Functional Reactive ABMS

Towards pure functional Agent-Based Modelling & Simulation

Jonathan Thaler
School of Computer Science
University of Nottingham
jonathan.thaler@nottingham.ac.uk

Peer-Olaf Siebers
School of Computer Science
University of Nottingham
peer-olaf.siebers@nottingham.ac.uk

***Abstract*—The pure functional paradigm has so far got not very much attention in ABS where the dominant programming paradigm is object-orientation, mainly through Java. In preliminary research of ours, we came to the insight that pure functional programming using Haskell without any additional libraries is very well suited to implement ABS but that in order to build more complex, reald-world models it would be necessary to leverage the basic concepts with functional reactive programming (FRP). In this paper we show how ABS can be fused with FRP to design an embedded domain specific language (EDSL) which allows to write specifications which (nearly) match pure functional code - the specification becomes Haskell code.**

As an example we focus on the simulation of a Wild-Fire

***Index Terms*—Agent-Based Simulation, Functional Programming, Haskell**

I. Introduction

TODO: talk about preliminary research and its drawbacks: need something like FRP / Yampa to leverage TODO: need a formal specification

TODO: this is a paper for working out the depth of one of the directions of the PhD: ACE, formulate research-questions

II. Background

A. Agent Based Simulation

We understand ABS as a method of modelling and simulating a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is of knowledge. Those parts, called agents, are modelled and simulated out of which then the aggregate global behaviour of the whole system emerges. So the central aspect of ABS is the concept of an agent which can be understood as a metaphor for a pro-active unit, situated in an environment, able to spawn new agents and interacting with other agents in a network of neighbours by exchange of messages [1]. It is important to note that we focus our understanding of ABS on a very specific kind of agents where the focus is on communicating entities with individual, localized behaviour from out of which the global behaviour of the system emerges. We informally assume the following about our agents:

- They are uniquely addressable entities with some internal state.
- They can initiate actions on their own e.g. change their internal state, send messages, create new agents, kill themselves.
- They can react to messages they receive with actions as above.
- They can interact with an environment they are situated in.

We will give a formal model of agents in a subsequent section (TODO: reference)

B. Functional Reactive Programming

So far we have considered only quite low-level approaches to structuring and composing functional programming: higher-order functions, laziness, monads and arrows. What we need is a programming paradigm built into pure functional programming which we can leverage to implement ABS. As already mentioned above, functional reactive programming (FRP) seems to be a highly promising approach. It is rather a lucky coincidence that Henrik Nilsson, one of the major contributor to the library Yampa, an implementation of FRP, is situated at the School of Computer Science of the University of Nottingham.

FRP is a paradigm for programming hybrid systems which combine continuous and discrete components. Time is explicitly modelled: there is a continuous and synchronous time flow. There have been many attempts to implement FRP in libraries which each has its benefits and deficits. The very first functional reactive language was Fran, a domain specific language for graphics and animation. At Yale FAL, Frob, Fvision and Fruit were developed. The ideas of them all have then culminated in Yampa, the most recent FRP library [2]. The essence of FRP with Yampa is that one describes the system in terms of signal functions in a declarative manner using the EDSL of Yampa. During execution the top level signal functions will then be evaluated and return new signal functions which act as continuations. A major design goal for FRP is to free the programmer from 'presentation' details by providing the ability to think in terms of 'modeling'. It is common that an FRP program is concise enough to also serve as a specification for the problem it solves [3].

Yampa has been used in multiple agent-based applications: [4] uses Yampa for implementing a robot-simulation, [5] implement the classical Space Invaders game using Yampa, [6] implements a Pong-clone, the thesis of [7] shows how Yampa can be used for implementing a Game-Engine, [8] implemented a 3D first-person shooter game with the style of

Quake 3 in Yampa. Note that although all these applications don't focus explicitly on agents all of them inherently deal with kinds of agents which share properties of classical agents: game-entities, robots,... Other fields in which Yampa was successfully used were programming of synthesizers, network routers, computer music development and has been successfully combined with monads [9].

