Improving Performance of Simulation Software Using Haskell's Concurrency & Parallelism

By
Nikolaos Bezirgiannis

Supervisor
dr. Wishnu Prasetya

Supervisor
dr. Ilias Sakellariou

Supervisor
prof. dr. Johan Jeuring

MSc Thesis

Dept. of Information and Computing Sciences,
Utrecht University

September 5, 2013

Abstract

Computer simulation has been widely applied in many areas, ranging from physics to biology and from economics to transportation. This tremendous applicability has the effect that, in most of the cases, simulations are constructed by scientists with non-computer expertise. These scientists are striving for ease of development and fast execution speed. We construct two frameworks, an agent-modelling framework and a pure parallel discrete-event simulation framework, both written in the Haskell programming language. Haskell language offers fast and expressive parallel and concurrency mechanisms, that we take advantage of. We benchmark our implementations against well-established competing frameworks and present the output results. The end-goal of this project is to satisfy the simulation user's criteria of usability and speed.

Acknowledgements

I would like to thank my supervisors for their guidance and helpful suggestions. I would also like to thank my girlfriend, Joëlle Walgers, for her tremendous support.

This work is funded by the Greek State Scholarship Foundation and its European scholarship programme.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Contents

1 Introduction **4**

1.1 Applications of Simulation 5

1.2 System Dynamics 7

1.3 Discrete Event Simulation (DES) 7

1.4 Research Question 9

1.5 Contribution 9

1.6 Outline 10

2 Introduction to Haskell Technologies **11**

2.1 What is Haskell 11

2.2 Green Threads 12

2.3 Mutable Variables (MVar) 12

2.4 Software Transactional Memory (STM) 14

3 Agent-based Simulation (ABM) **17**

3.1 What is ABM 17

3.2 NetLogo 19

3.3 Failed attempt in Erlang 24

4 HLogo: A parallel clone of NetLogo **26**

4.1 Introduction 26

4.2 The HLogo language 27

4.3 The HLogo Execution Model 30

4.4 Benchmark Results 35

4.5 Related Work 36
4.6 Appendix 38

5 Parallel and Distributed Discrete-Event Simulation (PDES) 43
5.1 What is PDES 43
5.2 Conservative approach 46
5.3 Optimistic approach 53

6 HDES: A lazy PDES framework 56
6.1 Introduction 56
6.2 The HDES language 58
6.3 Implementation 62
6.4 Benchmarking 63
6.5 Related Work 68
6.6 Appendix 69

7 Conclusions and Future Work 72
7.1 HLogo 72
7.2 HDES 73

References 75

Chapter 1

Introduction

Our world is surrounded by complex systems which are most of the times hard to comprehend and reason for. Yet we somehow find a way to deal with how they actually work and operate them in a satisfactory manner. Examples of such systems can be found everywhere; in human society (markets, organizations, language, internet), in biology (cells, organ – e.g. brain, immune system, organisms, populations, ecosystem), in physics (turbulence, weather, percolation, sandpile) and many other areas.

What makes a system complex? Burian (2013) defines a system to be an entity with smaller parts and relations between these parts. Following this, a *complex system* is a system composed from relatively many mutually related parts. Yet we find this definition not complete enough; according to the Merriam-Webster dictionary, under the entry for complex we discover the synonymous phrase "a group of things that are connected in *complicated* ways". So what makes a complex system to be regarded as complicated?

As Greek philosopher Aristotle famously said, "The whole is more than the sum of its parts". That means that there are emergent behaviours of a system that cannot be observed when examining its pieces individually. So there must be something complicated going on with the relations of a system.

It appears that, most of the times, the large number of relations of the parts of a system create a kind of "network" of serious complications. For that sole reason, complex systems are intrinsically hard to describe or understand. If we prefer a proper formal definition, we could say that a complex system is too complicated to have an analytical (exact, precise) representation, given the mathematical tools we have at hand (algebra, calculus).

And even if we had a representation for such a system, there are certain circumstances where it can become impractical to measure the performance of an actual system; reasons can be the costs for building such a system (moon landing) or the time required to execute it (meteorology). Still though, we can aim

for an approximate estimation of such a concrete system, that can arguably be adequate enough. That's exactly where simulation comes to play.

The term simulation refers to the imitation of the processes of a concrete system. Such a system can live in the real-world or, on an opposite direction, be completely fictional. In simulation, we try to estimate numerically the real performance of a system, which unfortunately, as said before, is too complicated to have an analytical solution. In that context, simulation is applied to give answers to specific questions, i.e. decision support (examples of such questions are: how fast is the system? , where is its bottleneck?, how can we maximize its throughput?)

Simulation is often confused with the term *Modelling*. A model according to Law (2007) is a set of assumptions about how the system works. From a computer science perspective, we could say that a model is the algorithms and/or equations that best describe the behaviour of the system. Modelling is the process/methodology of finding the most suitable such algorithms that describe the system, whereas Simulation refers to the correct execution of these algorithms and collecting the model's performance results, so we can later make decisions upon them.

It is not strange why the simulation technology started to blossom around the time the first computers came to existence. Apparently, it is extremely hard to execute these equations/algorithms simply on pen and paper; an automated infrastructure and a fast math solver that computer systems provide can aid in this. It all began half a century ago, with the notorious Manhattan project, during World War II. The goal was to model the process of a nuclear detonation and the result of it can still be witnessed today in the popular, from the other end, Monte Carlo method. For the rest of the thesis we are only considering computer simulation and we inadvertently use the simple term simulation to refer specifically to computer simulation.

1.1 Applications of Simulation

Transportation

Consider an unexpected hurricane affecting air traffic of a country. How the decision making of this problem is influenced? Should airplanes be allowed to depart, only to circle on the destination? Should airplanes be remain grounded? Should there be some kind of rerouting to alternative destinations? How much will be the frustration of the passengers (average waiting time)?

Economics and Social Sciences

Examples of these include, the effects of a surprising announcement on the stock market, how a declaration of war will affect the economy, what is the better suited economic model for the eurozone. A famous example of application in the Social Sciences, is how the ghetto communities are formed even with high tolerance between the factions.

Biology

A possible spread of a virus is very catastrophic. Simulations can investigate cases of virus contamination in the population, how the spreading works, what are the best quarantine systems.

Computer Systems

An example could be the designing of a next-generation internet protocol. How many web surfers are expected? How much will be the tolerance of delay? What will happen in unexpected failures of the network?

Weather Forecast

Simulation models with a huge size are created and executed, to make future predictions about the weather in a country or even the change in the world climate.

Military

In wargaming, different military tactics and strategies are evaluated, so as to determine the gear that is required. Simulation can be also used as a training facility for the military personnel.

Simulation can be categorized to two distinct types, based on how the time evolves; continuously (System Dynamics) or in discrete steps (Discrete Event Simulation).

1.2 System Dynamics

The model in System Dynamics (also known as Continuous Simulation) is represented as a set of differential equations. The rate of change of the time of the model is on a continuous scale. Let's consider for example the famous Lotka - Volterra equations that appeared in 1925 to describe originally the population of foxes and rabbits:

$$\frac{dx}{dt} = Ax - Bxy$$
$$\frac{dy}{dt} = -Cy + Dxy$$

where

x prey population

y predator population

A the growth rate of prey

B the rate at which predators eat their prey

C the death rate of predators

D the growth of predators by consuming prey

A rather controversial application of System Dynamics was the large scale computer simulation of the so-called Limits to Growth Meadows (1972), that tried to estimate the future economic and population growth of our planet.

On a computer, a system dynamics simulation is implemented using a specific (small) interval for the dt increments, so in practice it is demoted to a time-stepped simulation. A time-stepped simulation, contrary to the Discrete Event Simulation described below, advances time in equal intervals. On each interval, the simulation updates the state of the model and estimates its behaviour.

1.3 Discrete Event Simulation (DES)

The state of a simulated model is comprised of one or more distinct state variables. The type of each state variable can be arbitrary and is defined by the user. For example, if we were modelling a queuing service facility of a bank, we would have as state variables:

status of the counter (idle or busy, so of type `Boolean`)
number of customers waiting (type `Int`)
arrival time for each customer (type `list of Double`)

In Discrete Event Simulation, the state variables, in contrast with Continuous Simulation (System Dynamics), change instantaneously in discrete points in time. These time points are signaled by an event. A firing of an event (event handling) may change some or all the state variables of the model. Consider the bank example, and the events happening at each time point:

Event	Time Point (s)
`CustomerArrival`	10.5
`CustomerService`	11.5
`CustomerDepart`	15
`SimulationStop`	22

The event `CustomerArrival` increments the number of customers waiting by 1. The event `CustomerService` makes the status of the counter busy, and decrements the number of customers waiting by 1. The event `CustomerDepart` computes the time of service of the customer. The event `SimulationStop` does not change any state variables, but simply terminates the simulation.

The logic of what gets changed upon event firing is solely determined by the event handler, that has to be provided by the simulation user.

On every model, there is an extra state variable, the *simulation clock*, that holds the current time point of the simulation. During initialization, the simulation clock usually (but not necessarily) takes the value 0, although this restricts the type of the simulation clock to be a number instance (int, double …). We later depart from this restriction to allow arbitrary types that can be ordered, like for example a `Date` type.

The simulation must always advance forward in time up until the stop of simulation, and the simulation engine will assure us that there is no violation by "time travelling" to the past.

But when does the execution of a simulation stop? The stopping condition can be anything of:

- a maximum simulation time. When the simulation clock reaches this time, the simulation halts.
- when there are no more future events to be scheduled.
- an arbitrary stopping condition. For example when a state variable becomes 0.

An example of the execution can be better visualized with Figure 1.1, a diagram of time where event jumps happen:

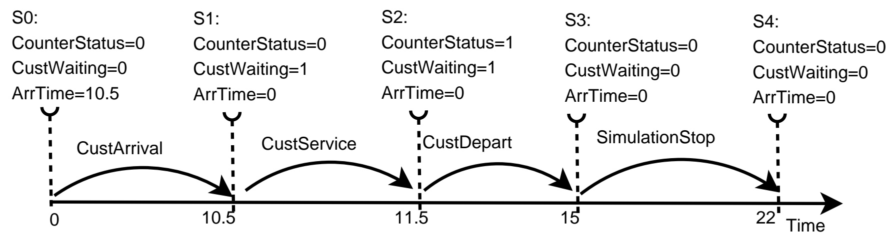

Figure 1.1: Illustration of advancing the state of the simulation by event handling

Internally, the simulation engine manages an event queue, that holds any future events of the simulation. At every loop, the engine picks the next earliest event and pass is it to the user-supplied event handler. After the event handler is finished, the engine updates the state variables and progresses to the next event in the queue.

1.4 Research Question

The current software for Discrete Event Simulation has certain limitations:

1) The simulation user often has to specify her model in an obscure and uneasy programming environment.
2) There is not enough exploit of parallelism as there should be based on the recent “booming” of multicore.

The research question that we investigate is whether an embedded implementation in a lazy functional language would improve that.

1.5 Contribution

We provide a better environment for the user to write Discrete Event Simulations in. The extra benefit is that the user can get speedup in execution, if her model adheres to the design of our parallel engine, which employs the ubiquitous, by our times, Symmetric Multiprocessing technology (SMP).

The result of our work is realized through two distinct (but related) programs. The first is `HLogo`, a clone of the NetLogo agent-modelling framework, that offers certain language advantages compared to NetLogo, and execution speedup in common model cases. The other program is named `HDES`, a Parallel Discrete

Event Simulation (PDES) framework, that strives to be both simple in its interface and also fast enough to compete with other PDES software. Both are written in Haskell and try to exploit the language's parallel and concurrency capabilities.

1.6 Outline

The rest of this thesis is organised as follows. The next chapter will lay down a short introduction to the Haskell programming language and deal with the parallel and concurrency technologies behind it. In chapter 3, we will discuss what agent-based modelling (ABM) is, introduce the reader to the NetLogo platform, and describe an initial naive clone of NetLogo in the Erlang programming language. Chapter 4 introduces the **HLogo** language and framework, which is our proposed clone of NetLogo that enables parallel execution. The chapter includes details to the altered execution model and the results after benchmarking the framework against NetLogo. Chapter 5 talks about Parallel and Distributed Discrete-Event Simulation (PDES) and the various synchronization algorithms. We also provide an trivial synchronization algorithm implemented in Erlang. People familiar with PDES may as well skip this chapter. The next chapter 6, proposes yet another PDES framework, called **HDES**, which has lazy evaluation and is implemented solely ontop of the Haskell Programming language. We provide documentation for the HDES language and discuss its implementation. We later try to test the lazy features of the HDES framework and benchmark it against a competing state-of-the-art PDES framework. The benchmarking results are provided. Chapter 7 concludes this thesis.

If you are eager to work with the implementation code, you can get git repositories of HLogo, HDES and the sources of the thesis at:

- HLogo repository: git clone git@repo.bezirg.net:/hlogo.git
- HDES repository: git clone git@repo.bezirg.net:/hdes.git
- Thesis repository: git clone git@repo.bezirg.net:/msc_thesis.git

Chapter 2

Introduction to Haskell Technologies

2.1 What is Haskell

Haskell[1] is a programming language first standardized by a comittee back in the year 1987. The programming paradigm that Haskell follows is called functional and that is for a reason; functions are the main programming concept used. Functions are treated as first-class citizens inside the language.

Compared to other common functional languages, like Scheme and OCaml, we could say that Haskell is a *pure* functional language. That means that the user cannot in any way intermix code with side-effects inside pure mathematical expressions. This restriction comes with the benefit of *referential transparency*: the user can replace an expression with its value without changing the behaviour of the program. Mathematical methods such as *equational reasoning* can then be safely applied to Haskell code.

Haskell is one of the few programming languages that offers non-strict (*lazy*) semantics. Its evaluation strategy departs from the usual *call-by-value* of the strict languages. Haskell's *call-by-need* evaluation order dictates that the arguments to a function need not be evaluated before the function is called; instead they are substituted in the function body and only evaluated if their result is needed by the expression. This evaluation strategy also employs *memoization*; the values of the function arguments are "cached" for any subsequent calls. Memoization can potentially lead to execution speedup.

The type system of Haskell is characterized by strong static-typing. The language uses the Hindley-Milner method to automatically infer the most general

[1] The Haskell Programming Language homepage http://haskell.org.

(principal) types for expressions. The *parametric polymorphism* of the language allows for writing functions generically; functions that can handle any arguments without depending on their specific types. Haskell also offers support for the so-called *typeclasses*. These are not the usual classes found in Object-Oriented programming languages. Typeclasses are instead used by the language for enabling *ad-hoc polymorphism*, that is found in other general-programming languages, such as C++ and Java.

2.2 Green Threads

Lightweight threads (also known as green threads) are virtual threads managed by a virtual machine (or by a language's advanced runtime-system), instead of the Operating System. We could say, in terms of OS concepts, that green threads exist solely in user space than the kernel space. For this reason, they have a smaller memory footprint than normal native threads and faster thread activation and synchronization. This allows us to spawn millions of green threads in a single system without running out of memory. However, this does not come without a price; green threads usually tend to be slower in terms of IO and context switching compared to native threads.

In reality, the virtual machine (or runtime system) initially spawns as many native threads as the processor capability of the CPU (being a dual core or quad-core etc.) and keeps them activated. Then on, it picks some (of the possible many) green threads and schedules them for execution by assigning them each to an active native thread. If the system is unicore (processor capability equals one) then as you can guess there is effectfully concurrency going on with however no parallelism gain, since there is only one native thread scheduled at a time.

There exist implementations of green threads in different programming languages, such as CPython, Go, Occam and Haskell. Haskell lacks a Virtual Machine and the threading is rather handled by the Runtime System and its process scheduler. Haskell's green threads coupled with Symmetric Multi Processing (multicore) may allow a faster program execution if carefully designed.

The SMP capabilities of Haskell have greatly benefited from a recent change to the Glasgow Haskell Compiler (GHC). GHC is the default Haskell compiler and tries to stay in the forefront of Haskell's evolution. The change affects the generational-copying garbage collector (GC). The GC can now act in parallel and make use of any extra cores of the system to speed up its process of Garbage Collecting.

2.3 Mutable Variables (MVar)

An other concurrency technology of Haskell besides the well-known Software Transactional Memory (STM), described in 2.4, is the so-called MutableVariable

(or MVar for short). MVar is the lowest-level communication abstraction there is in the Haskell ecosystem (Marlow 2013). It can only be used for shared memory communication; thus by definition cannot be applied in a distributed setting. The interface when working with MVars is extremely simple:

```
newMVar :: a -> IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
```

An MVar can be thought of as a memory cell that it is either empty or full. The `newMVar` operation creates an MVar and stores a value to it that was passed as the argument. The `putMVar` operation stores a new value to an already-created MVar. The MVar memory cell has to be empty before storing the new value. If, however, the MVar is currently full, the thread that calls putMVar will wait until it becomes empty. The `takeMVar` construct takes a value out of the memory cell, thus dimming the cell empty, and returns it to the caller. During a `takeMVar` operation, if the MVar is not full, then the thread that called `takeMVar` will have to wait until a value is "put" back, so it can again take it. When multiple threads are blocked on an MVar, they are woken up in FIFO order. This is useful for providing fairness properties of abstractions built using MVars (for more about this see the GHC documentation on MVars). The blocking/waiting of threads is internally realized with good-old imperative locks on the memory cells.

Although MVars are now a great part of the GHC compiler, the MVar implementation first appeared in the no-longer-maintained `hbc` Chalmers' Haskell compiler, which happens to be the first Haskell compiler ever, written by Lennart Augustsson.

