Concurrent ABS with STM

A functional approach

JONATHAN THALER and THORSTEN ALTENKIRCH, University of Nottingham, United Kingdom

TODO: select journals - ACM Transactions on Modeling and Computer Simulation (TOMACS): https://tomacs.acm.org/ - ?

Additional Key Words and Phrases: Agent-Based Simulation, Software Transactional Memory, Functional Reactive Programming, Haskell

ACM Reference Format:
Jonathan Thaler and Thorsten Altenkirch. 2019. Concurrent ABS with STM: A functional approach. 1, 1 (July 2019), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

novelty in our case the use of FRP

We present case-studies in which we employ the well known SugarScape [1] and agent-based SIR [3] model to test our hypothesis. The former model can be seen as one of the most influential exploratory models in ABS which laid the foundations of object-oriented implementation of agent-based models. The latter one is an easy-to-understand explanatory model which has the advantage that it has an analytical theory behind it which can be used for verification and validation.

The aim of this paper is:

This paper makes the following contributions: - FRP and STM - compares 3 approaches: non-concurrent, low-level locking, STM

The structure of the paper is:

2 BACKGROUND

TODO: be very careful, i copied some sentences directly from the relevant papers The whole concept of our approach is built on the usage of Software Transactional Memory (STM), where we follow the main paper [2] on STM [1].

Concurrent programming is notoriously difficult to get right because reasoning about the interactions of multiple concurrently running threads and low level operational details of synchronisation primitives and locks is *very hard*. The main problems are:

- Race conditions due to forgotten locks.
- Deadlocks resulting from inconsistent lock ordering.
- Corruption caused by uncaught exceptions.
- Lost wakeups induced by omitted notifications.

Worse, concurrency does not compose. It is utterly difficult to write two functions (or methods in an object) acting on concurrent data which can be composed into a larger concurrent behaviour. The reason for it is that one has to know about internal details of locking, which breaks encapsulation and makes composition depend on knowledge about their implementation. Also it is impossible to compose two functions e.g. where one withdraws some amount of money from an account and the

[1]We also make use of the excellent tutorial http://book.realworldhaskell.org/read/software-transactional-memory.html.

Authors' address: Jonathan Thaler, jonathan.thaler@nottingham.ac.uk; Thorsten Altenkirch, thorsten.altenkirch@nottingham.ac.uk, University of Nottingham, 7301 Wollaton Rd, Nottingham, NG8 1BB, United Kingdom.

2019. XXXX-XXXX/2019/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

other deposits this amount of money into a different account: one ends up with a temporary state where the money is in none of either accounts, creating an inconsistency - a potential source for errors because threads can be rescheduled at any time.

STM promises to solve all these problems for a very low cost. In STM one executes actions atomically where modifications made in such an action are invisible to other threads until the action is performed. Also the thread in which this action is run, doesn't see changes made by other threads - thus execution of STM actions are isolated. When a transaction exits one of the following things will occur:

(1) If no other thread concurrently modified the same data as us, all of our modifications will simultaneously become visible to other threads.
(2) Otherwise, our modifications are discarded without being performed, and our block of actions is automatically restarted.

Note that the ability to *restart* a block of actions without any visible effects is only possible due to the nature of Haskells type-system which allows being explicit about side-effects: by restricting the effects to STM only ensures that no uncontrolled effects, which cannot be rolled-back, occur.

STM is implemented using optimistic synchronisation. This means that instead of locking access to shared data, each thread keeps a transaction log for each read and write to shared data it makes. When the transaction exits, this log is checked whether other threads have written to memory it has read - it checks whether it has a consistent view to the shared data or not. This might look like a serious overhead but the implementations are very mature by now, being very performant and the benefits outweigh its costs by far.

Applying this to our agents is very simple: because we already use Dunai / BearRiver as our FRP library, we can run in arbitrary Monadic contexts. This allows us to simply run agents within an STM Monad and execute each agent in their own thread. This allows then the agents to communicate concurrently with each other using the STM primitives without problems of explicit concurrency, making the concurrent nature of an implementation very transparent. Further through optimistic synchronisation we should arrive at a much better performance than with low level locking.

2.1 STM primitives

STM comes with a number of primitives to share transactional data. Amongst others the most important ones are:

- TVar - A transactional variable which can be read and written arbitrarily.
- TArray - A transactional array where each cell is an individual shared data, allowing much finer-grained transactions instead of e.g. having the whole array in a TVar.
- TChan - A transactional channel, representing an unbounded FIFO channel.
- TMVar - A transactional *synchronising* variable which is either empty of full. To read from an empty or write to a full TMVar will cause the current thread to retry its transaction.

