Translating from First Order Recurrence Relations to an Agent-Based Representation

Leo Carlos-Sandberg
Supervisor: Dr Christopher D. Clack

August 3, 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 as message passing throughout a simulation. Agent-based models however tend to be less well understood and accepted by those outside computer science, 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 the definition of a simple recurrence relation language, to define simulations, and the design of a step-by-step process by which a set of recurrence relations defined in this language can be converted into an agent-based model representation, the InterDyne simulator is chosen as a target representation for this transformation.

Contents

1 Introduction

Complex systems have been an area of interest for a long time in many fields such as physics, biology and chemistry [REFS????]. More recently this is true in the fields of economics and finance [REFS?????], especially as the financial markets have become increasingly automated and interconnected. Of particular interest is emergent behaviour in complex systems given by name "".

Emergent behaviour can occur in a large number of ways, but more often the behaviour of interest is created over a time period. In general there are two ways of viewing systems in time, these are either continuous (where the system progress through time smoothly) or discrete (where the system takes equal quantised steps through time, with each step advancing the system).

For discrete time systems two particular modelling techniques that can be used are, recurrence relations and agent-based models.

Both of these techniques have been extensively used in the modelling of complex systems, in a larger range of fields [REFS???]. Recurrence relations have a number of advantages such as [REFS???]:

- Giving a formal mathematical definition of the whole system.
- Showing an obvious link between equations within the system.
- Giving a static representation of the system.

Agent-based modelling benefits include [REFS???]:

- Being able to encode unique agents, with varying levels of complexity.
- Giving a dynamic representation of the system's evolution.
- Plainly showing the passing of information between agents.
- Easily expandable by adding more agents.

The ability of agent-based models to track the communication between agents and their bottom up approach makes them particularly adept at analysing emergent behaviour.

As mentioned there has been a growing interest into complex systems and emergence in finance, a large part of this interest is on regulating these systems to stop the creation of destructive emergent behaviour [REFS????]. This has been especially true when looking at high-frequency trading, which has been accused of causing a number of negative effects within the markets, including flash crashes [REFS???].

Though both agent-based models and recurrence relations have been used within finance, there is a notable preference among researchers grounded in economics to use recurrence relations.[1] This preference appears to be caused by agent-based models not resonating with these researchers,

This lack of acceptance presents a problem for work on emergence within the financial system; agent-based models are a strong tool in investigating emergent behaviour, ignoring them seriously limit the ability to research this area, and a large amount of research has already been done in this area using agent-based model, if this work is only accepted by a subsection of the community its impact will be substantially lessened.

This paper seeks to present a method whereby agent-based models will resonate better with experts such as economists, and with the hope that they will be used more frequently and the research based upon them will be better accepted.

This raises the question, how can agent-based models be explained to a sceptic in such a way that they will accept them? This papers approach is to connect agent-based models and recurrence relations together, allowing the already accepted recurrence relations to become the formalism of the agent-based model.

[1] This is discussed further in Section 2.2.

For this approach to work, there has to be confidence that the connection between the recurrence relations and agent-based model, has a significance and is correct. Hence the next question is what is a meaningful connect? A correctness preserving transformation was chosen as a connection, allowing the recurrence relations and agent-based model to be two views of the same system, hence giving meaning to the agent-based model of the system in terms of recurrence relations. This approach is coined as the two views.

This transformation takes a set of recurrence relations and turns them into an agent-based model, this direction has been chosen opposed to the reverse as the aim is to encourage sceptics to engage with agent-based models, so allows them to start with the familiar before the transformation.

To achieve this transformation from a set of recurrence relations to a agent-based model, a number of conceptual challenges needed to be addressed. These problems are discussed in greater depth in Section 3.2 but including:

- The two paradigms of the models are completely different and opposing.
- How are function calls related to message passing?
- How can the idea of agents having a infinite list through out time of their values be derived from the recurrence relations?
- How can the idea of private and public information introduced, and what data should fall into each category?
- How can public broadcast data be introduce, and what data classifies to be treated as such?
- How can time limited information be introduced?
- How can name control be introduced in a tangible way into a recurrence relations that have no notion of agents.
- How can output messages containing message information be introduced?
- How can the recurrence relations be split in a way which does not splinter the model?
- How can small, and hence more susceptible to prove of correctness, steps be used to transform the model?

To create a transformation that allows these issues to be overcome and to increase the easy in proofing the correctness of this conversion, a step-by-step approach was taken, where the transformation is done in a series of small correctness preserving steps.

This papers aim is hence to create the design of each of these steps, in the process of transforming from a set of recurrence relations to an agent-based model.

Creating this transformative method has two important aspects, increasing the acceptance of agent-based models and creating a model for viewing systems in two perspectives. Increasing the acceptance of agent-based models has a number of benefits including:

- Increasing the use of agent-based models in new research, where their benefits would be more suited. Hence increasing the ease in which emergence can be investigated within the financial system.
- Allow for work already done using agent-based models to be re-evaluated with a greater understanding to the methods used.

The creation of the two views model always a number of unique benefits by providing a new tool for analyse of complex systems:

- Adds a new technique for viewing and modelling complex systems.
- Adds a new tool that can be used for hypotheses formalisation and communication of ideas between researchers.
- Adds a model that is suitable for both static and dynamic analyses.

The timing of this paper fits with the increased interest in agent-based models within finance, for modelling the complex behaviour of the markets [REFS???????]. This work is designed to be able to support other work done using agent-based models and help further the understanding of agent-based models during this time of interest.

To show a conversion between the two types of models a particular example of each must be chosen, for this paper a custom recurrence relation language was designed to be transformed into the agent-based model known as InterDyne. InterDyne was chosen in particular due to it being designed from the bottom up to model and investigate emergent behaviour within the financial markets, this software was also a convenient choice due to access to the source code and local expertise.

This papers layout is as follows. First a background is given detailing, emergent behaviour generally and in the financial markets, a description and review of the literature of both recurrence relations and agent-based models, and a in-depth description of InterDyne. Secondly a description and analysis of the problems with agent-based models and how a two-view approach addresses them. Thirdly a description of the design, formal syntax, and use, of the custom recurrence relation language. Fourthly the design and a example of the step-by-step transformation between the two models. Fifthly a validation that the transformed example does indeed represent a InterDyne simulation. Sixthly a test of a number of examples for correctness of functionality. Lastly the paper is concluded and summarised with notes for further work given.

