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/ - ?

TODO: re-measure SIR: need more measurements and average them

TODO: implement remaining Sugarscape chapters Pure TODO: implement remaining Sugarscape chapters STM TODO: implement synchronous Pure interactions (returning the value of the computation of the synchronous interaction to the surrounding SF) TODO: implement synchronous STM interactions (returning the value of the computation of the synchronous interaction to the surrounding SF) TODO: measure Sugarscape all chapters and compare them

TODO: write performance discussion TODO: write the background section TODO: write STM and ABS TODO: write Introduction TODO: write conclusion TODO: write further Research

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), 11 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 [8] and agent-based SIR [13] 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: - STM to concurrent ABS in the context of FRP - compares 3 approaches: non-concurrent, low-level locking, STM

The structure of the paper is:

2 BACKGROUND

TODO: Lock Free Data Structures using STM [6] TODO: The Limits of Software Transactional Memory [15] TODO: Composable Memory Transactions [9] TODO: Transactional memory with data invariants [10] TODO: A survey on parallel and distributed multi-agent systems for high performance computing simulations [16] TODO: Software Transactional Memory vs. Locking in a Functional Language [4] TODO: Modeling Software Transactional Memory with AnyLogic [11]

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 [9] on STM [1].

[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

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 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

In his masterthesis [3] the author investigated Haskells parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo simulation package, focusing on using Software Transactional Memory for a limited form of agent-interactions. *HLogo* is basically a re-implementation of NetLogos API in Haskell where agents run within IO and thus can also make use of STM functionality. The benchmarks show that this approach does indeed result in a speed-up especially under larger agent-populations. The authors thesis can be seen as one of the first works on ABS using Haskell. Despite the concurrency and parallel aspect our work share, our approach is rather different: we avoid IO within the agents under all costs, build on FRP and explore on a more conceptual level the use of STM rather than implementing a ABS library.

There exists some research [5, 17, 18] of using the functional programming language Erlang [2] to implement ABS. The language is inspired by the actor model [1] and was created in 1986 by Joe Armstrong for Eriksson for developing distributed high reliability software in telecommunications. The actor model can be seen as quite influential to the development of the concept of agents in ABS which borrowed it from Multi Agent Systems [19]. It emphasises message-passing concurrency with share-nothing semantics (no shared state between agents) which maps nicely to functional programming concepts. Erlang implements light-weight processes which allows to spawn thousands of them without heavy memory overhead. The mentioned papers investigate how the actor model can be used to close the conceptual gap between agent-specifications which focus on message-passing and their implementation. Further they also showed that using this kind of concurrency allows to overcome some problems of low level concurrent programming as well. Also [3] ported NetLogos API to Erlang mapping agents to concurrently running processes which interact with each other by message-passing. With some restrictions on the agent-interactions this model worked, which shows at using concurrent message-passing for parallel ABS is at least *conceptually* feasible.

TODO:

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*:

```haskell
-- 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.

```haskell
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

Our first case study is the SIR model which is a very well studied and understood compartment model from epidemiology [12] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population [7].

In it, people in a population of size N can be in either one of three states *Susceptible*, *Infected* or *Recovered* at a particular time, where it is assumed that initially there is at least one infected person in the population. People interact *on average* with a given rate of β other people per time-unit and become infected with a given probability γ when interacting with an infected person. When infected, a person recovers *on average* after δ time-units and is then immune to further infections. An interaction between infected persons does not lead to re-infection, thus these interactions are ignored in this model.

We followed in our agent-based implementation of the SIR model the work [13] but extended the by placing the agents on a discrete 2D grid using a Moore (8) neighbourhood TODO: cite my own PFE paper. In this case agents interact with each other indirectly through the shared discrete 2D grid by writing their current state on their cell which neighbours can read.

It is important to note that due to the continuous-time nature of the SIR model, our implementation follows the time-driven [14] approach and maps naturally to the continuous time-semantics and state-transitions provided by FRP. By sampling the system with very small Δt this means that we have comparatively very few writes to the shared environment which will become important when discussing the performance results.

