stack frame SF_{m+1} is created, we initialize $\texttt{Fixed}_{m+1}$ as follows: if $TP_m \neq NULL$, then $\texttt{Fixed}_{m+1} := \texttt{Fixed}_{TP_m}$. Otherwise, (i.e., if $TP_m = NULL$, i.e. $stack_j$ is empty and $m = 0$) then $\texttt{Fixed}_{m+1} := \emptyset$.
- a separate set $\texttt{Mutable}_{m+1}$, which contains pairs (x, w), where x is a shared variable, and $w \in D(T(x))$ is a valuation of x . This mutable set will contain such pairs for the set of shared variables that have been *used/written* up to now during the life of the outer-most active

[3] Technically, a transaction pointer, TP_{m+1}, is potentially an unbounded piece of data (the size of the stack is not bounded), and we can not cope with unbounded sized data for model checking. However, these transaction pointers are only a device we use to describe in a convenient way how implementations of this transactional paradigm would proceed without too much overhead. We can easily get semantically equivalent functionality as that given by TPs without requiring unbounded sized stack frames, albeit in a more cumbersome and less elegant way. Namely, we can copy all data from the most recent transactional stack frame into each non-transactional stack frame higher in the call stack (but prior to any higher transactional frame). This allows us to replace references using the transaction pointer by references to the current copy of the closest transaction frame. Each stack frame then contains only a bounded amount of data, regardless of the height of the stack. When a stack frame is popped this copy is used to update the the copy in the prior stack frame. Clearly, it is also semantically equivalent. But this is more cumbersome, especially notationally, so we opt for the more clear and convenient presentation using transaction points.

transaction. Again, when the stack frame SF_{m+1} is created, we initialize $\texttt{Mutable}_{m+1}$ just as with $\texttt{Fixed}_{m+1}$: if $TP_m \neq NULL$, then $\texttt{Mutable}_{m+1} := \texttt{Mutable}_{TP_m}$. Otherwise, (i.e., if $TP_m = NULL$ or $stack_j$ is empty and thus $m = 0$) then $\texttt{Mutable}_{m+1} := \emptyset$.

- a set $\texttt{Used}_{m+1}$ which containing the names of shared variables that have been *used* so far, again during the life of the outer-most active transaction. $\texttt{Used}_{m+1}$ is initialized in the same way as $\texttt{Fixed}_{m+1}$ and $\texttt{Mutable}_{m+1}$.
- a set $\texttt{Written}_{m+1}$ containing the names of shared variables that have been *written* so far, again, during the life of the outer-most active transaction. Again, $\texttt{Written}_{m+1}$ is initialized in the same way.

– **[Internal Transition]:** Internal transitions can occur when, in the current configuration $\psi = \langle \mathcal{U}, (stack_j, u_j)_{j=1}^n \rangle$, some process j is at a control node u_j, and there exists some internal edge $e = (u_j, v, g, \alpha)$, such that the guard g holds true in configuration ψ, i.e., $g(\mathcal{U}, stack_j, \gamma_j) = \texttt{True}$. The transition, after evaluating the guard (and seeing that it holds true), applies the assignment α, $(\mathcal{U}', stack'_j, \gamma'_j) := \alpha(\mathcal{U}, stack_j, \gamma_j)$. This of course, is done by first evaluating the relevant expressions on the in-scope copy of variables in $(\mathcal{U}, stack_j, \gamma_j)$, and then assigning the values simultaneously to obtain the updated left hand side $(\mathcal{U}', stack'_j, \gamma'_j)$. We then move to a new configuration where the current node on process j is set to v.
Crucial assumption: it is crucially assumed that the combined effect of both first evaluating the guard g and then carrying out the assignment α occurs atomically, meaning in particular that for an assignment of the form $[x_1, \ldots, x_d] := [expr_1, \ldots, expr_d]$, all assignments are made simultaneously and atomically. This is basically the *strong isolation* assumption.[4]
When a variable, x is read or updated by a guard or an assignment, if the variable is thread-local, then we read/update its value in the thread-local valuation γ_j. Then the variable x is a shared variable, there are a number of cases to consider, in order to decide what is the *in-scope* copy which is read/updated.
  - if the transaction pointer of the top frame $TP_m = NULL$, i.e., we are not inside a transaction on process j, then we simply read, or write to, the actual universal copy of the shared variable x. In other words, we modify the universal valuation $\mathcal{U}$.
  - Otherwise, in the case when $TP_m = k \neq NULL$, i.e., we are inside a transaction on process j.
    In this case, if the shared variable x is *read/used* by the guard or assignment (i.e., occurs in one of the the right-hand-side expressions) then:

