Design of A Conversion of both Semantics and Representation between Agent-Based Models and Recurrence Relations

Leo Carlos-Sandberg
Supervisor: Dr Christopher D. Clack

July 19, 2017

Abstract

This paper investigates a method of describing interacting systems from two opposing view points, recurrence relations and agent-based models. These two methods take fundamentally different approaches with recurrence relations being top-down, and agent-based models being bottom-up.
Connecting these two methods allows for a more complete investigation of emergent behaviour occurring within interacting systems. Agent-based models offer an attractive way of analysing emergent behaviour, with the ability to investigate individual interactions throughout a simulation. Agent-based models however tend to be less well understood and accepted by non-computer science experts, this is in contrast to recurrence relation which are normally well understood. Creating a correctness preserving link between recurrence relations and agent-based models allows for simulations to be understood in their recurrence relation representation and hence have their agent-based model representation accepted. This is important in fields such as finance as it opens up new tools for economists and regulators to use in understanding emergence in complex markets.
This research comprises of the creation of a recurrence relation language, allowing for simulations to be defined, and the fabrication of correctness preserving transformation software, that transforms the custom recurrence relation language into a agent-based model simulation, known as InterDyne.

Contents

1 Introduction

Financial markets exhibit many interesting and some times unexpected events, one type of these such events are flash crashes. A notable example of this event type, is the flash crash of 2010, in this crash the E-min S&P 500 equity futures market dropped in price by more then 5%, during which time market prices and rational valuations became disconnected, before rebound to close to its original price [1, 2]. This whole process occurred very rapidly, lasting approximately thirty-six minutes and has been described as“one of the most turbulent periods in their history” for the US financial markets [3].
A number of flash crashes have been show to originate from large sell orders, such as in the recent crash in the cryptocurrency ethereum [4]. The initial analysis of the 2010 flash crash came to a similar conclusion, with the joint report by the U.S. Commodity Futures Trading Commission (CFTC) and Securities & Exchange Commission (SEC), claiming that the crash was caused by a large sell order by a mutual fund[1] [1]. This initial finding is not fully accepted, even by the CFTC, who have since charged a trader with spoofing leading to the crash [5].
A number of other causes have been suggested, with a particularly interesting suggestion being emergent behaviour between high-frequency traders known as “hot-potato” trading. Though not suggested as the root cause “hot-potato” trading was mentioned by SEC and CFTC to have occurred during the time of the crash. The idea that emergent behaviour could lead to a crash is not novel, with emergence causing a number of phase transition within physical systems, and has been suggested to be able to move a market between a stable and unstable phase before [6].
The high-frequency traders being discussed here are algorithmic traders who act as market makers, both buying and selling, trading on the nanosecond scale. These traders can under certain conditions end up passing inventory in a cyclic pattern between each other, this is “hot-potato” trading [2].
Regulators struggle to find effective methods to reduce the likelihood of crash due to emergent behaviour. There is a lack of research on what directives could combat emergence induced flash crashes, partly because of the lack of agreement on exactly how these crashes are caused and if emergence can create them. Though research in this field is minimal, it has been shown that flash crashes can occur due to “hot-potato” trading [7, 8][3]. These studies took a bottom up approach by modelling the system as an agent-based model, and created a flash crash from “hot-potato” trading which was initiated by information delays.
Though these studies have managed to create a flash crash they use agent-based model methods, which have a number of draw backs leading to them not being commonly accepted by regulators[4], this makes them difficult to use in policy advisement.
This paper seeks to create a solution to this issue by constructing an automated transformation between a set of recurrence relations and an equivalent agent-based model. This solution will take a step-by-step approach, using a series of small correctness preserving transformations which when combined will transform between these two modelling techniques. Recurrence relations are more widely accepted by regulators and hence a should increase the acceptance of the related agent-based model, allowing this technique to be used to advice policies[5].
This method of transformation has a number of challenges associated with it:

- Recurrence relations and agent-based models are opposing models, with the former being top down and the latter being bottom up. Creating a conversion between entirely different models requires the fundamental mind set to be transformed.
- It is not clear if a conversion process can even be broken down into a number of small steps. A larger step maybe be needed to move between to two stages in the transformation. Though larger

[1]This process is explained in more detail in Section 2.3.1.
[2]Both high-frequency traders and “hot-potato” trading are discussed in more detail in Section 2.3.1.
[3]Study [8] will be discussed in greater detail in section 2.3.
[4]These draw backs will be discussed in more detail in Section 3.1.
[5]This rational will be discussed further in Section 3.

steps will on the whole be easier to create proving their equivalence is much more challenging then smaller steps.

This paper is organised as follows. First an agent-based model used for modelling the financial market is described, followed by a analyses of this model and how recurrence relations can be used to combat its down sides. Section 4 details a bespoke recurrence relation language for describing a system, and Section 5 describes the step-by-step processes in which this language is transformed. Section 6 concludes the paper, detailing further work to be undertaken.

2 Background

This section will provide a background for this paper, detailing the emergent behaviour that is of interest and the methods used to investigate this behaviour.

2.1 Emergent Behaviour

Emergent behaviour is a term used to describe macro-behaviour of a system that is not obvious from analysis of the micro-behaviour of the system, more formally this is behaviour that can not be predicted through analysis of any single component of a system [9].
A misunderstanding of emergence can lead to the fallacy of division, this is that a property of the system as a whole most also be a property of an individual component of the system; water for example has a number of properties including being able to be cooled down to become ice and heated to become steam, saying the same must also be true of a molecule of water however is incorrect. This concept continues into economics, being called the fallacy of composition, where what is true for the whole economy my not hold for an individual and vice versa [10].
A simple way to demonstrate emergence is in the Game of Life [11], which is an example of cellular automaton; this game takes place on an infinite two-dimensional grid in which cells can either be 'alive', coloured for example green, or 'dead', a different colour usually black. Wether a cell is 'alive' or 'dead' is based on a set of simple rules:

1. 'Alive' cells will transition to be 'dead' cells in the next time step if they have few than two 'alive' neighbours.

2. 'Alive' cells with two or three 'alive' neighbours remain 'alive' at the next time step.

3. 'Alive' cells will transition to be 'dead' cells in the next time step if they have more than three 'alive' neighbours.

4. 'Dead' cells with exactly three 'alive' neighbours will transition to 'alive' at the next time step.

With this simple set up very complex patterns evolving through time can be created, these patterns can be seen as emergence, with an individual cell not being able to encapsulate this behaviour. Natural phenomena similar to this is the formation of symmetries and patterns within snowflakes.
Emergent behaviour can be seen occurring naturally in many other cases, with physics offering a number of well explored examples. For instance the n-body problem [12], this historically is explained as n planets interacting in such a way as to produce complex behaviour, despite each individual body following Newtonian laws. An interesting aspect of the n-body problem is that it can be reduced down to three bodies and still exhibit complex emergent behaviour. This example shows that a system need not be overly complex or large to display emergent behaviour, and that by showing the existence of emergence in a simplistic system one can infer its presence in more complex versions of that system.