We could build higher abstractions based on the low-level MVars, such as unbounded buffered channels. A producer can store items to the channel with no particular limit on the number of stored items (except of memory limits ofcourse). A consumer (or multiple consumers) can take (read) one item at a time from that channel. The idea first came up in the Concurrent Haskell framework (Jones, Gordon, and Finne 1996). We can say that the definition of the `Chan` abstraction in Haskell is nearly crude. A `Chan` is a normal datatype holding two MutableVariables (MVars). The first MVar points to the read position, so it used by the consumer; the second MVar is called the write position to be used by the producer.

```
data Chan a = Chan
                    (MVar (Stream a))
                    (MVar (Stream a))
```

A `Stream` is a mutually-recursive datastructure that holds an MVar to an `Item` that itself holds another stream, thus forming a kind of chain of items. This chain can be better illustrated in Figure 2.1.

```haskell
type Stream a = MVar (Item a)
data Item a = Item a (Stream a)
```

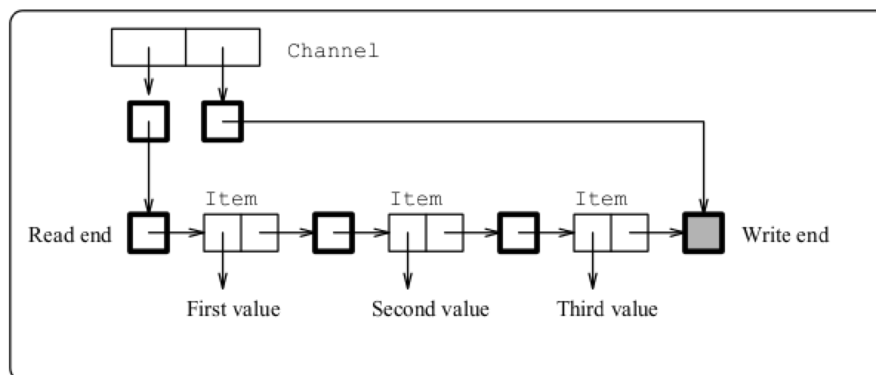

Figure 2.1: A channel with unbounded buffering implemented using MVars

MVars and particularly Chans are used extensively in the HDES framework, discussed in chapter 6, to provide the underlying communication.

2.4 Software Transactional Memory (STM)

Software Transactional Memory (STM) is a technology originating from the field of Distributed Databases. When it comes to concurrency, STM departs from the common locking mechanism and offers a concurrency paradigm that avoids locks altogether. When combined with Symmetric Multi-Processing (SMP) it can lead to considerable program speedup as reported by Perfumo et al. (2008).

This concurrency mechanism relies on the so-called *transactions*. A transaction is a sequence of reads and writes to variables (Transactional Variables, TVars for short). Transactional variables point to actual places in shared memory. TVars are CRUD (Create-Read-Update-Delete) mutable variables that can be created, read and and written only exclusively inside an atomic transaction. TVars are normally shared between transactions. Transactions are said to be applied *virtually* instantaneously, so no intermediate state changes can be witnessed by other transactions. People that are familiar with SQL databases may better treat STM transactions as the usual SQL atomic transactions albeit with a bigger focus on concurrency.

It is the common case that for each transaction a separate thread is spawned/assigned. Each transaction, thus each thread, keeps a separate log of reads and writes to TVars. At the end of the transaction, the thread checks for possible incosistencies in the log. If there was an incosistency during a

transaction — for example a transaction contains a read of a shared TVar that was written to in the mean time by another transaction — the transaction aborts with no effects to the shared memory, and retries later in the future. If there are no incosistencies, then the transaction is said to be succesful, and the associated code is actually committed (executed). Internally, the orchestration of a commit is realized by putting locks automatically in proper places in shared memory, although this process is absolutely hidden from the user.

Atomic transactions can be created by surrounding blocks of code with the term `atomically`. Let's consider a common example of bank accounts using the Haskell programming language. We want to make a financial transaction between two accounts:

```
send_money x = atomically (do
            v1 <- readTVar account1
            v2 <- readTVar account2
            writeTVar account1 (v1-x)
            writeTVar account2 (v2+x)
            )
```

The code of `send_money` runs on thread-1. The other, thread-2 runs the code:

```
check = atomically (do
    v1 <- readTVar account1
    v2 <- readTVar account2
    )
```

Note that the execution of the threads is interleaved. If we didn't have atomicity, then it could be the case that thread-2 possibly is in a state that can witness money leaving from account1 but not reaching account2.

STM, on the other hand, guarantees us that this case cannot happen, because these two blocks of code (`send_money` and `check`) cannot be interleaved when they have dependencies with each other. In reality, what happens is that the code inside atomically is optimistically run by each thread. The thread reads the transactional variables but delays writing to them; it only keeps a log of what has to be later written. In the end of the block, upon commiting the transaction, if the thread has found an inconsistency (e.g. a transactional variable has been changed in the mean time) it rollbacks and retries the whole transaction. In fact, the user herself can direct the thread to `retry`, somehow creating a kind of thread blocking. Consider the slightly altered but better `send_money'` function:

```
send_money' x = atomically (do
                v1 <- readTVar account1
                if (v1 < x)
```

```
    then retry
    else do
      v2 <- readTVar account2
      writeTVar account1 (v1-x)
      writeTVar account2 (v2+x)
)
```

The example is self-explanatory. The thread is blocked when the amount to transfer is less than the from-account. When another thread adds money to the from-account, the thread will be waken up and retry this transfer-transaction altogether.

STM is usually implemented as a software library on top of a programming language that, because a unique thread is usually spawned for each transaction, has to support lightweight (green) threads. If instead native OS threads are used by the language, this will limit the number of simultaneously active transactions.

Software Transactional Memory surprisingly was established a long time ago with Knight (1986). The author proposes a novel extension to Lisp and suitable hardware modifications to enable concurrency by avoiding locks. The idea was later refined by Shavit and Touitou (1995), who also coined the term Software Transactional Memory (STM), to allow the technology to operate only in-software.

In the Haskell programming language it was first implemented in Harris et al. (2005) and later refined in Harris and Peyton Jones (2006) and Discolo et al. (2006). STM in Haskell is modelled as a seperate monad, so as to ensure that transactional code cannot be run outside of transactions. With this restriction, the user cannot run arbitrary IO effects inside transactions, since these effects are not safe and cannot be rollbacked. This restriction is enforced by the Haskell's strong type system. Haskell's STM is arguably the best implementation there is, although it can be in many cases cumbersome to program in, since it is difficult for the user to combine the separate pure world of expressions with the STM monadic world. Its excellent implementation and safety guarantees are why we chose Haskell for our next generation NetLogo-clone, i.e. HLogo.

In Distributed Databases terms, STM offers reliable transactions for software. It cannot be regarded however a silver bullet, especially when STM is used for added parallelism to simulation software. It appears that there is a strong property in Parallel and Distributed Discrete Event Simulation theory, called the local causality constraint. STM apparently violates this property, which has as a result the non-repeatability of simulation experiments. More about local causality constraint in Section 4.1.

This shortcoming, however, did not restrain us from using Software Transactional Memory to speed up the execution of HLogo programs, discussed in Chapter 4.

Chapter 3

Agent-based Simulation (ABM)

3.1 What is ABM

Agent-based Simulation (ABM for short) departs from the usual Discrete-Event Simulation on how the particular model of the system is structured and layered. As you can guess, the basic building block of an ABM are *agents*.

There exist many different interpretations of what the term agent consists of. We find the Wooldridge (2009) definition quite satisfactory:

> The agents themselves are more or less "intelligent" chunks of computer code that are able to perceive and communicate with each other and react to stimuli in order to pursue their goals.

When compared to a regular Discrete Event Simulation, their only difference lies in the way their models are defined and expressed in a simulation. In ABM, we can say that a bottom-up approach is employed compared to a top-down approach in Discrete-Event Simulation. What that normally means is that the simulation user has to define the logic of each agent involved (bottom-up), instead of a single big monolithic event handler (top-down). From this follows that the behaviour of the system *emerges* from the particular communication and interactions of these sole agents. As ABM is generally implemented on top of a discrete-event simulation framework, some consider ABM to be a certain subclass of DES techonology.

ABM is also similar to Multi Agent Systems (a specific branch of AI and distributed systems), albeit having entirely different goals; the first for simulating

(imitating) models of systems and the latter for actually constructing such systems to be put in real practical use.

Continuing with which technologies ABM is linked to, we can claim that ABM can be also regarded as another kind of Spatial Modeling. Normally, the agents in ABM are situated in an artificial environment which has spatial characteristics (most of the times, either two- or three- dimensional).

An Agent-Based simulation can be better explained with an example. Consider an environment of initially 150 people (modelled as agents), of which 10 are infected (red) and the rest 140 are healthy but prone to infection (green). The agents move randomly in the 2D environment. At a later stage, we can view how the virus spreads through the agent population and how a certain part of it becomes immune to the virus (gray). The Figure 2.1 illustrates how good is ABM when it comes to visualizing a simulation:

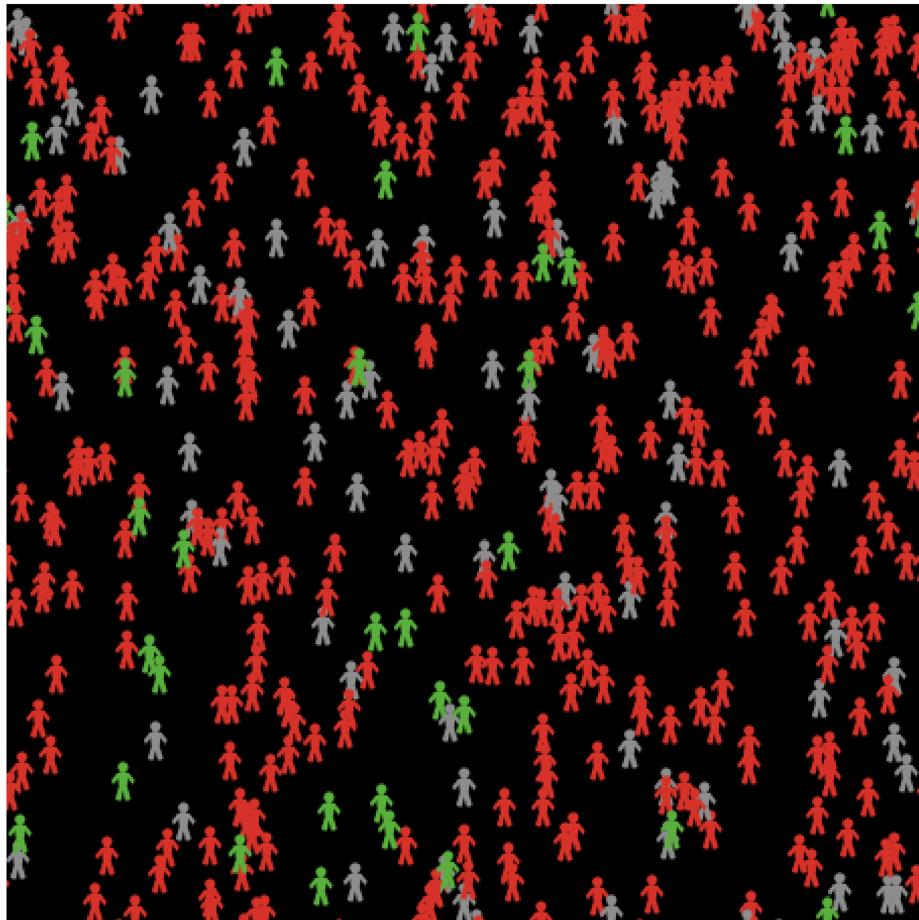

Figure 3.1: An example of Agent-Based Modeling simulating a virus infection

Agent Based Modeling (ABM) has gained traction lately, especially with its ease of use and wide area of applicability. We have witnessed ABM already being applied to areas mostly related to social sciences Epstein and Axtell (1996) and to a lesser extent to ecology Grimm et al. (2005), biology Pogson et al. (2006), physics Wilensky (2003) etc. Agent Based Modeling is a simulation technique that has as its building block the autonomous intelligent agent. ABM offers the right kind of high-level abstraction to construct from the ground up large agent populations that exhibit interesting (from a simulation perspective) emergent patterns.

When is ABM appropriate against a crude Discrete Event Simulation model? According to Salamon (2011) an Agent-based modeling is a suitable method for the study of a problem where we know the characteristics of the atomic parts and are interested in the behaviour of the entire system.

As ABM stays close to the theory of MultiAgent System (MAS), it is well established that often ABM code involves serious AI programming. In our thesis, we are not concerned in program expressivity and any AI techniques, but only keep focus on the usability of ABM and its implementation details.

3.2 NetLogo

NetLogo[1] is arguably the most well-known and appreciated ABM platform. It provides to the simulation user a simple-to-use programming language and an intuitive modeling environment that includes an IDE to accommodate the writing of NetLogo code as well as a GUI to watch the live execution of the compiled model.

NetLogo sprang off in 1999 by its creator Uri Wilensky as an advanced replacement to a similar framework, named StarLogoT. Since then, it has picked up a lot in the simulation community, mostly on the fact of its ease of use and rich built-in models library. Areas of the models include Biology, Computer Science, Earth Science, Social Science, Chemistry and others. An example of a Cellular Automaton (Game of Life) live simulation in the NetLogo IDE is shown in figure 3.2.

Besides simple 2D environments as the one in 3.2, NetLogo also supports 3D lattices with 3D-graphically enhanced views.

NetLogo code was recently open-sourced under the GNU General Public Licence (GPLv2). Internally, NetLogo is implemented in the Scala programming language. A Scala-written compiler translates NetLogo code to Java bytecode to be later run in a Java Virtual Machine (JVM). NetLogo can be considered with this method partially compiled. They are currently working towards a full

[1] Wilensky, U. 1999. NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.

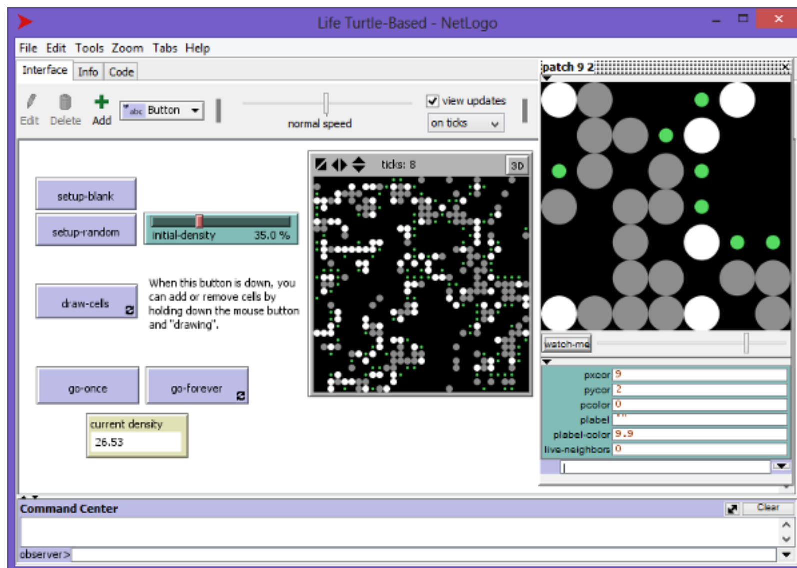

Figure 3.2: NetLogo Integrated Environment and visualization of a Game Of Life simulation

compiler, as well as a full-blown Javascript (in-browser) implementation (check Teletortoise). To be absolutely precise, according to their wiki:

> NetLogo does include a compiler that generates Java byte code. However, this compiler does not yet support the entire language, so some parts of user code are interpreted. We are working on expanding the compiler to support the entire language. Note that our compiler generates Java byte code, and Java virtual machines have "just-in-time" compilers that in turn compile Java byte code all the way to native code, so much user code is ultimately translated to native code.

3.2.1 The NetLogo language

The NetLogo framework suite defines also its own programming language, derived as a Logo dialect.

Logo is a graphic-oriented educational programming language, where the famous *turtle graphics* originate from. It is a simplistic though extremely powerful language — it even has Lisp macros. That is the reason why Logos' nickname is *Lisp without the parentheses.* It's first implementation was a Logo interpreter written in Lisp.

NetLogo's dialect departs from a standard Logo specification in that it has lexical scoping instead of the usual dynamic scoping of Lisp-like languages, the

code gets compiled to bytecode (or native code) instead of being interpreted and even has support for multiple turtles, instead of just a single one of most Logo implementations. From the other hand, it lacks any metaprogramming by not supporting backquotes (atoms) and macros. As far as we are aware of, NetLogo's implementation does not offer tail-call optimization (TCO), which is unfortunate.

The agents in NetLogo are situated in a 2D or a 3D spatial environment. There are essentially 3 types of agents in NetLogo: turtles, patches and links. Turtles, as in turtle graphics, move around the environment drawing shapes to the screen. Turtle agents can be dynamically created and destroyed at run-time of the simulation execution. A turtle during creation is given a unique Int number for referencing, called the `who` value. Patches are stationary agents and draw only on the tile they occupy. They are only spawned at the start of the simulation and cannot be later destroyed. A link connects two turtle agents together and can also visually represent this connection (for example as a directed line). The links, as it happens with turtles, can also themselves be created and destroyed at runtime. Every agent can communicate and interact with any other agent of the environment.

The language comes with a vast standard library for essential math programming, statistics, input/output and plotting to the screen. Three of the many primitives of the standard library, namely `ask`, `of` and `with` are responsible for the communication and interaction between the agents of the system. The first primitive is used by the current executing agent to instruct a set of agents to do something. The agent can use the `of` primitive to query for variables and attributes of other agents. The `with` primitive is a special case of the `of` primitive, and returns a subset of a given agentset that satisfy a certain predicate. The primitives are combined and illustrated in the following example:

```
ask turtles with [colour = red] [
    print [pxcor] of patch-here
    ]
```

In the example, every turtle that is colored red is instructed to print the x coordinate of the patch it sits on.

This is the most essential kind of interaction, that instructs other agents to perform an action.

On example 2, turtle 4 "asks" turtle 5 of its colour and prints it to the command line.

```
ask turtle 4 [print [color] of turtle 5]
```

The `of` primitive simply acquires information from another agent:

```
if [xcor] of turtle 3 < 0 [print "x coordinate of turtle 3 smaller than 0"]
```

The `with` construct takes a predicate and returns the set of agents that satisfy the predicate:

```
print count turtles with [color = red]
```

It comes that the `with` construct can be defined in terms of `of`. We spare you from its implementation.