This leads to the conclusion that Yampa is mature, stable and suitable to be used in functional ABS. This and the reason that we have the in-house knowledge lets us focus on Yampa. Also it is out-of-scope to do a in-depth comparison of the many existing FRP libraries.

1) Yampa: The central concept of Yampa is the one of a signal-function which can be understood of a mapping from an input-signal to an output-signal. Whether the signal is discrete or continuous does not matter, Yampa is suited equally well to both kinds. Signal-functions are implemented in Yampa using continuations which allow to freeze program-state e.g. through closures and partial applications in functions which can be continued later:

```
type DTime = Double

data SF a b = SF { sfTF :: DTime -> a -> (SF a b, b) }
```

Such a signal-function, which is called a *transition function* in Yampa, takes the amount of time which has passed since the previous time step and the current input signal (a). It returns a *continuation* of type SF a b determining the behaviour of the signal function on the next step and an output signal (b) of the current time-step.

Yampa provides a top-level function, running in the IO-Monad, which drives a signal-function by providing both input-values and time-deltas from callbacks. It is important to note that when visualizing a simulation one has in fact two flows of time: the one of the user-interface which always follows real-time flow, and the one of the simulation which could be sped up or slowed down. Thus it is important to note that if I/O of the user-interface (rendering, user-input) occurs within the simulations time-frame then the user-interfaces real-time flow becomes the limiting factor. Yampa provides the function embedSync which allows to embed a signal function within another one which is then run at a given ratio of the outer SF. This allows to give the simulation its own time-flow which is independent of the user-interface. We utilized this in the implementation of Recursive ABS (see Chapter **??**).

Additional functionality which Yampa provides is the concept of Events which allow to implement changing behaviour of signal-functions at given points in time. An event can be understood to be similar to the Maybe-type of Haskell which either is an event with a given type or is simply NoEvent. Yampa provides facilities to detect if an event has fired and also provides functions to switch the signal-function into a new signal-function with changed behaviour. Another feature of Yampa is its EDSL for time-semantics: integration over time, delay, accumulation, holding, firing events after/now/repeatedly.

C. Related Research

The amount of research on using the pure functional paradigm using Haskell in the field of ABS has been moderate so far. Most of the papers look into how agents can be specified using the belief-desire-intention paradigm [10], [11], [12]. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in [13]. It comes with very basic features for ABS but only allows to specify simple state-based agents with timed transitions. [12] which discuss using functional programming for DES mention the paradigm of functional reactive programming (FRP) to be very suitable to DES. [14] and [15] present a domain-specific language for developing functional reactive agent-based simulations. This language called FRABJOUS is human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell but is compiled to Yampa code - a FRP library for Haskell - which they claim is also readable. It seems that FRP is a promising approach to ABS in Haskell, an important hint we will follow in the section below.

Tim Sweeney, CTO of Epic Games gave an invited talk in which he talked about programming languages in the development of game-engines and scripting of game-logic [16]. Although the fields of games and ABS seem to be very different, in the end they have also very important similarities: both are simulations which perform numerical computations and update objects in a loop either concurrently or sequential [1]. In games these objects are called *game-objects* and in ABS they are called *agents* but they are conceptually the same thing. The two main points Sweeney made were that dependent types could solve most of the run-time failures and that parallelism is the future for performance improvement in games. He distinguishes between pure functional algorithms which can be parallelized easily in a pure functional language and updating game-objects concurrently using software transactional memory (STM).

The thesis of [18] constructs two frameworks: an agent-modelling framework and a DES framework, both written in Haskell. They put special emphasis on parallel and concurrency in their work. The author develops two programs with strong emphasis on parallelism: HLogo which is a clone of the NetLogo agent-modelling framework and HDES, a framework for discrete event simulation.

D. NetLogo

One can look at NetLogo as a functional approach to ABMS which comes with its own EDSL. Our approach differs fundamentally in the following way - no side-effects possible - no direct access to other agents, communication happens through asynchronous messages or synchronized conversations - powerful time-semantics which NetLogo completely lacks

[1] Gregory [17] defines computer-games as *soft real-time interactive agent-based computer simulations*

E. Examples

1) SIRS: [ ] SIRS: timed transitions using after, occasionally sending messages, transitions on messages

2) Wildfire: [ ] WildFire: rate transitions, occasionally sending messages, transition on messages

III. The formal Agent-Model

Functional programming in Haskell with its strong static type-system can be seen as to be much more formal than object-oriented programming in Java and C++. Where in the latter one e.g. a Class-Diagram in UML would be created, in our functional approach we create a formal model of our agent which can then be easily translated to Haskell.