[4] It may appear too strong to assume that the sequence of *both* guard evaluation, and assignment, occur atomically. This is not, strictly speaking, necessary. We can easily rephrase the TSM model definition so that internal edges either have trivial guards which are always true, or trivial assignments which do nothing, without changing the expressive power of the TSM model. (For this, note that multiple edges, even on a single process, may be enabled at any given configuration. In other words, we have non-determinism available in the TSM model even at the level of a single process.)

* if x is not already on the $\texttt{Written}_k$ list of the current transaction frame (i.e., the stack frame SF_k pointed to by $TP_m = k$), read its current value w from the universal copy, $\mathcal{U}$, and add (x, w) to the set $\texttt{Mutable}_k$ of the current transaction frame. Furthermore, we also add (x, w) to the $\texttt{Fixed}_k$ set of the current transaction frame. Finally, we add the variable x to $\texttt{Used}_k$.
* if x is already in the set $\texttt{Written}_k$, then simply look up and use its current value in $\texttt{Mutable}_k$. (A single valuation (x, w) will already be in $\texttt{Mutable}_k$, because x was already written to during the life of this transaction frame.)

If a variable x is *updated* with a value w, i.e., x it is an assignment assigns x some expression which evaluates to w, then:

* if x isn't already in $\texttt{Written}_k$, add it to $\texttt{Written}_k$.
* update the value of x in $\texttt{Mutable}_k$. In other words, put the pair (x, w) to $\texttt{Mutable}_k$, overwriting any earlier pair (x, w') in $\texttt{Mutable}_k$ if such a pair exists.

– **[Return]:** Such a transition is enabled if in the current configuration, on some process j, the current node is an exit node, i.e., $u_j = ex_c^j$. There are two cases to consider:

- *Return from an ordinary (non-transactional) procedure:* In this case the topmost stack frame, SF_m on $stack_j$ is not a transactional frame, i.e., $TP_m \neq m$. In this case, for effecting this transition, we pop the stack frame SF_m, and we update the current node on process j to the target v of the call edge $e_m = (u, v, g, c)$ which is also available in SF_m.
- *Return (either committed or automatically aborted) from a transactional procedure:* In this case, $TP_m = m$. If the current transaction frame is not the only transaction frame remaining (meaning $TP_{m-1} = d \neq NULL$), then we treat this exit similarly to a non-transactional procedure exit, except that we move the values of the fixed and mutable valuations of shared variables to their fixed and mutable copy in the next outer transaction frame (overwriting older values if necessary), and we do likewise with the used/written sets. In other words, given that $TP_{m-1} = d$, we assign $\texttt{Fixed}_d := \texttt{Fixed}_m$, $\texttt{Mutable}_d := \texttt{Mutable}_m$, $\texttt{Used}_d := \texttt{Used}_m$, and $\texttt{Written}_d := \texttt{Written}_m$.[5]

  If, on the other hand, this transaction frame is the only remaining one on the call stack (i.e., $TP_{m-1} = NULL$), we perform the following 1-step *compare-and-swap*: Pop the stack frame SF_m, and use the call edge e_m

[5] Note that what this means is that there is never an automatic run-time abort of a nested transaction. There are only explicit aborts of nested transactions. Such aborts can simply be viewed as exception handling mechanisms, which allow the effect of the nested transaction on shared memory to be "erased". However, it should be noted that, inevitably, the nested transaction can nevertheless have "side effects" even if aborted. In particular, the nested transaction may have updated thread-local variables. Such side effects can easily be used to program, e.g., an explicit *retry* mechanism for transactions in this model, so separate constructs for retry are not needed.

which is stored on SF_m to restore the context and assign a new node for process j.