2 Background

This section covers a description of emergent behaviour, feedback loops as type of this behaviour and their appearance in the financial markets. Recurrence relations and agent-based models are introduced as a method for modelling emegernt behaviour, and they are described. InterDyne is introduced as an example of an agent-based model designed for the financial markets. Its applicability to modelling aspects of the financial market will be discussed and description of its design will also be given.

2.1 Emergent Behaviour

Emergent behaviour is a term used to describe macro-level behaviour of a system that is not obvious from analysis of the micro-level behaviour of the system, more formally this is behaviour that can not be predicted through analysis of any single component of a system [1].

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 [2].

A simple way to demonstrate emergence is in the Game of Life [3], 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 [4], 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.

The emergent behaviour within the n-body problem is caused by interaction dynamics, this is the communication between different elements of a system. The interactions here takes the form of gravitational pulls, if these pulls were not present then the system as a whole, and every individual present would maintain a constant velocity, unless they physically collided [5].

Though this example deals with a physical phenomenon, interaction based emergence is present in many different systems. Interactions in these systems can take the form of verbal and visual communication in social systems with negative emergent behaviour in this case being the break down of social cooperation [6]. The financial markets can be thought of as a complex system, with interaction dynamics, hence one can assume that the markets would exhibit emergence. This is true and the financial markets have seen to exhibit a large selection of emergent behaviour, such as the formation of patterns, bubbles and crashes [7]. These particular examples derive from feedback loops with in the markets, in a similar process to that of the interaction between short range and long range feedback loops in chemical reactions [8].

Feedback loops are a prominent type of emergent behaviour that can occur from interaction dynamics. 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.

In finance this could be seen as a simple trader who decides how much to sell based solely on their inventory at the previous time step.

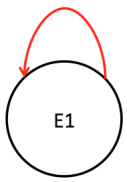

Figure 1: *Simple feedback loop with an entity supplying its input from its output.*

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.

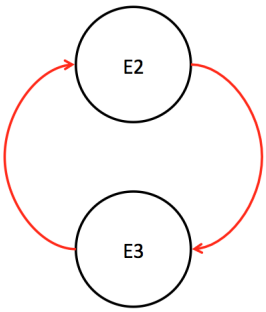

Figure 2: *Feedback loop between two entities, with each output being transformed by the other entity before becoming an input.*

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.

Since feedback loops can be destabilising and damaging to the system in which they occur, there is interest in studying this emergence in the aim to prevent monetary loss and damage to the economy.

A notable form of emergence behaviour, due to feedback loops, that takes place within the electronic markets are flash crashes. A flash crash is an event during which time the trading price of a security drops very rapidly, becoming disconnected from its fundamental, before then recovering [9]. A particularly famous flash crash is that of 2010, in which the E-min S&P 500 equity futures market dropped in price by more then 5%, before rebound to close to its original price [9, 10]. 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 [11].

Research done by Ref [12, 13], describes how the crash may have unfolded due to a feedback loop between High-Frequency Traders (HFTs), known as 'hot-potato" trading.

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 [14]. 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 [15].

The feedback loop of "Hot Potato" trading, 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 sold 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 [16].

This constant selling and buying of inventory can artificially inflate the trading volume of the market, changing how many traders operate and potentially leading to a flash crash.

Flash crashes have occurred on a number of occasions and in a large selection of markets, with a more recent example being a crash of the cryptocurrency ethereum [17].

2.2 Methods for Modelling Emergent Behaviour in Finance

There are many different methods available for modelling system that may exhibit emergent behaviour, such as: 1-period, 2-period and multi-period models, probabilist models, like hidden Markov models, ordinary differential equations and partial differential equations [REFS??????]. This paper focusses on two models that can be used for discrete non-equilibrium system, recurrence relations and agent-based models.

Both recurrence relations and agent-based models have been used in modelling the financial markets and have had a number of papers published on them.

Recurrence relations have seen wide spread use in economics appearing in a large selection of journals, including numerous times in top journals [18–27]. This wide spread use of the technique implies the acceptance of recurrence relations within financial modelling.

Agent-based models also have a large selection of papers published on them relating to economics and finance [28–30]. However these papers are noticeably absent from top economics journals (with some exceptions, such as Ref. [31] and [32]) and tend to be published in the Journal of Economic Behavior & Organization and the Journal of Economic Development & Control [33, 34]. The lack of agent-based models present in top economic journals combined with the number of papers published in other journals shows the lack of wide spread acceptance for this technique [35, 48, 50].

Here these two modelling techniques will be described.

2.2.1 Recurrences Relations

Recurrence relations connect a discrete set of elements in a sequence, these elements are normally either numbers or functions, they can be used to define these sequences or 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 [36].

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 [37].

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 parameters, 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.

Recurrence relations have a number of benefits given by their construction, such as:

- Giving a formal mathematical definition of the whole system.
- Showing an obvious link between equations within the system.
- Giving a static representation of the system.

A formal mathematic definition of the whole system being model is given when using recurrence relations due to three factors: the way in which they model the whole system as a entity, the static view point they give to the system and the mathematical stye which they take [38].

Links within sets of recurrence relations are hard coded into the equations, making these relationships amenable to static analysis.

This model lays out both the functionality of each equation and the relation between them in such a way to give a static representation of the system as a whole.

This model however has difficulties in design as it can be challenging in creating a large continuous description of a system containing many components, also the function calls as a method of information passing can make dynamically tracking the flow of information during a simulation hard [REFS????].

2.2.2 Agent-Based Models

A modelling technique that takes a more dynamic approach 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 [39].

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 [40], herding among traders [41], and fundamentalists [42].

Agent-based models are particularly suited to system which, contain a number of autonomous components. Each component can be modelled independently and then allowed to interact through messages sent between each other. This allows for a obvious visual design of the system, such as that shown in Fig. 3.