An agent can be seen as a tuple $< id, s, m, ec, b >$.

- **id** - the unique identifier of the agent
- **s** - the generic state of the agent
- **m** - the set of messages the agent understands
- **ec** - the *type* of the environment-cells the agent may act upon
- **b** - the behaviour of the agent

A. Id

The id is simply represented as an Integer and must be unique for all currently existing agents in the system as it is used for message-delivery. A stronger requirement would be that the id of an agent is unique for the whole simulation-run and will never be reused - this would support replications and operations requiring unique agent-ids.

B. State

Each agent may have a generic state comprised of any data-type, most likely to be a structure.

```
data SIRSState = Susceptible | Infected | Recovered
data SIRSAgentState = SIRSAgentState {
    sirsState :: SIRSState,
    sirsCoord :: SIRSCoord,
    sirsTime :: Double
}
```

It is possible that the agent does not rely on any state s, then this will be represented by the unit type (). One wonders if this makes sense and asks how agents can then be distinguished between each other. In functional programming this is easily possible using currying and closures where one encapsulate initial state in the behaviour (see below), which allows to give each agent an individual initial state.

C. Messages

Agents communicate with each other through messages (see below) and thus need to have an agreed set of messages they understand. This is usually implemented as an ADT.

```
data SIRSMsg = Contact SIRSState
```

D. Environment-Cell

The agent needs to know the generic type of the cells the environment is made of to be able to act upon the environment. Note that at the moment we only implemented a discrete 2d environment and provide only access and manipulation to the cells in a 2d discrete fashion. In the case of a continuous n-dimensional environment this approach needs to be thoroughly revised. It is important to understand that it is the *type* of the cells and not the environment itself.

E. Behaviour

The behaviour of the agent is a signal-function which maps an AgentIn-Signal to an AgentOut-Signal. It has the following signature:

```
type AgentBehaviour s m e = SF (AgentIn s m e) (Ag
```

AgentIn provides the necessary data to the agent-behaviour: its id, incoming messages, the current state s, the environment (made out of the cells ec), its position in the environment and a random-number generator.

AgentOut allows the agent to communicate changes out of the behaviour: kill itself, create new agents, sending messages, state s, environment (made ouf of the cells ec), environment-position and random-number generator.

IV. Environment

So far we only implemented a 2d-discrete environment. It can be understood to be a tuple of $< b, d, n, w, cs >$.

- **b** - the optional behaviour of the environment
- **d** - the dimensions of the environment: its maximum boundary extending from (0,0)
- **n** - the neighbourhood of the environment (Neumann, Moore)
- **w** - the wrapping-type of the environment (clipping, horizontal, vertical, both)
- **cs** - the cells of the environment of type c

We represent the environment-behaviour as a signal-function as well but one which maps an environment to itself. It has the following signature:

```
type EnvironmentBehaviour c = SF (Environment c) (
```

This is a regular SF thus having also the time of the simulation available and is called after all agents are updated. Note that the environment cannot send messages to agents because it is not an agent itself. An example of an environment behaviour would be to regrow some good on each cell according to some rate per time-unit (inspired by SugarScape regrowing of Sugar).

The cells are represented as a 2-dimensional array with indices from (0,0) to limit and a cell of type c at every position. Note that the cell-type c is the same environment-cell type ec of the agent.

Each agent has a copy of the environment passed in through the AgentIn and can change it by passing a changed version of the environment out through AgentOut.