In this case study we compare the performance of the following implementations under varying numbers of CPU cores:

(1) State Monad - This is the original implementation we also discuss in TODO: cite my own PFE paper. In it the discrete 2D grid is shared amongst all agents using the State Monad. Agents are run sequentially after another thus ensuring exclusive read/write access to it. Because we are neither running in the STM or IO Monad there is no way we can run this implementation concurrently.
(2) STM Monad - This is the same implementation like the State Monad but instead of sharing the discrete 2D grid in a State Monad, agents run in the STM Monad and have access to the discrete 2D grid through a *TVar*. This means that the reads and writes of the discrete 2D grid are exactly the same but happen always through the *TVar*. Also each agent is run within its own thread, thus enabling true concurrency when the simulation is actually run on multiple cores (which can be configured by the Haskell Runtime System).
(3) IO Monad - This is exactly the same implementation like the STM Monad but instead of running in STM, the agents now run in IO. They share the discrete 2D grid using an *IORef* and have access to an *MVar* to synchronise access to the it. Also each agent is run within its own thread.
(4) RePast - To have an idea where the functional implementation is performance-wise compared to the established object-oriented methods, we implemented a Java version of the SIR model using RePast with the State-Chart feature. This implementation cannot run on multiple cores concurrently but gives a good estimate of the single core performance of imperative approaches. Also there exists a RePast High Performance Computing library for implementing large-scale distributed simulations in C++ - we leave this for further research as an implementation and comparison is out of scope of this paper.

Each experiment was run until $t = 100$ and stepped using $\Delta t = 0.1$ except in RePast for which we don't have access to the underlying implementation of the state-chart and left it as it is. For each experiment we conducted 8 runs on our machine (see Table 1) and report both the average

OS	Fedora 28 64-bit
RAM	16 GByte
CPU	Intel Core i5-4670K @ 3.40GHz x 4
HD	250Gbyte SSD
Haskell	GHC 8.2.2
Java	OpenJDK 1.8.0
RePast	2.5.0.a

Table 1. Machine and Software Specs for all experiments

	Cores	μ Duration (sec)	σ Duration (sec)
State	1	100.33	0.434
STM	1	53.182	0.393
STM	2	27.817	0.555
STM	3	21.776	0.388
STM	4	20.201	0.789
IO	1	60.564	0.265
IO	2	42.779	0.421
IO	3	38.614	0.397
IO	4	41.914	1.073
RePast	1	10.822	0.377

Table 2. Experiments on constant 51x51 (2601 agents) grid with varying number of cores.

and standard deviation. Further, we checked the visual outputs and the dynamics and they look qualitatively the same to the reference implementation of the State Monad TODO: cite my own PFE paper. In the experiments we varied the number of agents (grid size) and the number of cores when running concurrently - the numbers are always indicated clearly.

5.1 Constant Grid Size, Varying Cores

In this experiment we held the grid size constant to 51 x 51 (2601 agents) and varied the cores where possible. The results are reported in Table 2.

Comparing the performance and scaling on multiple cores of the STM and IO implementations shows that the STM implementation significantly outperforms the IO one and scales better to multiple cores. The IO implementation performs best with 3 cores and shows slightly worse performance on 4 cores as can be seen in Figure 1. This is no surprise because the more cores are running at the same time, the more contention for the lock, thus the more likely synchronisation happening, resulting in more potential for reduced performance. This is not an issue in STM because no locks are taken in advance.

Comparing the reference *State* implementation shows that it is the slowest by far - even the single core STM and IO implementations outperform it by far. Also our profiling results reported about 30% increased memory footprint for the State implementation. This shows that the State Monad is a rather slow and memory intense approach sharing data but guarantees purity and excludes any non-deterministic side-effects which is not the case in STM and IO.