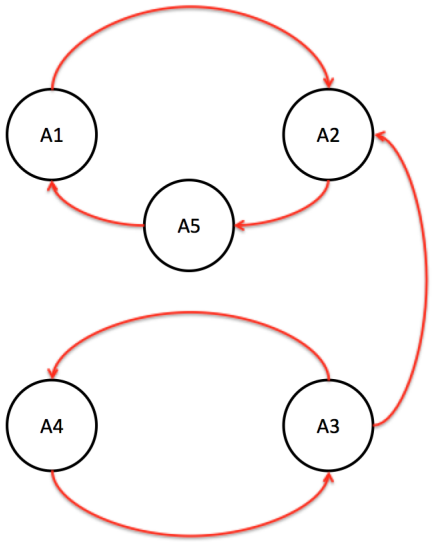

Figure 3: *Visualisation of the allowed interactions between five different agents, represented by black circles, with red lines representing the message paths.*

The main aspect of this model is the agents used. Though every system modelled can have drastically different agents, there are a few characteristics which should be in place for a agent to be considered an agent [43, 44]:

- An agent is self contained, unique and identifiable, this requires an agent to have boundaries which can easily be used to determine what is and what is not part of the agent.
- An agent autonomous and it can act independently during its interactions with other agents. Its has behaviour and decisions that can be associated with information acquired during communication with other agents.
- An agents state varies over time, representing variables associated with its current position.
- An agent behaviour is influenced by dynamic interactions with other agents.

This modelling technique has a number of benefits originating from its construction, these include:

- Being able to independently create each agent to varying degrees of complexity.
- Giving a clear visual representation of the systems interconnectedness.
- Giving a dynamic representation of the system.
- Easily expandable by adding more agents.

Since each agent in the system can be completely autonomous and independent from each other, save for message passing, this allows for them to be created individually. Hence each agent can describe its relevant component to a relevant degree of complexity, making the creation of a simulation more intuitive and sectioned. This makes agent-based models in many cases the most natural way for describing systems composed of "behavioural" entities [45].

Do due the set up of this model the a network representing the communication between different agents is easy to construct, this allows for topologies to be visualised and can aid in the creation of a simulation.

The use of messages and bottom up approach this model takes allows for a more dynamic view of a system to be achieved, messages can be more easily tracked through out the systems evolution allowing for events to be more easily pinpointed and analysed.

The use of independent agents makes this technique very amenable to expansion, new agents can be added normally with little to no change to previously existing agents. This allows these simulations to be confidently expanded to look at more complex systems, or to add new elements to an existing system.

These aspects of this approach make the modelling technique well qualified for modelling and analyse of emergent behaviour within systems of interacting components. However this approach does contain some draw backs including a one-to-many problem (where though some high level behaviour may be noted in the system this does not mean the only way of achieving this behaviour is the current system design) and the lack of a formal definition for the system as a whole [REFS???].

2.3 InterDyne

An example of an agent-based model used in modelling the financial markets is InterDyne. InterDyne is bespoke simulator created by Clack and his research team at UCL [46], it is a general-purpose simulator for exploring emergent behaviour and interaction dynamics within complex non-equilibrium 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 through a harness 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 without proper definition (allowing operations to be defined in a number of time steps) or they can be equated to a real time (usually with the smallest time gap needed being a single time step and then all other timings being integer multiples of this). This discrete time is most important to message passing, with 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. An agent can send private messages that are only received by a single other agent (one-to-one messages), or it can send public broadcast messages received by any number of other agents, subscribed to a channel (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[2], 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. 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.[3]

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

InterDyne has been designed with modelling flash crashes in the financial markets in mind and has a number of features that make it well suited to this purpose.

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 whether 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 for messages to be delayed is needed to facilitate hypotheses involving delays as a factor for emergent behaviour. Delays have been suggested to have caused "hot potato" trading during a flash crash, these delays can exist due to processing time and transmission time of messages [10].

InterDyne allows for both symmetric and asymmetric, delays. These delays can be static or dynamic, with dynamic delays requiring a special intermediary agent.

2.3.1.3 Storing Messages

InterDyne facilitates analyses of simulations be allowing for the messages between agents to be stored and viewed as a trace. This can include all messages as well as timings, messages can also be sent directly to the harness which will be added to the trace file.

[2] Unless instructed to, using a static delay.

[3] If an agent receives multiple messages at the same time step and the pseudo-random element is not being used, these messages will order based on the identifiers of their sender agents.

2.3.1.4 Message Ordering

The order in which messages are processed can be very important. For example in an exchange, 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 emergent 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.

2.3.2 InterDyne Detailed Operation

InterDyne is a written in the functional language Haskell. The structure of an InterDyne model is show in Fig. 4, this structure contains a number of agents and a simulator harness. These agents send two types of messages, either one-to-one or one-to-many (broadcast) messages. Both these messages are sent to the harness, the harness then resends these messages to the appropriate agents, one-to-one messages are sent to their target agent and one-to-many messages are sent to any agent subscribed to the relevant broadcast channel. Messages can also be sent directly to the harness and not rerouted to another agent. At the end of the simulation the harness will produce a trace file containing information on all the messages sent for post-hoc analysis.

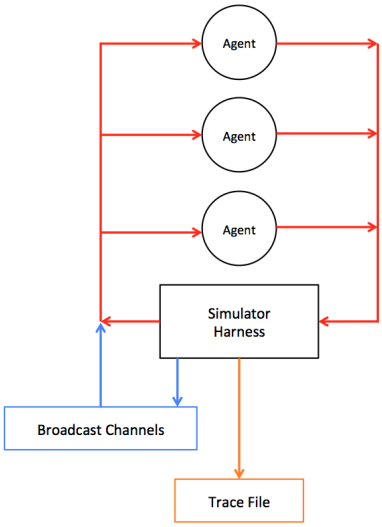

Figure 4: *Structure of an InterDyne simulation containing three agents. The messages sent by the agents to the harness and from the harness are coloured red. The trace file and messages sent to the trace file are coloured orange. The messages sent from the harness to the broadcast channels and then back to the agents are coloured blue [47].*

InterDyne has a intrinsic delay, which means any message sent will not be received until the next time step, message sent at t will be received the earliest at $t + 1$. This is in part due to the fact that the harness compiles all messages from a single time step before sending them to their targets and hence initiating the next time step. The harness can be seen as the driver in the simulation sending out the message and forcing the agents to send there next message.

Longer delays can also be added using InterDynes topology, this allows for any two agents to be selected (a sender and a receiver) and for a delay between them to be given. This topology takes the form of a directed graph, with the agents acting as the nodes and the interaction routes as the links, this allows for the delays to be asymmetrical. Using this a delay for a path could also be set as a abort message, making a particular communication route unusable.