2.1.1 Interaction Dynamics

The emergent behaviour that is of interest to this paper is caused by interaction dynamics, this is the communication between different elements of a system. For instance in the above example of an n-body system, communication takes the form of gravitational interactions. Therefore the emergent behaviour of the system and the reason why each body does not have zero acceleration, is do to the gravitational interactions between the different bodies. Gravity acts as a messenger in this system, telling each body about the mass and momentum of the others, and hence causing the other bodies to react, removing this messenger also removes the emergence and all the bodies would continue with constant velocities unless they physically collide [13].
Though this example deals with a very physical system, emergence may be created due to any form of communication between entities. An example of this is emergent behaviour in social systems where individuals can communicate and observe each other, but do not physically effect one another, this can lead to the forming or break down of social cooperation [14].

2.1.2 Feedback Loops

Emergent behaviour from interaction dynamics can take a number of forms, with a prominent type being feedback loops.
Feedback loops are where the input information to an entity is in some way dependent on the output information of that same entity, normally from a previous moment in time. In their simplest form this can just be a single entity supplying an input to its self, shown in Fig. 1. For example if in Fig. 1, $E1$ is a function that creates an output that is twice its input, given the initial input 1 this would create the series of inputs: $1, 2, 4, 8, ...$

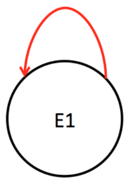

Figure 1: *Simple feedback loop with an entity supplying its input from its output.*

Feedback loops can be encompass multiple entities, in Fig. 2 a feedback loop is shown that encloses two different entities. A output from $E2$ is passed to $E3$ which in turn creates a new input for $E2$, though $E2$ input is not directly its own output, it does depend upon it.

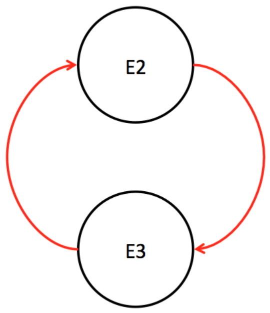

Figure 2: *Feedback loop between two entities, with each output being transformed by the other entity before becoming an input.*

A loop such as the one in Fig. 2, is only considered a feedback loop if information is passed through out the entire loop. If $E3$ produced a constant output, or an output that does not depend upon its input from $E2$, then this would not be a feedback loop as the new input to $E2$ does not depend on its output.

These example are very simple, feedback loops can be much more complex, encompassing any number of entities, each of whom can have very complex algorithms for transforming their inputs. Feedback loops can operate across time, meaning that an event in the past can eventually feedback to a present decision. For a feedback loop containing a large number of entities the time scale on which the feedback occurs can be come significantly large.

Though feedback loops are often assumed to be a negative property, some can be stabilising due to a benign effect.

Feedback loops can be present in a system in two ways, either they can be a constant fixture, called a static feedback loop, or they can form and change, called a dynamic feedback loop. A static feedback loop is present in the system from the start whether this is intentional and known, or unintentional and unknown to the members of the system. Dynamic feedback loops may not be present at the start and can form and change over time, with new entities joining or leaving them, allowing them to increase or decrease in size or effect, to split or merge, or to disappear.

Due to the potential complexity of feedback loops both in construction and in time, they can be difficult to detect, therefore methods are usually used to expose them. For static loops, forms of static analysis can be used such as, analysing initial setup, this is possible since the loops do not change through out time. Dynamic loops can be much harder to observe and analyses, an important aspect to detecting these loops is the interactions, messages sent between different entities within the system. Since the loops can evolve over time being able to track and analyse these messages over a time series is vitally important for the analysis of these loops, this time dependent analyse is called dynamic analyse.

2.1.3 Emergence in Financial Markets

Though emergence is prevalent in many disciplines, this paper is focussed on emergence in financial markets. The financial markets can be thought of as a large complex interacting system, which is the ideal environment for emergent behaviour. Events such as the formation of patterns, bubbles and crashes can all be seen as emergent behaviour caused by a variety of feedback loops [15]. This is similar to the complex interaction between short range and long range feedback loops in chemical reactions leading to pattern formation [16].

2.2 Agent-Based Models

There are a number of different techniques to model emergent behaviour in complex systems, one popular method is agent-based modelling. Agent-based modelling can be considered more of a mind set then a rigid methodology, this involves describing the system in question in terms of its components and then allowing these to interact. Agent-based models allow a system to be described naturally and are hence the canonical approach to modelling emergent phenomena. This method is a bottom up approach, allowing for each component of the system, agent of the model, to be created to a relevant degree of abstraction [17].
Agent-based models have been used to model a wide range of emergent behaviour including in the financial markets, examples of this are, noise traders [18], herding among traders [19], and fundamentalists [20].

2.3 InterDyne

InterDyne is bespoke simulator created by Clack and his research team at UCL [21], it is a general-purpose simulator for exploring emergent behaviour and interaction dynamics within complex systems. InterDyne design is that of an agent-based model interacting via a harness. This creates a structure of individual autonomous agents who interact through messages sent to one another.
Similar to other agent-based models InterDyne operates in discrete-time rather then continuous time. These quantised time chunks which move the simulation forward can be left with out proper definition, simply having operations defined in a number of time steps, or they can be equated to a real time usually with the smallest time gap needed be a single time step and then all other timings being integer multiples of this. This discrete time is most important to message passing, meaning messages between agents are only sent on a integer time step.
Massages in InterDyne are just small packets of data, such as a series of numbers. Agents can only communicate via these messages, meaning that any emergent behaviour observed, that is not directly due to one agent, must be caused by linking of agents mediated by these messages. An agent can send private messages that are only received by a single other agents, one-to-one messages, or can send broadcast messages received by a number of agents, one-to-many messages. To facilitate this a communication topology can be made for InterDyne, this is done in the form of a directed graph determining which agents can communicate with each other. Due to the directional nature of these messages this topology could allow an agent to send messages to another but not be able to receive messages from that same agent. Messages have a defined order to them, an agent will, unless otherwise instructed, always process messages in the order in which they arrive. To change the order in which messages arrive delays can be added to communication paths between agents, this can be a static delay which always applies to messages sent from one agent to another, meaning this will arrive a set number of time steps later. Or a more complex dynamic delay, which is achieved by using another agent to mediate the passing of these messages delaying by an amount decided on in some internal logic. All messages in InterDyne are passed through a harness, this does not alter the messages or delay them[6], but does store the messages and their order which can be used in post analyse.
Each of the agents within an InterDyne simulation can be completely unique and modelled to different levels of complexity, as is the case with most agent-based models, allowing system components to be created to the level needed for the required experiment. As a whole InterDyne simulations are deterministic, repeated experiments will return identical results. However non-determinism can be added via the agents, making some part of an agent stochastic will lead to repeated experiments on the whole returning different results. A pseudo-random element can also be added by instructing InterDyne to randomly sort the message order for any agent receiving multiple messages in one time step. This is only pseudo-random as, as long as the same seed is used each run of the simulation will order the rearranged messages in the same way.

