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

Leo Carlos-Sandberg
Supervisor: Dr Christopher D. Clack

August 11, 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 3.3 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.

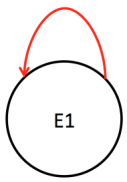

Figure 1: *Simple feedback loop with an entity supplying its input from its output.*

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.

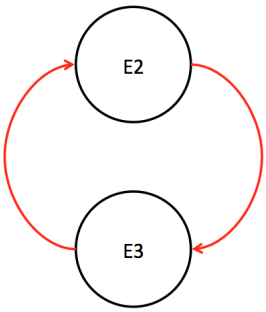

Figure 2: *Feedback loop between two entities, with each output being transformed by the other entity before becoming an input.*

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 [35–37].

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 [38].

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 [39].

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 [40].

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 [41].

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 [42], herding among traders [43], and fundamentalists [44].

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.

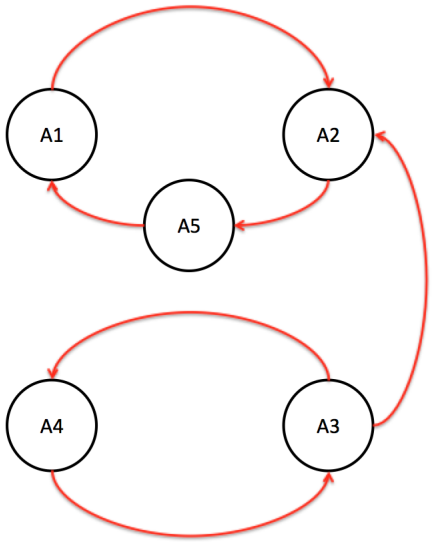

Figure 3: *Visualisation of the allowed interactions between five different agents, represented by black circles, with red lines representing the message paths.*

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 [45, 46]:

- 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 [47].

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 [48], 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.

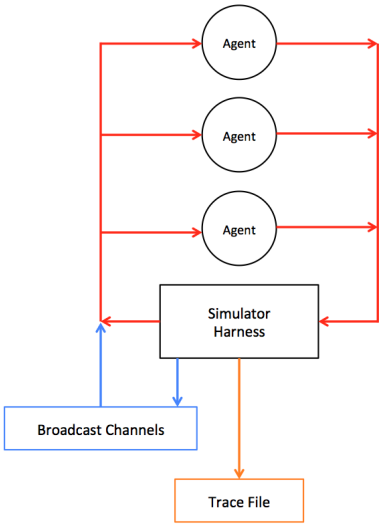

Figure 4: *Structure of an InterDyne simulation containing three agents. The messages sent by the agents to the harness and from the harness are coloured red. The trace file and messages sent to the trace file are coloured orange. The messages sent from the harness to the broadcast channels and then back to the agents are coloured blue [49].*

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.

Figure 6:

Running an InterDyne experiment is done by calling the simulator with relevant inputs, an example example of this is shown in Fig. 7.

```
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 7: *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. [36], 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.

This section how a two-views approach can be created and problems associated with the design of this method.

3.1 Method for Achieving a Two-Views Approach

A two-views approach, where recurrence relations define the formalisation of an agent-based model, must consist of a connection between the two models. This connection must be able to show that the recurrence relations and the agent-based model represent each other.

A translation between between a set of recurrence relations representing a system to an agent-based model representing the same system. This will give a connection between the two models which is able to demonstrate the equivalence of the recurrence relations and the agent-based model, hence allowing for the former to act as the formalism for the latter.

An important consideration in creating a translation between the two methods is, which direction this translation should take. For the purpose of defining the formalisation of an agent-based model and hence increasing its acceptance for a reader, a recurrence relations to agent-based model trajectory is preferred. This ordering will allow a read to start with a formalised description and translate this formalisation to cover an agent-based model, as opposed to starting with an unformalised model and translating to a formalised one.

The next consideration is how this translation will be achieved. Inspiration can be drawn from other works investigating the design of translations, such as Refs. [50], [51] and [52]. These three papers discuss translations between program representations in a way in which, the final representation will faithfully represent the initial. The approach taken by these papers is to increment the translation in step-by-step manner, where each step slightly transforms the previous to be closer to the final form. The benefit of this method is that if each step on the translation can be proven to be correct then the whole translation can be seen as correct.

A two-views approach can hence be created by translating in a step-by-step fashion from a recurrence relation language to an agent-based model. The agent-based model used in this translation is InterDyne and the recurrence relation language used is discussed in Section 4. InterDyne was selected due to a number of factors, most importantly InterDyne was designed from the bottom up to model and investigate emergent behaviour with the financial markets, with research already being conducted using it to this end [12], this software was also a convenient choice due to access to the source code and local expertise.

3.2 Review of Similar Work

The idea of using recurrence relations to give a formal definition to an agent-based model is not a novel one and falls under the general heading of using formal systems to prove properties of agent-based models. Work in this area has covered formal using game theory, discrete systems, classifier agents and equation-based models [53]. The use of recurrence relations falls under the classification of an equation-based model, these have been used in a number of papers looking to give a formalisation to agent-based models. Here three papers, Refs [54], [36] and [46], looking at this area will be reviewed and discussed, along with DEVS formalism introduced in Ref. [55].

Before reviewing these papers, it is worth considering the more obvious question of does a formalism need to be given by another modelling technique to allow for the analyses of an agent-based model? Reference [53] present a method formalism for the analysis of agent-based models, looking in particular at performance.

This paper looks to prove findings from agent-based models, in particular the focus is on existing agent-based software such as Swarm, Repast 3 and others. This paper notes that these pieces of software use a modular imperative architecture with factored agents, spaces, a scheduler, logs, and an interface. Hence this paper presents a theoretical formalism for analysing a modular imperative agent-based model.

This this formalism is presented using an abstract machine specification of the modular imperative agent-based modelling architecture. A Random Access Stored-Program (RASP) architecture was chosen for the abstract machine, this architecture had been created in an earlier paper (Ref. [56]) and

was chosen due to its similarities to real computer systems. This is opposed to other options including Turing machines and λ-calculus, as they are less well matched with real computer systems.

One of this papers motivations for using this technique over an equation-based model formalisation, is that the authors feel that this formulation of the agent-based model in terms of an equation-based model would be unnatural and possibly unrecognisable to those accustomed to agent-based models. This consideration is not as important to the work presented in this dissertation, since here the aim is to familiarise those accustomed with equation-based models to agent-based models and not the reverse. The same argument applies to the use of RASP over alternatives such as λ-calculus, this dissertation is more interested in creating a formalism which is relatable to economist over computer scientists.

The first paper to be examined is Ref. [54], this paper considers the formalisation of agent-based models by creating their implementation guided by equation-based models, an (equation-based model)-oriented design.

This paper looks at an example with interacting entities with given strategies, however these strategies could change upon interactions. In this example the equation-based model took a probabilist and macro-view approach where the agent-based model, due to its nature, to a micro-view approach. The analyses of these two models showed that though the systems appeared to be the same the micro-view approach could create slightly different outcomes depending on the number of agents chosen. This difference comes from the fact that the individual agents in the agent-based model were created to match a macro-level description given by the equation-based model, however due to the autonomous nature of the agents only their general behaviour can match and in certain circumstances even this did not correctly produce the behaviour.

This paper concludes summarising the benefits of this approach in validating an agent-based model, here an agent-based model is known to be a faithful representation of the system as long as the equation-based model is a faithful representation. However the question of whether the age-based model is correctly implemented still rests with that model.

The main difficulties faced by this paper appear to be the transformation from a macro-level view (equation-based model) to a micro-level (agent-based model) view, this is particularly apart due to the statistical nature of the equation-based model. Though they succeeded in creating a formalisation of their model the differences in results between runs still raises questions over the equality between the two view points. To avoid this issue, taking a more micro-level focussed equation-based model should suffice.

The next paper is Ref. [36], this paper is based on a similar principle to this project in that it seeks to bridge the conceptual gap around agent-based models in economics. It tackles this problem by introducing an equation-based formalism to describe both agent-based models and analytical models, though of interest here is their work on former.

The agent-based model is formalised by specifying how each agent's state within the model changes between each time step. This is done by defining agent attributes using vectors of state variables and then applying time-indexed difference equations to specify each agents state change. This allows for expression of the agent-based model in terms of difference equations, which they then use to look at parameter estimation.