Additionally, the following functions are provided:

- atomically :: STM a → IO a - Performs a series of STM actions atomically. Note that we need to run this in the IO Monad, which is obviously required when running an agent in a thread.
- retry :: STM a - Allows to retry a transaction immediately.
- orElse :: STM a → STM a → STM a - Tries the first STM action and if it retries it will try the second one. If the second one retries as well, orElse as a whole retries.

3 RELATED WORK

TODO: hlogo masterthesis TODO: erlang papers

4 STM AND ABS

For a proof-of-concept we changed the reference implementation of the agent-based SIR model on a 2D-grid as described in the paper in Appendix ??. In it, a State Monad is used to share the grid across all agents where all agents are run after each other to guarantee exclusive access to the state. We replaced the State Monad by the STM Monad, share the grid through a *TVar* and run every agent within its own thread. All agents are run at the same time but synchronise after each time-step which is done through the main-thread.

We make STM the innermost Monad within a RandT transformer:

```
type SIRMonad g   = RandT g STM
type SIRAgent g   = SF (SIRMonad g) () ()
```

In each step we use an *MVar* to let the agents block on the next Δt and let the main-thread block for all results. After each step we output the environment by reading it from the *TVar*:

```
-- this is run in the main-thread
simulationStep :: TVar SIREnv
               -> [MVar DTime]
               -> [MVar ()]
               -> Int
               -> IO SIREnv
simulationStep env dtVars retVars _i = do
  -- tell all threads to continue with the corresponding DTime
  mapM_ (`putMVar` dt) dtVars
  -- wait for results, ignoring them, only [()]
  mapM_ takeMVar retVars
  -- read last version of environment
  readTVarIO env
```

Each agent runs within its own thread. It will block for the posting of the next Δt where it then will run the MSF stack with the given Δt and atomically transacting the STM action. It will then post the result of the computation to the main-thread to signal it has finished. Note that the number of steps the agent will run is hard-coded and comes from the main-thread so that no infinite blocking occurs and the thread shuts down gracefully.

```
createAgentThread :: RandomGen g
                  => Int
                  -> TVar SIREnv
                  -> MVar DTime
                  -> g
                  -> (Disc2dCoord, SIRState)
                  -> IO (MVar ())
createAgentThread steps env dtVar rng0 a = do
    let sf = uncurry (sirAgent env) a
    -- create the var where the result will be posted to
    retVar <- newEmptyMVar
    _ <- forkIO (sirAgentThreadAux steps sf rng0 retVar)
    return retVar
  where
    agentThread :: RandomGen g
                => Int
                -> SIRAgent g
                -> g
                -> MVar ()
                -> IO ()
    agentThread 0 _ _ _ = return ()
    agentThread n sf rng retVar = do
      -- wait for next dt to compute next step
      dt <- takeMVar dtVar

      -- compute next step
      let sfReader = unMSF sf ()
          sfRand   = runReaderT sfReader dt
          sfSTM    = runRandT sfRand rng
      ((_, sf'), rng') <- atomically sfSTM

      -- post result to main thread
      putMVar retVar ()

      agentThread (n - 1) sf' rng' retVar
```

5 CASE STUDY 1: SIR

- maps nicely to continuous time-semantics and state-transitions provided by FRP - STM results in considerable performance boost

The implementation of the susceptible, infected and recovered agents are the same except that instead of accessing the environment through *get* and *put*, we use *readTVar*, *writeTVar* and *modifyTVar*.

We checked the visual outputs and the dynamics and they look qualitatively the same to the reference implementation. Summarizing the performances:

Implementation	Avg. Duration
State Monad	101.15 sec
STM Single-Core	52.75 sec
STM 4-Core	20.57 sec

When we scale up the grid-size for the 4-core version we get the following results (single run in each case, gives only a rough estimate):

Grid-Size	Duration
51 x 51	21 sec
101 x 101	92 sec
151 x 151	188 sec
201 x 201	305 sec
251 x 251	530 sec

Further we investigated how well STM scales to multiple cores by running the 51 x 51 on 1-4 cores:

Cores	Duration
1	52 sec
2	28 sec
3	22 sec
4	21 sec

5.0.1 Comparison to IO implementation. We also implemented the same model running in the IO Monad and doing the synchronisation all 'manually' aquiring and releasing the locks explicitly. In this case, the environment is shared using an IOVar and the access is synchronised using an MVar. We get the following performance (single run in each case, gives only a rough estimate):

