Reducing Fair Stuttering Refinement of Transaction Systems

Rob Sumners

Advanced Micro Devices

robert.sumners@amd.com

November 16th, 2015

Overview

Specifications of Reactive Systems

Reactive Systems refer to any system which has an ongoing interaction with some environment.. and as such their specification is defined as properties of infinite sequences of behaviors as opposed to singular results. Specifications of reactive systems can thus be factored into two types of properties:

Safety Properties

Safety properties can only be refuted by some finite prefix of an infinite run.. (i.e. nothing *bad* happens)

Progress Properties

Progress properties can only be refuted by some infinite suffix of a run.. (i.e. something *good* always eventually happens)

Verifying State Transition Systems

In the case of State transition systems (defined by an initial state predicate `(init x)` and a next-state transition relation `(next x y)`), property verification entails:

Verifying Safety Properties

Safety properties can generally be verified by proving that the *not bad* states are an invariant of the system. For finite state systems, one can do an exhaustive search of the reachable states to determine if any bad states are reached.

Verifying Progress Properties

Progress properties require showing that no infinite sequences of *not good* states can be reached. Yuck.. For finite state systems, one must search if any *not good* cycles are reachable.. alternatively, add a progress measure to transfer the progress property to a safety property.

Targeting Transaction Systems

Transaction Systems

For the fun of it, we will focus on asynchronous transaction sytems with the following characteristics..

- the state of a transaction system is simply a list of transaction states
- asynchronous update: transaction systems update one transaction state at a time
- each transaction starts in an initial state and eventually reaches a `done` state
- for simplicity, we will assume that no transactions are deleted or spawned.. but this is not limiting

Transactions in these systems are usually fairly simple in isolation, but the complexity arises from how they interact and interleave their computation.. all types of examples.. but how do we allow transactions to interact?

Specification of Transaction Systems

For an implementation transaction system, we will assume the definition of a much simpler specification transaction system and prove that the implementation is a *fair stuttering refinement* of the specification.. but what the heck does that mean?

Well.. in a bit..

We also want to define transaction systems in a way which affords an effective procedure/algorithm for checking for fair stuttering refinements.. more on that later.. in a bit..

Refinement and Stuttering Refinement

Refinement

An implementation is a *refinement* of a specification if every run of the implementation can be matched by a run of the specification

Stuttering Refinement

An implementation is a *stuttering refinement* of a specification if every run of the implementation can be matched by a run of the specification with an allotment for finite stutter in the implementation

...but this does not guarantee fairness...

Fair run

A run of a transaction system is fair if at all times, for every transaction in the state, there is a future time where the transaction will be selected to try to take its next step

Fair Stuttering Refinement

Fair Stuttering Refinement

An implementation is a *fair stuttering refinement* of a specification if every fair run of the implementation can be matched by a fair run of the specification with an allotment for finite stuttering on the implementation side

It is a nice, compact, and intuitive way to express safety and progress properties we would like to prove and has the following assurances:

- Implementation transactions are verified to progress in a way consistent with the specification transactions
- Implementation transactions are verified to never deadlock or starve and always make progress to completion
- Implementation transactions can still hide spatial and temporal details of the implementation

Fair Stuttering refinement is also transitive.. so we can chain refinements.. more on that too..

How Do Transactions Interact?

There are many ways in which transactions can potentially interact:

- shared memory or state
- message passing
- locks, mutexes, etc.
- spawning and joining..
- ... many many possible ways

But unfortunately, all of these standard means for transaction interaction introduce the potential for strong state correlation between transactions as they run and this makes writing efficient procedures and algorithms far more difficult.

So, we allow a transaction to only *block* another transaction (more precisely, block certain transitions of the other transaction) and this is the only interaction we allow.

Defining Transaction Systems

A transaction system is defined by two predicates on transaction states (init x) and (next x y c k).. (init x) simply tests if x is an initial transaction state..

Transaction next state relations

The predicate (next x y c k) returns T when a transition exists from x to y with a valid clock c and the transition is not blocked by k

```
(defun done (a)           (and (exists (x c k) (next x a c k))
                               (not (exists (y c k) (next a y c k)))))
(defun next-total (x y c k) (or (next x y c k) (and (done x) (init y))))
(defun next-all (x y c l)   (forall (i) (next-total x y c (nth i l))))
(defun good-clk (c l)       (forall (i) (all-clks-< (nth i l) c)))
(defun n-c (x y l)          (exists (c) (and (good-clk c l) (next-all x y c l))))
(defun next-trans (x y l)   (if (exists (z) (n-c x z l)) (n-c x y l) (= x y)))
(defun next-sys (l m i)     (and (equal-lists-except-index l m i)
                                 (next-trans (nth l i) (nth m i) l)))
(defun init-sys (l)         (forall (i) (init (nth i l))))
```

Key Benefit: State Smoothness Property

A key property of this style of transaction systems is the following "State Smoothness" property:

State Smoothness Property

For any list of transactions `L`, if any sublist of `L` is not reachable, then `L` is not reachable

This is very important in reducing the scope of our searches for failures, but it comes at the price of requiring a certain discipline in the definition of transactions and certain sacrifices... namely, any tightly coupled programs or processes will need to be merged into larger transactions.. Yikes?