Adding delays is done through run time arguments, here two run time arguments must be added, one stating the maximum delay in the system and the other listing the delays. To show the maximum delay the argument $(Arg(Str\ \text{“}maxDelay\text{”}, 10)))$ is used, where 10 is the maximum delay. To add the list of delays the argument $(DelayArg(Str\ \text{“}DelayArg\text{”}, delay))$ is used, where *delay* is a function that takes two agents and returns the delay between the first and the second.

Messages sent between agents can in theory be as complex as needed, these messages do however have to comprise of these components:

- A tag indicating the type of message being sent, e.g. a broadcast message.
- A tuple of two integers, with the first being the sending agents ID and the second being either the receiving agents ID or the receiving broadcast channels ID.
- The data that is being sent.

An example of this is a one-to-one message, which would look like $Message\ (1, 2)\ data$, where *data* is the information being sent from agent 1 to agent 2

A broadcast message, which looks like $Broadcastmessage\ (3, 1)\ broadcastdata$, can be received by agents subscribed to its channel, in this case 1. All agent subscriptions have to be announced at the beginning of a simulation and can not be changed during it. This is done by adding the subscription channel to the list in a tuple passed as an argument, for example this could look like $(agent1, [1])$, for agent 1 subscribed to broadcast channel one.

Empty messages can also be sent, if the message is required to be known to be intentionally empty and not a mistake the a message can be sent containing “Hiaton”. This is done at the beginning of the run to allow for the harness to send a first message.

InterDyne is designed in such a way that agents both taken in and produce a seemingly infinite list of messages. This is achieved through lazy evaluation in Haskell, which in short means that any element in the list will not be calculated until it is absolutely needed. This allows for an infinite list as long as no agent tries to read further into the list then what as already been calculated. In achieving this an agent must at each time step, allows read in a message (even if it is then not used) and produce a list of messages (even if this is a empty list). An agent will hence iterate over the list of incoming messages, at each time step adding a new output message to its list of output messages.

Agents in InterDyne are topically, though not required to be, written in two parts, a wrapper function and a logic function. The wrapper will manage the reading of inbound messages to the agent and generate its outbound messages, it will also update the local state of the agent. While a logic function is called by the wrapper to calculate the messages to be sent and there content. A agent written following this design could have a form similar to that of the agent show in Fig. 5. This agents wrapper recursively calls its self on the inbound message list, consuming the head of the list[4] and producing a list of outbound messages using a logic function.

```
1 wrapper st args ((t, msgs, bcasts) : rest) myid = [m] : (wrapper st args rest myid)
2                                                    where
3                                                    m = logic (t, msgs, bcasts)
```

Figure 5: *Simple wrapper function, which creates a list of output messages by iterating through a list of inbound messages and calling the sub-function logic.*

[4]This design dictates that the head of the list is the messages for the current time step.

The simulator harness is an intrinsic part of the simulator, it drives the simulation, handles message passing and produces the output trace file.

Running an InterDyne experiment is done by calling the simulator with relevant inputs, an example example of this is shown in Fig. 6.

```
exampleExperiment
  = do
    sim 100 myargs (map snd myagents)
    where
    myargs   = [ (Arg (Str "maxDelay", maxDelay)),
                 (DelayArg (Str "DelayArg", delay))
               ]
    myagents = [ (agent1, [1]),
                 (agent2, []),
                 (agent3, [2,3])
               ]
    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
```

Figure 6: *A run of an experiment in InterDyne containing delays.*

3 Description and Analysis of the Problem

Investigating emergent behaviour originating from interaction dynamics, within a complex system, relays on viewing the communication between different components of the system as messages being passed. These messages can then be analysed to assess the communication and its pattern that lead to the creation of the emergent behaviour, this approach is amenable to the discovery of phenomena, such as dynamic feedback loops. A model investigating emergent behaviour hence must be able to view communication as messages being passed between different components. For results from this model to be broadly accepted, in fields such as economics, the model as a whole must also be well defined.

Agent-based models naturally support a message passing view making them well suited to analysing and investigating emergent behaviour. However most agent-based models, though having each individual agent be fully specified, do not have a well defined formalisation for the system as a whole. This makes it challenging to perform system wide analytics on most agent-based model. Recurrence relations naturally support a system wide analysis, due to the full definition of their models. However their formulation does not support a message passing view of communication between components, making it difficult to analyse emergent behaviour with them directly.

This creates a problem as neither model is perfect for the task at hand. A resolution to this problem suggested by Ref. [48], is to give a formal definition of the specification of an agent-based model as a set of recurrence relations.This can be seen as viewing a system in two ways, as a recurrence relation model and as a agent-based model, a two-views approach.

3.1 Method for Achieving a Two-Views Approach

- How can this solution be achieved (translation form rr to abm)
- How will this translation be done (research review conclude step by step aproach)
- what is the project? (make rr language, convert in a step-by-step manner to InterDyne)

3.2 Challenges with the Two-Views Approach

- what are the challenges with this method (list in intro)
- The two paradigms of the models are completely different and opposing.
- How are function calls related to message passing?
- How can the idea of agents having a infinite list through out time of their values be derived from the recurrence relations?
- How can the idea of private and public information introduced, and what data should fall into each category?
- How can public broadcast data be introduce, and what data classifies to be treated as such?
- How can time limited information be introduced?
- How can name control be introduced in a tangible way into a recurrence relations that have no notion of agents.
- How can output messages containing message information be introduced?
- How can the recurrence relations be split in a way which does not splinter the model?
- How can small, and hence more susceptible to prove of correctness, steps be used to transform the model?

4 Bespoke Recurrence Relation Language

The first step in this translation is selecting a recurrence relation, which to translate from. Recurrence relations are used across a wide range of disciplines and as such have many different forms of notation [REFS????]. It was however decided to create a bespoke recurrence relation language for use in this translation, this was done for a number of reasons:

1. Need to have a strict set of rules for the language to allow for automated parsing.
2. Need to restrict the functionality of the language to keep it as simple as possible, to facilitate the automated translation. Only introducing complexity where necessary.
3. Need to maintain sufficient functionality for modelling complex systems in the language.