- If, on the other hand, the *compare* fails, i.e., values in the universal copy and fixed copy on the stack of shared memory are not the equal then do exactly the same as done by an explicit *abort* operation, which is specified next.

– **[Abort]:** An abort transition can occur, either because an *explicit* abort node is encountered and terminates a transaction, meaning in the current configuration some process j is at an abort node, i.e., $u_j = ab_c^j$, or it can also occur because (as described at the end of the previous [Return] case), the compare-and-swap at the end of a transaction failed, resulting in a memory conflict, in which case the run-time system will do an automatic abort.
In both cases, we do the following. If the abort is being carried out on process j, and the top stack frame on $stack_j$ is SF_m (this is necessarily a transactional frame) then we pop SF_m off $stack_j$, and we restore the new node (just as in the "Return" case).
We also do the following important thing: if this was a nested transaction, meaning that $TP_{m-1} = d \neq NULL$, then we must update $\texttt{Fixed}_d$, $\texttt{Mutable}_d$, and $\texttt{Used}_d$ with the new values of shared variables that were read for the first time (in the context of transaction frame SF_d) inside this nested transaction SF_m. We do this update as follows: For every pair $(x, w) \in \texttt{Fixed}_m$ such that there does not exist any valuation (x, w') of variable x in $\texttt{Fixed}_d$, add (x, w) to $\texttt{Fixed}_d$, and likewise add (x, w) to $\texttt{Mutable}_d$ and add x to $\texttt{Used}_d$. This updating is done so that these shared variables, whose universal copy has already been read once inside the context of the transaction frame SF_d (in fact inside SF_m) will not be read again.

This completes the description of the semantics of TSMs. As usual, the set of configurations, Q, and the transition relation, Δ, of a TSM, A, together define an ordinary (and in general infinite-state) transition system $T_A = (Q, \Delta)$. A *run* of A is a (finite or infinite) sequence $\rho = \rho_0 \stackrel{e_0}{\rightarrow} \rho_1 \stackrel{e_1}{\rightarrow} \rho_2 \ldots$, where ψ_0 is the initial configuration and for all i, $\psi_i \in Q$ and $(\psi_i, e_i, \psi_{i+1}) \in \Delta$, where the event e_i is a specific edge, return, commit, or abort, operation that is enabled in state ρ_i and such that executing it yields state ρ_{i+1}. For a run ρ, we will use $\rho(i, j)$, $i \leq j$, to denote the segment $\rho_i \ldots \rho_j$ of the run. We refer to segments $\rho(0, j)$ as *initial segments* of ρ.

A.3 Stutter-invariance

Stutter-invariance was first considered in [16]. For a (finite or infinite) words $w = w_0 w_1 w_2 \ldots$ over an alphabet Σ, with $w_i \in \Sigma$ for all i, and a mapping $f : \mathbb{N} \mapsto \mathbb{N}^+$, from natural numbers to positive natural numbers, let $w^f \doteq w_0^{f(0)} w_1^{f(1)} w_2^{f(2)} \ldots$, where, as usual, w_i^n denotes the concatenation of n copies of the letter w_i. Recall that a language $L \subseteq \Sigma^\infty = \Sigma^\omega \cup \Sigma^*$ over an alphabet Σ is *stutter-invariant* iff for all (finite or infinite) words $w = w_0 w_1 w_2 \ldots$, and for all mappings, $f : \mathbb{N} \mapsto \mathbb{N}^+$, $w \in L$ iff $w^f \in L$. Two words w and $w' \in \Sigma^\infty$, over