Interaction happens through global or per agent variables. NetLogo also supports local variables, which as we said before, are lexically scoped.

```
globals [var1 var2 var3] ;; list of global variables
turtles-own [tvar1 tvar2 tvar3] ;; list of per-turtle variables
patches-own [pvar1 pvar2 pvar3] ;; list of per-patch variables

ask turtle 4 [set var1 3]

ask turtle 5 [set var1 var1 + 1]
```

NetLogo's breeds are a way to categorize turtles. Each breed can have its own set of attributes, forming some kind of primitive Object Oriented Programming (OOP).

```
breed [cats cat]
breed [mice mouse]
cats-own [energy] ;; per cat variable
mice-own [speed] ;; per mouse variable
```

Links is a unique feature to associate two agents together. A link can itself be regarded as an agent, and as like a turtle, be dynamically created and destroyed on run-time.

NetLogo offers a kind of structured programming, by allowing the users to define their own procedures (functions) with the form:

```
to proc_name arg1 arg2 arg3 rest
    ...
    return 3
end
```

NetLogo allows variable arguments in procedures (polyvariadic functions) , similar to what the Common Lisp does with its `&rest` keyword.

Besides procedures, NetLogo has the usual control flow structures of other imperative languages: `ifelse` , `foreach`, `while`, `loop`.

Suprisingly, NetLogo lacks a module system. This becomes a burden during large deployments, but it is justified in the sense that inexperienced users are often put off by modular programming.

3.2.2 Execution of NetLogo code

NetLogo's execution order is the usual call-by-value. Its execution model is completely sequential; there is no concurrency or parallelism involved. That means that when an "ask" primitive is called, it simply jumps from an agent context (most of the times that is the top-level/observer context) to another agent context, creating a kind of chain of dependencies (values) between agents. This model is hard to parallelize, because these dependencies will create a lot of race conditions if they happen to be overlooked by the different computing processors.

If we would try to visualize the execution sequence of the example give before, we could come up graphically with the Figure 3.1.

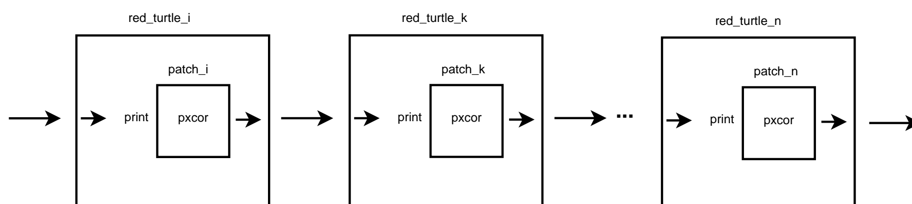

Figure 3.3: The execution context-passing for the red-turtle example

As you can witness yourself, there is room for improvement in this kind of NetLogo programs. For our case example, instead of waiting on each agent to finish executing and then pass execution to the next agent, thus creating a sequence of executions, we could execute each agent's associate code in parallel and collect their results. This is the main idea behind HLogo with considerations of course on certain topics of atomicity and replication, discussed later on, in the next section.

On the other hand this execution model is completely deterministic; consequent simulation runs of the same model will yield exactly the same results. This comes as a strong advantage if the simulation user is looking for repeatability of simulations. This is often the case when simulation users have to distribute their work and reproduce their model results at the other end to prove their correctness.

3.3 Failed attempt in Erlang

Our first attempt in this thesis was to clone NetLogo to a complete Erlang-written implementation. Erlang is an impure functional dynamically-typed language with a strong incline towards concurrency. The Erlang programming language has something that comes to close to green threads, called green processes. Despite having also a small memory footprint, green processes differ from the usual green threads in that the memory is not shared between any processes; communication is achieved strictly by message-passing.

When trying to implement this clone, we decided to model each agent (being a turtle, patch, or a link) as an Erlang process. This creates a nice and simple one-to-one mapping between agents and processes for the execution model. However, as discovered later, it happens to be inherently wrong. We include this failed attempt into our thesis only for stating the simple fact that the one-to-one mapping, which is the first thing that usually comes to mind (and the most natural intuition afterall), does not work in practice.

We could say that the implementation was easy and straightforward, firstly because of the built-in process mechanism that Erlang provides and secondly for reasons that Erlang matches the characteristics of the NetLogo language. Erlang, albeit functional, is an impure strict dynamically-typed language, so the user can easily mix side-effects with pure expressions. Since NetLogo also happens to be impure, strict, and dynamically-typed, programming in both those languages can be done semantically in the same exact way, minor some differences in the syntax. We only note this now, because this, unfortunately, isn't the case for our next-generation Haskell-based clone of NetLogo, called HLogo, that is discussed later on.

In this way, we've managed to port almost 100% of the NetLogo's API and standard library to the Erlang clone. Only later, when we moved forward to writing regression and unit testing against our API, we discovered that there is an intrinsic error in our execution model. This can again be better illustrated with an example. Take a look at this absurd however absolutely valid NetLogo code:

```
ask turtle 0
    [ask turtle 1
        [ask turtle 0 [show color]]]
```

What is going on, is that turtle 0 asks turtle 1 that asks back turtle 0 to print turtle 0's color. This, when executed in NetLogo, will actually print a color number.

When we express this in our Erlang clone, we have:

```
p:ask(p:turtle(0), fun () ->
    p:ask(p:turtle(1), fun () ->
```

```
        p:ask(p:turtle(0), fun () -> p:show(p:color()) end)
        end)
    end)
```

If we actually ran this in our implementation, we get a *deadlock* upon execution. Why is this? Consider, mentally, what are the processes which are going to be spawned here. With the outer call to ask the main process (shell process here) asks the process turtle0 to execute the code of the outer lambda `fun`. The process turtle0 picks up the `fun` and executes it, blocking on it, so it can return the result of the `fun` back to the main process. Upon execution of the outer `fun`, turtle0 process asks turtle1 process to execute the middle lambda `fun`. The turtle1 process picks up the code and asks back turtle0 process to execute some code (in this case, show its color) and blocks for turtle0's reply. However, the process turtle0 cannot execute what the turtle1 wants to, because it has already blocked waiting for an answer from turtle1! So, both processes are blocked waiting on each other's results, leading on a deadlock. This cannot be easily alleviated, because of the inherent fault in the execution model of the Erlang clone.

Because of this, the general idea was scrapped, but still you can look at it as a working but limited NetLogo clone at these repositories:

- The ported NetLogo API to Erlang:
  git clone git@repo.bezirg.net:/erlogolang.git

We also built around it a fully functional GUI, that stays as close as possible to NetLogo's GUI. Our intention, as future work, is to port this GUI also to HLogo. For now, you can get it from:

- The GUI of the NetLogo:
  git clone git@repo.bezirg.net:/erlogogui.git

Our next clone was written in Haskell and by having a totally different model of execution, does not suffer from these kinds of deadlock.

Chapter 4

HLogo: A parallel clone of NetLogo

Abstract

Agent Based Modeling has become quite popular to the simulation community for its usability and wide area of applicability. However, speed is not usually a trait that Agent Based Modeling is characterized of attaining. We propose yet another Agent Based Modeling framework, called HLogo, that by being a direct clone of the NetLogo platform, strives to actually be as easy to use as NetLogo, without any compromise to execution speed. The case is that HLogo, by exploiting parallel multicore execution through the technology of Software Transactional Memory, can be in certain cases faster than its NetLogo counterpart.

Keywords: Agent simulation platforms, Agent programming languages, Parallel Simulation, Software Transactional Memory

4.1 Introduction

The aforementioned success of ABM sprang off numerous ABM implementations of computer frameworks that alleviate the development of these Agent based models (for a comparison of frameworks see Tobias and Hofmann 2004; or Railsback, Lytinen, and Jackson 2006; also Castle and Crooks 2006). While ABM research has prior been focused on the methodology by Salamon (2011), ease of use by Wilkerson-Jerde and Wilensky (2010), portability by Grimm et al. (2006) and expressivity by Sakellariou, Kefalas, and Stamatopoulou (2008) of Agent Based Models, little has been done in the actual performance of the

available ABM platforms and frameworks. The problem has been stated before in P. F. Riley and Riley (2003), and although some solutions since then have been offered, there has been no wide consensus yet established.

With our work we strive to speed up the Agent Based Modeling and Simulation execution by using yet again another concurrent method called Software Transactional Memory, discussed earlier in Section 2.4. We apply our ideas to the NetLogo suite. We have to note, however, that our ideas are in no sense restricted to NetLogo; our intuition is framework-agnostic and thus could be easily applied to other ABM frameworks too.

We contribute a parallel clone of NetLogo, the HLogo language and framework that

- utilizes many processor cores to speedup the execution of Agent Based Models.
- has non-deterministic execution, in case the user demands it.
- stays as close as possible to the NetLogo philosophy.
- tries to be faster in most circumstances than NetLogo.
- offers a domain specific language embedded in the Haskell programming language which lifts certain restrictions of the original NetLogo implementation.

4.2 The HLogo language

The HLogo framework defines a simulation language to interface with the core of the altered execution model, which is described later on. This domain specific language (DSL) is also embedded in Haskell; that means that certain characteristics of the HLogo language are borrowed/inferred from the host language, i.e. Haskell. Embedded DSLs come with certain advantages and shortcomings, although however the Hlogo language tries to stay as close as possible syntactically and semantically with the original NetLogo language.

In terms of syntax, the HLogo and NetLogo are quite similar, because of the simple juxtaposition of function application in both languages. The only difference lies in precedence of binary operators: in NetLogo binary operators behave as normal functions consuming input as it comes, where in HLogo binary operators have lower precedence than normal prefix function notation. This shortcoming can be alleviated by using Haskell's application operator (`$`). For example in NetLogo we write `print 1 + 3`, whereas in HLogo we have to write `print $ 1 + 3`.

In Haskell, functions are first-class citizens. For this reason, procedures (functions) in HLogo are introduced as local (or top-level) variables; there is no extra special syntax for the introduction of procedures. For example, in NetLogo we have to write:

```
to-report inc x
    report x + 1
end

to main
    print inc 3
end
```

whereas in HLogo we write:

```
inc x = x + 1
main = print (inc 3)
```

Semantically the languages are quite different. NetLogo has a traditional strict evaluation with call-by-value evaluation order. HLogo borrows the lazy evaluation (call-by-need evaluation order) from Haskell. Call-by-need evaluation order can optimize away repetitive calls to the same expression, called sharing optimization. Also a lazy evaluation scheme allows a more expressive and powerful language, since constructs such as infinite lists can be expressed: `print $ take 10 $ [1..]`. The main advantage of lazy evaluation, however, is that expressions that are not needed will not be evaluated, leading to speed up. Consider in NetLogo:

```
to slow_computation
    ...
end

to main
    slow_computation
    report 3
```

based on strict evaluation the `slow_computation` has to be evaluated, whereas that is not the case in HLogo:

```
slow_computation = ...
main = do
    slow_computation
    report 3
```

According to Lennart Augustsson, a pionneer and advocate of DSLs, it is critical for embedded DSLs to be hosted in a lazy language. Lazy host languages allow us to express primitives of the DSL, that could not be expressed otherwise in a strict language without using any kind of meta-programming (macros). For example, it is straightforward to define the ubiquitous `ifthenelse` construct in HLogo (Haskell), than writing it in a C host. In HLogo, `ifthenelse` is defined simply as:

```
ifthenelse p t f = case p of
                     True -> t
                     False -> f
```

When it comes to typing, NetLogo is a strong dynamically typed language; HLogo is instead statically typed by default with optional dynamic typing (with the use of Haskell's Data.Dynamic module). Type checking mostly comes down to personal taste and preference, but we could say that in certain cases HLogo is safer than NetLogo:

```
to-report test_typechecking
  let non_number "3"
  report 1 + non_number
end
```

This will throw a type error only at runtime, whereas in HLogo:

```
test_typechecking = do
    let non_number = "3"
    report (1 + non_number)
```

the example will not typecheck and so no executable will be produced from the HLogo's compile phase.

We have managed to port almost 100% of the API of the NetLogo standard library. What remains for HLogo is to provide statistics plotting. The definition of globals, breeds and links, etc is realized with Template Haskell macro code:

```
globals ["global1", "global2"]
turtles_own ["t1", "t2"]
patches_own ["p1", "p2"]
links_own ["l1", "l2"]
breed ["mice", "mouse"]
breeds_own "mice" ["m1", "m2"]
```

NetLogo supports variable number of arguments to be passed to certain builtin primitives, much like what the `&rest` keyword does in Common Lisp. Currently, HLogo does not support such polyvariadic functions, but there are solutions to emulate these in Haskell with Oleg Kiselyov's polyvariadic functions or Liquid Haskell or the way it is done with Haskell's built-in printf procedure.

HLogo has rudimentary support for visualizing the simulation environment with the help of the powerful diagrams package. The simulation user in HLogo has to call the procedure `snapshot` in any places in her code. During execution of the simulation program, files will be created under the current directory with

names `snapshot_tick_number.eps`. By default, the output of visualizations is Encapsulated Postscript, but `diagrams` support other outputting, such as `png`, `svg`, `GTK`. We are working on extending HLogo with a *live* visualization using the GTK toolkit.

What HLogo offers that NetLogo currently lacks of, is the `atomic` primitive (the center theme of HLogo and this chapter). The simulation user wraps the code as in `atomic ( code )` that she considers should be run safely during parallel execution of STM (see Section 2.4). STM in Haskell is modelled as a seperate monad, so as to ensure that transactional code cannot be run outside of transactions. With this restriction, the user cannot run arbitrary IO effects inside transactions, since these effects are not safe and cannot be rollbacked. This restriction is enforced by the Haskell's strong type system. Haskell's STM is arguably the best implementation there is, although it can be in many cases cumbersome to program in, since it is difficult for the user to combine the separate pure world of expressions with the STM monadic world. Its excellent implementation and safety guarantees are why we chose Haskell for the HLogo framework.

4.3 The HLogo Execution Model

Although the API calls of NetLogo and the standard library that comes bundled with the platform have remained intact during porting to the HLogo implementation, the execution model has been severely modified. It is the case that the typical sequential execution model which comes with NetLogo has been altered to accomodate parallel execution.

Let's take a look how HLogo exploits STM and green threads to speed up the NetLogo simulation code. Consider the 3 communication & interaction primitives of NetLogo, discussed earlier. Their general form is:

> **ask** agents code
> code **of** agents
> agents **with** code

We observe a pattern of code segments that have to be run by multiple agents. In the case of the `ask` primitive, an agent instructs (asks) a set of agents to perform some actions (defined in the `code` segment). We could utilize STM for the `ask` primitive by wrapping each code segment in a transaction. Thereby, we can safely infer that there will be no inconsistencies created between the agents. For each agent in the instructed agentset we create a transaction containing the wrapped code. We, then, appoint to each created transaction a separate green thread, so as to benefit from concurrent execution. We could visualize the altered execution of `ask` with Figure 4.1.

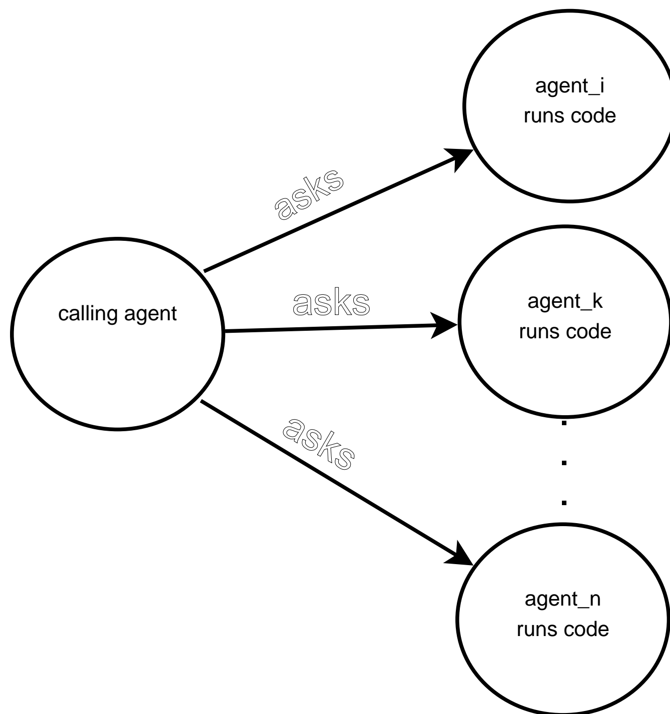

Figure 4.1: Visualized execution of the ask primitive in HLogo. Each circle depicts a separate transaction/thread

We have to note that after spawning a transaction/thread for each agent, the calling agent resorts back to continue executing on its own thread. In other words, the calling agents do not have to wait for the spawned agents to finish execution.

Somewhat different to the `ask` semantics is the execution of the `of` primitive. Here, also the calling agent spawns a transaction wrapped in a unique thread for every agent from the agentset. The added difference is that the calling agent has to wait for the results of the spawned threads; After collecting them, it returns a list of the accumulated results. The execution can be better depicted in Figure 4.2.