V. Functional Reactive ABS

the fundamental problem is that unlike in oo e.g. java there are no objects and no implicit aliases through which to acess and change data: method calls are not there in FP. we must silve the problem of how to represent an agent and how agents can interact with each other

using example SIRS or Wildfire TODO: show how simple state-transition agents work using switch TODO: show how the time-semantics can be used

1) Representing an agent and environment - there are no classes and objects in Haskell.
2) Interactions among agents and actions of agents on the environment - there are no method-calls and aliases in Haskell.
3) Implement the necessary update-strategies as discussed in our paper **??**, where we only focus on sequential- and parallel-strategies - there is no mutable data which can be changed implicitly through side-effects (e.g. the agents, the list of all the agents, the environment).

A. Messaging

As discussed in the literature reflection in Chapter **??**, inspired by the actor model we will resort to synchronized, reliable message passing with share nothing semantics to implement agent-agent interactions. Each Agent can send a message to an other agent through AgentOut-Signal where the messages are queued in the AgentIn-Signal and can be processed when the agent is updated the next time. The agent is free to ignore the messages and if it does not process them they will be simply lost. Note that due to the fact we don't have method-calls in FP, messaging will always take some time, which depends on the sampling interval of the system. This was not obviously clear when implementing ABS in an object-oriented way because there we can communicate through method calls which are a way of interaction which takes no simulation-time.

B. Conversations

The messaging as implemented above works well for one-directional, virtual asynchronous interaction where we don't need a reply at the same time. A perfect use-case for messaging is making contact with neighbours in the SIRS-model: the agent sends the contact message but does not need any response from the receiver, the receiver handles the message and may get infected but does not need to communicate this back to the sender. A different case is when agents need to transact in the time-step one or multiple times: agent A interacts with agent B where the semantics of the model (and thus messaging) need an immediate response from agent B - which can lead to further interactions initiated by agent A. The Sugarscape model has three use-cases for this: sex, warfare and trading amongst agents all need an immediate response (e.g. wanna mate with me?, I just killed you, wanna trade for this price?). The reason is that we need to transact now as all of the actions only work on a 1:1 relationship and could violate ressource-constraints. For this we introduce the concept of a conversation between agents. This allows an agent A to initiate a conversation with another agent B in which the simulation is virtually halted and both can exchange an arbitrary number of messages through calling and responding without time passing (something not possible without this concept because in each iteration the time advances). After either one agent has finished with the conversation it will terminate it and the simulation will continue with the updated agents (note the importance here: *both* agents can change their state in a conversation). The conversation-concept is implemented at the moment in the way that the initiating agent A has all the freedom in sending messages, starting a new conversation,... but that the receiving agent B is only able to change its state but is not allowed to send messages or start conversations in this process. Technically speaking: agent A can manipulate an AgentOut whereas agent B can only manipulate its next AgentIn. When looking at conversations they may look like an emulation of method-calls but they are more powerful: a receiver can be unavailable to conversations or simply refuse to handle this conversation. This follows the concept of an active actor which can decide what happens with the incoming interaction-request, instead of the passive object which cannot decide whether the method-call is really executed or not.

C. Iteration-Strategies

Building on the foundations laid out in my paper about iteration-strategies in Appendix **??**, we implement two of the four strategies: sequential- and parallel-strategy. We deliberately ignore the concurrent- and actor-strategy for now and leave this for further research [2]. Implementing iteration-strategies using Haskell and FRP is not as straight-forward as in e.g. Java because one does not have mutable data which can be updated in-place. Although my work on programming paradigms in Appendix **??** did not take FRP into account, general concepts apply equally as well.

1) Sequential: In this strategy the agents are updated one after another where the changes (messages sent, environment changed,...) of one agent are visible to agents updated after. Basically this strategy is implemented as a variant of fold which allows to feed output of one agent (e.g. messages and the environment) forward to the other agents while iterating over the list of agents. For each agent the agent-behaviour signal-function is called with the current AgentIn as input to retrieve the according AgentOut. The messages of the AgentOut are then distributed to the receivers AgentIn. The environment of the agent, which is passed in through AgentIn and returned through AgentOut will then be passed forward to all agents i + 1 AgentIn in the current iteration and override their old environment. Thus all steps of changes made to the environment are visible in the AgentOuts. The last environment is then the final environment in the current

[2] Also both strategies would require running in the STM-Monad, which is not possible with Yampa. The work of Ivan Perez in [9] implemented a library called Dunai, which is the same as Yampa but capable of running in an arbitrary Monad.

iteration and will be returned by the callback function together with the current AgentOuts.