Their work shows the ability to prefer estimations on agent-based models, counter to some criticism that has been levelled at agent-based models. Some difficulty in this approach can come from agent-based models with complicated algorithmic behaviour within the agents, specifying these models as difference equations can be challenging task. The work in this paper does not directly suffer from this problem as all systems are first described as recurrence relations before being translated into agent-based models, meaning that the equation view will only be as complex as the original writer intends.

Th last paper to be looked at is Ref. [46], this paper investigates relating Forrester's System Dynamics (SD) model to an agent-based model. System dynamics models consist of a set of difference equations, so this can be seen as a comparison between difference equations and agent-based models.

This paper presents a formal specifications for both the Forrester's and agent-based models, these specifications are then compared to show their equivalence. Was equivalence has been proven the specification of a Forrester's model can be used to derive an agent-based model that is an correct representation of the same system.

This technique its applied to an SIR epidemic model, creating an Forrester's and agent-based model representation. These two descriptions of the system are then evaluated and have their results compared. The findings show that though both models show similar behaviour, there is not an exact match between them. This is due to the probabilistic nature of the system being modelled and the different approach that an agent-based model takes to these kind of systems.

This paper, though showing equivalence, does so in a more abstract manner then a direct translation between the two models. The difference in experimental results from the two models is down to the use of describing the system in a probabilist fashion, this can be avoided by describing a system in a more micro-level view.

A discussion relating to the formalism of agent-based models would be amiss with out mentioning the Discrete Event System Specification (DEVS) [55]. DEVS is a theoretical formalism designed for analysing complicated simulations, which it does by describing simulations as hierarchical Moore machines, with additional timing information added to the state transitions. DEVS has a few variants including classic DEVS (this is the core abstraction), coupled DEVS (allows for hierarchical nesting of the Moore machines [57]) and Dynamic Structuring DEVS (attempts to allow for dynamic behaviour [58]).

DEVS is able to provide a theoretical foundation for analyses of a number of critical computational properties of agent-based models, including properties such as decidability. In theory DEVS has the ability to represent any computational model, including agent-based models, as following the Church-Turing thesis any computing system as complex as a Moore machine (which DEVS is) can be used to calculate any function. However though DEVS is capable of representing an agent-based model it is not necessarily the best approach. Despite the work done in Ref. [58], DEVS is still not an ideal match for models that can dynamically behaviour, such as agent-based models where agent behaviour can be dynamically generated. Classic DEVS imposes boundaries between states and behaviours making dynamic behaviour difficult to represent, where as Dynamic Structuring DEVS introduces new constraints that have a similar effect in limiting utility. DEVS also does not include conventions of agent-based models that impact the limits of their computational properties, making it again difficult to use DEVS as a formal basis for agent-based modelling [53].

3.3 Challenges with the Two-Views Approach

The approach taken in this project of creating a step-by-step translation between a recurrence relation description of a system to a agent-based model description on the same system has a number of challenges associated with it. In this section the difficulties with this project will be described.

As noted in Section 3.2, when looking at work done by other researchers in this area, a main challenge was how to model systems that have no counter part in a agent-based model. This problem exists mostly due to modelling probabilist systems and macro-level systems, which are counter to the view an agent-based model takes. In this project however the focus will be on systems which are modelled at the micro-level and without probabilist aspects, though stochastic aspects will be allowed within the micro-level view.

An problem overshadowing most aspects of this project is that the two paradigms being translated between are completely different and in some ways opposing. Recurrence relation models take a analytic approach where as agent-based models take a dynamic one.

The formulation of how information is exchanged within these two models is also in contrast. Recurrence relations take a static view with information passed via function calls, opposed to the agent-based model approach of dynamically sending the information between agents using message passing. This represents a difference in the way information is seen in these models, in recurrence relations information is freely available and the equation wishing to use it simply has to make a function call.

Where as in agent-based modelling information is given by its creator to only other agents selected by its creator, hence agents can not use information which has not been already sent to them.

The agent-based model selected for use, InterDyne, operates by making every agent take in an infinite list of their inputs and output an infinite list of their outputs. Creating this view of the system from a recurrence relation model, which is created as a snapshot of a current time step is challenging.

In InterDyne both direct and broadcast messages exist, these two messaging types can be seen as the use of private and public date in that order. Recurrence relations do not have an inbuilt method of distinguishing between different types of data as all information is passed by direct function calls, reconciling these two view points creates another challenge.

In recurrence relations information from any previous time step can be easily accessed, where as in InterDyne information is only available for the time step at which it was received. This creates the idea of time limited information with InterDyne that is not present within a recurrence relations model.

Agent-based models have name control, indicating the different agents and functions within them. However introducing name control to recurrence relations, that have no sense of agents, in a why in which it is tangible to the agent-based model is a challenge.

The introduction of sending information in the form of a messages as a concept, to recurrence relations poses its own difficulties. Recurrence relations information is passed directly as values, where as messages contain information such as message type, and to and from fields.

There has been criticism that agent-based models splinter a model, finding a way to split the recurrence relations into agents with out splintering the model creates another challenge.

This approach is to create a series of small steps between a recurrence relation model and an agent-based model, the smaller the steps are the easier it is for a reader to follow the process. However a challenge is finding how to create these small steps as it may often be easier to create large but less comprehensible steps.

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. 8, 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 8: *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. 9, 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 9: *Layout of a simulation.*

An experiment is written as a function call, with a certain set of arguments. Figure 10 shows a experiment that will return the value of the recurrence relation i_f1 at time 3.