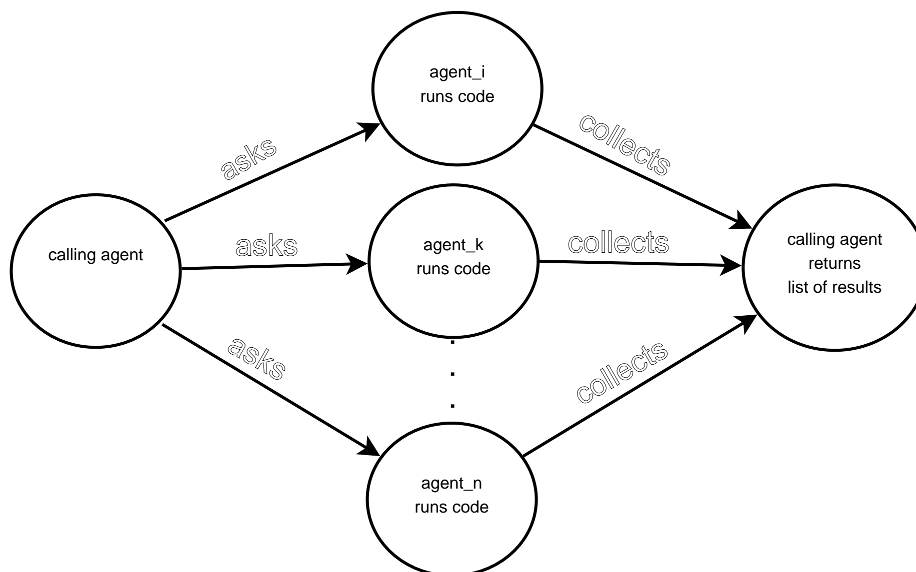

Figure 4.2: Visualized execution of the "of" primitive in HLogo

As we said in Chapter 2 when we introduced NetLogo, the `with` primitive is a special case of the `of` primitive and thus its implementation follows that of the `of` implementation (we spare you from the depiction of the `with` execution). The collected results are filtered by the calling agent and the filtered list is returned to the caller.

Compared to the clone written in Erlang, the agent is not modeled as a separate thread, but simply as a set of Transactional Variables. We use the algebraic data type (ADT) syntax of Haskell to define 3 different records for the 3 different types of agents: turtle, patches and links. A sample of the patch datatype defined in Haskell follows:

```
data Patch a = MkPatch {
      pxcor_ :: Int,              -- on creation
```

```
        pycor_ :: Int,              -- on creation
        pcolor_ :: TVar Double,
        plabel_ :: TVar String,
        plabel_color_ :: TVar Double,
        pvars_ :: Array Int (TVar a),
        pgen :: TVar StdGen
        }
             deriving (Eq)
```

It follow similar for the definition of the `Turtle` and `Link` datatypes.

Each attribute of the agent is a specific record. Attributes that do not change over time, in this case `pxcor` and `pycor`, do not have to be wrapped inside a Transactional Variable, since they are defined on creation time and are not mutated.

Each agent has also an extra attribute (`tvars` for turtle , `pvars` for patch and `lvars` for link) that is an immutable array holding transactional variables that point to the agent-specific attributes declared with `turtles_own`, `patches_own` and `links_own` respectively. For now, we are constrained by the implementation only to allow homogenous arrays, i.e. arrays with each element having the same type. This can be later alleviated in future work with the Haskell's support for heterogenous collections.

The agents have been augmented with a `gen` field, which is a Transactional Variable holding a random generator feed. This enables us to overcome the limitations of the default Random implementation that comes with the Haskell language specification. Specifically, by using a distinct field per agent to hold a generator, we are made sure that there are not going to be race conditions during random generation computations; a statement that cannot be guaranteed with the standard `System.RandomIO` module.

The engine of HLogo creates during initialization three dictionaries holding the agents data structures and manipulates them during execution. The types of the dictionaries are given below:

```
type Patches = Map (Int, Int) Patch

type Turtles = IntMap Turtle

type Links = Map (Int, Int) Link
```

The `Patches` dictionary is a single mapping from (xcor,ycor) keys to `Patch` records. This dictionary as well as the `Patch` records that it holds are created during initialization time and no patches can be destroyed or added during execution, following accordingly the NetLogo specification. The `Turtles` dictionary is a mapping from Int keys (which are essentially the `who` values) to the `Turtle`

records. The `Links` datastructure maps from `(end1, end2)` which are the ends of the link (essentially `who` values of two patches) to the Link records. Links can as well be destroyed and created during execution time.

4.3.1 Issues

There are certain issues that come with the aforementioned execution model. It appears that the HLogo execution model does not satisfy repeatability of simulation runs. The issue conversely boils down to this; consider the simple case in an STM setting, where two threads simultaneously are competing for acquiring one shared resource (reading or writing to it). The example is depicted in Figure 4.3.

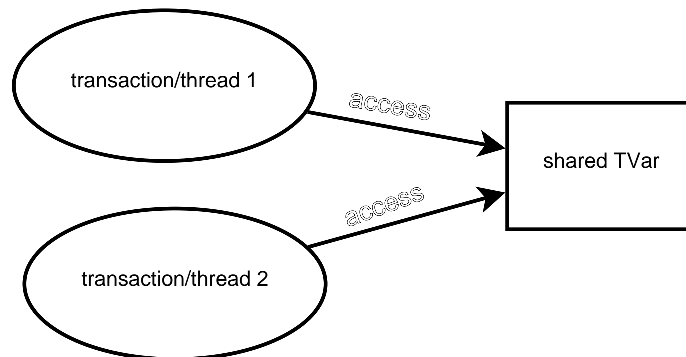

Figure 4.3: A possible occurence in an STM setting. Each circle is a different transaction/thread. The square is the shared transactional variable (TVar) resource.

STM guarantees that the threads will acquire the resource in sequence, though not dictating which exact order of sequence, being it first the thread 1 then the thread 2 or the other way around. In that sense the order of acquiring the resource is non-deterministic; consecutive runs might yield a different order and thus a totally different result. The repeatability of the simulation cannot be, in any sense, guaranteed.

This issue of non-repeatability is related to the fundamental property of Parallel and Distributed Discrete Event Simulation theory (more about this in chapters 5 and 6. This problem is not a fault of the implementation but it is inherent in STM and distributed databases in general; STM violates by definition the local causality constraint. For this sole reason, this Haskell clone may be better regarded appropriately as a multi-agent system (MAS) than a standard simulation framework.

4.3.2 Runtime implementation

The implementation of the described execution was done in the Haskell programming language and ended up to be completely straightforward. Haskell is one of the few, *pure* functional programming languages out there. The reason behind choosing Haskell was not ofcourse its "unpopularity", rather the strong and quality-standard STM library provided by the language. Another reason is the excellent green threads implementation in the Glasgow Haskell Compiler's (GHC) runtime system.

4.4 Benchmark Results

We ran two simple benchmarks to compare the execution speed of NetLogo to that of our own framework, HLogo. In every case, we are varying the number of agents (population) and number of processor cores utilized. The benchmarks were run on a quad-core Intel processor in Windows 7 64bit.

The first benchmark is the "sheep" benchmark. N sheep turtles move around a 100x100 torus eating grass (turning patches from green to brown). Grass regrows at a certain level (from brown to green). The benchmarking stops after 1000 ticks. The results:

Population	netlogo-parsing	netlogo	hlogo1core	hlogo2core	hlogo4core
100 sheeps	17.1	12.4	14.9	9.5	7.2
250 sheeps	18.0	13.4	19.6	12.3	8.6
500 sheeps	19.4	14.7	26.9	15.8	10.7
1000 sheeps	21.6	17	40.3	23.2	14.7
2000 sheeps	24.5	19.9	64.5	35.6	21.7
3000 sheeps	26.3	21.7	87.8	47.9	27.7

Note: the `netlogo-parsing` section shows the whole time of parsing the NetLogo language, compiling it down to JVM bytecode and then executing it, whereas `netlogo` refers only to the execution time (so as to be fair with respect to hlogo execution). For HLogo, On 4 cores, we also enable hyperthreading, essentially having 8 threads. You can visualize the results with Figure 4.4.

The second benchmark we evaluated is called "redblue" benchmark. N turtles are moving forward 1 step on every tick. If they are on a red patch, they also turn left by 30 degrees. If they are on a blue patch, they turn right by 30 degrees. The benchmarking stops after 1000 ticks. The results are:

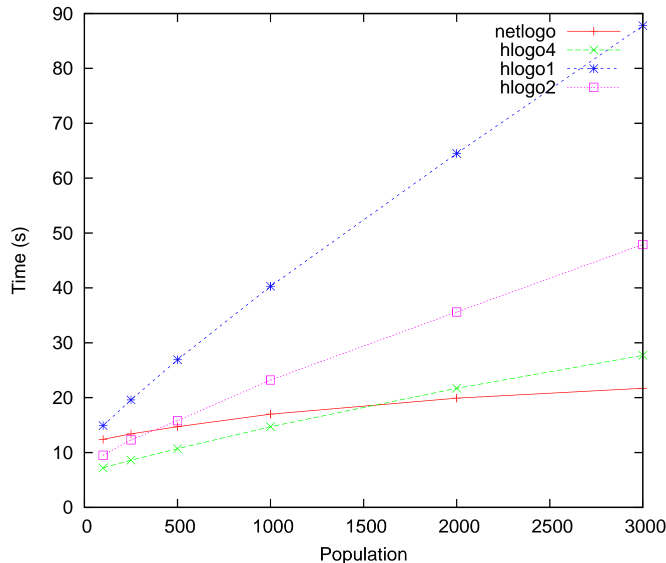

Figure 4.4: Sheep benchmark results

Population	netlogo-parsing	netlogo	hlogo1core	hlogo2core	hlogo4core
1000 turtles	5.3	1.4	2.8	1.8	1.2
2500 turtles	6.8	2.9	9	3.5	2.3
5000 turtles	9.7	5.8	15.9	7.2	4.2
10000 turtles	15.4	11.5	29.1	14.2	7.5
20000 turtles	28.8	24.9	55.2	28	14.5
30000 turtles	43.8	39.9	72.5	42	21.2

Again the results are illustrated in Figure 4.5.

We can clearly witness a speed gain in HLogo when we switch from a single core to more and more cores. This gives us the impression that Software Transactional Memory works and thus will give us performance benefits. But is this enough to “beat” NetLogo code? When the number of agents/turtles remain low, HLogo wins in every case against NetLogo. This is this case for both for the `sheep` and `redblue` benchmark. However, in the `sheep` benchmark, when the population of sheep increases, the performace of HLogo execution degrades dramatically and ends up being slowest even compared to the sequential version of NetLogo.

The explanation for this observation can be easily derived when we take into account how STM works. If there are threads using (competing for) the same resource, then there are going to be eventual incosistencies. These incosistencies lead to retries of the STM transactions until the conflicting agents (threads) are resolved. And that is what happens apparently in the case of the `sheep` benchmark. When there are agents (sheep in our case) in a spatial environment (2D in our case) and compete for the same resources (grass), there are going to be some conflicts. These conflicts lead to transaction retries. In a densely populated environment the transaction retries are going to be high enough to affect the performance of execution.

You can resort to the Appendix to analyze the benchmarking code of NetLogo and HLogo and check the visualization results.

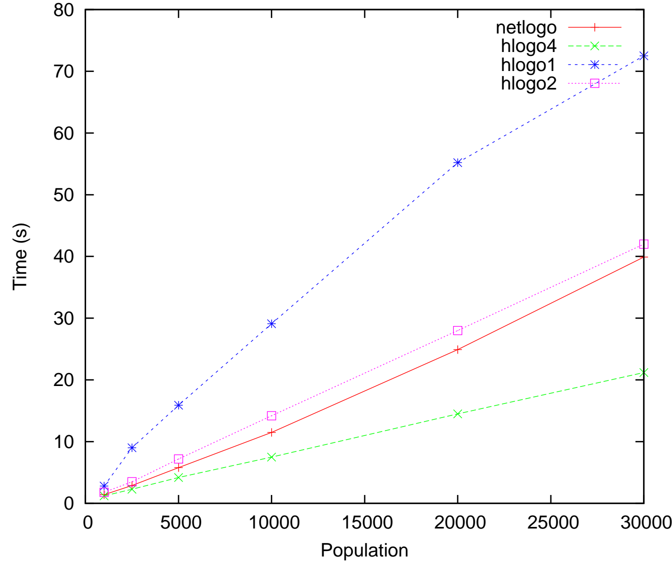

Figure 4.5: Redblue benchmark results

through Distributed Discret-Event Simulation. The key problem as they state is the decomposition of the environment which leads to the problem of fair load balancing of the distributed machines. P. F. Riley and Riley (2003) propose another Distributed Agent Simulation Environment called SPADES. SPADES tries to address the concerns of Artificial Intelligence when desinging agent systems, while having distributed execution and repeatability of results. Massaioli, Castiglione, and Bernaschi (2005) use another parallelization technology, called OpenMP, to speed up the execution of Agent Based Models. However, these technique restricts the implementation of ABM frameworks to be only that which provide an OpenMP implementation, i.e. C, C++, Fortran. Also, it adds the requirement to the simulation user to annotate her simulation code with extra OpenMP pragmas, which is kind of off-putting to say at least. SASSY by M. Hybinette et al. (2006) is a scalable agent based simulation system that sits as a middleware between an agent-based API and a Paralel Discrete Event simulation (PDES) kernel. The difference in SASSY compared to Logan and Theodoropoulos (2001) and P. F. Riley and Riley (2003) is that the ABM framework can be built up from existing standard PDES kernels. D'Souza, Lysenko, and Rahmani (2007) propose an innovative method of executing mega-scale Agent-Based Models in the massively parallel Graphics Processing Unit (GPU). Although, it is well established that this method can lead up to considerable speed gains, we feel that the type of Agent based models that can be run on this platform is restricted. Another similar framework is Flame GPU, built on-top the FLAME ABM framework, and has succesfully been applied on the famous EURACE project to simulate the european economy described in Deissenberg,

van der Hoog, and Dawid (2008).

A totally different approach, but one that we can say “works” every time, is called *parameter sweeping*. It is often the case that the exact same agent based model has to be executed multiple times, with only certain differences to input parameters or random seeds. This execution can be done automatically and in parallel as has been shown by Koehler, Tivnan, and Upton (2005), who implements parameter sweeping for the NetLogo platform.

Concerning ABM and Haskell, we could clearly say that our framework is the first Agent Based Modeling framework being defined in the Haskell programming language. There have been, however, other Haskell simulation packages on Hackage: Aivika is a Haskell library that provides extensive system dynamics and discrete event simulation. Event-monad, as the name suggests, provides an event monad and monad transformer. It can be used as a low-level helper library to build a simulation framework. Users can create an event-graph simulation system and schedule events to it. It is not actively developed. Per se, it does not employ any parallelism, but it could theoretically be used together with a parallel strategy to exploit parallelism. Hasim is a library for process-based Discrete Event Simulation in Haskell. It does not employ any kind of parallelism.

4.6 Appendix

NetLogo code for the sheep benchmark

```
breed [sheep a-sheep]
turtles-own [energy]
patches-own [countdown]

to setup
  reset-timer
  clear-all
  ask patches [ set pcolor green ]
  ;; check GRASS? switch.
  ;; if it is true, then grass grows and the sheep eat it
  ;; if it false, then the sheep don't need to eat
  if grass? [
    ask patches [
      set countdown random grass-regrowth-time
      set pcolor one-of [green brown]
    ]
  ]
  create-sheep initial-number-sheep
  [
    set color white
    set size 1.5  ;; easier to see
    set label-color blue - 2
    set energy random (2 * sheep-gain-from-food)
    setxy random-xcor random-ycor
  ]
  reset-ticks
end

to go
  if ticks > 1000 [print count sheep print timer stop]
  if not any? turtles [ stop ]
  ask sheep [
    move
    if grass? [
      set energy energy - 1
      eat-grass
    ]
  ]
  if grass? [ ask patches [ grow-grass ] ]
  ;set grass count patches with [pcolor = green]
```

```
  tick
end

to move  ;; turtle procedure
  rt random 50
  lt random 50
  fd 1
end

to eat-grass  ;; sheep procedure
  ;; sheep eat grass, turn the patch brown
  if pcolor = green [
    set pcolor brown
    set energy energy + sheep-gain-from-food  ;; sheep gain energy by eating
  ]
end

to grow-grass  ;; patch procedure
  ;; countdown on brown patches: if reach 0, grow some grass
  if pcolor = brown [
    ifelse countdown <= 0
      [ set pcolor green
        set countdown grass-regrowth-time ]
      [ set countdown countdown - 1 ]
  ]
end
```

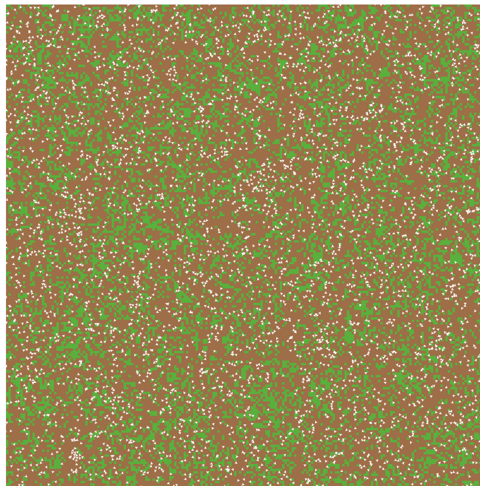

Figure 4.6: Sample NetLogo visualization output for the sheep benchmark

HLogo code for the sheep benchmark

```
import Framework.Logo

globals []
patches_own ["countdown"]
breeds ["sheep", "a_sheep"]
breeds_own "sheep" ["senergy"]

-- Model Parameters
grassp = True
grass_regrowth_time = 30
initial_number_sheep = 100
initial_number_wolves = 0
sheep_gain_from_food = 4
wolf_gain_from_food = 20
sheep_reproduce = 4
wolf_reproduce = 5

setup = do
```

```
  ask (atomic $ set_pcolor green) =<< patches
  when grassp $ ask (atomic $ do
                     r <- random grass_regrowth_time
                     c <- liftM head (one_of [green, brown])
                     set_countdown r
                     set_pcolor c
                   ) =<< patches

  s <- atomic $ create_sheep initial_number_sheep
  ask (atomic $ do
          s <- random (2 * sheep_gain_from_food)
          x <- random_xcor
          y <- random_ycor
          set_color white
          set_size 1.5
          set_label_color (blue -2)
          set_senergy s
          setxy x y
       ) s
  atomic $ reset_ticks


go = forever $ do
  t <- ticks
  when (t > 1000) (unsafe_sheep >>= count >>= unsafe_print_ >> stop)
  ask (do
         move
         e <- senergy
         when grassp $ do
            atomic $ set_senergy (e -1)
            eat_grass
       ) =<< unsafe_sheep
  when grassp (ask grow_grass =<< patches)
  atomic $ tick

move = atomic $ do
  r <- random 50
  l <- random 50
  rt r
  lt l
  fd 1

eat_grass = do
  c <- pcolor
  when (c == green) $ do
              atomic $ set_pcolor brown
              atomic $ with_senergy (+ sheep_gain_from_food)

grow_grass = do
  c <- pcolor
  when (c == brown) $ do
               d <- countdown
               atomic $ if (d <= 0)
                        then set_pcolor green >>
                             set_countdown grass_regrowth_time
                        else set_countdown $ d -1
run ['setup, 'go]
```