[6]Unless instructed to using the static delay.

InterDyne is created to be particularly amenable to dynamic analyses of its simulations, this is achieved in part by all messages being sent via the harness allowing them to be stored in order.

2.3.1 Applicability to Finance

Though InterDyne is a general purpose simulator, its main use thus far has been the exploration of financial markets. In particular InterDyne has been used to explore "Flash Crash" of 2010, during which market prices and rational valuations became disconnected, with some stocks trading as low as a penny per share, this lead to frenzied trading and irrational prices which spread between markets causing a massive price crash [1]. This event lasted around 36 minutes and has been described as"one of the most turbulent periods in their history" for the US financial markets [3].
The hypotheses for this crash which InterDyne exists to investigate, is that this crash is an emergent phenomenon caused by the interaction between High Frequency Traders (HFTs) within the market.
HFTs are a subset of algorithmic traders who normally participate in the market as arbitrageurs or market makers, they invest in ultra-high speed technology allowing them to detect, analyses and react to market condition in nanoseconds [22]. This means HFTs can trade huge quantities of assets in very short time frames, with some estimates stating that 10-40% of all trades where initiated by them during 2016 [23].
The type of interaction between these traders suggested to have caused the crash is "Hot Potato" trading, this is when inventory imbalance is repeatedly passed between HFTs market makers. A market maker is a trader who is required to have both a bid and a ask on the order book at all times, this means in theory that they are constantly buying and selling, a high frequency market maker as expected should be buying and selling very very often. Market makers make a profit from the spread and not long positions, hence they want to keep inventories low to avoid the market moving against them. To achieve this market makers have strict inventory limits that if they pass will cause them to go into what is know as a "panic state", during this state the trader will sell of an amount of its inventory to return back into its normal trading region. This inventory now solid by the market maker can be bought by another market maker causing them to in turn go into "panic" and sell, this process is "Hot Potato" trading and can in theory continue indefinitely [24].
"Hot Potato" trading was observed in the market during the "Flash Crash" [1], this is thought to have been caused by a combination of an initial large sell order by a mutual fund and delays in communication between HFTs maker makers and the exchange on which they were operating on, causing them to buy more inventory then they wanted and go into a "panic state" and hence a "Hot Potato" feedback loop. This section explains in more detail this hypotheses and how InterDyne is set up to investigate it.

2.3.1.1 Deterministic

The deterministic nature of InterDyne allows for experiments to be run multiple times with the same result always returned, this allows for changes to the experiment setup to be investigated. For example changing the number of traders in the market and comparing this to a previous run allows for an investigation into how many traders are required for emergent behaviour to be observed.
This becomes particularly interesting when comparing the interactions between market makers to that of the n-body problem, like with this problem one could expect emergent behaviour might occur to some extent in a large group of market makers, however the question of wether the emergence persists in a comparable market to the three-body problem and how this compares to a larger market can be investigated.

2.3.1.2 Message Delays

Allowing the delaying of messages is intrinsically important to the investigation of the hypotheses since the existence of delays is proposed as one of the main aspects in the "Hot Potato" trading that occurred. Delays exist between all aspects of the market which can account for the processing time of the different

elements and the transmission time of messages between them. Some of these delays will be static but it has been preposed that the delays related to the exchange actually increased during the crashing, further worsening the situation [1].
Static delays in built in InterDyne can be used to investigate the crash to see if it can occur without the need for dynamically varying delays. Dynamic delays created with agents can then be used to further investigate the events that occurred during the crash, allowing a situation to be set up where as more messages are sent to and from the exchange its delays increase. Asymmetric delays can be specified between two agents allowing further investigation into the environment in which a crash is mostly likely to occur and how delays could be altered to reduce this out come.

2.3.1.3 Message Passing

To observe the decided system level behaviour, a flash crash, two different methods can be use; the behaviour can be encoded into the program forcing it occur at a system level, or the system can be setup to allow the behaviour to emerge at the system level. For a true understanding of emegernt behaviour the latter approach is more relevant, this requires the different agents within the simulation to be able to communicate directly with one another. In modelling the financial market these communications are in the form of messages sent between different entities, for example a trader could send a message to an exchange detailing a limit order they wish to issue and an exchange could send back a message containing a confirmation of this order.
These messages allow interaction dynamics to occur within the simulation and hence for emergent behaviour derived from interaction dynamics to naturally present within the system.

2.3.1.4 Storing Messages

Due to the nature of emegernt behaviour being usually unexpected, it can be very difficult to to deduce what low level structures and operations gave rise to this system level phenomenon. This is especially true when modelling a financial system that has a large number of interacting agents all sending and receiving messages, some of which can be delayed changing their expected order of arrival. The delays in the system have been suggested to have influenced the emergence of behaviour within the system, hence in investigating these systems it is important to take into account not only message counter-parties but also message timings. InterDyne facilitates post simulation analyse into this by being able to produce a trace for all time steps of the full information of; messages sent by any agent, messages received by any agent and messages being delayed before being delayed to an agent.

2.3.1.5 Discrete Time

Though it is easy to assume that the financial markets operate in continuous time this is in fact not always the case. For electronic markets that trade through an exchange their time is set by the exchange, orders are not processed and messages are not sent back till the exchange decides to do so. These electronic exchanges them selfs operate in discrete time, this is unavoidable and is a product of the systems being run on computers, a computer runs based on an internal clock that ticks in discrete intervals based on a change in a square-wave oscillating voltage. This change in voltage is so fast that to a human it seems continuous, however HFTs operate themselves at such high speeds that the system clock time gaps are comparable and hence need to be considered. Therefore models simulating HFTs interactions to this detail must take account of this discrete time, hence InterDynes discrete time nature is a good match to model HFTs interactions.

2.3.1.6 Message Ordering

The order in which messages are processed can be very important, for an exchange, for example, it can change whose limit order has priority at a given price and whose market order executes the lowest prices. Changing these factors can make or break feedback loops within the system, meaning if message ordering is not properly dealt with the correct emegernt behaviour may not be observed. Hence InterDyne stores messages in the order they are received by an agent, taking into account delays to the messages. This however can not be done when multiple messages are received at the same time step, due to the nature of discrete time there is no way for the agents to know which message arrived first, therefore two options are presented by InterDyne; messages are ordered according to their agent identifier or messages are randomised and executed in the emerging order. This randomisation is handled in the same manner every run of the simulation [7] hence resulting in the same order and therefore not effecting the deterministic nature of the experiment.

2.3.1.7 Agents

A benefit of agent based models of other alternatives is the ability to encode agents as unique traders instead of having to model the general behaviour of a number of traders. This allows for unique behaviour of varying complexity to be given to different agents, facilitating experimentation with different trading strategies, allowing different questions to be investigated, such as, does a trading strategy need to be complex for emergence to be observed?