More on Clocks and Blocks...

Clocks are natural numbers which are guaranteed to be greater than any existing clock value stored in a field of any transaction state. Clocks can be copied into the transaction state and then used later for the sake of unique indentification or to block based on ordering of transactions.

Transaction blocking could represent a variety of real effects on transaction execution from protocol implementation to resource availability. But transaction blocking can also be selective (only blocking some transaction updates) and this can be used to communicate information.. although in a limited way.

More Implications and Limitations

There are (at least) two big limitations in how we define transactions.. no direct communication between transactions and no shared storage.

Actually, because there are no stores in general and all transactions must make progress to completion, the issue is even more dire.. there is no way to hold or store data indefinitely. As we will see in the simple shared memory specification, the last value written to an address may just eventually flutter away on the wind..

Some of this is alleviated through specification.. make the lack of persistence part of the specification but in a manner which is clear so that there is no chance of hiding real failures in the implementation.

In general, this is not suited for all systems and definitions.. best suited for systems with small transactions which have a lot of protocol and resouce blocks as the main concern of design complexity.

Example Specification: Shared Memory 1

For our shared memory specification, every transaction is a read or write with a source, address, and data, and goes through the following states:

- *idle* — initial state with no instruction to perform
- *launch* — instruction received and ready to begin processing
- *precom* — instruction has arrived at ordering point.. waiting to commit
- *commit* — the instruction has been chosen to be performed next for a given address
- *visible* — the effects of the instruction are now "visible" to others
- *retire* — the instruction has been retired (along with its effects)

There is no shared state and no shared memory — so, the current state of memory is defined for each address by the last transaction for that address to become *visible*.. but note that due to our limitations on transactions.. there is no way to KEEP the last *visible* state around so your last written value to an address may just vanish..

Example Specification: Shared Memory 2

```
(defun mem-next (x y c k)
  (and (cond ((is-idle x)    (is-launch y))          ;; state update
             ((is-launch x)  (is-precom y))
             ((is-precom x)  (is-commit y))
             ((is-commit x)  (is-visible y))
             ((is-visible x) (is-retire y)))
       (= (l-clk y) (if (is-launch y) c (l-clk x)))  ;; launch clock update
       (= (p-clk y) (if (is-precom y) c (p-clk x)))  ;; precom clock update
       (implies (or (is-write-cmd x) (is-visible x) (is-commit x))
                (= (data x) (data y)))               ;; when data remains the same
       (= (list (cmd x) (src x) (addr x))            ;; other fields are static
          (list (cmd y) (src y) (addr y)))
       (cond ((is-launch x)                          ;; blocks by other reads/writes
              (implies (and (is-launch k)            ;; memory ordering requirements
                            (= (src x) (src k))
                            (or (= (addr x) (addr k))
                                (= (cmd x)  (cmd k))))
                       (<= (l-clk x) (l-clk k))))
             ((is-precom x)
              (cond ((is-precom k)                   ;; must pick in order to be fair
                     (implies (or (= (src x) (src k))
                                  (= (addr x) (addr k)))
                              (<= (p-clk x) (p-clk k))))
                    ((is-commit k)                   ;; can only commit one at a time
                     (/= (addr x) (addr k)))
                    ((is-visible k)                  ;; copy data from last visible
                     (implies (and (is-read-cmd x)
                                   (= (addr x) (addr k)))
                              (= (data k) (data y))))
                    (t t)))
             ((is-commit x)                          ;; ensure that only one is visible
              (implies (is-visible k) (/= (addr x) (addr k))))
             (t t))))
```

Example Implementation: Cache Coherence 1

Yet another version of the German Cache Coherence protocol with the following transaction components and updates:

- *instruction state* similar to the shared memory but updated relative to cache actions
- *local cache state* in idle/valid/invalid and carries addr and data
- *req channel state* in send/transit/delivered and carries cmd/src/addr
- *inv channel state* in send/transit/delivered and carries addr
- *iack channel state* in send/transit/delivered and carries addr/data
- *grant channel state* in send/transit/delivered and carries addr/data
- *host-mem state* in idle/received/pick/wait-rsp/send and holds src/addr/data

Execution first checks local cache state to see if read/write addr is already there.. if not, send a req. to host-mem, and then wait for the grant. The host-mem will check directory state to determine if it can grant or must first invalidate.. sends out invalidate requests, waits for acks, and then sends the grant.