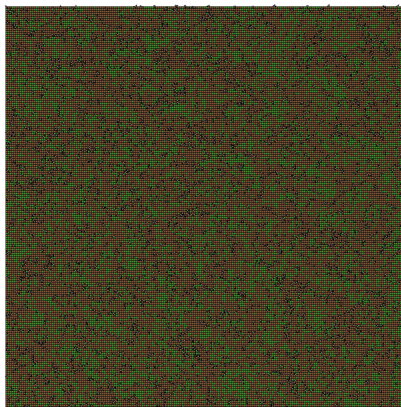

Figure 4.7: Sample HLogo visualization output for the sheep benchmark

NetLogo code for the redblue benchmark

```
to setup
  clear-all
  reset-timer

  ask patches [set pcolor one-of [black black black black
                                  black black black black
                                  red blue]]

  create-turtles 1000
  ask turtles [setxy random-xcor random-ycor]
  reset-ticks
end

to go
  if (ticks = 1000) [print timer stop]
  ask turtles [behave]
  tick
end

to behave
  let p pcolor
  fd 1
  ifelse (p = red)
    [lt 30]
    [if (p = blue) [rt 30]]
end
```

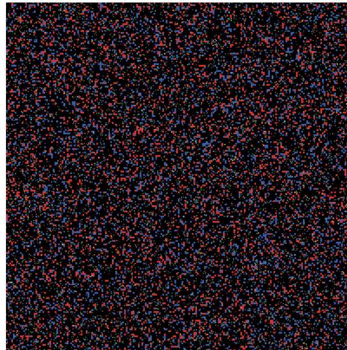

Figure 4.8: Sample NetLogo visualization output for the redblue benchmark

HLogo code for the redblue benchmark

```
import Framework.Logo

globals []
patches_own []
turtles_own []

setup = do
  ask (do
        [c] <- unsafe_one_of [black, black, black, black,
                              black, black, black, black,
                              red, blue]
        atomic $ set_pcolor c) =<< patches
  atomic $ create_turtles 1000
  atomic $ reset_ticks

go = forever $ do
  t <- ticks
  when (t==1000) $ stop
  ask (behave) =<< turtles
  atomic $ tick

behave = do
  c <- pcolor
  atomic $ fd 1 >> if c == red
```

```
                    then lt 30
                    else when (c == blue) (rt 30)

run ['setup, 'go]
```

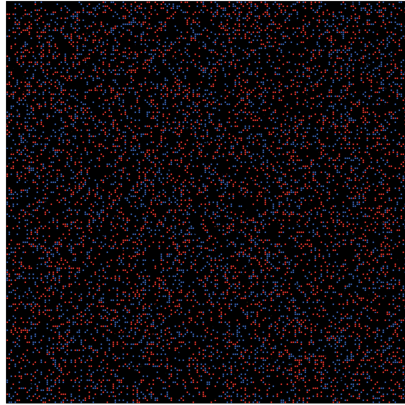

Figure 4.9: Sample HLogo visualization output for the redblue benchmark

Chapter 5

Parallel and Distributed Discrete-Event Simulation (PDES)

5.1 What is PDES

Up until now, when describing Discrete Event Simulation, we tried to avoid mentioning how the simulation software is mapped down to actual execution hardware. Discrete Event Simulation is normally executed on single-processor hardware systems. Since DES models are usually laid down in a completely sequential style; there is no point of employing any parallelism on sequential software. From the other hand, Parallel and Distributed Discrete Event Simulation (PDES for short), as the name suggests, is concerned with how to specify the simulation models in a concurrent fashion so as to exploit parallelism and how to properly execute them on parallel hardware.

To be more precice, Parallel and Distributed Simulation (PDES) refers to the execution of a simulation on a computer system comprised of multiple machines, interconnected through a network. There are certain reasons in that PDES is used:

Faster execution It is the common case that users consider PDES when there are computational limitation on executing the simulation on a single machine.

Distributed memory It may as well be the case that the model which is simulated is so large, it cannot fit in a single memory system. For that reason the model is decomposed into parts and distributed in multiple

machines, each with a single memory module, forming a kind of distributed memory setting.

Geographical distribution Users at different geographical places can participate on a simulation run, and interact with other participants (called human-in-the-loop). An example of this can be military training (wargaming).

Heterogenous simulators are simulation systems that have been built inprivate and later connected to communicate and interact with each other, hence creating a larger simulation environment. PDES can alleviate the connection of these disparate simulation systems.

Fault tolerance If a particular processor fails, another processor of the simulation system can pick up its work, thus offering resistance to failure.

Applications of PDES technology can be divided into two distinct categories: those that are concerned with *analytic simulations* and those dealing with the so-called *virtual environments*. In an analytic simulation the user is interested in measuring the performance of the modelled system by capturing certain statistics about its behaviour. Note that the simulation user in analytic simulations is simply an observer to the model and does not interact at all with it. Applications of an analytic simulation can be the modelling of the aiport traffic while collecting statistics about the airplane delays, a transportation system with measures of its traffic congestion, or a telecommunication network and its percentage of packet loss. Virtual environments are a more recent development of PDES technology and mainly departs from analytic simulation in that the simulation user plays an active role during the simulation execution, often in terms called *human-in-the-loop*. Examples of virtual environments are an airplane simulator for training pilots, or military exercises of a simulated battlefield with many participating human-in-the-loop soldiers. Virtual environments are frequently compared to video games; the differencen lies in that whereas video games main goal is to provide entertainment, virtual environments simulations strive to certainly be pedagocical. For our HDES framework we solely focus on analytic parallel simulations. The reason is that virtual environments, being real-time simulations, are difficult indeed to parallelize/speed up their execution.

PDES is designed to run in many different types of hardware, ranging from shared-memory multiprocessors (under the technology of Symmetric Multi Processing (SMP)) to distributed clusters and grid systems, as well as SIMD machines (Single Instruction - Multiple Data), which modern Graphical Processing Units (GPUs) are designed for. For our own HDES framework, we focus only on shared-memory execution on the common nowadays SMP technology and leave the distributed execution of HDES for future work.

Before we can execute a simulation to parallel-enabled hardware, we first have to transform the model so as to be able to run in this kind of hardware. This is usually achieved by decomposing the simulated model into several submodels, called logical processes (LPs). Each group of LPs is assigned to different processors and the communication between logical processes is realized with simple

message passing. A message is a time-stamped remote (non-local to the LP) event that has a sender LP and a destination LP.

The decomposition procedure can be clarified with an appropriate example. Imagine that we are currently modelling the airport system of Europe. Airplanes can arrive to an airport, land to the runway and later depart for a different destination airport. The model is composed of multiple airports in many different countries. We could assign to each airport a Logical Process (a submodel) and treat the `Landed` and `Departure` events as local to each LP, whereas the `Arrival` event will be a message (remote event) passed from the source airport/LP to the destination airport/LP.

Each Logical Process has a set of state variables associated with it. We should make sure that state variables of any LP are not being shared with any other logical processes. For the airport example, an LP could possibly have 3 state variables, namely, the number of airplanes on the air (of type `Int`), the number of airplanes on the ground (of type also `Int`) and a `Boolean` variable to indicate if the runway is currently free.

A fundamental property in PDES theory, that should hold in every PDES implementation, is the so-called *local causality constraint*. According to R. M. Fujimoto (2001):

> A discrete-event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging time stamped messages obeys the local causality constraint if and only if each LP processes events in a global timestamp order.

Local causality constraint has the effect of:

- Parallel/distributed execution yields the same results as sequential execution
- The results of the simulation are reproducible

A single instance of a violation of the property is called a *causality error*. As a result of violating the local causality constraint property, the PDES implementations have to often face the so-called *synchronization problem*. The synchronization problem has an even greater effect on heterogeneous hardware. For example, consider a simulated model comprised of two LPs, each mapped to a distinct processor. If processor-1 is much faster than processor-2, then the LP1 will advance forward in a faster pace than LP2. This situation will lead into many more violations of the local causality constraint, compared to an identical processor setup.

There have been developed two distinct categories of distributed algorithms to tackle the synchronization problem. The conservative category of algorithms is the topic of the next section. With the conservative approach the causality

errors caused by out-of-ordered processed events are avoided at all costs. In optimistic algorithms however, described later in Section 5.4, causality errors are allowed to happen but later the simulation engine has to go back and correct them.

5.2 Conservative approach

The conservative algorithms historically predate those of the optimistic approach. Also, the conservative aproach is generally regarded as slower compared to its optimistic rival. Still, though from a theoretical perspective it can be of great benefit to discuss how the conservative algorithms try to solve the synchronization problem. Here we are going to describe the first conservative algorithm proposed, appeared in Misra (1986).

We first have to make some assumptions for the execution of the simulation engine. According to R. M. Fujimoto (2001) the simulation engine should satisfy the following three constraints:

1) an LP will not send remote messages in decreasing timestamp order
2) the network infrastructure guarantees that the messages are delivered in the order they were sent
3) the communications are reliable; i.e. the remote messages will eventually be delivered

Each Logical Process holds a FIFO queue for every link to other Logical Processes. There, the incoming messages are dispatched and stored to the correct FIFO queue. Ofcourse, the LP has to define also a single FIFO queue to store any locally produced events.

The naive central approach for running the simulation engine is written below in pseudocode:

```
while (simulation is not over) {
    wait for each FIFO to contain at least one message
    remove message M with the smallest timestamp
    clock := time of M
    handle M
}
```

However, this approach will certainly lead to deadlocks where the Logical Processes will mutually wait for incoming messages, while forbidding them to send any outgoing messages. There is a way to avoid this deadlocks at all by devising a different algorithm, called the Chandy/Misra/Bryant Null-Message protocol algorithm described in Misra (1986).

On every loop of the simulation engine, the Logical Process sends “fake” messages, called *null messages* that are used only for synchronization purposes and not take part in simulation; hence the name “null”. A null message acts as a guarantee from the sender LP to the receiver LP; the sender LP promises to the receiver LP that it will not send any new messages to the receiver for a certain amount of time in the future. This amount of time is called for obvious reasons the *lookahead* value. The lookahead value is usually static and is pre-declared by the user for each LP link/connection. In the airport example the user may have a certain setup between 3 LP-airports as:

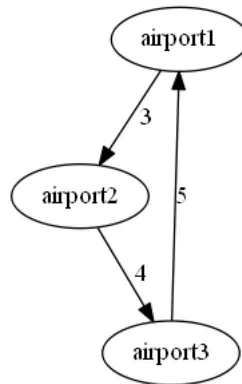

Figure 5.1: Example with 3 airports and their lookahead values

The nodes of the graph are the LP-airports and the contain the lookahead values. For this particular airport example it may be better to consider the lookahead value as the distance between two airports.

What is left is to modify the main loop of the simulation engine, so as in the end of the loop to send a null message to each neighbouring LP with a timestamp (`now + lookahead_ij`), where i is the current sender LP and j is the remote receiver LP):

```
while (simulation is not over) {
    wait for each FIFO to contain at least one message
    remove message M with the smallest timestamp
    clock := time of M
    handle M
    send to each neighbour j:
         null message with timestamp (now+lookahead_ij)
}
```

5.2.1 Example in Erlang

Each *logical process* will itself be an instance of a *simulation engine*. The simulation engine's role is to receive events in discrete time, process them in timestamp order, and schedule new events in the future to itself or other logical processes (for example by sending an event message to a remote simulation engine). The simulation engine stops when there are no more events to process, or a certain condition is met.

Why did we choose Erlang to implement such a simulation? The difficult part of implementing a simulation engine is not how to process the incoming events (they are just linked to arbitrary code which gets executed by the engine), but more how to easily communicate between logical processes (sending and receiving events). This can be easily done in Erlang, since *message-passing* is a first-class citizen of the language. Events are simply modelled as Erlang messages, and logical processes, likewise, are implemented as Erlang processes. Another reason is that, in Erlang, running on many processors (SMP) or on multiple distributed machines is transparent, that is we do not have to write extra code to handle these distinct cases.

In our implementation, we will use a conservative non-zero-lookahead mechanism, influenced by the Chandy/Misra/Bryant null message protocol algorithm described by Misra (1986).

Conservative mechanisms are easier to implement by the simulation developer, but require extra (lookahead) information from the end-user; optimistic mechanisms on the other hand don't require such information by the end-user, but are much more difficult to implement. We can say that, in most cases, an optimistic approach is faster in execution time than a conservative simulation. In this section, however, we only demonstrate a conservative mechanism written in Erlang.

In action, the simulation program will be comprised of two entities: the simulation application and the simulation engine. The simulation application has the model specification in it, not mathematically defined, but rather through a computer language. The simulation engine is also written in a programming language (since it has to be executed); it takes the the simulation application as input and runs it accordingly. The two entities can, but don't have to, be hosted on the same programming language. In this case, we are choosing Erlang for both the simulation application and the simulation engine.

The definition of the simulation application is writeen in an Erlang module, that must follow the OTP principles, thus defined as a `sim_proc` behaviour. What the application is responsible for is the definition of state (the state record), simulation initialization (the `init` function), a series of callbacks (`handle_event` function) and simulation termination cleanup (the `terminate` function). The event callbacks are simply associating a possible incoming event to specific code

that should be executed. In this example, the simulation application is modelling an airport, which schedules arrival, landed and departure events.

Here, we are going to show you how a simulation application (aka simulation model) is written and structured using this proposed experimental Erlang framework. Worth noting is the difference between the simulation application and the rest part of the framework, the simulation engine. We could say that the simulation engine is the heart of the simulation, where events are scheduled and executed in order. The simulation application is merely user code, linked to events, which is passed to the simulation engine for execution. This code, as you can guess, must be specified in the Erlang language.

The model is comprised of multiple Erlang modules, one for each Logical Process definition. This is in contrast to what is the common case in a sequential simulation, where the code of the model is accumulated in a single file location. The reason for choosing a multi-module approach is that the application stays as close as possible to the Erlang style guideline. In our framework, each Erlang application module corresponds to an individual Logical Process (LP). When each LP is instantiated (spawned in Erlang terms), it will be assigned its own simulation engine and event code to be executed. That means that every LP will have a distinct time clock and event queue.

5.2.2 An example of a simulation application module

For our example, we our modelling a network of airports, following the same example as in the book of R. M. Fujimoto (2001). Each LP will be an individual airport, each with a single runway, where aircraft land to and depart from.

We first start the module file as usual, by defining the following directives:

```
-module(abd).
-behaviour(sim_proc).
-compile(export_all).
```

The above mean that we specify a Logical Process (that is, airport in our case), named `abd`, and we want its whole API to be exported (`-compile(export_all)` directive).

Next, we declare our constants for our program simulation. We have R, which determines the time the runway is in use for an airplane to land. The constant G specifies the time the aircraft after landing, stays in the ground and travels to the gate for its next departure.

```
%% constants
-define(R, 10). % time runway in use to land aircraft
-define(G, 5). % time required at gate
```

After that, we define a record that holds the state of the LP. The LP state is also influenced by the performance measures we set up in our simulation study. In this case, we have to declare 3 distinct state variables, the 1st to count the planes that are in the air, the 2nd for the number of planes that have landed and finally a boolean value to state if the runway is currently free:

```
%% state variables
-record(state, {in_the_air,
                on_the_ground,
                runway_free}).
```

What follows, is the `init` function that will initialize our LP state, schedule initial events to the simulation engine and ultimately declare what are the incoming and outgoing connections (links) to other airports. Suppose that, for the outgoing links, we have to provide a non-zero lookahead value.

```
init(_Args) ->
    %% initialize state_variables
    State = #state{in_the_air = 0,
                   on_the_ground = 0,
                   runway_free = true},

    %% schedule initial event
    sim_proc:schedule(arrival, 30),
    sim_proc:schedule(arrival, 10),

    %% incoming links
    sim_proc:link_from(ord),

    %% outgoing links with lookahead
    sim_proc:link_to(lax, 3),

    {ok, State, 40}.
```

The code is self-explanatory. We initially have 0 planes in the air, 0 on the ground and the runway is not currently in use. Next, we schedule two arrival events on time 30 and 10 respectively. We have an incoming link with the ord airport; that means that we are expecting incoming flights that start from ord and reach our destination. There is also an outgoing link to the lax airport, with the lookahead value being 3 units of time. This means that departures scheduled from the abd airport, to arrive to the lax airport will take at least 3 units of time to reach the lax airport. So, in this case, the lookahead can be determined by the distance between two airports.

The final line in the `init` function, `{ok, State, 40}`, simply returns the state and specifies an endpoint for the simulation of the abd airport; so at time 40 we

are stopping the simulation of the abd LP and we are not taking into account any events after the end time.

We continue to define the event handling functions. We make heavy use of the pattern-matching capabilities of the Erlang language, to pattern-match on the distinctive events of the LP. First, we handle the arrival events to our abd airport:

```
handle_event(arrival, State) ->
    sim_proc:println("Arrived"),
    In_the_air_ = State#state.in_the_air + 1,

    Runway_free_ = case State#state.runway_free of
                       true -> sim_proc:schedule(landed, ?R),
                               false;
                       false -> false
                   end,
    {ok, State#state{in_the_air = In_the_air_,
                     runway_free = Runway_free_}};
```

We print to output and increment the `In_the_air` variable by 1. We then check if the runway is free. If this is the case, then we schedule a future land event after `R` units of time. We then continue, and return the new state.