2.3.2 InterDyne Detail Operation

InterDyne is an agent-based model simulator written in the Haskell language. An InterDyne simulations structure, shown in Fig. 3, consistent of a number of independent agents, sending messages to a "Simulation Harness", this harness then (i) sends these messages to the relevant counter party or parties [8], (ii) saves these messages to a trace file.

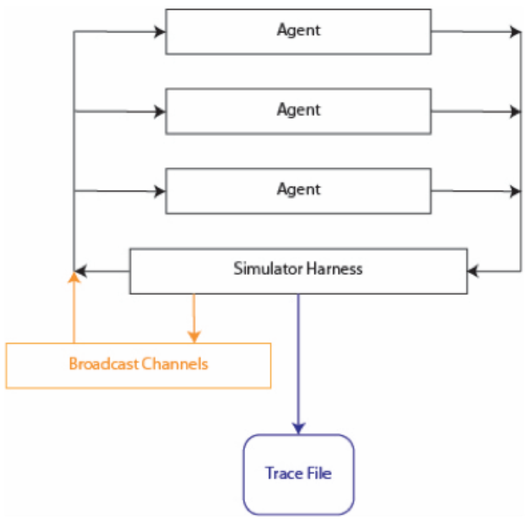

Figure 3: *Structure of an InterDyne simulation containing three agents.*

By set up each agent is required to both consume and produce a message at every time step, this is done by sending each agent a potentially-infinite list of messages and requiring it to create a potentially-infinite list of messages. The inbound list contains every message the agent will receive through out the entire simulation, ordered by time, and the outbound list contains every message the

[7] This can be changed using a new seed if so desired

[8] In the case of broadcast messages

agent will ever send, order by time. This is possible due to lazy evaluation of the language that InterDyne is written in, This means that only messages being used are evaluated. Since no agent should attempted to use a message from the future [9], only messages that have been created so far are read.
Since the set up of these list requires an element for every time step and an agent my not need to send a message during a time step, empty messages can be sent, this will result in an agent either receiving or sending an empty list. It may be the case in a simulation that an agent could be generating empty messages by mistake, to differentiate between a message empty by mistake or by choice, a message can be created containing "Hiaton" demonstrating that it is empty by choice.
A InterDyne simulation once created can be run by executing the "sim" function with relevant arguments, this function has a type,

```haskell
sim :: Int -> [Arg_t] ->  [(Agent_t, [Int])] -> IO()
```

corresponding to, number of time steps, list of "runtime arguments", and a two tuple with the agents function and broadcast channels to which it is subscribed.
Agents within a simulation are defined by ID numbers, with the first agent having ID=1, followed by the second agent with ID=2, ID=0 is used to represent the "simulator harness". These ID numbers are used to specify which agents messages are coming from and going to. This however can be difficult to keep track of within large experiments for the experimenter, therefore InterDyne allows for agents to be referred to by a name. This example shows InterDyne being run with the use of names for identification:

```haskell
import Simulations.RuntimeArgFunctions as RTAFuns

exampleExperiment :: IO ()
exampleExperiment
  = do
    sim 60 myargs (map snd myagents)
    where
    myargs = [ convert ]
    myagents = [ ("Trader",  (traderWrapper, [1])),
                 ("Broker",  (brokerWrapper, [3])),
                 ("Exchange",(exchangeWrapper, [2,3]))
               ]
    convert = RTAFuns.generateAgentBimapArg myagents
```

This simulation will run for 60 time steps, with three agents subscribed to a number of broadcast channels, for example the third agent is subscribed to channel 2 and 3. There is a single run time argument convert, that can convert an ID to a name and a name to an ID. This is a simplistic call of the function sim, but shows how an experiment run could be setup and executed.
In the example one can notice that the agents called all were to referred to as wrappers, this is because agents in InterDyne are typically, though not necessarily, written in two sections; "wrapper" and "logic" functions. The "wrapper" function, called in the example, can be thought of as the true agent, it handles message receiving and sending, as well as updating the local state of the function, hence this function is the one that interacts with the other agents and the harness. The "logic" function rests inside the "wrapper" function and computes the messages to be sent, hence this contains the functionality of the agent. An example of a "wrapper" function containing a "logic" function is shown here:

```haskell
wrapper_i :: Agent_t
wrapper_i st args ((t, msgs, bcasts) : rest) myid
     =  [m]  : (wrapper_i st args rest myid)
        where
        m = logic_i (t, msgs, bcasts)
```

This agent will output a list containing message m, which is the output of the "logic" function for that given time period based on the received messages. The wrapper is a recursive function with each recursion being a new time step. It takes as inputs, st the local stat variable, args the run time

[9]If they do an error will be thrown.

arguments, a list of all messages to it, and its ID. The list of messages has a three tuple for each time, containing the current time, list of all messages sent to the agent directly, and a list of all messages sent to the agent from broadcast channels.
InterDyne supports a wide range of messages in an array of complexities, to allow for a variety of interactions to take place. Messages, which are either one-to-one or broadcast, must all contain a pair of integers which represent the sending agents ID and the receiving agent or broadcast channels ID, the rest of the message depends on the type of message being sent. For example one could send a one-to-one message containing a list of (key, value) pairs,

```
Message (Int, Int) [Arg_t]
```

or a message carrying a string,

```
Datamessage (Int,Int) [Char]
```

Broadcast messages content is defined by a broad type, which defined further within its type definition,

```
Broadcastmessage (Int, Int) Broadcast_t
```

As previously stated ID=0 represents the harness, messages sent to this ID are printed to an output file, output is also achieved in InterDyne by storing all messages of type Datamessage, which are saved to a different file then those sent to ID=0.
As mentioned earlier InterDyne allows for both a topology of allowed interactions and delays along interaction paths to be defined. This can be done by passing two runtime arguments to the “sim” function [10], (i) a function that when given two agent IDs will return a delay for the interaction between them in time steps, and (ii) the maximum delay that returned by this function, i.e. the maximum delay present in the system. The previous example expanded with delays is shown here:

```
import Simulations.RuntimeArgFunctions as RTAFuns

exampleExperiment :: IO ()
exampleExperiment
  = do
    sim 60 myargs (map snd myagents)
    where
    myargs   = [ convert,
                 (Arg (Str "maxDelay", maxDelay)),
                 (DelayArg (Str "DelayArg", delay))
               ]
    myagents = [ ("Trader",   (traderWrapper, [1])),
                 ("Broker",   (brokerWrapper, [3])),
                 ("Exchange",(exchangeWrapper, [2,3]))
               ]
    convert = RTAFuns.generateAgentBimapArg myagents
    delay 1 2 = 1
    delay 1 x = error "illegal interaction"
    delay 2 x = 2
    delay 3 2 = 3
    delay 3 x = error "illegal interaction"
    maxDelay = fromIntegral 3
```

Once delays have been specified in this manner all messages, one-to-one and broadcast, are delayed by the stated amounts between the defined agents. Dynamic delays are not shown here, but are achieved with use of another agent acting as a go between [21].