What comes a bit as a surprise is that the single core RePast implementation significantly outperforms *all* other implementations, even when they run on multiple cores and even with

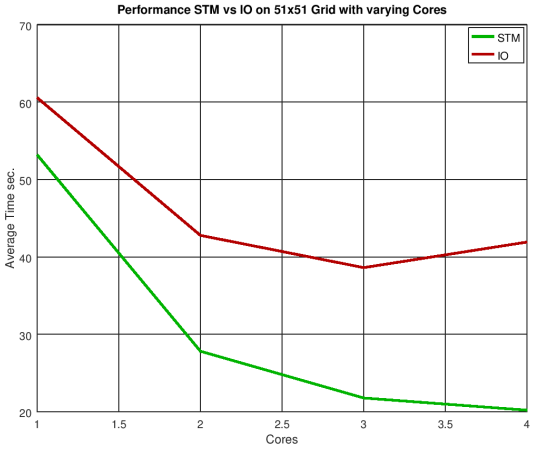

Fig. 1. Comparison of performance and scaling on multiple cores of STM vs. IO. Note that the IO implementation performs worse on 4 cores than on 3.

Grid-Size	μ Duration (sec)	σ Duration (sec)
51 x 51 (2601)	20.201	0.789
101 x 101 (10201)	74.493	0.524
151 x 151 (22801)	168.47	1.783
201 x 201 (40401)	TODO	TODO
251 x 251 (63001)	TODO	TODO

Table 3. STM Monad experiments on varying grid sizes on 4 cores.

Grid-Size	Avg. Duration	σ Duration
51 x 51 (2601)	TODO sec	TODO sec
101 x 101 (10201)	TODO sec	TODO sec
151 x 151 (22801)	TODO sec	TODO sec
201 x 201 (40401)	TODO sec	TODO sec
251 x 251 (63001)	TODO sec	TODO sec

Table 4. IO Monad experiments on varying grid sizes on 4 cores.

RePast doing complex visualisation in addition (something the functional implementations don't do). We attribute this to the conceptually slower approach of functional programming, and maybe we could have optimised parts of the code but we leave this for further research.

5.2 Varying Grid Size, Constant Cores

In this experiment we varied the grid size and used constantly 4 cores. The results for STM are reported in Table 3. The results for IO are reported in Table 3. The results for Repast are reported in Table 5 - note that these experiments all ran on a single (1) core and were conducted to have a rough estimate where the functional approach is in comparison to the imperative.

Grid-Size	Avg. Duration	σ Duration
51 x 51 (2601)	TODO sec	TODO sec
101 x 101 (10201)	TODO sec	TODO sec
151 x 151 (22801)	TODO sec	TODO sec
201 x 201 (40401)	TODO sec	TODO sec
251 x 251 (63001)	TODO sec	TODO sec

Table 5. Repast experiments on varying grid sizes on a single (1) core.

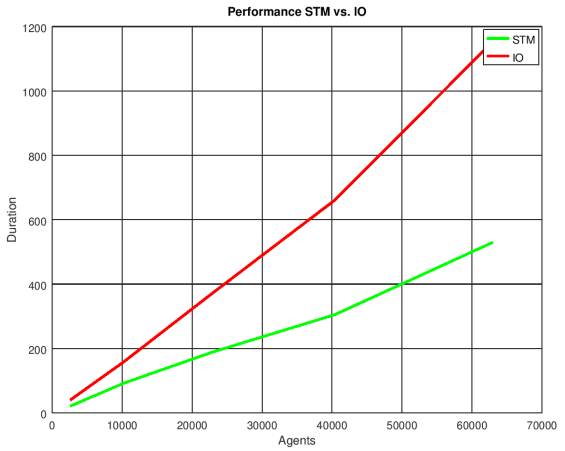

Fig. 2. Comparison of STM (Table ??) and IO performance (Table ??). TODO: put into a single performance comparison figure: State, STM, IO and RePast

The Figure 2 clearly indicates that STM outperforms the low level IO implementation by a substantial factor and scales much smoother.

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.

5.3 Retries

TODO 4 cores Averaging 8 runs again

5.4 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).

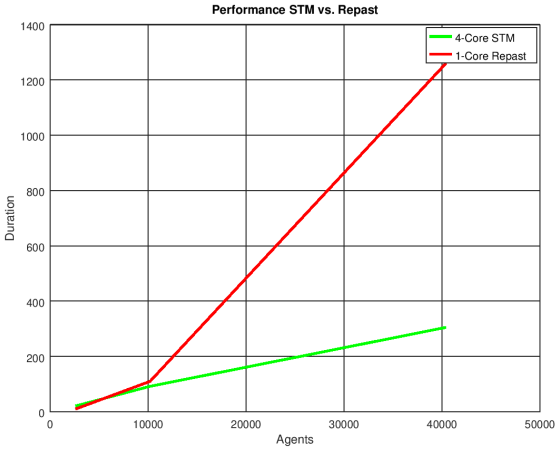

Fig. 3. Comparison of 4-Core STM and single core Repast (Table ??).