The code for the landed event is similar:

```
handle_event(landed, State) ->
    sim_proc:println("Landed"),
    In_the_air_ = State#state.in_the_air - 1,
    On_the_ground_ = State#state.on_the_ground + 1,
    sim_proc:schedule(departure, ?G),
    Runway_free_ = case In_the_air_ > 0 of
                       true -> sim_proc:schedule(landed, ?R),
                               false;
                       false -> true
                   end,
    {ok, State#state{in_the_air = In_the_air_,
                     on_the_ground = On_the_ground_,
                     runway_free = Runway_free_}};
```

We write to output and change the state variables accordingly. We don't forget to schedule a departure events after `G` units of time. If there are still aircraft circling over the airport waiting to land, we pick the next in line and schedule it for landing. We then return the updated state.

Then we handle the departure events:

```
handle_event(departure, State) ->
    sim_proc:println("Departed"),
    On_the_ground_ = State#state.on_the_ground - 1,
    sim_proc:schedule(lax, arrival, 5),

    {ok, State#state{on_the_ground = On_the_ground_}};
```

We decrement the state variable `On_the_ground` by 1 and schedule an arrival at the remote lax airport in 5 units of time — which is fine, since it is being larger than the 3 lookahead value specified in `init`.

We then consider stop events, possibly generated by the local or a remote LP. We could ignore a stop event, or do what is advisable and stop the simulation of the LP by returning `{stop, State}`, instead of an `{ok, State}`.

```
handle_event(stop, State) ->
    {stop, State}.
```

When the LP stops, either running out events (normal termination) or running out out simulation time (timeout termination), the terminate callback function is called to handle last code before the exit of process (similar to a destructor in the OO-world). Here we simply announce the termination in the standard output.

```
terminate(normal, _State) ->
    sim_proc:println("Finished simulation");

terminate(timeout, _State) ->
    sim_proc:println("Timeout reached").
```

Following closely the OTP guidelines, We can still consider messages sent to the Logical Process that are not events. We can handle these `out-of-order` messages with the `handle_info` callback function, as:

```
handle_info(_Info, State) ->
    {noreply,State}.
```

In this example, we simply ignore these kind of messages, but in a different situation we might consider handling them, thus creating a reactive/interactive Erlang process.

What is left is to show the *hot-code loading* capabilities of the Erlang VM. In practice, hot-code loading means that we could change the simulation code dynamically during the run-time of the simulation framework. In our case, we simply don'tt react in possible code changes. The benefits of hot-code loading for simulation purposes are not clear and, so, are left to be explored in the future.

```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Ofcourse, we have to write appropriate `sim_proc` behaviour processes also for the rest two airports that we model. But we can skip this for now, since it follows the same principles.

5.3 Optimistic approach

Comparing to the conservative approach that avoids causality errors, the optimistic algorithms allow any causality violations but provide a way to later recover from them. In the end, a correct timestamp order is still imposed. The theory behind optimistic algorithms originate from distributed database systems and microprocessor technology. The first to appear and most widely applied optimistic algorithm is the so-called Jefferson's Time Warp mechanism (Jefferson et al. 1987). In this section we are going into detail of how this algorithm is defined and executed.

First we have again to establish some assumptions about our simulation execution environment. As in the conservative approach we have to make sure that no state variables are shared between the Logical Processes. The communication infrastructure used is assumed to be reliable; messages will eventually arrive at their destination. A constraint that optimistic algorithms lift compared to the conservative approach is that now, the Logical Processes do not require to send messages in timestamp order and the network medium does not have to guarantee that the messages arrive in the order they were sent. The LPs can be dynamically allocated and destroyed and the simulation user does not have to specify lookahead values for each LP link/connection. This tends to be a serious burden for the simulation user in the conservative approach.

In the TimeWarp mechanism there are two distinct submechanisms: the *local control* and the *global control* mechanism. As their name suggest the local control mechanism is executed locally inside each LP, whereas the global control mechanism require participation of all LPs and an extra processing unit, called the *controller*.

Let's first take a look at the *local control* mechanism. Instead of using separate FIFO queues for each connection, in the optimistic approach the engine holds only a single FIFO queue for incoming as well as local events, called the *event list*. The events have to be sorted inside the list; usually according to ascending timestamp order. A *needle* is used to point to the index of the last processed event inside the list. The events are not discarded after being processed; only the needle moves forward.

A *straggler message* is a remote message arriving out of order. That means that the simulation engine has **optimistically** moved more forward than it

should have to. The needle points to a later event than the straggle message's timestamp. In that case, we say that the LP has to *rollback*. You can take a graphical look at a possible event list in the Figure 5.2.

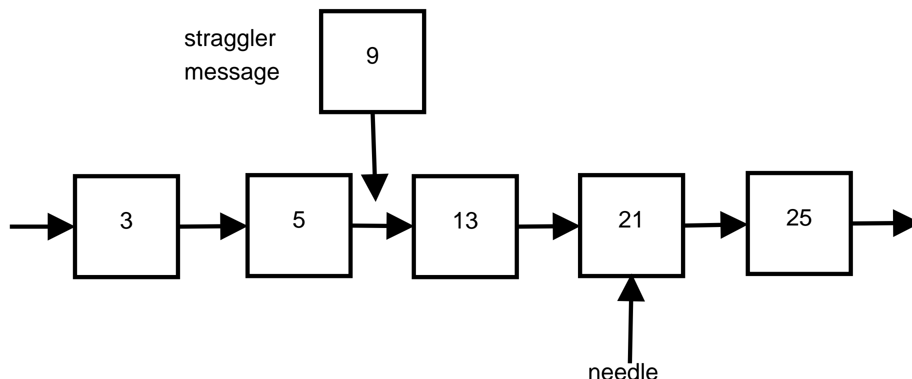

Figure 5.2: An example of the event list in TimeWarp algorithm

In our example the *25* and *21* events have to be rollbacked. During a rollback operation, the state prior the execution of a rollbacked event is recovered. This recovery is achieved with *intermediate state saving*. On each cell that the event occupies in the event list we store a copy of the state variables prior to executing the event. In our example, if we consider 3 state variables and a simple copy-state saving approach the event list could have been represented as in Figure 5.3.

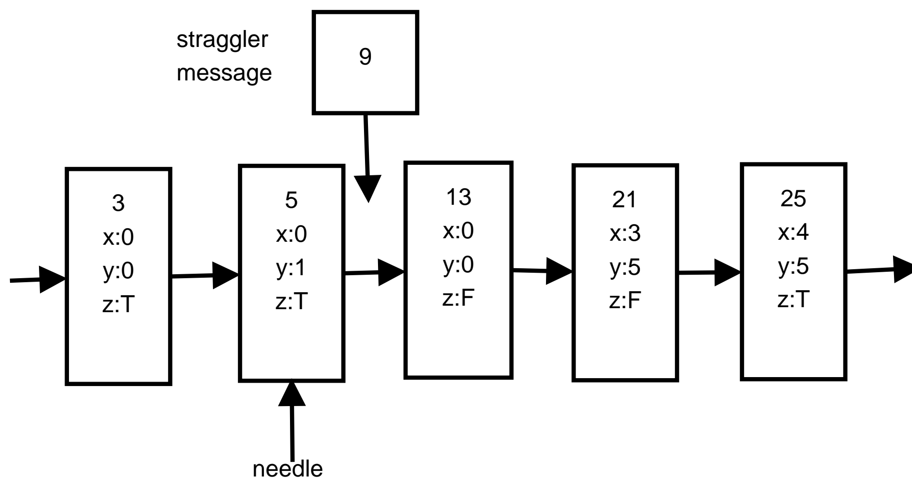

Figure 5.3: An example of intermediate state saving

During rollbacking any remote messages that have been spawned by rollbacked

events have to be cancelled. This is achieved by sending a so-called *anti-message* for each cancelled message. Anti-messages have the same contents as the original messages with the only difference in an enabled flag. When an LP receives an anti-message, that means that the original message that the LP received has to be cancelled. If the LP has already processed that message then it has to also rollback prior to the received message. To distinguish a rollback caused by an out-of-order message (*primary rollback*) we categorize a rollback caused by an anti-message as *secondary rollback*. In the case that the message was not processed by the LP yet, the antimessage will annihilate the original message and both will be removed from the event list.

In the local control mechanism, which we just described, the event list size will be ever-increasing. Also, any IO that was fired during event processing has been delayed and has to be somewhere in the future enforced. The *global control* mechanism is introduced to tackle exactly these two cases. A helper processing unit (usually a distinct process or thread in the system) is spawned and manages the mechanism. This unit is called the *controller*. The *controller* periodically (the time period is defined by the user) issues a *Global Virtual Time* (or GVT) computation. GVT, in simple words, is the globally smallest needle throught all the LPs of the simulation system. The intuition is that all events that have a timestamp smaller than the GVT can be safely committed (their IO can be enforced). After an event is committed, its memory in the event list can be reclaimed. This process is called *fossil collection* and is similar in mind with what the *garbage collector* achieves in a high-level programming language.

The simple case of the global control mechanism in Time Warp dictates that each Logical Process in the system stops-the-world and be forced to take part in the GVT computation. This approach is called *synchronous GVT*, since the LPs have to wait for the global control mechanism to finish so they can resume their own local control mechanism. There have also been proposed *asynchronous* algorithms, where the LP that currently participates in the GVT computation can still process its event list and advance forward in time. Theoretically, the asynchronous algorithms can achieve faster execution results compared to synchronous algorithms.

Chapter 6

HDES: A lazy PDES framework

Abstract

Parallel and Distributed Simulation (PDES) is the focus of the Discrete-Event Simulation theory on speeding up the performance of simulations by employing parallelism and/or distributed computation. We argue that the current PDES technology suffers from the problem of unnecessary computation; the evaluation is not driven by the performance measures of the simulation user and instead are inadvertently strictly evaluated. Instead we propose yet another PDES framework, called HDES, that harnesses the lazy evaluation of its host language, i.e. Haskell, to offer an even greater performance execution of parallel simulations. The HDES embedded domain specific language (EDSL) has the added benefit of being more expressive than conventional PDES languages, thanks to the suggested lazy semantics.

Keywords: Parallel Discrete Event Simulation, PDES, Parallel and Distributed Simulation, PADS, Time Warp, Haskell

6.1 Introduction

The recent focus of Discrete Event Simulation theory has been the execution of simulation models in absolute parallel and distributed environment setups. Models are currently designed to imitate systems even larger and more complex; this leads to a need for more computational power by the simulation hardware.

Pallel and Distributed Simulation technology has been laid down to mitigate this need.

The Moore's law as we know it has been surprisingly accurate in its predictions during the history of computing. However, there has been a recent doubt of whether computing performance can keep up with Moore's transistor count growth. Contemporary processors are utilizing more and more processing cores and the network connectivity has become so fast and cheap that it is now relatively easy to interconnect physically distant computer systems. This simulation hardware have to be accompanied by relevant simulation software. Here is where Parallel and Distributed Discrete Event Simulation (PDES) comes to play.

Each model that we simulate must have an associated list of performance measures that the simulation user is particularly interested in. After the user has constructed the model in some domain-specific programming language or after some consecutive execution runs of the same constructed model, she realizes that she is only after a part of the whole list of performance measures. The user has then to "dig" into the code again, removing any references to the unnecessary code. Otherwise the user will most likely experience a delay in retrieving the performance results, due to the needless computations in the simulated model. Based on the PDES frameworks that are currently on the "market", even if we ask for a part of the performance measure list, the simulation frameworks will still have to compute its entirety. This leads to a performance degradation, that we in other case would not experience if we have excluded the nonessential perfomance measures.

Another problematic scenario happens in the case where we try to glue multiple models together to create a multimodel (or supermodel). This can be better understood with an example. Consider two already-made distinct models, an airport and a train station connected together. When the airport model was created, there was a separate model of buses that was connected to the airport. Because our current setup does not include a bus terminal connection to the airport, the unnecessary scheduling of buses in the airport model will not have to be computed and updated.

There is a way to address the problem without having the simulation user to resort to any code changes. By using a *lazy evaluation* strategy, the simulation program will not compute any values that are not asked for displaying by the simulation user. Currently, the only industrial-strength programming language that provides lazy semantics is the Haskell language, which can be categorized as a purely-functional statically-typed language. We contribute a brand-new PDES framework, written in Haskell, named HDES that

- has lazy evaluation. Besides the benefit of speeding up the simulation by not computing unnecessary performance measures, a lazy evaluation strategy enables the simulation user to express a larger variety of computer simulation programs

- employs an *optimistic* PDES algorithm to speedup the execution of simulations
- utilizes multiple processor cores (Symmetric Multi Processing)
- offers a simple domain specific language embedded in the Haskell programming language with an exremely intuitive interface

6.2 The HDES language

The HDES framework defines a simulation language to interface with the PDES execution kernel, described later. This domain specific language (DSL) is embedded in Haskell; that means that certain characteristics of the HDES language are borrowed/inferred from the host language, i.e. Haskell. Embedded DSLs come with certain advantages and shortcomings. We, the implementors of the language, don't have to write new compilers and tools around the language; instead we use the ones already supplied from the Haskell ecosystem. We also inherit the good attributes of a functional language such as Haskell, that is, its great abstraction mechanism and its strong static type system. However, an embedded DSL is also constrained by the syntax of its host language.

We could lay down our model specification in a modular fashion using multiple modules, but for sake of clarity in this example we put the whole specification in one sole module. Let's open a file with the name `Main.hs` and start writing our model specification. We first have to import the framework's API and kernel:

```haskell
import Framework.DES
```

We then have to specify how many Logical Processes (LP) we have in our model and what are their names. Note that each LP name has to be unique per model specification. In our specific example, we try to model an airport network, with 3 distinct airports identified by their names `ABD`, `LAX` and `ORD`.

```haskell
lps ["abd", "lax", "ord"]
```

The Logical Processes will have to handle local and remote events between each other. For our case, we specify an `Event` datatype that holds the possible values that an event can take, i.e. an `Arrival` of an airplane, a `Landed` event and a `Departure` event to a remote destination airport.

```haskell
data Event = Landed | Arrival | Departure
           deriving (Eq, Ord)
```

The datatype is self-explanatory. We used the Algebraic Data Type (ADT) syntax to provide an enumeration of all possible values. We also derive for our

`Event` datatype an `Eq` instance, so that we can compare for equality and an `Ord` so as to give priorities to simultaneous events. Simultaneous events are events that happen in the exact same time. Here, we give priority to airplanes that are about to land than those that are just arriving to the airport.

Our framework is polymorphic to the kind of Event datatype. The simulation user is free to choose any datatype that has an Eq and Ord instance. The user could as well go with the more common `type Event = String`, and use simple strings for event names — that is the approach that most C-based PDES frameworks take. However, this approach is arguably slower because during event handling, the pattern-matching on a string datatype will be slower than a complete enumeration of events, as it is done in our case.

What follows is the `Time` datatype. For sake of simplicity, we use an integer for our time.

```
type Time = Int
```

We can witness that HDES is also polymorphic and not-restrictive on the Time datatype, compared to the rest of the HDES frameworks. Indeed, time in HDES can be any datatype that can be ordered (provided an instance to the `Ord` typeclass). We could have as well used `DateTime` for our datatype.

The last datatype we have to define is the `State` datatype. `State` is a Haskell record with each field being a state variable of the Logical Process. Here the `State` has the same structure for all 3 airports, but no state variables are shared between the Logical Processes/airports; this a fundamental property of Parallel and Distributed Discrete Event Simulation (PDES) theory. In our example we have 3 state variables: `in_the_air` holds the number of airplanes that have arrived and are currently circulating around the airport, `on_the_ground` holds the number of airplanes that have already landed and has type integer, and `runway_free` indicates if the runway of the airport is currently free for landing.

```
data State = State
           { _in_the_air :: Double,
             _on_the_ground :: Int,
             _runway_free :: Bool,
           } deriving (Show)
```

Note that each Logical Process can have a different State structure than other LPs. We could as well have defined three datatypes `State1`, `State2` and `State3` for our airport models.

In HDES, we also make use of the excellent fclabels package which allows us to define lenses (setters and getters in the OO terminology) for our data types and overcome the weakness of Haskell's built-in record support. We just have to introduce a new macro expression in our code:

```
mkLabels [''State]
```

Now there are three extra functions available to manipulate the state:

```
get :: (state :-> a) -> Sim time event state a
set :: (state :-> a) -> a -> Sim time event state ()
modify :: (state :-> a) -> (a -> a) -> Sim time event state ()
```

Each function takes as the first argument the lens of the datatype. We use these 3 functions extensively in our code to easily manipulate our state variables.

Any constants of the model can be simply supplied as top-level definitions in Haskell. In our case `r` is the time the runway is in use to lande the aircraft and `g` is the time taken by the airplane to go to its gate.

```
r = 10 -- time runway in use to land aircraft
g = 5 -- time required at gate
```

Then we define the statistics structure that should be collected/computed from the state variables during the simulation run. These performance measures, for now, can be summation of state variables (summ) , their accumulation (accu) or aggregation (aggr). In our case, it is:

```
stats = (|
          summ _in_the_air ,
          accu _in_the_air ,
          summ _on_the_ground ,
          accu _on_the_ground
        |)
```

We continue by defining the model logic for each LP/airport. An LP is defined as a monadic action in the `Sim` monad with a type signature `def_lp :: Sim time event state a`. A monad is a fundamental composable construct in Haskell that sequences effects. Our Sim monad is polymorphic, as we said before, on the time, event and state datatypes, as well as the returning result `a` of the monad. In the first part of the definition of the LP, we initialize the state and create initial events. We then define an inner `handle` function that should pattern-match on each event declared in the provided `Event` datatype. Last, the `def_lp` has to return this `handle` function, so the HDES kernel can pick it up for dispatching the events.