3 Description and Analysis of the Problem

Though InterDyne is a functional simulation platform and has been used to run a number of experiments, most notably in Ref. [8] where a model was presented showing a flash crash caused by interaction

[10]Both the arguments must be given.

dynamics, it possesses a concern when discussing these results, especially to non-experts in computational modelling.
This section will discuss the main draw backs of InterDyne and a method for countering them.

3.1 Why InterDyne is Not Enough

As already mentioned InterDyne is an agent-based model and hence possesses the same limitations as do other models of this type. A number of these limitations are of particular concern to economists, making agent-based models and their results not commonly accepted by them [25, 26]. The most significant limitations with this models are:

1. Producing high-level behaviour or emergent behaviour from a set of base rules, only shows that those create it and does not show that those are the only rules that could exhibit this behaviour. This is often referred to as the inverse function problem.

2. Tracking the affect of an input parameter on the output of the agent-based model can be very difficult, and parameter-estimation may be done in a fashion that will not represent all of the possible outcomes of the model.

3. Despite each agent within a simulation being fully specified the model as a whole will lack a formal definition.

The first limitation is shared by man modelling techniques and is a by product of studying emergent behaviour, more then one scenario may lead to the same emergence. This limitation should be considered more as a consideration when analysing results from experimentation, it is generally speaking useful to find a set of conditions that lead to an emergent behaviour, but should be noted that it can not be said that it is the only one without further research.
The second limitation can be reduced by analysis of the sensitivity of the outcomes to parameter selections, this is shown in Ref. [25].
The third limitation is of particular interest, though the other limitations can lead to questions of the validity of findings from agent-based models, the understanding of the actual model and experiment being undertaken is hindered by this limitation. A resolution to this problem was put forward by Ref. [25], in which a formal definition of the specification of the an agent-based model was given in terms of a set of recurrence relations. This solution not only provides a formal definition of an agent-based model but also does so in a way which is relatable to non-programmers, such as economists.

3.2 Recurrences Relations

Recurrence relations connect a discrete set of elements in sequence, these elements are normally either numbers or functions, they can be used to define these sequences of produce the elements in them. They can be seen as equations that give the next term in a sequence based on the previous term or terms, hence defining said sequence. Recurrence relations are often used to define coefficients in series expansions, moments of weight functions, and members of families of special functions [27].
The most simplistic form of a recurrence relation is one where the next term depends only on the immediately preceding term. If the n^{th} element in the sequence is defined as x_n, then this recurrence relation can be written as,

$$x_{n+1} = f(x_n), \tag{1}$$

where $f()$ is a function that calculates the next term based on the previous one. A recurrence relation does not just have to depend on its immediate previous term and can depend on any number of terms further back in the sequences, for example a recurrence relation depending on terms from two and three steps before can be written as,

$$x_{n+1} = f(x_{n-1}, x_{n-2}), \tag{2}$$

with $f()$ now taking two inputs to produce the new term in the sequence [28].
Recurrence relations can also be used to define a sequence through time, in the simplest case the enumerate n, can be set to represent time t, this is applicable to discrete time as it requires set steps between the different times. Just as in the previous examples, the simplest recurrence relation is,

$$x_{t+1} = f(x_t), \tag{3}$$

where x_t is the term at time t and $f()$ gives the term at $t+1$ based on the term at t. Again this can be expanded to include terms from a number of previous time steps, allowing the memory of the sequence to be shown.
A recurrences relation for defining a sequences may as well as depending upon previous terms, also depend upon some parameter, α, this would give, in its simplest case,

$$x_{n+1} = f(x_n; \alpha). \tag{4}$$

The next term in the sequence may not only depend on previous terms within its own sequence and paramters, it can also be conditional on another sequence. For example one sequence through out time, x, may depend on another sequence through out time, y, a simple recurrence relation for this could be,

$$x_{t+1} = f(x_t, y_t), \tag{5}$$

with the sequence for y possibly depending on its own recurrence relation. The sequence for x may not even directly depend on its own sequence and could solely depend on y,

$$x_{t+1} = f(y_t). \tag{6}$$

Though it could also indirectly depend on its self, if y was defined by a recurrence relation depending on x, such as,

$$y_{t+1} = f(x_t). \tag{7}$$

These cross sequence associations allow for complex interactions to be represented as time series defined by recurrence relations, this is a method which can be applied to agent-based modelling.
Reference [25] showed how an agent-based model looking at the microstructure of the financial markets, can be exhibited in the form of recurrence relations. In their approach each agent with in an n agent model is considered to be well described by a state variable, $x_{i,t}$. The state variable describes the agent at each time step, where i is the identity of the agent ($i \in 1, 2, ..., n$), and t is the time step at which the agent is being described, i.e. $x_{1,5}$ describes the first agent at the fifth time step. These state variables are defined by the recurrence relation,

$$x_{i,t+1} = f_i(x_{i,t}, x_{-i,t}; \alpha_i), \tag{8}$$

where $f_i()$ is a function unique to agent i, that can take the state of the agent at the previous time step, $x_{i,t}$, the state of any other agent at the previous time step, $x_{-i,t}$[11], and a bespoke parameter α_i.
This method works by describing each agent as a sequence of state variables, these sequences are then interlinked by having dependencies to each other, allowing both the agents and their interactions to be represented. A simple example of this could be two agents, x_1 and x_2, that are coupled to each other and only depend on the others value at the previous time step and a parameter, represented by the recurrence relations,

$$x_{1,t+1} = f_1(x_{2,t}; \alpha_1), \tag{9}$$

$$x_{2,t+1} = f_2(x_{1,t}; \alpha_2). \tag{10}$$

[11]The $-i$ is used to refer to all other agents in the system. So if $i = 1$, $-i$ refers to agents $2, 3, 4, ..., n$, for an n agent system.

This formulation only shows relationships to the previous time steps, however information delays can be added to the system by allowing the recurrence relations to have a longer memory. A recurrence relation that uses a state variable from three time steps ago, could be considered to have a time delay of three time steps in receiving this information. The generalisation of the model needed to allow for this, is a recurrence relation of the form,

$$x_{i,t+1} = f_i(x_{i,t}, x_{i,-t}x_{-i,t}, x_{-i,-t}; \alpha_i). \tag{11}$$

This is similar to Eq. 8 but contains two new terms $x_{i,-t}$ and $x_{-i,-t}$, these terms are used to refer to any time steps before the previous one, time t. This more general equation can now reference any state variable from any previous time, from any agent, as well as its bespoke parameter, to compute the next term in its sequence.
This models can be solved for macro-level properties by iteratively solving each state variable, $x_{i,t}$, given some initial conditions. This method of assessing the model gives it a formal definition, and one that is accessible by a larger range of experts then classic agent-based models. Recurrence relations in a number of forms are commonly used in economics and finance and hence are familiar and relatable to economists, this makes them a far more effective tool for describing models to these domain experts then agent-based models.
This iterative method for solving the equations makes any dependencies between systems apparent, as if $x_{2,t}$ is present in the definition of $x_{1,t+1}$, one can say that $x_{1,t+1}$ depends on $x_{2,t}$. This transparency of dependencies makes this formulation amenable to static analysis, allowing the recurrence relations to be investigated to return their static dependencies.
Although all recurrence relations will contain some form of recursive, since this is an intrinsic property, this will not necessarily be negative or destabilising. For example a dependency on a state at a previous time step may not be destabilising to the system, however the reverse would be a case where the state variable at $t + 1$ depended on its self, this would be a destabilising loop. Static analysis hence could be used to detect hard-wired destabilising loops, such as the one just mentioned.
Static analyse of the system can become difficult when the functions $f_i()$ contain conditional statements relating to their inputs, this can cause inputs not to be used at certain times hence meaning dependencies will not be as obvious as simply the inputs given, this is a problem of determining wether a function will necessarily evaluate all its arguments [29].
Though through static analysis recurrence relations can be used to view the history of the state variables of a single agent, however the history of messages between pairs of agents is not easily detectable with static analysis. The interaction history is found through dynamic analysis which is not well suited to recurrence relations, as they lack a clear ability to follow the passing of messages within the system.