Σ are *stutter-equivalent*, if there is some $f : \mathbb{N} \mapsto \mathbb{N}^+$ such that either $w' = w^f$ or $w = (w')^f$. Equivalently, w and w' are stutter-equivalent if they agree on all stutter-invariant properties, i.e., for any stutter-invariant language L they are either both contained in L or both not. LTL formulas without the next operator express precisely all stutter-invariant LTL languages, and there are similarly easy syntactic restrictions of Büchi automata for describing all stutter-invariant ω-regular properties (see [20, 11]).

A.4 Proof of Theorem 1

Theorem 1. *All TSMs are stutter-serializable.*

Proof. Given any run ρ of a TSM, $\mathcal{A}$, let $\rho(0, j) = \rho_0 \dots \rho_j$ be an initial segment of ρ such that ρ_j is the state just after a transaction on some process k terminates successfully and commits. Let the transitions $\rho_{i_1} \stackrel{e_{i_1}}{\rightarrow} \rho_{i_1+1}$, $\rho_{i_2} \stackrel{e_{i_1}}{\rightarrow} \rho_{i_2+1}, \dots, \rho_{i_m} \stackrel{e_{i_1}}{\rightarrow} \rho_{i_m+1} = \rho_j$ be precisely those transitions of the run associated with the execution of a particular transaction on process k from start to finish (i.e., from just before the transaction call to just after its successful commit). Note that the only transition among these that can possibly change the universal valuation is the final (commit) transition $\rho_{i_m} \stackrel{e_{i_1}}{\rightarrow} \rho_{i_m+1}$.

We now inductively construct a new run ρ', by induction on the length j of such initial segments of ρ whose last state is the state just after a successful commit of a transaction. Given such an initial segment, we delay the events $e_{i_1}, \dots, e_{i_{m-1}}$ and reschedule them just before e_{i_m}. In other words, the new event sequence will look as follows:

$e_1 e_2 \dots e_{i_1-1} e_{i_1+1} e_{i_1+2} \dots e_{i_2-1} e_{i_2+1} \dots e_{i_{m-1}-1} e_{i_{m-1}+1} \dots e_{i_m-1} e_{i_1} e_{i_2} \dots e_{i_m}$.

We claim that these events remain enabled after this reordering and generate a valid initial segment of a run, $\rho'(0, j) = \rho'_0 \dots \rho'_j$, and that furthermore $\rho'(0, j)[\mathcal{U}]$ is stutter-equivalent to $\rho(0, j)[\mathcal{U}]$, and finally that the last state ρ'_j is exactly equal to ρ_j. The reasons these claims hold was basically described in the sketch proof in the body of the paper: since the only communication between processes is through the universal copy of shared memory, no event on processes other than k is disabled because of the delay of the non-committing events inside this transaction on process k, and furthermore, all the events in the transaction will remain enabled after the delay because, since the compare-and-swap of the commit operation e_{m-1} succeeded in the original run, the universal values of all shared variables remain the same as they were when each read inside the transaction took place. Note that stuttering of the universal values can be introduced (or removed) by this reordering of events, but the final states are identical, because every process' view of the initial segment $\rho'(0, j)$ is exactly the same as $\rho(0, j)$. By induction on the length of initial segment of ρ which end in a successful commit, we can convert longer and longer initial segments, and thus the entire run ρ, into a new run ρ' such that $\rho'[\mathcal{U}]$ is stutter-equivalent to $\rho[\mathcal{U}]$ and such that ρ' contains only serialized successful transactions. (Note that there is no danger that an event will be indefinitely delayed by this reordering, because

the only events that are delayed in the reordering are non-commit events of a transaction which commits successfully, and they are only delayed a finite number of steps until just prior to the successful commit.) □

A.5 Proof of Lemma 1

Lemma 1. *The set $\mathcal{G}_c$ is computable for every transactional procedure A_c.*

Proof. Recall that there are only finitely many possible GSPs. Moreover, even though a transaction may internally be a recursive procedure, since we are considering sequential executions of the transaction in isolation from all other processes we can treat the transaction just like a standard Recursive State Machine (RSM) (or, equivalently, Pushdown System) and use the well known algorithms for model checking them (see [2]) in order to determine whether an execution corresponding to each given generalized summary path is possible (i.e., valid) or not, as follows.