    As noted in the third point, this language as to have a certain level of functionality.
    It was decided that the language should have the following function:

- 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 index into a list.
- The ability to use brackets.

- The ability to declare variables.
- The ability to define a unique name for each recurrence relation.
- The ability to define unique names for groups of recurrence relations.
- The ability to define a recurrence relation that uses inputs.
- The use of sub functions within recurrence relations.
- The ability for recurrence relations to pattern match initial conditions.
- The use of where blocks.
- The use of if else statements.
- The ability to specify an entire experiment.

4.1 Syntax

The formal syntax for this language is shown in Fig. 7, this syntax is written in a style consistent with type definitions within the Miranda language. This style is that of a type followed by its values, the left hand side of ::= is the type and can take the form of any one of its values, on the right hand side.

A simulation is defined as type *simulation*, which has two parts. A list of recurrence relations (list of type $definition$) and information about the simulation (of type *experiment*).

A definition of a recurrence relation is defined as type $definition$, has one values $Function$. The value $Function$ contains to list of characters which hold the group name and the name of the recurrence relation, a list of the arguments (type *argument*) and a expression for the functionality of the recurrence relation (type *expression*).

A argument, whether it be used for the definition of a recurrence relation or used for calling one, has one value $Argument$. This value is associated with an expression of the actual argument (type *expression*).

The *experiment* must take the value of $Experiment$ which has two parts, a list of initial conditions (type *initcon*) and a function call for the result of the simulation (type *experimentrun*).

An initial condition for the recurrence relations is listed as a value $Initcon$ which has a name and an expression associated with it (type [*char*] and *expression* respectively), this name is used to connect the initial condition to the relevant recurrence relations.

The *experimentrun* can have two values, either being empty (value $Emptyrun$) or to have a function call (value $Experimentrun$). If there is a function call this has an associated *expression* with it.

An *expression* can produce any expression for a recurrence relation that the language will allow, *expression* can recursively call itself to create more complex expressions or can call other types such as *argument*, $subdefinition$, *op* and $specfunc$.

The use of operations is through the type *op*, this type has a value relating to all operations that use two values.

Two operations are allowed that only use one value, returning the head of a list (value $Listhead$) and returning the tail of a list (value $Listtail$).

The type $subdefinition$ has two values for either internal variables in a where statement ($IntVariable$) or internal functions within a where statement ($IntFunction$).

```
simulation ::= Simulation [definition] experiment

definition ::= Function [char] [char] [argument] expression

argument::= Argument expression

experiment::= Experiment [initcon] experimentrun

initcon::= Initcon [char] expression

experimentrun::= Emptyrun|Experimentrun expression

expression::= Emptyexpression
              |Ifthenelse expression expression expression
              |Brackets expression
              |List [expression]
              |Operation expression op expression
              |IntFunct [char] [argument]
              |ExtFunct [char] [char] [argument]
              |IntVar [char]
              |ExVar [char] [argument]
              |Specialfunc specfunc expression
              |Number num
              |Where expression [subdefinition]

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

specfunc::= Listhead|Listtail

subdefinition ::= IntVariable [char] expression |  IntFunction [char] [argument]
    expression
```

Figure 7: *Formal definition of syntax within bespoke recurrence relation, written in style of Miranda type definitions.*

4.2 Naming Conventions

This language has a strict naming convention, as well as some restricted names. Naming conventions are in place for recurrence relation definitions, where blocks, sub-functions, variables, initial conditions and if statements.

A recurrence relation be defined in four parts, *name arguments* = *expression*. The name must be formatted in the manner *group_name*, where *group* is the name of the group the recurrence relation belongs to and *name* is the name of the particular recurrence relation. Multiple recurrence relations can belong to the same group, and hence have the same group name, however only one relation in each group can have a set name. This language requires that all recurrence relations take at least one argument, and that time t has to be the first argument. Time t does not need to be utilised within the relation but it must be in the definition.

A recurrence relation can have sub definitions relating to it, this is done through the use of a where block. A where block is written after the expression for a recurrence relation and starts with the keyword *where*. After this local definitions for internal functions and variables can be defined.

Internal functions must be formatted in the manner *_name*, where *name* is the name for the function. These functions can only be accessed by the recurrence relation they are within.

Local variables can be defined using the formate *name*, this name can be anything as long as it does not clash with restricted names.

Initial conditions are listed in a special location in the layout, which will be show in Sec. 4.3, and are defined in three parts $name = expression$. This name can be seen as a variable and can be anything not in the restricted name list.

If statements with in this language are written as $if\ condition\ then\ expression\ else\ expression$. Where if the condition is true, then the first expression is run and if not the second expression is.

There are a number of names and naming conventions which can not be used and can be seen as restricted, these are:

- The use of _, unless it is being used for its designated purpose in recurrence relations and internal functions.
- The use of the name *main*.
- The use of the name *init*.
- The use of the name *where*, unless used to make a where statement.
- The use of symbols, unless for their mathematical purpose.

4.3 Well Formed Programs

This language has a strict style and layout, this covers a number of aspects: layout, experiment, initial conditions and recurrence relations.

The layout of an simulation should match that shown in Fig. 8, with *main* signifying the experiment, *init* the initial conditions and *where* the list of recurrence relations.

```
main
"Experiment"

init
"Initial Conditions"

where
"Recurrence Relations"
```

Figure 8: *Layout of a simulation.*

An experiment is written as a function call, with a certain set of arguments. Figure 9 shows a experiment that will return the value of the recurrence relation i_f1 at time 3.

```
i_f1 3
```

Figure 9: *Function call for value at time 3 in recurrence relation 1_f1.*

The initial conditions, which use will be shown later, are defined as shown in Fig. 10. There is no defined limit to the amount of initial conditions that can be defined.

```
q = 4
k = 16
```

Figure 10: *Definition of two initial conditions, q and k.*

A simple recurrence relation can be written as shown in Fig. 11, this recurrence relation belongs to group i, is named $f1$ and takes one argument which is time t. The expression of this relation is to call itself at the previous time step and to add that to a call for another recurrence relation, from group j, at the previous time step.

```
i_f1 t = (i_f1 (t-1) ) + (j_f1 (t-1))
```

Figure 11: *A simple recurrence relation, belonging to group i that adds its value at the previous time step to that of j_f1 at the previous time step.*