```
def_lp = do
   -- initialize state
   -- spawn initial events
```

```
  -- define the handler function with type handler :: Event -> Sim ()
  let handle Event1 = do
                   ...
  let handle EventN = do
                   ...

  return handle  -- return the constructed handler
```

Inside the handle function the user may want to spawn **local** events using one of these two functions:

> **schedule** *event time*
> **scheduleNow** *event time*

where the Now version treats the time argument as time+Now. To spawn **remote** events to other Logical Process the user can use the following similar functions:

> **scheduleR** *lp event time*
> **scheduleRNow** *lp event time*

For example, here is the definition of the model logic for the ABD airport:

```
def_abd = do
  -- Initialize state
  put (State 0 0 True)

  -- Schedule initial events
  schedule Arrival 1

  -- Create the event handler
  let handle Arrival = do
            dump "abd: Arrived"
            modify in_the_air (1+)
            rf <- get runway_free
            when rf $ do
              scheduleNow Landed r
              set runway_free False

      handle Landed = do
            dump "abd: Landed"
            modify in_the_air (subtract 1)
            modify on_the_ground (1+)
            scheduleNow Departure g
```

```
          ita <- get in_the_air
          rf <- if ita > 0
               then do
                 scheduleNow Landed r
                 return False
               else return True
          set runway_free rf

    handle Departure = do
          dump "abd: Departed"
          modify on_the_ground (subtract 1)
          dump "abd: Should schedule arrival at lax on 5"
          n <- now
          when (n < 1000) $
               scheduleRNow lax Arrival 5

  return handle
```

We do similar for the definition of the `LAX` and `ORD` airports.

We finish the module by providing a main function, that will start the framework's core, spawn a green thread for each LP and execute the simulation while collecting statistics:

```
main = run [("abd", abd, def_abd `collect` stats)
           ,("lax", lax, def_lax `collect` stats)
           ,("ord", ord, def_ord `collect` stats)
           ] -- Order does not matter
```

6.3 Implementation

In the heart of the PDES execution engine lies an algorithm with optimistic synchronization. The algorithm is based on the original Time Warp algorithm proposed in (Jefferson and Sowizral 1985), albeit slightly modified to accomodate our functional programming needs. Specifically, wherever the original algorithm uses side-effects and mutable arrays, we have to replace them with immutable data structures, since Haskel is a *pure* functional language by default. For the implementation we tried to stay as simple as possible, since it is relatively hard to provide a bug-free optimistic algorithm; usually such an algorithm requires extensive testing and debugging. Haskell helped here, because you can better reason about your algorithm, when your program is referentially transparent, i.e. side-effect free. On the other hand, however, it generally tends to be hard to express an imperative algorithm in a pure functional language setting. The purity that Haskell offers means that the datastructures we can use, have to be

immutable. For representing the event list datastructure we used the immutable Data.Sequence. Sequence container types are implemented internally with 2-3 finger trees that enable the addition to both ends of the list in constant time $O(1)$. If we would have gone another way with Haskell mutable IO arrays, (as described here), we would certainly gain a lot in speed and memory, however, we would risk the clarity of our implementation.

The HDES framework can be used only with shared-memory systems. There is unfortunately no way currently to have a distributed setting, like other PDES frameworks do by exploiting the MPI architecture. To benefit from parallel execution, the simulation user has to run HDES in a Symmetric Multi Processing (SMP) -enabled machine. There is no restriction on the number of core processors that can be utilized by HDES; however for the best performance possible, it is advised to match the number of CPU cores with the number of Logical Processes of the model.

For each Logical Process (LP) in the simulated model, a distinct lightweight Haskell thread will be created. The communication of the LP threads is realized with *MutableVariables* (MVars). MVars are being used for the communication betwen LPs and the Global Virtual Time (GVT) controller. MVar channels in particular are used extensively in the HDES framework to provide the necessary communication between Logical Processes (LPs), thus forming a kind of low-level message passing, albeit implemented in shared-memory.

We could use these channels as our Logical Processes mailboxes. For each LP we create a unique channel where other LPs can produce events (send remote messages) and the LP itself can consume/process them.

Since we are restricted by the memory of the computer system, we have to periodically run *fossil collection* to claim any unnecessary memory. The fossil collection runs on each LP after every GVT computation completes. The benefit of using a high-level language as Haskell, is that we don't have to deal manually with managing memory; instead we rely on the automatic garbage collector (GC) of the language to do it for us.

6.4 Benchmarking

6.4.1 Testing Laziness

There are 3 airports, namely `ABD`, `LAX`, and `ORD`. The framework kernel as described before, will launch 3 threads in total, one for each airport (Logical Process). We initially schedule one airplane (job) that starts from ABD and travels to LAX and then to ORD and then back to ABD, essentially forming a circle. During the whole simulation execution, there is only 1 job scheduled, so normally we will not expect to witness any parallel benefits, because one thread

exclusively is executing at any time; the rest two threads are stalled. Also no time warp rollback is expected to trigger.

To test if laziness actually works, we introduce a bogus computation on every iteration of the Logical Process and we store it in a state variabe. The evaluation of each Logical Process, as discussed before, is *driven* only from its performance measures. So we expect the extra bogus state variable will stay unevaluated and its memory allocation will be discarded on the periodic fossil collection. The computation that was introduced is evaluating the 32th number for the fibonacci sequence `fib 32`. Such a call takes around 0.8s to finish in our test computer system on compiled code. The fibonacci function is naively defined in Haskell as:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

We annotate the previous `State` datastructure with two fib-related variables, to update and hold the result of the `fib 32` computation.

```
data State = State
    { ...
      _arg_fib :: Int,
      _res_fib :: Int
}
```

The performance measures (statistics) specification remains the same:

```
stats = (|
          summ _in_the_air ,
          accu _in_the_air ,
          summ _on_the_ground ,
          accu _on_the_ground
        |)
```

You can check that the `arg_fib` and `res_fib` state variables are excluded from the statistics gathering. If our hypothesis is correct, then these state variables will not be computed at all, because of laziness employed, resulting in speedup. The intuition is that the user is only interested in some state variables at any time, and he/she declares that in the *stats* data structure. The state variables that are not required, will not be computed.

Since Haskell is lazy by default, the comparison must be done against a strict version of the code. It is the case that the Haskell user can introduce strictness points to her code, declaring which part of the code must be evaluated eagerly. In our case, we copy our construct model module file to a different place and make the following change to the state structure:

```haskell
data State = State
{   ...
    _arg_fib :: !Int,
    _res_fib :: !Int
} deriving (Show)
```

The `fib` state variables are annotated with the `!` symbol. This means that the Runtime System will evaluate them to Weak Head Normal Form (WHNF). Since the fib state variables both point to a primitive type (Integer in this case), they will be in reality evaluated to Normal Form (NF), because of the usual compiler optimizations of unboxing primitive types.

For both runs of the lazy and strict programs, we have to pass to them a GVT interval parameter (in microseconds). Every defined interval, the framework controller will compute the Global Virtual Time (GVT) of the simulation, and announce it to the LPs. Then on, each LP will determine, based on GVT, which processed event is safe, and then commit its associated IO. Usually, it is kind of hard to determine an appropriate GVT interval, because it is model-dependent. If the interval is too large, then this may slow down the execution, because of periodically low commiting IO. On the other hand, if the interval is too smal, the LPs cannot "peacefully" continue their work without being continuously "interrupted" by the controller on small periodic times.

In this model, because we have only 1 job executing at any time, it does not really matter if we go with a small GVT interval, since this will not greatly affect the LPs; there is not much work to be executed in the first place.

With a GVT interval of 1000 microsecs we have:

Model	HDES (secs)
LazyAirport	0.625
StrictAirport	23.07

So it is certainly clear than on a lazy setting, the slow fibonacci computations are not executed, thus benefitting from Haskell's lazy evaluation order.

6.4.2 Benchmarking against ARTIS

ARTIS is a recent state-of-the-art Parallel and Distributed Discrete-Event Simulation framework (PADS) written in C, that offers a time warp (optimistic parallel simulation) implementation, like HDES. It is closed source and employs system processes for parallel simulation (SMP on the same physical machine)

and MPI for distributed simulation. Here we compare our HDES with only the parallel section of ARTIS, since HDES is only parallel and not distributed (yet).

We ported an example that comes bundled with the ARTIS package. There are 3 airports (milan, bologna, rome) initially having 2 airplanes each, and these airplanes fly over on one of the 2 neighbouring airports randomly. Arrival events schedule Landed events and Landed Events schedule Departure events on a neighbouring airport. The simulation stops at time 2000. In the end, we count the total departures for each airport.

To begin with, let's compare the style of programming in ARTIS to that of our HDES framework. There are definitely certain limitations when programming with ARTIS. The event types in ARTIS are simple strings. This tends to be slower than using an enumeration or an Algebraic Datat Type (ADT). Also it does not allow any typos to be catched by the compiler at compile-time. Another limitation is that the ordering of events is achieved with simple string comparison (since events are strings). The simulation user, compared to HDES, cannot provide her own ordering of simultaneous events. The simulation user has the extra burden to explicitly include the clock in each LP's State structure. Beside this, the initialization clock is done to 0. This limits the clock to be a `Num` instance and only take positive values. In HDES the initialization of the clock is done to `undefined` or better thought as $-\infty$. The simulation user can as well use negative values as the initial clock value. The user is not even restricted of having Num instances, i.e. Int, Long, Float, Double. In ARTIS each remote event (message) have to be annotated by the user with the information of the sender. In HDES this is achieved implicitly. For safety reasons, the ARTIS framework redefines the `printf` procedure with some "magic" macros.

The greatest limitation in our opinion is that there are no safety guarantees in ARTIS when intermixing IO code with Simulation code; the user can include in her simulation code that is unsafe and can not be rollbacked. In HDES, however, because of Haskell's strong type system, the user cannot mix unsafe IO code in the simulation code and this is enforced by the type system itself. Overall, we could say the experience in programming with HDES and its high-level domain specific language is arguably better than using low-level ARTIS C code.

The results were run on a dual core 64-bit linux machine.

Setup	Time
ARTIS (3 processes, in reality 2-threaded)	2.123s
HDES (1 thread)	1.493s
HDES (2 threads)	0.612s

Note: The ported code, although the same, does not yield the same results for

ARTIS as well as HDES. The reason is the peculiarity of ARTIS for ordering simultaneous events. However, we assume almost the same workload for both HDES and ARTIS, since they have similar number of departures for each airport, as show below:

Airport	PDES	Total Departures
milan	ARTIS	553
milan	HDES	572
bologna	ARTIS	551
bologna	HDES	564
rome	ARTIS	567
rome	HDES	578

When comparing the artisAirport example for the HDES framework, we observer that the speedup is superlinear (greater than the expected linear):

Setup	Time
HDES (1 core)	1.493s
HDES (2 cores)	0.612s

Actual speedup ~= 2.44 > than linear speedup = 2

This looks strange in the beginning, but there is an intuitive reason behind it. The optimal way to run a parallel simulation is to match the number of processes (threads) with the number of LPs. In this way, the number of rollbacks would be minimum. For example, in this case, we have 1 core assigned with 3 threads (because of 3 LPs). At each time 1 thread is executing for some time, then another is picked up from the scheduler. It is more likely to rollback because 1 thread advances forward, where the others are stalled.

With the HDES framework we can collect engine statistics about the number of total rollbacks and the number of gvt rounds of a simulation run.

Setup	Time	Rollbacks
HDES (1 proc thread, 3 green threads)	1.493s	5230
HDES (2 proc threads, 3 green threads)	0.612s	997
HDES (3 proc threads, 3 green threads)	0.779s	982

We observe reduction on the rollbacks when we match the number of processor threads to the number of LPs. But we don't get any better execution speed when we have 3 processor threads, because our system is a dual-core. If we had a 4-core machine, we could possibly observe a reduction on time too.

6.5 Related Work

One of the oldest, but still operating, PDES frameworks is the Georgia Tech's Time Warp (GTW) implementation (Das et al. 1994). A contributor to GTW has been the author of the great book on Parallel and Distributed Systems (R. M. Fujimoto 2001). GTW, as well our own HDES framework, is limited to shared-memory multiprocessors and does not offer any kind of distributed execution. P. F. Riley and Riley (2003) proposes the System for Parallel Agent Discrete Event Simulator (SPADES), a distributed agent simulation environment with software-in-the-loop execution. SASSY by M. Hybinette et al. (2006) is another agent based simulation system that sits as a middleware between an agent-based API and a Paralel Discrete Event simulation (PDES) kernel. The Parallel Real-time Immersive network Modeling Environment (PRIME) is a recent project by the Modeling and Networking Systems Research Group from the Florida University. PRIME Liu, Li, and He (2009) is an open-source optimistic PDES framework mainly written in C++ and uses MPI for distributed simulation. (Perumalla 2005) proposes μsik, a low-level micro-kernel to build expressive PDES frameworks ontop. The kernel offers conservative, optimistic as well as mixed (conservative-optimistic) algorithms. What is astonishing is that in μsik the simulation user can change the deployed PDES algorithm on-the-fly.

The group of Parallel and Distributed Simulation (PADS) at the University of Bologna has been developing for the past years numerous PDES frameworks in many different languages. Their results include an industrial-strength Parallel and Distributed Simulation framework written in C, called ARTIS Bononi et al. (2005). Many high-level platforms, e.g. GAIA and LUNES, have been designed on top of the lower-level ARTIS framework. Unfortunately the simulation engine of ARTIS is closed-source. There have been two other, open-source this time, PDES implementations from the group; one implemented in Google's language Go (D'Angelo, Ferretti, and Marzolla 2012), and another in the Ericsson's programming language Erlang. These two implementations, however, are no longer developed or maintained. For an older survey of PDES platforms see (Low et al. 1999).

Imperative style with eager evaluation seems to be the driving force up until now for implementing PDES technology. Simply put, there was not any work done based in a lazy functional language setting before. For this reason, we consider our HDES platform, written in Haskell, as original work. However, there have been general simulation frameworks in the past implemented in Haskell with, unfortunately, no parallelism gain. Aivika is a Haskell library that provides

extensive system dynamics and discrete event simulation. Event-monad, as the name suggests, provides an event monad and monad transformer. It can be used as a low-level helper library to build a simulation framework. Users can create an event-graph simulation system and schedule events to it. It is not actively developed. Per se, it does not employ any parallelism, but it could theoretically be used together with a parallel strategy to exploit parallelism. Hasim is a library for process-based Discrete Event Simulation in Haskell. It does not employ any kind of parallelism.

Functional Reactive Programming (FRP for short) has been gaining great attention in the Haskell and the functional community in general. FRP can model systems that change/progress in time. Also FRP can deal with the so-called signals, i.e. the handling of discrete events. In this sense we could say that FRP is another sideview of Discrete-Event Simulation. Although it is well established before that FRP technology can be used for simulation in functional languages and while many FRP libraries have been proposed for Haskell and other programming languages, none of we know of is addressed for the simulation crowd.

6.6 Appendix

6.6.1 ARTIS code taken from their repository `airports_tw.c`

```
1   void    _printf(const char *fmt, ...)
2   {
3       int size;
4       va_list li;
5       char    buf[1024];
6
7
8       va_start(li, fmt);
9       vsprintf(buf, fmt, li);
10      va_end(li);
11
12      size = strlen(buf);
13      fseek(stdout, Pinfo->offset, SEEK_SET);
14
15      fputs(buf, stdout);
16      Pinfo->offset += size;
17  }
18  //
19  #define printf  _printf
20
21
22  //  Arrival (to the local LP) of an airplane
23  void    arrival_event_handler (int src, char *from, char *id)
24  {
25
26      //  Check the runway
27      if (Pinfo->runway_isfree){
28          //  It is free
29          Pinfo->in_the_air++;        // one more airplane is waiting for landing
30          Pinfo->runway_isfree = 0;   // lock the runway
31
32          TW_Schedule(Pinfo->clock+LAND_T, (void *)create_submsg ("LAN", from, id),
33                      sizeof(SubMsgHeader));
34          printf ("[%s]:   %.1f Waiting for landing, flight %s from %s\n", cnames[LP],
35                      Pinfo->clock, id, from);
36          fflush(stdout);
37      }
38      else {
39          //  If the source of the plane is not the local airport, then
40          //  one more airplane is waiting for landing
41          if(src != LP)   Pinfo->in_the_air++;
42
43          //  Schedule a new arrival event in future
44          TW_Schedule(Pinfo->clock+WAIT_T, (void *)create_submsg ("ARR", from, id),
45                      sizeof(SubMsgHeader));
```

```
46          printf ("[%s]:   Runway busy, the flight %s from %s is waiting\n", cnames[LP], id, from);
47          fflush(stdout);
48      }
49  }
50
51  //  Landing (to the local LP) of an airplane
52  void    landed_event_handler (char *da, char *id)
53  {
54      //  One less on air, one more on the ground
55      Pinfo->in_the_air--;
56      Pinfo->on_the_ground++;
57
58      //  Schedule the next departure (of the plane)
59      TW_Schedule (Pinfo->clock + LOAD_T, (void *)create_submsg ("DEP", "", id),
60                   sizeof(SubMsgHeader));
61
62
63      printf ("[%s]:     %.1f  Landed flight %s from %s\n", cnames[LP],
64              Pinfo->clock, id, da);
65      fflush(stdout);
66
67      //  Now the runway is free
68      Pinfo->runway_isfree = 1;
69  }
70
71  //  Departure (from the local LP) of an airplane
72  void    departure_event_handler (char *id)
73  {
74      int dest;   // destination airport (LP)
75      int ret;    // return value
76
77      //  One less on the ground
78      Pinfo->on_the_ground--;
79
80      //  Choose a random destination from the 2 available
81      dest = rnd_bool() == 0 ? NEIGHBOUR1 : NEIGHBOUR2;
82
83      //  Send to the destination airport (LP) the arrival event
84      ret = TW_Send(dest, Pinfo->clock + FLIGHT_T, (void *)create_submsg ("ARR", cnames[LP], id),
85                    sizeof(SubMsgHeader) );
86
87      //  Check the sending operation
88      if(ret >= 0) {
89          //  If OK then trace out the departure
90          printf ("[%s]: %.1f Departed flight %s with destination %s\n",  cnames[LP],
91                  Pinfo->clock, /*clock + FLIGHT_T,*/ id, cnames[dest]);
92          fflush(stdout);
93      }
94      else {
95          //  Else print an error message
96          //  FIXME
97      }
98  }
99
100 int main(int argc, char* argv[])
101 {
102     SubMsgHeader    *submsg;    // event message
103
104     char    msg[1024];      // message buffer
105     char    id[20];         // airplane identification string
106     int from;           // source LP (of a message)
107     int     i;          // temporary variable
108     long    ret;            // return value
109     double  Ts;         // time variable
110
111     //  Setup the TIME WARP synchronization algorithm
112     LP = TW_Init(cnames[atoi(argv[1])], argv[2], 5000, (void *)Pinfo,
113                  sizeof(*Pinfo), end_clock, GVT_THRESHOLD);
114
115     //  Initialize the airports identifiers
116     NEIGHBOUR1  = (LP + 1) % 3;
117     NEIGHBOUR2  = (LP + 2) % 3;
118     printf(" %s <- [%s] -> %s\n", cnames[NEIGHBOUR1], cnames[LP], cnames[NEIGHBOUR2] );
119
120     //  Initialization of the current LP state
121     Pinfo->offset   = 0;
122     Pinfo->SEED     = SEEDS[LP];        // random numbers generator, seed initialization
123     Pinfo->clock        = 0.0;          // simulated time initialization
124
125     //  Model variables initialization
126     Pinfo->runway_isfree    = 1;    // the runway is free
127     Pinfo->on_the_ground    = 2;    // 2 planes (in each airport) are on the ground
128     Pinfo->in_the_air   = 0;    // no airplanes are on the air
129     Pinfo->total_flies  = 0;    // total number of departed flies (from this airport)
130
131     //  Boostrapping the simulation model
132     for(i=0; i<PLANES_NUMBER; i++) {
133         //  Scheduling the first batch of departures
134         sprintf(id, "%s-%d", cnames[LP], i);
135         //  Sending the departure events
136         TW_Schedule(Pinfo->clock+WAIT_T+i,
```