2) Parallel: The parallel strategy is *much* easier to implement than the sequential but is of course not applicable to all models because of it different semantics. Basically this strategy is implemented as a map over all agents which calls each agent-behaviour signal-function with the agents AgentIn to retrieve the new AgentOut. Then the messages are distributed amongst all agents. A problem in this strategy is that the environment is duplicated to each agent and then each agent can work on it and return a changed environment. Thus after one iteration there are n versions of environments where n is equals to the number of agents. These environments must then be collapsed into a final one which is always domain-specific thus needs to be done through a function provided in the environment itself.

D. Environment

TODO: again cite my own work where I discussed the problem of environments

Each agent has a copy of the environment passed in through the AgentIn and can change it by passing a changed version of the environment out through AgentOut. In the sequential update-strategy the environment of the agent i will then be passed to all agents i + 1 AgentIn in the current iteration and override their old environment. Thus all steps of changes made to the environment are visible in the AgentOuts. The last environment is then the final environment int he current iteration and will be returned by the callback function together with the current AgentOuts. In the parallel update-strategy the environment is duplicated to each agent and then each agent can work on it and return the changed environment. Thus after one iteration there are n versions of environments where n is equals to the number of agents. These environments must then be collapsed into a final one which is always domain-specific thus needs to be done through a function provided in the environment itself. In both the sequential and parallel update-strategy after one iteration there is one single environment left. An environment can have an optional behaviour which allows the environment to update its cells. This is a regular SF thus having also the time of the simulation available. Note that the environment cannot send messages to agents because it is not an agent itself. An example of an environment behaviour would be to regrow some good on each cell according to some rate per time-unit (inspired by SugarScape regrowing of Sugar).

E. Time-Semantics

The main reason for building our pure functional ABMS approach on top of Yampa was to leverage the powerful time-semantics of Yampa which allows us to implement important concepts of ABMS:

state-chart: agents are at all time of their life-cycle in one state and can switch between multiple states using transitions timed transitions: transition to another state/behaviour happens at a discrete time rate transitions: transition happens with a given rate message transition: transition upon receiving a given message

VI. Discussion

advantages: - no side-effects within agents leads to much safer code - edsl for time-semantics - declarative style - sequential and parallel - powerful time-semantics - arrowized programming is optional and only required when utilizing yampas time-semantics. if the model does not rely on time-semantics, it can use monadic-programming by building on the existing monadic functions in the EDSL which allow to run in the State-Monad which simplifies things very much - when to use yampas arrowized programing: time-semantics, simple state-chart agents - when not using yampas facilities: in all the other cases e.g. SugarScape is such a case as it proceeds in unit time-steps and all agents act in every time-step

disadvantages: - very steep learning curve for non-functional programmers - learning a new EDSL - think ABMS different: when to use async messages, when to use sync conversations

[ ] important: increasing sampling freqzency and increasing number of steps so that the same number of simulation steps are executed should lead to same results. but it doesnt. why? [ ] hypothesis: if time-semantics are involved then event ordering becomes relevant for emergent patterns. there are no tine semantics in heroes and cowards but in the prisoners dilemma [ ] can we implement different types of agents interacting with each other in the same simulation ? with different behaviour funcs, digferent state? yes, also not possible in NetLogo to my knowledge. but they must have the same messages, emvironment

[ ] Hypothesis: we can combine with FrABS agent-based simulation and system dynamics

VII. Conclusion and future research

further research - verification & validation - switch to Dunai to allow usage of Monadic programming in the arrows

In his 1st year report about Functional Reactive GUI programming, Ivan Perez [3] writes: "FRP tries to shift the direction of data-flow, from message passing onto data dependency. This helps reason about what things are over time, as opposed to how changes propagate". This of course raises the question whether FRP is *really* the right approach, because the way we implement ABS, message-passing is an essential concept. It is important to emphasis that agent-relations in interactions are never fixed in advance and are completely dynamic, forming a network. Maybe one has to look at message passing in a different way in FRP, and to view and model it as a data-dependency but it is not clear how this can be done. The question is whether there is a mechanism in which we have explicit data-dependency but which is dynamic like message-passing but does not try to fake method-calls? Maybe the concept of conversations (see above) are a way to go but we leave this for further research at the moment.