Initial conditions can added to recurrence relations using pattern matching, as shown in Fig. 12. The recurrence relation is first written with the value for which the initial condition will be sent and this is sent to equal the initial condition, then the full recurrence relation is written.

```
i_f1 0 = q
i_f1 t = (i_f1 (t-1) ) + (j_f1 (t-1))
```

Figure 12: *Recurrence relation containing pattern matching for initial conditions.*

An if statement, shown in Fig. 13, can contain very complex components it is hence good practice to encase each expression with in a set of brackets to maintain legibility and correct functionality.

```
j_f1 t = if (t<0) then (k) else (25 + ((j_f1 (t-1)) * (i_f1 (t-1))) + (j_f2 (t-1)))
```

Figure 13: *Recurrence relation using an if statement.*

A where statement is used to provided additional functionality to a recurrence relation and can be used to make a expression more legible and help reduce human errors. Figure 14 shows the use of a where statement to add an internal variable and an internal function to a recurrence relation.

```
i_f1 t = a * (_sf1 t)
         where
         a      = 5
         _sf1 x = x*2
```

Figure 14: *A recurrence relation containing a where statement to add a internal function and a internal variable.*

This style and layout is brought together to produce a full simulation, as shown in Fig. 15.

```
main
i_f1 3

init
q = 4
k = 16

where
i_f1 0 = q
i_f1 t = (i_f1 (t-1) ) + (j_f1 (t-1))

j_f1 0 = k
j_f1 t = if (t<0) then (k) else (25 + ((j_f1 (t-1)) * (i_f1 (t-1))) + (j_f2 (t-1)))
j_f2 0 = k+q
j_f2 t = (j_f1 (t-3)) + 27
```

Figure 15: *A simulation in the bespoke recurrence relation language.*

5 Translation from Recurrence Relations to InterDyne

- Re say what the convertor is and what is meant to do
- Re say project scope and that here the converter is only being designed
- say that is takes a step by step approach and why
- why? so that the steps can be shown to be correctness preserving later more easily
- say what the steps are going to be

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. 16.

```
main
i_f1 3

init
q = 4
k = 16

where
i_f1 0 = q
i_f1 t = (i_f1 (t-1) ) + (j_f1 (t-1))

j_f1 0 = k
j_f1 t = if (t<0) then k else (25 + ((j_f1 (t-1)) * (i_f1 (t-1))) + (j_f2 (t-1)))
j_f2 0 = k+q
j_f2 t = (j_f1 (t-3)) + 27
```

Figure 16: *Simple example of recurrence relations written in custom defined language.*

5.1 Recurrence Relation Parser (done)

The recurrence relations shown in Fig. 16 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. 17.

```
This is the parse tree
```

Figure 17: *Simple example of recurrence relations written in custom defined language.*

5.2 Infinite List Outputs (done)

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.18.

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

Figure 18: *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.19.

```
Main
i_f1_list ! 3

Init
q = 4
k = 16

Where
_createlist f t = (f t):(_createlist f (t+1))

i_f1_list = _createlist i_f1 0
            where
            i_f1 0 = q
            i_f1 t = (i_f1_list ! (t-1) ) + (j_f1_list ! (t-1))

j_f1_list = _createlist j_f1 0
            where
            j_f1 0 = k
            j_f1 t = if (t<0) then k else (25 + ((j_f1_list ! (t-1)) * (i_f1_list ! (t
    -1))) + (j_f2_list ! (t-1)))

j_f2_list = _createlist j_f2 0
            where
            j_f2 0 = k+q
            j_f2 t = (j_f1_list ! (t-3)) + 27
```

Figure 19: *Recurrence relations with with infinite list outputs.*

5.3 Wrapper Function (done)

```
Main
(i_wrapper ! 3) ! 0

Init
q = 4
k = 16

Where
_createlist f t = (f t):(_createlist f (t+1))
_createwrapper lists = (map hd lists) : (_createwrapper (map tl lists))

i_wrapper = _createwrapper [i_f1_list]
            where
            i_f1_list = _createlist i_f1 0
                        where
                        i_f1 0 = q
                        i_f1 t = (i_f1_list ! (t-1) ) + ((j_wrapper ! (t-1)) ! 0)

j_wrapper = _createwrapper [j_f1_list , j_f2_list]
            where
            j_f1_list = _createlist j_f1 0
                        where
                        j_f1 0 = k
                        j_f1 t = if (t<0) then k else (25 + ((j_f1_list ! (t-1)) * ((
    i_wrapper ! (t-1)) ! 0)) + (j_f2_lst ! (t-1)))
          j_f2_list = _createlist j_f2 0
                      where
                      j_f2 0 = k+q
                      j_f2 t = (j_f1_list ! (t-3)) + 27
```

Figure 20: *Recurrence relations with with wrapper functions.*

5.4 Outputs (done)

```
Main
((outputs ! 3) ! 0) ! 0

Init
q = 4
k = 16

Where
_createlist f t = (f t):(_createlist f (t+1))
_createwrapper lists = (map hd lists) : (_createwrapper (map tl lists))

outputs =   _createwrapper [i_wrapper, j_wrapper]

i_wrapper = _createwrapper [i_f1_list]
            where
            i_f1_list = _createlist i_f1 0
                        where
                        i_f1 0 = q
                        i_f1 t = (i_f1_list ! (t-1) ) + (((outputs ! (t-1)) ! 1) ! 0)

j_wrapper = _createwrapper [j_f1_list, j_f2_list]
            where
            j_f1_list = _createlist j_f1 0
                        where
                        j_f1 0 = k
                        j_f1 t = if (t<0) then k else (25 + ((j_f1_list ! (t-1)) * (((
    outputs ! (t-1)) ! 0) ! 0)) + (j_f2_lst ! (t-1)))
            j_f2_list = _createlist j_f2 0
                        where
                        j_f2 0 = k+q
                        j_f2 t = (j_f1_list ! (t-3)) + 27
```