Specifically, for a given GSP, G, we can use an LTL property to describe the possible sequence of (simultaneous) atomic reads corresponding to G. Intuitively, we can use an LTL formula with nested untils to describe a sequence like the following: "the first shared variable that is read (atomically) is x_{i_1} with a value read of w_{i_1}, and thereafter there are no reads of unused shared variables until x_{i_2} and x_{i_3} are read simultaneously, with values w_{i_2} and w_{i_3}, and thereafter .....". Furthermore, the LTL formula corresponding to G can also describe the values according to G: of thread-local variables at the beginning and end of the execution, the values of shared variables that are committed (if any) to $\mathcal{U}$ at the end, and whether the transaction did indeed commit or abort.

Model checking such an LTL formula for G against the Recursive State Machine corresponding to A_c will decide whether the given GPS, G, is valid or not. We can do this check separately for every possible GPS, to compute the set $\mathcal{G}_c$ of valid ones. We can do so because there are only finitely many GSPs. (There are in fact more efficient ways to compute the valid GSPs in an aggregate way, using standard techniques from program analysis, but we don't need these for the result.) □

A.6 Construction of B_r, B'_r, and B, in the proof of Theorem 2.

We now describe how the finite summary state-machine, B_r, for process r, discussed in the proof of Theorem 2, can be constructed. A state of B_r contains a valuation of local memory, a *partial* valuation of universal shared memory (the reason why it is partial will be made clear shortly), and a control location (control node) which corresponds either to a control node of the main procedure of process r, or to some auxiliary control nodes which are needed as intermediate states in the middle of mimicking the execution of generalized summary paths. Also, the states having such auxiliary control nodes will additionally need to keep an extra *fixed* partial valuation of shared memory, as described below.

Intuitively, the state machine B_r mimicks the execution of the "main" procedure, $A^r_{p_r}$ of process r, except recall that the main procedure can only make transactional calls. Suppose an enabled call $e = (u, v, g, c)$ to a transaction, A_c, is encountered, in a state where the thread-local valuation and (partial) universal valuations of variables are γ and σ'.[6] Rather than actually executing the call e and maintaining a call stack, B_r non-deterministically "guesses" some valid GSP, $G = (\gamma_{start}, R, \gamma_{finish}, status, \sigma) \in \mathcal{G}_c$ such that $\gamma_{start} = \gamma$. (If no such valid GSP exists, then that means no halting behavior of this transaction is possible if it is executed starting in the memory "environment" given by γ. In such a case, B_r effectively "hangs" as a process.) If such a GSP, G does exist, and if B_r has guessed G non-deterministically, it then goes through a sequence of d intermediate auxiliary states, which reflect the sequence of d (potentially simultaneous) reads $R = R_1, \ldots, R_d$ that occur in G, and the *partial* valuations of shared memory that they define. The intermediate auxiliary state associated with read R_i has partial valuation R_i of shared memory These intermediate states also retain another *fixed* partial valuation. Namely, the j'th intermediate state in the sequence contains fixed partial valuation $F_j = \cup_{i=1}^{j} R_i$. Finally, at the end of the auxiliary sequence we update local valuation γ to γ_{finish}, and update the partial valuation of universal shared memory to a new valuation σ'', consistent with *status* and σ, as follows. If $status =$ `commit`, then we non-deterministically either set $\sigma'' := \sigma$ or $\sigma'' := \emptyset$[7], and if $status =$ `abort` then $\sigma'' = \emptyset$. The control node is then set to v, the return point of the call.[8] Transitions of B_r are labeled by either the edge e of the TSM that enabled them, or by `commit`, `automatic-abort`, `manual-abort` if it is a transaction's commit step, automatic abort step, or manual abort step, respectively, or by `return` if the transition corresponds to a return from an ordinary procedure call. If the transition corresponds to an indermiate auxiliary step within a transaction, it has a special label `auxiliary`. This completes our description of B_r. (We forgo more formal notation defining the states and transitions of B_r, which is rather tedious.) We can readily observe that since there is only finite data, and since the length d of such generalized summary paths is bounded, the number of states of B_r is finite.