[3] main author of the paper [9]

REFERENCES

[1] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. Wiley Publishing, 2009.

[2] H. Nilsson, A. Courtney, and J. Peterson, "Functional Reactive Programming, Continued," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '02. New York, NY, USA: ACM, 2002, pp. 51–64. [Online]. Available: http://doi.acm.org/10.1145/581690.581695

[3] Z. Wan and P. Hudak, "Functional Reactive Programming from First Principles," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: ACM, 2000, pp. 242–252. [Online]. Available: http://doi.acm.org/10.1145/349299.349331

[4] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, Robots, and Functional Reactive Programming," in *Advanced Functional Programming*, ser. Lecture Notes in Computer Science, J. Jeuring and S. L. P. Jones, Eds. Springer Berlin Heidelberg, 2003, no. 2638, pp. 159–187, dOI: 10.1007/978-3-540-44833-4_6. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-44833-4_6

[5] A. Courtney, H. Nilsson, and J. Peterson, "The Yampa Arcade," in *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '03. New York, NY, USA: ACM, 2003, pp. 7–18. [Online]. Available: http://doi.acm.org/10.1145/871895.871897

[6] H. Nilsson and I. Perez, "Declarative Game Programming: Distilled Tutorial," in *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '14. New York, NY, USA: ACM, 2014, pp. 159–160. [Online]. Available: http://doi.acm.org/10.1145/2643135.2643160

[7] G. Meisinger, "Game-Engine-Architektur mit funktional-reaktiver Programmierung in Haskell/Yampa," Master, Fachhochschule Obersterreich - Fakultt fr Informatik, Kommunikation und Medien (Campus Hagenberg), Austria, 2010.

[8] C. Mun Hon, "Functional Programming and 3d Games," Ph.D. dissertation, University of New South Wales, Sydney, Australia, 2005.

[9] I. Perez, M. Brenz, and H. Nilsson, "Functional Reactive Programming, Refactored," in *Proceedings of the 9th International Symposium on Haskell*, ser. Haskell 2016. New York, NY, USA: ACM, 2016, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2976002.2976010

[10] T. De Jong, "Suitability of Haskell for Multi-Agent Systems," University of Twente, Tech. Rep., 2014.

[11] M. Sulzmann and E. Lam, "Specifying and Controlling Agents in Haskell," Tech. Rep., 2007.

[12] P. Jankovic and O. Such, "Functional Programming and Discrete Simulation," Tech. Rep., 2007.

[13] D. Sorokin, *Aivika 3: Creating a Simulation Library based on Functional Programming*, 2015.

[14] O. Schneider, C. Dutchyn, and N. Osgood, "Towards Frabjous: A Two-level System for Functional Reactive Agent-based Epidemic Simulation," in *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium*, ser. IHI '12. New York, NY, USA: ACM, 2012, pp. 785–790. [Online]. Available: http://doi.acm.org/10.1145/2110363.2110458

[15] I. Vendrov, C. Dutchyn, and N. D. Osgood, "Frabjous: A Declarative Domain-Specific Language for Agent-Based Modeling," in *Social Computing, Behavioral-Cultural Modeling and Prediction*, ser. Lecture Notes in Computer Science, W. G. Kennedy, N. Agarwal, and S. J. Yang, Eds. Springer International Publishing, Apr. 2014, no. 8393, pp. 385–392, dOI: 10.1007/978-3-319-05579-4_47. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-05579-4_47

[16] T. Sweeney, "The Next Mainstream Programming Language: A Game Developer's Perspective," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 269–269. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111061

[17] J. Gregory, *Game Engine Architecture, Third Edition*. Taylor & Francis, Mar. 2018.

[18] N. Bezirgiannis, "Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism," Ph.D. dissertation, Utrecht University - Dept. of Information and Computing Sciences, 2013.