Example Implementation: Cache Coherence 2

The German protocol implementation updates the shared memory instruction state at the following points:

- *launch* — updates immediately on instruction retrieval..
- *precom* — updates either on availability in local cache or on arrival to host-mem
- *commit* — updates either on availability in local cache or when all iacks have been received
- *visible* — updates on availability in local cache
- *retire* — when the instruction completes and the cache is cleared

For reference, the common German fail case of not ordering Invalidate and Grants from host-mem in the same channel will result in a fail of the check that blocks are preserved in the spec... (i.e. the block guarding the entry to *visible* state breaks if we can reach a state where two caches have been granted exclusive access).

Reducing Fair Stuttering Refinement 1

Goal: reduce proof of fair stuttering refinement of two transaction systems to a series of smaller checks on a bounded number of transactions

`init-done-maps-to-init-done`

init/done transaction states in the implementation map to init/done transaction states in the specification

`next-stutter-match`

every implementation transaction step is matched by a specification transaction step or stutters

`next-no-unbounded-stutter`

every stuttering transaction sequence is bounded (e.g. no cycles) between matching states

`block-preserved-at-match-steps`

any block in the specification must be matched by a block in the implementation at any step in the implementation which must be matched

Reducing Fair Stuttering Refinement 2

`no-reachable-block-cycles`

we cannot reach any set of transaction states which are mutually blocking.. no reachable deadlocks

`no-reachable-starvation`

every minimum blocking transaction set for a blocked trans. state cannot be reached or cannot complete to done

`well-founded-trans-refinement`

`well-founded-trans-refinement` is the conjunction of these 7 properties

```
;; abusing notation here as spec and impl are functions..
(implies (and (well-formed-system-p spec)
              (well-formed-system-p impl))
         (iff (well-founded-trans-refinement spec impl)
              (fair-stuttering-refinement spec impl)))
```

Reducing Fair Stuttering Refinement 3

"Soundness" overview

From the first five properties, we can ensure a stuttering refinement by proving that the specification transaction will always eventually match.. Define a measure on the implementation states based on block-dependence being acyclic (e.g. there is always one state which is unblocked). In order to ensure fairness, we need to define a measure for each transaction id which exists due to the no-reachable-starvation property.. this is a bit more complicated but inuitively, progress is ensured because some transaction which is blocking must be able to make progress and must then eventually be blocked itself.

"Completeness" overview

Each of the properties failing leads directly to a minimal run of the implementation system which cannot be matched by the specification or gets stuck in an infinite loop or causes a transaction to be blocked every time it tries to take a step.

A Checker for Finite State Transactions 1

User provides a list `systems` of system names. For each system name `'sys` in `systems`, the user must define predicates `(sys-init x)` and `(sys-next x y c k)` in ACL2 prior to executing the checker.
The checker then performs the following steps for `systems`:

- *typecheck* — fixpoint iteration through system functions accumulating type information.. fail on inconsistencies.
- *bddorder* — fixpoint iteration again through system functions accumulating ordering information.. build final order.
- *translate* — translate the system functions to BDDs using the BDDorder and type information from previous steps.
- *check-wftr* — perform the `well-founded-trans-refinement` checks between each system starting from specification.

A Checker for Finite State Transactions 2

Several things to note about *typecheck*, *bddorder*, and *translate*

- *typecheck* passing will ensure that the system functions have effective finite domain (bounded number of cars,cdrs,gets and limited operations on atoms).
- types computed for a more concrete system must be a "subtype" of the type computed for the more abstract systems.. this allows an implicit mapping of states by dropping the state components in the concrete system which are not relevant at the abstract level and allows us to use one transaction type for all systems.
- *bddorder* fixpoint iteration is computed across all system functions... needed so we can use one bddordering for transactions for all systems.
- *bddorder* produces strong equivalence classes between transaction fields that need interleaving as well as precedences which determine the final layout.