Figure 21: *Recurrence relations with with wrapper functions.*

5.5 Messages

```
Main
((outputs ! 3) ! 0) ! 0

Init
q = 4
k = 16

Where
_createlist f t = (f t):(_createlist f (t+1))
_createwrapper lists = (map hd lists) : (_createwrapper (map tl lists))

outputs =  _createwrapper [i_wrapperfun outputs,    j_wrapperfun outputs]

i_wrapperfun inputs = _createwrapper [i_f1_list]
                      where
                      i_f1_list = _createlist i_f1 0
                                  where
                                  i_f1 0 = q
                                  i_f1 t = (i_f1_list ! (t-1) ) + (((hd inputs) ! 1) !
    0)

j_wrapperfun inputs = _createwrapper [j_f1_list, j_f2_list]
                      where
                      j_f1_list = _createlist j_f1 0
                                  where
                                  j_f1 0 = k
                                  j_f1 t = if (t<0) then k else (25 + ((j_f1_list ! (t
    -1)) * (((hd inputs) ! 0) ! 0)) + (j_f2_lst ! (t-1)))
                      j_f2_list = _createlist j_f2 0
                                  where
                                  j_f2 0 = k+q
                                  j_f2 t = (j_f1_list ! (t-3)) + 27
```

Figure 22: *Recurrence relations with with wrapper functions.*

5.6 Queues

5.7 Harness

```
Main
sim_harness 3

Init
q = 4
k = 16

Where
_createlist f t = (f t):(_createlist f (t+1))
_createwrapper lists = (map hd lists) : (_createwrapper (map tl lists))

sim_harness t = return
                where
                return  = ((outputs ! t) ! 0) ! 0
                outputs =  _createwrapper [i_wrapperfun outputs,    j_wrapperfun outputs
    ]



i_wrapperfun inputs = _createwrapper [i_f1_list]
                      where
                      i_f1_list = _createlist i_f1 0
                                  where
                                  i_f1 0 = q
                                  i_f1 t = (i_f1_list ! (t-1) ) + (((hd inputs) ! 1) !
    0)

j_wrapperfun inputs = _createwrapper [j_f1_list , j_f2_list]
                      where
                      j_f1_list = _createlist j_f1 0
                                  where
                                  j_f1 0 = k
                                  j_f1 t = if (t<0) then k else (25 + ((j_f1_list ! (t
    -1)) * (((hd inputs) ! 0) ! 0)) + (j_f2_lst ! (t-1)))
                      j_f2_list = _createlist j_f2 0
                                  where
                                  j_f2 0 = k+q
                                  j_f2 t = (j_f1_list ! (t-3)) + 27
```

Figure 23: *Recurrence relations with with wrapper functions.*

5.8 Input Messages

```
Main
sim_harness 3

Init
q = 4
k = 16

Where
_createlist f t = (f t):(_createlist f (t+1))
_createwrapper lists = (map hd lists) : (_createwrapper (map tl lists))
_createinput id outputs = outputs

sim_harness t = return
                where
                return  = ((outputs ! t) ! 0) ! 0
                outputs =  _createwrapper [i_wrapperfun (inputs ! 0), j_wrapperfun (
    inputs ! 1)]
                inputs  = [_createinput 1 outputs, _createinput 2 outputs]

i_wrapperfun inputs = _createwrapper [i_f1_list]
                      where
                      i_f1_list = _createlist i_f1 0
                                  where
                                  i_f1 0 = q
                                  i_f1 t = (i_f1_list ! (t-1) ) + (((hd inputs) ! 1) !
    0)

j_wrapperfun inputs = _createwrapper [j_f1_list, j_f2_list]
                      where
                      j_f1_list = _createlist j_f1 0
                                  where
                                  j_f1 0 = k
                                  j_f1 t = if (t<0) then k else (25 + ((j_f1_list ! (t
    -1)) * (((hd inputs) ! 0) ! 0)) + (j_f2_lst ! (t-1)))
                      j_f2_list = _createlist j_f2 0
                                  where
                                  j_f2 0 = k+q
                                  j_f2 t = (j_f1_list ! (t-3)) + 27
```

Figure 24: *Recurrence relations with with wrapper functions.*

5.9 Direct Messages

```
Main
sim_harness 3

Init
q = 4
k = 16

Where
_createlist f t = (f t):(_createlist f (t+1))
_createwrapper lists = (map hd lists) : (_createwrapper (map tl lists))
_createmsgwrapper lists = (map _tplhd lists) : (_createmsgwrapper (map _tpltl lists))
                          where
                          _tplhd (from, to, values) = (from, to, hd values)
                          _tpltl (from, to, values) = (from, to, tl values)
_createinput id outputs = (_getmsgs id (hd outputs) []):_createinput id (tl outputs)
                          where
                          _getmessages id []     msgs = msgs
                          _getmessages id (x:xs) msgs = _getmessages id xs (msgs++(
    _getsubmsgs x))
                                                                where
                                                                _getsubmsgs []
       id submsgs = submsgs
                                                                _getsubmsgs ((from, to, value):
    xs) id submsgs = if (to = id) then (_getsubmsgs xs id (submsgs++[(from, to, value)])
    ) else (_getsubmsgs xs id submsgs)
_getvalue (from, to, value) = value


sim_harness t = return
                where
                return  = ((outputs ! t) ! 0) ! 0
                outputs =  _createwrapper [i_wrapperfun (inputs ! 0), j_wrapperfun (
    inputs ! 1)]
                inputs  = [_createinput 1 outputs, _createinput 2 outputs]

i_wrapperfun inputs = _createmsgwrapper [(1, 2, i_f1_list)]
                      where
                      i_f1_list = _createlist i_f1 0
                                  where
                                  i_f1 0 = q
                                  i_f1 t = (i_f1_list ! (t-1) ) + (_getvalue ( (hd
    inputs) ! 0 ))

j_wrapperfun inputs = _createmsgwrapper [(2, 1, j_f1_list)]
                      where
                      j_f1_list = _createlist j_f1 0
                                  where
                                  j_f1 0 = k
                                  j_f1 t = if (t<0) then k else (25 + ((j_f1_list ! (t
    -1)) * (_getvalue ( (hd inputs) ! 0))) + (j_f2_lst ! (t-1)))
                      j_f2_list = _createlist j_f2 0
                                  where
                                  j_f2 0 = k+q
                                  j_f2 t = (j_f1_list ! (t-3)) + 27
```