```
137                     (void *)create_submsg ("DEP", cnames[LP], id),
138                      sizeof(SubMsgHeader) );
139     }
140
141     //   Main simulation loop, receives messages and calls the handler associated with them
142     while (!end_reached) {
143
144         //  Looking for a new incoming message
145         ret = TW_Receive( (void *)msg, &Ts, &from);
146
147         if(ret != END_SIMULATION) {
148
149             //  A message has been received, update the current simulated time
150             //      and call the appropriate message handler
151             submsg = (SubMsgHeader *)msg;
152
153             //  Updating the simulated time of the current state
154             Pinfo->clock = Ts;
155
156             //  Handlers
157             if (strcmp (submsg->event_type, "ARR")==0)              // arrival event
158                 arrival_event_handler (from, submsg->from, submsg->id);
159
160             if (strcmp (submsg->event_type, "LAN")==0)              // landing event
161                 landed_event_handler (submsg->from, submsg->id);
162
163             if (strcmp (submsg->event_type, "DEP")==0){             // departure event
164                 Pinfo->total_flies++;
165                 departure_event_handler (submsg->id);
166             }
167         }
168         else {
169             //  No more messages
170             printf ("[%s]: -- ending condition satisfied, clock=%.2f\n", cnames[LP],
171                     Pinfo->clock);
172             printf ("Total flights departed from %s: %d\n", cnames[LP],
173                      Pinfo->total_flies);
174             ftruncate(1, Pinfo->offset);
175
176             //  The simulation if finished
177             end_reached = 1;
178
179             //  The simulation is ended, we have to finalize the synchronization mechanism
180             TW_Finalize();
181         }
182     }
183
184     return 0;
185 }
```

Chapter 7

Conclusions and Future Work

We have described and discussed two simulation frameworks, namely HLogo and HDES, hosted in the same programming language, that is Haskell. HLogo is a clone of the famous in the simulation community NetLogo platform, that tries to exploit the Software Transactional Memory implementation of Haskell to speed up simulated models. And in certain specific cases it manages to succeed in its goal. HDES is a Parallel and Distributed Simulation (PDES) framework that employs a Time Warp optimistic algorithm. Its language, embedded in Haskell, is arguably a good basis for writing simulation software. By utilizing shared-memory parallelism, HDES manages to compete against a modern framework written in C and beat it.

7.1 HLogo

We have designed and implemented HLogo, a clone of the NetLogo framework that utilizes Software Transactional Memory to benefit in parallelism. The domain specific language of HLogo, although embedded in a totally different language (Haskell), tries to be as close as that of the original NetLogo. We have managed to port many NetLogo modules to HLogo and we benchmark two of those against the frameworks. Execution results show that HLogo is faster than NetLogo for most cases, particularly where the number of agents stay low. When the agent population grows as to produce significant number of STM conflicts, the performance of HLogo considerably drops.

For future work we want to investigate if the `atomic` construct can be automatically inserted in certain places of NetLogo programs without breaking its semantics; possible as the result of static program analysis. Another direction is

to create a source-to-source compiler (from a Logo-variant to Haskell) to make it easier for the simulation users to specify her models in. In this way, we may even achieve compatibility with already existing NetLogo-written source code.

Concerning the core of the execution model, we woud like to visit the possible execution of HLogo and its simulation runs in a distributed setting. Compared to a single physical multicore system, a distributed setting can enable Agent Based Models to be run in High-performance Computing (HPC); there is also the extreme case where the model cannot fit in a single shared memory and has to be distributed to many memory locations. Haskell technologies, such as Distributed Software Transactional Memory or Cloud Haskell could help us achieve this task.

The scheduling of workload to the threads does not involve any intelligence; it is simply turn-based (the next available thread picks the next agent). This naive method can lead to unnecessary STM conflicsts. By exploiting the spatial characteristics of the model we could better (more clever) assign the work to the threads, so that it minimizes the number of conflicts (retries). This work clustering could be static (on initialize) or adaptive (during runtime execution).

Our framework HLogo might be better considered as a proof of concept. In future work we would like to extend the NetLogo compiler by using the same codebase and adding on top of the execution model one of the many Scala STM implementations. We argue that the NetLogo language has then again to be extended with an `atomic` construct. Thus, we would benefit from staying as close as possible to the implementation of NetLogo, while still having the exact same syntax and utilizing the original Graphical User Interface.

7.2 HDES

We successfully designed and implemented an optimistic PDES framework in the Haskell programming language with the name HDES. The motivation initially was to exploit the inherit laziness of Haskell. The tests conducted show us that we have reached this goal. We ended up writing a domain specific language of HDES embedded in the host language Haskell. We argue that the HDES language is a better fit for regular simulation development, since its API is simple and intuitive. We compare and contrasted the HDES framework against a modern PDES platform, named ARTIS. We discussed the limitations of usual PDES implentations such as ARTIS and furthermore we benchmarked each framework for a common sample model (airport network). The results show that HDES can be upto twofold faster than its competitor ARTIS, while comprising half its code size.

For future work, we want to run more benchmarks, including in the comparison also other production-ready PDES frameworks. Concerning the HDES implementation, we would like in the future to replace our synchronous Time Warp

algorithm, with a more sophisticated and arguably faster algorithm, such as the asynchronous algorithms proposed by (Samadi, Muntz, and Parker 1987) and (Mattern 1989). A distributed execution of HDES is also planned, but it is is marked as considerable work, even with the latest advances of distributed computing in Haskell (Cloud Haskell). The Haskell programming language currently lacks good bindings to the de-facto MPI protocol.

Embedded domain-specific languages, such as HDES, inherit the semantics and the type system of the host language. The developer benefits from not having to implement error reporting for the language by herself. However, one disadvantage with this method is that error reporting, having primarily been designed specifically for the host language, becomes inadequate and confusing to say at least, when used for the designed DSL. A possible solution is to adopt the idea of (Heeren, Hage, and Swierstra 2003), where the developer has to write extra directives to drive error reporting without modifying the underlying compiler.

Another direction for improvements can be a GUI designed around HDES. Specifically we are interested to provide a user-friendly interface and appropriate dialogs for easily defining the performance measures/statistics.

We believe that the Haskell programming language can be a good host for simulation languages. It provides the right abstractions and its lazy execution model can be benefitial on most circumstances. Although there has been a vast 30-year-long research on Simulation theory, we can surprisingly still find room for improvements, by applying the theory to a high-level modern functional language.

References

Bononi, Luciano, Michele Bracuto, Gabriele D'Angelo, and Lorenzo Donatiello. 2005. "Scalable and efficient parallel and distributed simulation of complex, dynamic and mobile systems." In *Techniques, Methodologies and Tools for Performance Evaluation of Complex Systems*, 136–145. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1587702.

Burian, Jan. 2013. "Complex systems tutorial." http://eldar.cz/cognition/complex/.

Castle, C. J. E., and A. T. Crooks. 2006. "Principles and Concepts of Agent-Based Modelling for Developing Geospatial Simulations" (sep). *Centre for Advanced Spatial Analysis (UCL), UCL (University College London), Centre for Advanced Spatial Analysis (UCL): London, UK* http://eprints.ucl.ac.uk/3342/.

Das, Samir, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. 1994. "GTW: a time warp system for shared memory multiprocessors." In *Simulation Conference Proceedings, 1994. Winter*, 1332–1339. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=717527.

Deissenberg, Christophe, Sander van der Hoog, and Herbert Dawid. 2008. "EURACE: A massively parallel agent-based model of the European economy." *Applied Mathematics and Computation* 204 (2) (oct): 541–552. http://www.sciencedirect.com/science/article/pii/S0096300308003019.

Discolo, Anthony, Tim Harris, Simon Marlow, Peyton, and Satnamproxy Singh. 2006. "Lock -Free Data Structures Using STMs in Haskell. "Eighth International Symposium on Functional and Logic Programming (April)". http://research.microsoft.com/~simonpj/papers/stm/lock-free.htm.

D'Angelo, Gabriele, Stefano Ferretti, and Moreno Marzolla. 2012. "Time Warp on the Go." In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, 242–248. http://dl.acm.org/citation.cfm?id=2263057.

D'Souza, Roshan M., Mikola Lysenko, and Keyvan Rahmani. 2007. "SugarScape on Steroids: Simulating over a Million Agents at Interactive Rates." In *Proceedings of Agent2007 Conference. Chicago, IL.*

Epstein, Joshua M., and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up.* Brookings Institution Press.

Fujimoto, Richard M. 2001. "Parallel Simulation: Parallel and Distributed Simulation Systems." In *Proceedings of the 33nd Conference on Winter Simulation, 147–157. WSC '01. Washington, DC, USA: IEEE Computer Society.* http://dl.acm.org/citation.cfm?id=564124.564145.

Grimm, Volker, Uta Berger, Finn Bastiansen, Sigrunn Eliassen, Vincent Ginot, Jarl Giske, John Goss-Custard, et al. 2006. "A standard protocol for describing individual-based and agent-based models." *Ecological Modelling* 198 (1–2) (sep): 115–126. http://www.sciencedirect.com/science/article/pii/S0304380006002043.

Grimm, Volker, Eloy Revilla, Uta Berger, Florian Jeltsch, Wolf M. Mooij, Steven F. Railsback, Hans-Hermann Thulke, Jacob Weiner, Thorsten Wiegand, and Donald L. DeAngelis. 2005. "Pattern-Oriented Modeling of Agent-Based Complex Systems: Lessons from Ecology." *Science* 310 (5750) (nov): 987–991 http://www.sciencemag.org/content/310/5750/987.

Harris, Tim, 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*, 48–60. http://dl.acm.org/citation.cfm?id=1065952.

Harris, Tim, and Simon Peyton Jones. 2006. "Transactional Memory with Data Invariants." In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, Ottowa. http://ipaper.googlecode.com/git-history/658c8c23e7496300442f004a97ba1c5fe8b8c6f0/Simon-Peyton-Jones/stm-invariants.pdf.

Heeren, Bastiaan, Jurriaan Hage, and S. Doaitse Swierstra. 2003. "Scripting the type inference process." In *ACM SIGPLAN Notices*, 38:3–13. http://dl.acm.org/citation.cfm?id=944707.

Hybinette, Maria, Eileen Kraemer, Yin Xiong, Glenn Matthews, and Jaim Ahmed. 2006. "SASSY: a Design for a Scalable Agent-based Simulation System Using a Distributed Discrete Event Infrastructure." In *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, 926–933. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4117701.

Jefferson, David, Brian Beckman, Fred Wieland, Leo Blume, and Mike DiLoreto. 1987. *Time warp operating system.* Vol. 21. ACM. http://dl.acm.org/citation.cfm?id=37508.

Jefferson, David, and Henry Sowizral. 1985. "Fast concurrent simulation using the Time Warp mechanism." In *SCS Conf. Distributed Simulation*, 63–69.

Jones, Simon Peyton, Andrew Gordon, and Sigbjorn Finne. 1996. "Concurrent haskell." In *POPL*, 96:295–308. http://citeseerx.ist.psu.edu/viewdoc/

download?doi=10.1.1.47.7494&rep=rep1&type=pdf.

Knight, Tom. 1986. “An Architecture for Mostly Functional Languages.” In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 105–112. http://dl.acm.org/citation.cfm?id=319854.

Koehler, M. T. K., B. F. Tivnan, and S. Upton. 2005. “Clustered Computing with Netlogo and Repast J: Beyond chewing gum and duct tape.” In *Agent2005 conference, Chicago, IL.*.

Law, Averill M. 2007. *Simulation modeling and analysis*. Boston: McGraw-Hill.

Liu, Jason, Yue Li, and Ying He. 2009. “A large-scale real-time network simulation study using prime.” In *Winter Simulation Conference*, 797–806. http://dl.acm.org/citation.cfm?id=1995574.

Logan, Brian, and Georgios Theodoropoulos. 2001. “The distributed simulation of multiagent systems” 89 (2): Proceedings of the IEEE 89 (2): 174–185. 174–185. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=910853.

Low, Yoke-Hean, Chu-Cheow Lim, Wentong Cai, Shell-Ying Huang, Wen-Jing Hsu, Sanjay Jain, and Stephen J. Turner. 1999. “Survey of languages and runtime libraries for parallel discrete-event simulation” 72 (3): 170–186. http://sim.sagepub.com/content/72/3/170.short.

Marlow, Simon. 2013. “Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming”. O’Reilly Media, Inc.

Massaioli, Federico, Filippo Castiglione, and Massimo Bernaschi. 2005. “OpenMP Parallelization of Agent-based Models.” Parallel Computing 31 (10): 1066–1081.

Mattern, Friedemann. 1989. “Virtual time and global states of distributed systems” 1 (23): 215–226. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.4399&rep=rep1&type=pdf.

Meadows, Donella H., Donella H. Meadows, Jorgen Randers, and William W. Behrens III. 1972. “The Limits to Growth: A Report to The Club of Rome”. Universe Books, New York. http://www.ask-force.org/web/Global-Warming/Meadows-Limits-to-Growth-Short-1972.pdf.

Misra, Jayadev. 1986. “Distributed Discrete-event Simulation.” ACM Computing Surveys (CSUR) 18 (1): 39–65. http://dl.acm.org/citation.cfm?id=6485.

Perfumo, Cristian, Nehir Sönmez, Srdjan Stipic, Osman Unsal, Adrián Cristal, Tim Harris, and Mateo Valero. 2008. “The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment.” In , 67–78. New York, NY, USA: ACM. http://doi.acm.org/10.1145/1366230.1366241.

Perumalla, Kalyan S. 2005. “Msik-a Micro-kernel for Parallel/distributed Simulation Systems.” In Principles of Advanced and Distributed Simulation. PADS 2005. 59–68. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1443311.

Pogson, Mark, Rod Smallwood, Eva Qwarnstrom, and Mike Holcombe. 2006. “Formal agent-based modelling of intracellular chemical interactions” 85 (1): 37–45. http://www.sciencedirect.com/science/article/pii/S0303264706000335.

Railsback, Steven F., Steven L. Lytinen, and Stephen K. Jackson. 2006. “Agent-based Simulation Platforms: Review and Development Recommendations.” *SIMULATION* 82 (9) (sep): 609–623. http://sim.sagepub.com/content/82/9/609.

Riley, Patrick F., and George F. Riley. 2003. “Next generation modeling III - agents: Spades — a distributed agent simulation environment with software-in-the-loop execution.” In , 817–825. Winter Simulation Conference. http://dl.acm.org/citation.cfm?id=1030818.1030926.

Sakellariou, Ilias, Petros Kefalas, and Ioanna Stamatopoulou. 2008. “Enhancing NetLogo to Simulate BDI Communicating Agents.” In *Artificial Intelligence: Theories, Models and Applications*, ed. John Darzentas, George A. Vouros, Spyros Vosinakis, and Argyris Arnellos, 263–275. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-540-87881-0_24.

Salamon, T. 2011. *Design of Agent-Based Models*. Tomás Bruckner. http://books.google.com/books?hl=en&lr=&id=2rCdKnltaH8C&oi=fnd&pg=PA1&dq=Design+of+Agent-Based+Models&ots=CA7FBR2jpF&sig=k4i1bU0RCicEfSDNXDEtzkZKOwE.

Samadi, Behrokh, Richard R. Muntz, and D. S. Parker. 1987. “A distributed algorithm to detect a global state of a distributed simulation system.” In *Proc. IFIP Conference on Distributed Processing*.

Shavit, Nir, and Dan Touitou. 1995. “Software transactional memory.” In *Proceeding PODC '95 Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* , 204–213. New York, NY, USA: ACM. http://doi.acm.org/10.1145/224964.224987.

Tobias, Robert, and Carole Hofmann. 2004. “Evaluation of free Java-libraries for social-scientific agent based simulation.” In *Journal of Artificial Societies and Social Simulation*. http://jasss.soc.surrey.ac.uk/7/1/6.html.

Wilensky, Uri. 2003. “Statistical mechanics for secondary school: The GasLab multi-agent modeling toolkit” In *International Journal of Computers for Mathematical Learning* 8 (1): 1–41. http://link.springer.com/article/10.1023/A:1025651502936.

Wilkerson-Jerde, M., and Uri Wilensky. 2010. “Restructuring change, interpreting changes: The deltatick modeling and analysis toolkit.” In *Restructuring change, interpreting changes: The deltatick modeling and analysis toolkit*

Wooldridge, Michael J. 2009. *An introduction to multiagent systems*. Chichester, U.K.: John Wiley & Sons.