3.3 Two Views Approach

As can be seen from the previous discussions neither agent-based models, in the form of InterDyne, or recurrence relations offer an optimal modelling technique whose results can be analysed to the level desired. With InterDyne suffering from a lack of a formal definition and difficulties for static analysis, and recurrence relations being unsuitable to preform dynamic analyses on. Looking at the benefits of the two techniques, InterDyne is well suited to dynamic analyses, and recurrence relations provided a formal definition and are appropriate for static analyses, one can see that they match each others draw backs.
Therefore a more ideal model, would be representable in both recurrence relations and as an agent-based model. This model would be suited to both static analyses, while expressed as recurrence relations, and dynamic analyses, when in the form of an agent-based model. A model that can correctly transform between a set of recurrence relations and an agent-based model, would provide the agent-based model with a formal definition in the form of the recurrence relations. This method will be referred to as the two views approach, with recurrence relations, and agent-based modelling seen as two complementary

views of the same system.
To create a model tool which will allow for the two views approach, three aspects most be covered, an agent-based model,the recurrence relations, and a convertor for transforming between them. The agent-based model as already been created in the form of InterDyne and as already discussed fulfils all the needed requirements for the systems being modelled. The latter two aspects however have not been previously produced and will be the focus of this paper.

4 Bespoke Recurrence Relation Language

Recurrence relations are used across a wide range of disciplines and as such have many different forms of notation, since these recurrence relations are going to act as an import for the convertor program, having a wide range of possibly conflicting notations is not ideal. Therefore it was decided that a bespoke notation for the recurrence relation input should be used, this language will force recurrence relations to be written in a set form for input into the convertor.
In designing this language a few main considerations had to be taken into account:

1. The language needs to be able be both understood by and written by non-computer scientists, such as economists and mathematicians.
2. The language needs to be as simple as possible, to keep it easy to formally define and learn.
3. The language still needs to contain enough functionality to fully specify the problems being modelled.

The first consideration is taken into account by choosing the language to based in a more mathematical style then one that heavily relays on a computational style.The second and third consideration go hand in hand, requiring that the functionality of the language be as simple as possible while still allowing for the problem to be expressed. As such it is pertinent to decide what functionality is required by the language, this functionality is:

- The use of basic mathematical operations.
- The use of lists.
- The ability to call the head of a list.
- The ability to call the tail of a list.
- The ability to add to the head of a list.
- The ability to declare variables.
- The ability to define a unique name for each recurrence relation.
- The ability to define a recurrence relation that uses inputs.
- The use of where blocks.
- The use of if else statements.
- The ability to specify an entire experiment and initial conditions.

It was decided that this language would be created by using an already existing coding language and reducing its functionality down to what was required. This resulted in the recurrence relation language being a simplified version of the Miranda language, Miranda was chosen as it is a functional language whose structure is similar to that of a set of mathematical equations, hence making it easier to adapt for use by non-domain experts.

4.1 Syntax/Grammer

```


>program ::= Program [definition] experiment




>definition ::= Name [char] expression | Function [char] [char] [argument] expression |
    InterVariable [char] expression |  IntFunction [char] [argument] expression



>argument::= Argument expression




>experiment::= Experiment [globalvariables] experimentbody experimentrun



>globalvariables::= Globalvariables [char] [argument] expression



>experimentbody::= Emptybody|Expbody [argument] expression



>experimentrun::= Emptyrun|Exprun expression




>expression::= Emptyexpression ||primarly used to intiate loops and should not ever
    exist in the final out put
>               |Ifelse expression expression expression ||this is the if statement
    taking: if condition, true code, else code
>               |Brackets expression
>               |List [expression] ||for []
>               |Operation expression op expression
>               |Funint [char] [argument] ||internal function
>               |Funext [char] [char] [argument] ||externally defined functions
>               |Varint [char]
>               |Varex [char] [argument]
>               |Constantvar [char]
>               |Specialfunc specfunc expression
>               |Number num
>               |Where expression [definition]
>               |Mainfunc [argument] ||this is the actual program itself


>op::= Plus
>       |Minus
>       |Multiply
>       |Divide
>       |Lessthan
>       |Greaterthan
>       |Equals
>       |Notequals
>       |Lessequ
>       |Greaterequ
>       |Listadd
>       |Bang



>specfunc::= Listhead|Listtail
```

Figure 4: *Simple example of recurrence relations written in custom defined language*

4.2 Rules of Use

4.3 Examples of Use

Here the syntax for the above listed functionality will be explained, in ordered of listing.
Basic mathematical operations are allowed, plus $+$, minus $-$, times $*$, divide $\backslash$. The ability to make comments that will not be read is also allowed, a comment is started with "\\" and then the rest of the line is ignored.
The main method for storing and transferring information between recurrence relations will be in the form of lists, lists follow the same rules stipulated by the Miranda language. A list can be created by putting the elements in side "[]" and separating them with a comma, allowing lists to be written in forms similar to:

```
[1,2,3,4]

[[1,2,3],[4,5,6],[7,8,9]]

[[1,2,3],[4,5]]
```

However all elements in these lists must be of the same type.
The head of a list will be extractable using the command *hd*, this will return the first element of a list, for example *hd* called on the list $[1, 2, 3]$ would return 1. Similarly the tail of a list can be called using *tl*, for example calling *tl* on $[1, 2, 3]$ would return 3.
Elements can be added to a list using the command :, this can be applied in the form $4 : [1, 2, 3] = [4, 1, 2, 3]$.
Variables can be declared in the language, these are names that are attached to fixed values, and can be written in the form:

```
c_name = value
```

A variable name must state with the identifier *c_*, this indicates that the object is a constant, this underscore is then followed by the unique name of the variable, an equals, and then the value to which the variable refers.
A recurrence relation is defined in a similar way to a variable, but can take inputs and represents a function:

```
agent_name input = function
```

The name must state with a identifier of what agent, *agent_*, this recurrence relation belongs to, in this language all recurrence relations belong to an agent. A group of recurrence relations that belong to an agent will interact with each other but only have a single recurrence relation that interacts with a different agent, group of recurrence relations. After the agent identifier, the unique name of that recurrence relation within the agent is given, recurrence relations belonging to different agents may have the same name but with an agent the name must be unique. Then the inputs for this recurrence relation are given, there may be one or more inputs[12] separated with a space between them. Then after the equals the function for the recurrence relation is give, this function may call on other recurrence relations.
A where block is used to define functions and variables unique to a certain recurrence relation, one can be written as follows:

```
agent_name input = function
                   where {
                       function
                       variable
                       etc
                   }
```

[12]A recurrence relation with no inputs is just a variable.