Figure 25: *Recurrence relations with with wrapper functions.*

5.10 Output Message

```
Main
sim_harness 3

Init
q = 4
k = 16

Where
_createlist f t = (f t):(_createlist f (t+1))
_createwrapper lists = (map hd lists) : (_createwrapper (map tl lists))
_createmsgwrapper lists = (map _tplhd lists) : (_createmsgwrapper (map _tpltl lists))
                          where
                          _tplhd (from, to, values) = (from, to, hd values)
                          _tpltl (from, to, values) = (from, to, tl values)
_createinput id outputs = (_getmsgs id (hd outputs) []):_createinput id (tl outputs)
                          where
                          _getmessages id []     msgs = msgs
                          _getmessages id (x:xs) msgs = _getmessages id xs (msgs++(
    _getsubmsgs x))
                                                                where
                                                                _getsubmsgs []
       id submsgs = submsgs
                                                                _getsubmsgs ((from, to, value):
    xs) id submsgs = if (to = id) then (_getsubmsgs xs id (submsgs++[(from, to, value)])
    ) else (_getsubmsgs xs id submsgs)
_getvalue (from, to, value) = value


sim_harness t = return
                where
                return  = _createinput 0 outputs
                outputs =  _createwrapper [i_wrapperfun (inputs ! 0), j_wrapperfun (
    inputs ! 1)]
                inputs  = [_createinput 1 outputs, _createinput 2 outputs]

i_wrapperfun inputs = _createmsgwrapper [(1, 2, i_f1_list), (1, 0, i_f1_list)]
                      where
                      i_f1_list = _createlist i_f1 0
                                  where
                                  i_f1 0 = q
                                  i_f1 t = (i_f1_list ! (t-1) ) + (_getvalue ( (hd
    inputs) ! 0 ))

j_wrapperfun inputs = _createmsgwrapper [(2, 1, j_f1_list)]
                      where
                      j_f1_list = _createlist j_f1 0
                                  where
                                  j_f1 0 = k
                                  j_f1 t = if (t<0) then k else (25 + ((j_f1_list ! (t
    -1)) * (_getvalue ( (hd inputs) ! 0))) + (j_f2_lst ! (t-1)))
                      j_f2_list = _createlist j_f2 0
                                  where
                                  j_f2 0 = k+q
                                  j_f2 t = (j_f1_list ! (t-3)) + 27
```

Figure 26: *Recurrence relations with with wrapper functions.*

5.11 Broadcast Messages

6 Testing and Validation

- What is the validation for? checking that the conversion actual matches with an ABM (in this case InterDyne)
- do this by comparing types
- also by checking coding syntax

6.1 Type Comparison to InterDyne

6.2 Conversion to Haskell

6.3 Testing

- checking that this conversion still produces the same results as the RR

7 Conclusion

7.1 Further Work

- Build code
- correctness proof

References

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

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

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

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

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

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

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

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

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

[10] 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.

[11] 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.

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

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

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

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

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

[17] 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.

[18] Marian Kwapisz. On difference equations arising in mathematics of finance. *Nonlinear Analysis: Theory, Methods & Applications*, 30, 1997.

[19] Nancy L. Stokey. *Recursive Methods in Economic Dynamics*. Harvard University Press, 1989.

[20] Lars Ljungqvist and Thomas J. Sargent. *Recursive Macroeconomic Theory*. MIT Press, 2004.

[21] Charles M. C. Lee, James Myers, and Bhaskaran Swaminathan. What is the intrinsic value of the dow. *The Journal of Finance*, 1999.

[22] Myles M. Dryden. Share price movements: A markovian approach. *The Journal of Finance*, 1969.

[23] Stefan Niemann and Paul Pichler. Collateral, liquidity and debt sustainability. *The Economic Journal*, 2017.

[24] Sandeep Kapur and Allan Timmermann. Relative performance evaluation contracts and asset market equilibrium. *The Economic Journal*, 2005.

[25] Chaim Fershtman and Ariel Pakes. Dynamic games with asymmetric information: A framework for empirical work. *The Quarterly Journal of Economics*, 2012.

[26] Gauti B. Eggertsson and Paul Krugman. Debt, deleveraging, and the liquidity trap: A fisher-minsky-koo approach. *The Quarterly Journal of Economics*, 2012.

[27] Cars H. Hommes. Heterogeneous agent models in economics and finance. *Handbook of Computational Economics*, 2, 2006.

[28] Leigh Tesfatsion. Agent-based computational economics: modeling economies as complex adaptive systems. *Information Sciences*, 149, 2003.

[29] Leigh Tesfatsion. Introduction to the special issue on agent-based computional economics. *Journal of Economic Dynamics and Control*, 25, 2001.

[30] Simone Alfarano, Thomas Lux, and Friedrich Wagner. Estimation of agent-based models: The case of an asymmetric herding model. *Computational Economics*, 26, 2005.

[31] Thomas C. Schelling. Models of segregtion. *American Economic Review*, 59(2):488–493, 1969.

[32] Herbert Gintis. The dynamics of general equilibrium. *The Economic Journal*, 117:1280–1309, 2007.

[33] Matteo Richiardi. The future of agent-based modelling. *Economics Papers*, 2015.

[34] J. Doyne Farmer and Duncan Foley. The economy needs agent-based modelling. *Nature*, 460:685–686, 2009.

[35] Dirk Helbing and S. Balietti. *Social Self-Organization, Agent-Based Simulations and Experiments to Study Emergent Social Behavior*. Springer, 2012.

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

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

[38] John M. Hollerbach. A recursive lagrangian formulation of maniputator dynamics and a comparative study of dynamics formulation complexity. *IEEE Transactions on Systems, Man, and Cybernetics*, 10, November 1980.

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

[40] 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.

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

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

[43] Charles M. Macal and Michael J. North. Tutorial on agent-based modelling and simulation. *J. Simulation*, 4:151–162, 09 2010.

[44] Charles M. Macal. To agent-based simulation from system dynamics. In *Proceedings of the 2010 Winter Simulation Conference*, 2010.

[45] Ali Bazghandi. Techniques, advantages and problems of agent based modeling for traffic simulation. *International Journal of Computer Science Issues*, 9(3), 2012.

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

[47] Christopher D. Clack. Interdyne user manual. http://www.resnovae.org.uk/fccsuclacuk/images/docs/research/InterDyne User-Manual.pdf, 08 2017.

[48] 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.

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

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