Grid-Size	Retries Ratio
51 x 51	TODO
101 x 101	TODO
151 x 151	TODO
201 x 201	TODO
251 x 251	TODO

Table 6. Retries Ratio of STM Monad experiments on varying grid sizes on 4 cores.

(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] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.
[2] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (Sept. 2010), 68–75. https://doi.org/10.1145/1810891.1810910
[3] Nikolaos Bezirgiannis. 2013. *Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism.* Ph.D. Dissertation. Utrecht University - Dept. of Information and Computing Sciences.
[4] Fernando Castor, JoÃčo Paulo Oliveira, and AndrÃĺ L.M. Santos. 2011. Software Transactional Memory vs. Locking in a Functional Language: A Controlled Experiment. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11 (SPLASH '11 Workshops).* ACM, New York, NY, USA, 117–122. https://doi.org/10.1145/2095050.2095071

[5] Antonella Di Stefano and Corrado Santoro. 2005. Using the Erlang Language for Multi-Agent Systems Implementation. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT '05)*. IEEE Computer Society, Washington, DC, USA, 679–685. https://doi.org/10.1109/IAT.2005.141

[6] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2006. Lock Free Data Structures Using STM in Haskell. In *Proceedings of the 8th International Conference on Functional and Logic Programming (FLOPS'06)*. Springer-Verlag, Berlin, Heidelberg, 65–80. https://doi.org/10.1007/11737414_6

[7] Richard H. Enns. 2010. *It's a Nonlinear World* (1st ed.). Springer Publishing Company, Incorporated.

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

[9] 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

[10] Tim Harris and Simon Peyton Jones. 2006. Transactional memory with data invariants. https://www.microsoft.com/en-us/research/publication/transactional-memory-data-invariants/

[11] Armin Heindl and Gilles Pokam. 2009. Modeling Software Transactional Memory with AnyLogic. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques (Simutools '09)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 10:1–10:10. https://doi.org/10.4108/ICST.SIMUTOOLS2009.5581

[12] W. O. Kermack and A. G. McKendrick. 1927. A Contribution to the Mathematical Theory of Epidemics. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 115, 772 (Aug. 1927), 700–721. https://doi.org/10.1098/rspa.1927.0118

[13] 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

[14] Ruth Meyer. 2014. Event-Driven Multi-agent Simulation. In *Multi-Agent-Based Simulation XV (Lecture Notes in Computer Science)*. Springer, Cham, 3–16. https://doi.org/10.1007/978-3-319-14627-0_1

[15] Cristian Perfumo, Nehir SÃűnmez, Srdjan Stipic, Osman Unsal, AdriÃąn Cristal, Tim Harris, and Mateo Valero. 2008. The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-core Environment. In *Proceedings of the 5th Conference on Computing Frontiers (CF '08)*. ACM, New York, NY, USA, 67–78. https://doi.org/10.1145/1366230.1366241

[16] Alban Rousset, BÃĪnÃĪdicte Herrmann, Christophe Lang, and Laurent Philippe. 2016. A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Computer Science Review* 22 (Nov. 2016), 27–46. https://doi.org/10.1016/j.cosrev.2016.08.001

[17] Gene I. Sher. 2013. *Agent-Based Modeling Using Erlang Eliminating The Conceptual Gap Between The Programming Language & ABM.*

[18] Carlos Varela, Carlos Abalde, Laura Castro, and Jose GulÃąas. 2004. On Modelling Agent Systems with Erlang. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang (ERLANG '04)*. ACM, New York, NY, USA, 65–70. https://doi.org/10.1145/1022471.1022481

[19] Michael Wooldridge. 2009. *An Introduction to MultiAgent Systems* (2nd ed.). Wiley Publishing.

Received May 2018