Construction of the single RSM, $B'_{r'}$, corresponding to the unrestricted process r' is easier, because we don't have to incorporate summaries for it. We can retain all the stack processing done on one process of the TSM (including that done for transactions) on the call stack of the RSM, $B'_{r'}$. We omit a detailed description of $B'_{r'}$ here. (The only minor subtlety to observe is that states of $B'_{r'}$

[6] Such a call is enabled if we are at control node u and the guard g is satisfied by the thread-local valuation γ together with *any* full shared valuation σ^* which extends σ', i.e., such that $\sigma' \subseteq \sigma^*$.

[7] The two possibilities here correspond to either a successful commit or an automatic abort by the run-time system due to a memory conflict. Later in the construction of the asynchronous product B this non-deterministic choice is resolved by checking consistency with the universal shared valuations of other processes.

[8] In the final transition out of the auxiliary sequence we discard the final fixed partial valuation F_d. F_d is used later in the construction of B.

are also labeled by appropriate *partial* valuations of the universal copy of shared memory, as in the case of the summary state-machines B_r.)

Armed with the summary state machines B_r for all "top-transactional" processes, together with the RSM, $B'_{r'}$, corresponding to the single unrestricted process, we can construct the *asynchronous product RSM* $B = (\otimes_{r \neq r'} B_r) \otimes B'_{r'}$. Informally, this asynchronous product mimicks the interleaved execution of all these processes when they run concurrently, including the possible interleaving of steps of other processes in the middle of auxiliary steps associated with a transaction running on one of the processes. A state s of the asynchronous product B consists of a tuple of process states, one for each process, such that the partial valuations of universal shared memory in all of the process states in the tuple are consistent with each other (meaning no two of them give different values to the same variable). A state s of B also contains a universal partial valutation σ_s which is the union of the (consistent) universal partial valuations of each process. Transitions of B are determined by transitions of individual processes: they transform the process state for that process only, together with the universal partial valuation σ_s. A transition on process i is *enabled* at a state s, with process state s_i for process i, under the following conditions. If that process's transition is labeled by an edge e, then it is enabled if there is an extension of the thread-local (partial) valuation of that process's state, s_i, and an extension of the combined unversal (partial) valuation σ_s of all processes, which satisfies the guard of edge e.[9] Similary, if that process's transition is labeled by `return` or labeled `auxiliary`, then it is always enabled. A `commit` step is enabled if the fixed valuation F_d in the process state s_i is consistent with the universal valution σ_s. By contrast, a `automatic-abort` step is enabled if the fixed valuation F_d is inconsistent with σ_s. (This ensures that commit steps only occur if there is a a successful compare-and-swap at the end of the transaction.) Finally, `manual-abort` steps are always enabled. This completes our description of B (we forgo a more formal description.) It is not hard to see that since there is only one unrestricted process, B can itself be viewed as an RSM, with the call stack of $B'_{r'}$ now serving also as the call stack of the entire product RSM, B.

[9] Of course, since we only keep states s where partial valuations of universal memory are consistent, this also implicity implies that the transition is only enabled if the valuations of shared memory that it results in, based on what variables are written to, is consistent between different processes.