We interleave these BDD transaction variables across multiple transaction IDs.. so if we have transaction ordering *BV0, BV1, BV2, ...* then the final order will be *tr0.BV0, tr1.BV0, ..., tr0.BV1, tr0.BV1, ...*

A Checker for Finite State Transactions 3

In regard to clocks.. Clocks are unbounded natural numbers and can be copied into updated transaction states and thus break the finite domain assumption... But, as part of *typecheck*, we ensure that clocks can only be equated or linearly compared with other clocks with no constants.. This means that we can use a finite set of natural values for clocks as long as we have more values than variables and we bound every state set computation to not include more variables than possible clock values. moving on...

A Checker for Finite State Transactions 4

Checking *WFTR* proceeds in checking the requisite properties of each system with respect to the next system in the chain. The first several properties are fairly straightforward checks of building the appropriate BDD and testing whether it is empty or not. For the more involved checks, we compute a multi-transaction BDD and then use a backwards traversal to determine if the set is reachable:

- `block-preserved-at-match-steps` — check reach for 2-trans set `(x k)` where we can take a step from `x` (not blocked by `k`) which must be matched in the spec. and is blocked by `(map k)`
- `no-reachable-starvation` — check reach for n+1-trans set `(x k0 ... kn)` where `x` is minimally blocked by the `ki`. `x` must remain fixed in this case. if backward reach finds a path, then check forward reach to done states.
- `well-founded-trans-refinement` — check reach for incrementally larger trans-sets `(x0 ... xn)` where each `x` in the set is blocked by some subset and there is at least one dependence chain of length `n`

A Checker for Finite State Transactions 5

The multi-trans search procedure is the main “engine” of the check process. It primarily takes a BDD defining a set of states for some fixed number of transactions and tries to take backwards steps to push all transactions to init states. A few key optimizations are involved:

focused search

When computing backward search, we attempt to push back one focus transaction.. it will continue until its backward transition is blocked.. if so, then we change the focus to a transaction that blocked.. if the focus is blocked and we cannot pick any new blocking transactions, then the search fails immediately.

...and....

A Checker for Finite State Transactions 6

searching through spec.

We operate on the assumption that spec.s are significantly smaller systems and faster to search through than impl.s. We utilize this by jumping up to the spec, check to see if we can show the mapped states to be unreachable and if not, we will use the “path” in the spec. as a guide for searching in the impl.

spliting and reordering transaction BDDs

We also use BDD *splitting* to try and reduce the breadth of search we try to store in the BDDs as we go along, and we reorder the transactions in the BDDs to better utilize the previous state caches.

We also note that we can target starvation and cycle checks to include at least one transaction *block* which is not in the specification.

Future Work: General Improvements

There is a LOT of potential room for improving the efficiency of the algorithms in the checker and potentially extending the range of what could be defined as transactions.

One big overarchiing consideration is that the multi-trans searches are basically searches for the some interlock which keeps the bad states or sequences from being reached.. with some early computation to determine where the interlocks exist, it might greatly speed up the multi-trans checks.

Future Work: Easing the Burden of Definition

As presented, the restrictions on transaction interaction are severe and require some thought to work through..

I have some macros which allow system `next` functions to be defined as a set of “local” update rules which keeps the definition a bit more sane..

But it would be nice to come up with a more systematic way to extract a definition of transaction system (or at least partially) from a more realistic definition.

In addition, I would like to auto-generate a monitor which could detect inconsistencies between real implementation system and a transaction system for simulations of the real implementation.

Future Work: Proving Finite State Checker Correct

The primary interest is in proving soundness of the checker.. Basically the following steps:

- Define a proof generator from the typecheck output that proves the `init` and `next` functions are effectively-finite.
- Prove that the translator produces BDDs which are equivalent to the system functions on a finite domain
- Prove that when the Checker returns PASS then we have a `well-founded-trans-refinement`
- Prove that `well-founded-trans-refinement` implies `fair-stuttering-refinement`

I think most of this is tractable. I can see how to do the first two pieces and am working through the last piece, but proving the checker ensures `well-founded-trans-refinement` will be the most difficult step..

Future Work: From BDDs to Terms

BDDs are great when an effective ordering exists and the transaction states are sufficiently finite.. But, they can be inefficient and more importantly, the user has less control of the efficiency of BDD computations.. At a certain point, you just hope they work and if not, try to introduce another intermediate system or change how the systems are defined.

My preference would be to use terms to represent the states and use rewriting extended by new definitions and rules from the user.. There are many technical hurdles to making this happen (most pressing is the need to extend rewriting to effectively include first-order quantification)... May also need the user to supply some invariants of the reachable transaction states that we can prove are invariant..

But most of the pieces of `well-founded-trans-refinement` are well-suited to the potential of verifying with expanding next-state relations and rewriting.