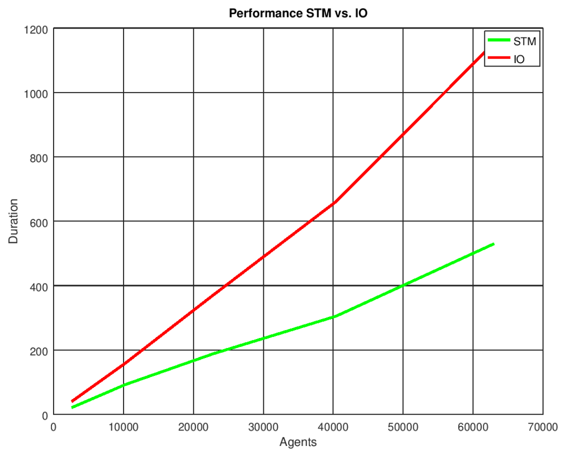

Fig. 1. Comparison of STM (Table 5) and IO performance (Table 5.0.1)

Grid-Size	Duration
51 x 51	38 sec
101 x 101	158 sec
151 x 151	370 sec
201 x 201	661 sec
251 x 251	1154 sec

The Figure 1 clearly indicates that STM outperforms the low level IO implementation by a substantial factor and scales much smoother.

Further we investigated how well IO scales to multiple cores by running the 51 x 51 on 1-4 cores (single run in each case, gives only a rough estimate):

Cores	Duration
1	61 sec
2	45 sec
3	40 sec
4	38 sec

Comparing the scaling to multiple cores of the IO (Table 5.0.1) and STM implementation (Table 5) shows no significant difference between STM and IO as can be seen in Figure 2.

5.0.2 Comparison to Java RePast single core. To have an idea where the functional implementation is performance-wise compared to the established object-oriented methods, we conducted a performance comparison with a Java implementation using RePast, running on a single-core. All parameters were the same and the simulation was run until virtual time t=100 was reached, on various grid-sizes. Due to the lack of proper timing facilities in RePast we measured the time by hand using a stopwatch. Although this is not very precise it gives a rough estimate and allows a very basic comparison, being precise enough if the difference is larger than 1 second. We measured the following:

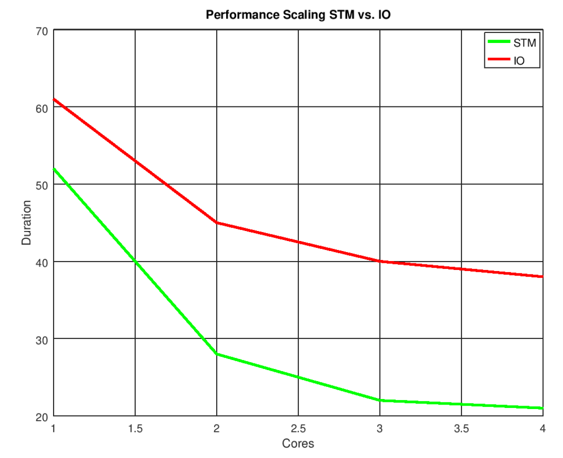

Fig. 2. Comparison of scaling IO (Table 5) and STM (Table 5) to multiple cores.

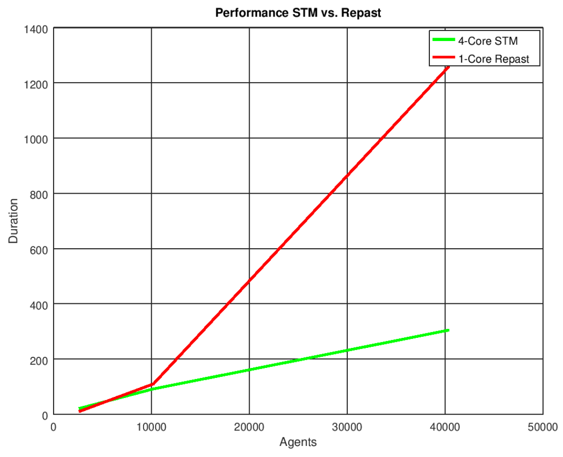

Fig. 3. Comparison of 4-Core STM and single core Repast (Table 5.0.2).