Inside a *where* block which is identified by any number of functions and variables can be defined, with each new one appearing on a new line.
This language facilitates the use of if else statements, this allows for different behaviour based on the inputs to the recurrence relation, this can be written in the form:

```
agent_name input = myif (condition) then (function/value)
                   else
                   (function/value)
```

Using the command *myif* declares that an if else statement is being used, then within brackets the condition for the if statement is given, this can use $<, >, \leq, \geq, =$, etc. If this statement is true then the *then* part of the statement is executed, if this is false the *else* part is used.
An experiment can be defined with initial conditions in the following way:

```
run_main = []//This is where the different experiment runs go

//This is where initial conditions go

main runnumber =

//This is where the experiment goes

where
{
    //This is where the definition of each recurrence relations go
}
```

The design of this is one where the *where* block contains all the definitions for the recurrence relations in the experiment. Above this the *main* defines an experiment to run, such as a recurrence relation that you want to iterate ten times. The initial conditions are defined above this, these can have multiple different values depending on an input parameter. At the top *run_main* gives a list of the *main* with different values of *runnumber* which corresponds to different values of the initial conditions.

4.3.1 Simple Example

```
run_main = [main 0, main 1]



var_init runnumber = myif (runnumber=0) then (10) else (20)




main runnumber =

[i_f1 1, i_f1 2, i_f1 3]

where
{



i_f1 t = (i_f2 t) + (j_f1 t 1)

i_f2 t = myif (t<12) then (10) else (20)




j_f1 t a = myif (a<2) then (9) else (j_f2 t)

j_f2 t = fun t 3
         where{
         fun t a = (k_f1 a)+t}




k_f1 t = t + (var_init runnumber)


}
```

tuples exist but can not be used by the user

5 Recurrence Relation to InterDyne Converter

This section details the design of the transformation from a set of recurrence relations written in the previously discussed custom language into an InterDyne simulation. This transformation processes as been split into a number of smaller steps, with each step taking the expression of the system given by the previous step and modifying it to be closer to the InterDyne formalisation.
The transformation has been split into small steps to avoid any large jumps in logic between different representations. Jumps in logic can make it difficult to follow the transformation and more challenging to prove the correctness of the transformation.
The aim of this section is to show the design of each of these transformative steps, and not to implement them into code or prove their correctness, however in some case the code may also be shown.
To assist in the explanation of the transformation a simple example will be used through out this section Recurrence Relation Example 1 (RRE1), this example is shown in Fig. 5.

```
run_main = [main 0]

srt_i_f1 0 = 0

srt_j_f1 0 = 0

srt_j_f1 2 = 0

main runnumber =

[i_f1 3]

where
{
i_f1 t = i_f1 (t-1) + j_f1 (t-1)

j_f1 t = j_f2 t + j_f1 (t-1)

j_f2 t = myif (t<4) then (10) else (0)
}
```

Figure 5: *Simple example of recurrence relations written in custom defined language.*

5.1 Recurrence Relation Parser

The recurrence relations shown in Fig. 5 are text, and hence the first step in transforming them is to give them a representation within code. This is done by a parser in two parts, lexical analysis and syntax analysis.

Lexical analysis takes in the text file as a list of characters and converts this to a list of tokens representing items, such as function names.
Syntax analysis takes this list of tokens and stores it in a parse tree that enforces structure. This parse tree can be seen in Fig. 4.

5.2 Infinite List Outputs

A large difference between recurrence relations and InterDyne, is that in InterDyne communication is done using list, and not function calls. These lists are infinite, with a each element representing a value at a different time step. Though these lists are infinite in practice they are only as long as the highest time reached, as values for any time after this are not needed and hence will not be calculated. To make the recurrence relations have a similar approach to this a wrapper function for each relation needs to be made, this function will produce a infinite list containing the value of the relation at each time step, this wrapper can be seen in Fig.6.

```
agent_function args = _createlist 0 args
                        where
                        _createlist t args = (_sublogic t args):_createlist (t+1)
                        _sublogic   t args = recurrance_relation
```

Figure 6: *Simple example of recurrence relations written in custom defined language.*

To access a value for a particular time of a relation, the bang operator, !, now has to be used. This operator applied to a list will return a specified element of the list, for example in $List!n$ the n^{th} element will be returned. Therefore the experiment will now be written in the form shown in Fig.7.

```
run_main = [main 0]

srt_i_f1 0 = 0

srt_j_f1 0 = 0

srt_j_f1 2 = 0

main runnumber =

[i_f1!3]

where
{
i_f1 = _createlist 0
        where
        _createlist t = (_sublogic t):_createlist (t+1)
        _sublogic   t = (i_f1!(t-1)) + (j_f1!(t-1))

j_f1 = _createlist 0
        where
        _createlist t = (_sublogic t):_createlist (t+1)
        _sublogic   t = (j_f2!t) + (j_f1!(t-1))

j_f2 = _createlist 0
        where
        _createlist t = (_sublogic t):_createlist (t+1)
        _sublogic   t = myif (t<4) then (10) else (0)
}
```

Figure 7: *Recurrence relations with with infinite list outputs.*

5.3 Wrapper Function

```
run_main = [main 0]

srt_i_f1 0 = 0

srt_j_f1 0 = 0

srt_j_f1 2 = 0

main runnumber =

[(i_wrapper!3)!0]

where
{
i_wrapper = _createlistw 0
            where
            _createlistw t = [_f1!t]:_createlistw (t+1)

            _f1 = _createlist 0
                  where
                  _createlist t = (_sublogic t):_createlist (t+1)
                  _sublogic   t = (i_f1!(t-1)) + ((j_wrapper!(t-1))!0)

j_wrapper = _createlistw 0
            where
            _createlistw t = [_f1!t, _f2!t]:_createlistw (t+1)

            _f1 = _createlist 0
                  where
                  _createlist t = (_sublogic t):_createlist (t+1)
                  _sublogic   t = (j_f2!t) + (j_f1!(t-1))

            _f2 = _createlist 0
                  where
                  _createlist t = (_sublogic t):_createlist (t+1)
                  _sublogic   t = myif (t<4) then (10) else (0)
}
```