Grid-Size	Java Repast	4-Core Haskell
51 x 51	10 sec	20 sec
101 x 101	110 sec	92 sec
201 x 201	1260 sec	305 sec

As can be seen in Figure **??**, while on a 51x51 grid the single-core Java RePast version outperforms the 4-core Haskell STM version by about 200%, the figure is inverted on a 201x201 grid where the 4-core Haskell STM version outperforms the single core Java Repast version by 400%. We can conclude that the single-core Java RePast version clearly outperforms the Haskell STM 4-core version on small grid-sizes but that the Haskell STM version scales up with increasing grid-sizes and clearly outperforms the RePast version with increasing number of agents.

5.1 Conclusions

Interpretation of the performance data leads to the following conclusions:

(1) Running in STM and sharing state using a TVar is much more time- and memory-efficient than running in the State Monad.
(2) Running STM on multiple cores concurrently leads to a significant performance improvement (for that model).
(3) STM outperforms the low level locking implementation, running in the IO Monad, substantially and scales much smoother.
(4) Both STM and IO show same scaling performance on multiple cores, with the most significant improvement when scaling from a single to 2 cores.
(5) STM on single core is still slower than an object-oriented Java implementation on a single core.
(6) STM on multiple cores dramatically outperforms the single-core object-oriented Java implementation on a single core on instances with large agent numbers.

6 CASE STUDY 2: SUGARSCAPE

- main difficulty: synchronous agent-interactions - STM: not clear yet but retry factor of 5

7 PERFORMANCE DISCUSSION

8 CONCLUSION

Using STM for concurrent, large-scale ABS seems to be a very promising approach as our proof-of-concept has shown. The concurrency abstractions of STM are very powerful, yet simple enough to allow convenient implementation of concurrent agents without the nastiness of low level concurrent locks. Also we have shown by experiments, that we indeed get a very substantial speed-up and that we even got linear performance scaling for our model.

Interestingly, STM primitives map nicely to ABS concepts: using a share environment through a *TVar* is very easy, also we implemented in an additional proof-of-concept the use of *TChan* which can be seen as persistent message boxes for agents, underlining the message-oriented approach found in many agent-based models. Also *TChan* offers a broadcast transactional channel, which supports broadcasting to listeners which maps nicely to a pro-active environment or a central auctioneer upon which agents need to synchronize.

Running in STM instead of IO also makes the concurrent nature more explicit and at the same time restricts it to purely STM behaviour. So despite obviously losing the reproducibility property due to concurrency, we still can guarantee that the agents can't do arbitrary IO as they are restricted to STM operations only.

Depending on the nature of the transactions, retries could become a bottle neck, resulting in a live lock in extreme cases. The central problem of STM is to keep the retries low, which is directly influenced by the read/writes on the STM primitives. By choosing more fine-grained / suitable data-structures e.g. using a TArray instead of an Array within a TVar, one can reduce retries significantly. We tracked the retries in our proof-of-concept using the stm-stats library and arrived at a ratio of 0.0% retries - note that there were some retries but they were so low that they weren't significant.

9 FURTHER RESEARCH

Despite the promising proof-of-concept, still there is more work needed:

- So far we only looked at a model which is very well-behaved in STM, leading to a retry ratio of 0. It is of interest to see how STM performs on a model with much more involved read/write patterns e.g. the Sugarscape.
- So far we only looked at a time-driven model. It would be of fundamental interest whether we can somehow apply STM and concurrency to an event-driven approach as well. We hypothesise that it is not as striking and easy due to the fundamental sequential approach to even-processing. Generally one could run agents concurrently and undo actions when there are inconsistencies - something which STM supports out of the box.
- So far we only looked at asynchronous agent-interactions through TVar and TChan: agents modify the data or send a message but don't synchronise on a reply. Also a receiving agent doesn't do synchronised waiting for messages or data-changes. Still, in some models we need this synchronous way of agent-interactions where agents interact over multiple steps within the same global time-step. We yet have to come up with an easy-to-use solution for this problem using STM.

going towards distribution using Cloud haskell.

ACKNOWLEDGMENTS

The authors would like to thank

REFERENCES

[1] Joshua M. Epstein and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up.* The Brookings Institution, Washington, DC, USA.

[2] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05).* ACM, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

[3] Charles M. Macal. 2010. To Agent-based Simulation from System Dynamics. In *Proceedings of the Winter Simulation Conference (WSC '10).* Winter Simulation Conference, Baltimore, Maryland, 371–382. http://dl.acm.org/citation.cfm?id=2433508.2433551

Received May 2018