Figure 8: *Recurrence relations with with wrapper functions.*

5.4 Harness

```
run_main = [main 0]

srt_i_f1 0 = 0

srt_j_f1 0 = 0

srt_j_f1 2 = 0

main runnumber =

[((sim_harness!0)!3)!0]

where
{
sim_harness = allmsg
              where
              allmsg  = _addsrtmsg restmsgs
              _addsrtmsg restmsgs = []                  ||adds startmsg to restmsgs :
    doesnt do it so have to make fun
              restmsgs = _createmsgs 1
              _createmsgs time = msg:_createmsgs (time+1)
                                 where
                                 msg = [_msgtowrap 1, _msgtowrap 2]
                                 _msgtowrap id = [(i_wrapper 1 (_createmsgs time) 1)!
    time, (j_wrapper 1 (_createmsgs) 2)!time]

i_wrapper t allmsg id = _outmsg:i_wrapper (t+1) (tl allmsg) id
                        where

                        redmsg = hd (allmsg!(id-1))

                        _outmsg = [_f1!t]

                        _f1 = _createlist 0
                              where
                              _createlist t = (_sublogic t):_createlist (t+1)
                              _sublogic   t = (i_f1!(t-1)) + ((redmsg!1)!0)

j_wrapper t allmsg id = _outmsg:j_wrapper (t+1) (tl allmsg) id
                        where

                        redmsg = hd (allmsg!(id-1))

                        _outmsg = [_f1!t, _f2!t]

                        _f1 = _createlist 0
                              where
                              _createlist t = (_sublogic t):_createlist (t+1)
                              _sublogic   t = (j_f2!t) + (j_f1!(t-1))

                        _f2 = _createlist 0
                              where
                              _createlist t = (_sublogic t):_createlist (t+1)
                              _sublogic   t = myif (t<4) then (10) else (0)
}



to look back in time you have to pass a list contaiing those values through the
    recursion of the wrapper, however wont this valiate the type defintion of the
    wrapper?

How should a fucntion access its own value at a previous time step? this should be by
    passing a list through the recursion as well?

All fucntions within a wrapper can access all other functions within the same wrapper at
     time step t, but can only access functions in other wrappers at t-1
This is needed for both functionality and makes sense when creating agents with multiple
     parts which can be seen to operate at the same time

How do I send a message to the harness at this stage sense to and from dont exist?
```

5.5 Direct Messages

5.6 Broadcast Messages

6 Validation

6.1 Type Comparison to InterDyne

6.2 Conversion to Haskell

7 Testing

7.1 Example 1

7.2 Example 2

7.3 example 3

8 Conclusion

8.1 Further Work

References

[1] U.S. Commodity Futures Trading Commission, U.S. Securities, and Exchange Commission. Findings regarding the market events of may 6, 2010. https://www.sec.gov/news/studies/2010/marketevents-report.pdf, September 2010.

[2] Dragos Bozdog, Ionut Florescu, Khaldoun Khashanah, and Jim Wang. Rare events analysis for high-frequency equity data. *Wilmott Journal*, pages 74–81, 2011.

[3] Andrei Kirilenko, Albert S. Kyle, Mehrdad Samadi, and Tugkan Tuzun. The flash crash: The impact of high frequency trading on an electronic market. Working Paper, SSRN. http://ssrn.com/abstract=1686004 (accessed May 13, 2017)., 2014.

[4] Arjun Kharpal. Ethereum briefly crashed from $319 to 10 cents in seconds on one exchange after ‘multimillion dollar’ trade. http://www.cnbc.com/2017/06/22/ethereum-price-crash-10-cents-gdax-exchange-after-multimillion-dollar-trade.html, June 2017.

[5] Brian Louis and Janan Hanna. Flash crash trader sarao pleads guilty to fraud, spoofing. https://www.bloomberg.com/news/articles/2016-11-09/accused-flash-crash-trader-sarao-to-plead-guilty-in-chicago, November 2016.

[6] Hankyu Moon and Tsai-Ching Lu. Network catastrophe: Self-organized patterns reveal both the instability and the structure of complex networks. *Scientific Reports*, 2015.

[7] Tommi A. Vuorenmaa and Liang Wang. An agent-based model of the flash crash of may. *Working Paper*, 2014.

[8] Christopher D. Clack and Dmitrijs Zaparanuks Elias Court. Dynamic coupling and market instability. Working Paper, 2014.

[9] John S. Osmundson, Thomas V. Huynh, and Gary O. Langford. Emergent behavior in systems of systems. In *Conference on Systems Engineering Research (CSER)*, 2008.

[10] Norman Ehrentreich. *Agent-Based Modeling: The Santa Fe Institute Artificial Stock Market Model.* Springer, 2008.

[11] Eugene M. Izhikevich et al. Game of life. *Scholarpedia*, 10(6):1816, 2015.

[12] Douglas C. Heggie. The classical gravitational n-body problem. astro-ph/0503600, 2005.

[13] Isaac Newton, I. Bernard Cohen, and Anne Whitman. *The Principia: Mathematical Principles of Natural Philosophy.* Univ of California Press, 1999.

[14] Dirk Helbing. *Social Self-Organization.* Springer, 2012.

[15] Jochen Fromm. Types and forms of emergence. *arXiv:nlin/0506028*, 2005.

[16] Alan Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London*, 237(641):37–72, 1952.

[17] Eric Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *The National Academy of Sciences*, 99(3), 2002.

[18] J. Bradford De Long, Andrei Shleifer, Lawrence H. Summers, and Robert J. Waldmann. Noise trader risk in financial markets. *Journal of Political Economy*, 98(4):703–738, 1990.

[19] Alan Kirman. Ants, rationality, and recruitment. *The Quartely Journal of Economis*, 108(1):137–156, 1993.

[20] Jeffrey A. Frankel and Kenneth A. Froot. Chartists, fundamentalists and the demand for dollars. *National Bureau of Economic Research*, 1986.

[21] Christopher D. Clack. The interdyne simulator (2011-). http://www.resnovae.org.uk/fccsuclacuk/research, 11 2016.

[22] Peter Gomber, Martin Haferkorn, and Bus Inf Syst. High-frequency-trading. *Business & Information Systems Engineering*, 5:97–99, April 2013.

[23] Irene Aldridge and Steven Krawciw. *Real-Time Risk: What Investors Should Know About FinTech, High-Frequency Trading, and Flash Crashes.* Wiley, 2017.

[24] Elias Court. The instability of market-making algoritms. MEng Dissertation, 2013.

[25] R. Leombruni and M. Richiardi. Why are economists sceptical about agent-based simulations? *Physica A: Statistical Mechanics and its Applications*, 355(1):103–109, 2005.

[26] M. Gould, M. Porter, and S. Williams. Limit order books. *Quantitative Finance*, 13(11):1709–1742, 2013.

[27] Jet Wimp. *Computation with Recurrence Relations.* Pitman Advanced Publishing Program, 1984.

[28] Duane Q. Nykamp. Recurrence relation definition. http://mathinsight.org/definition/recurrence_relation, 6 2017.

[29] Chris Clack and Simon L. Peyton Jones. *Strictness analysis — a practical approach*, pages 35–49. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.