```
i_f1 3
```

Figure 10: *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. 11. There is no defined limit to the amount of initial conditions that can be defined.

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

Figure 11: *Definition of two initial conditions, q and k.*

A simple recurrence relation can be written as shown in Fig. 12, 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 12: *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. 13. 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 13: *Recurrence relation containing pattern matching for initial conditions.*

An if statement, shown in Fig. 14, 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 14: *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 15 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 15: *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. 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: *A simulation in the bespoke recurrence relation language.*

5 Design of a Translation from Recurrence Relations to InterDyne

This section details the design of a step-by-step translation between the previously described bespoke recurrence relation language and the InterDyne simulator. This is broken down into eight individual steps, each of these steps alters the output of the previous step to resemble more closely an InterDyne simulation. Each of these steps should be a small enough change from the previous iteration that the transformation, which they preform is digestible.

The eight steps required for this translation are to introduce:

1. Infinite list output.
2. Grouping into agents.
3. A global output list.
4. Agent input lists.
5. Targeted information passing.
6. A simulator harness.
7. Runtime arguments.
8. Message communication.

These eight steps translate the bespoke recurrence relation model into that of an InterDyne model, with some limited functionality. This translation also does not use the functionality of broadcast messages within InterDyne, all information passing within the recurrence relations is considered to be private and is hence sent as a one-to-one message. This translation does support expanded recurrence relation that take more then one argument, as well as recurrence relations that are first degree or higher.[5]

These steps will be shown in this section by translating an example written in the bespoke recurrence relation language, shown in Fig. 17. This example consists of four recurrence relations (i_f1, j_f2, j_f2 and k_f1) split into three groups (i, j and k). Each of these relations uses the allowed pattern

[5] A first-degree recurrence relation references the previous time step $(t-1)$, a second-degree relation references the one prior to this $(t-2)$ and so on.

matching to define their initial conditions in terms of q and k, which are defined in the section *init*. The experiment, defined under *main*, is to return the value of the relation i_f1 at the time step 3.

```
main
i_f1 3

init
q = 4
k = 16

where
i_f1 0 = q
i_f1 t = (i_f1 (t-1)) + (j_f1 (t-2))
j_f1 0 = k
j_f1 t = cond (t<0) (k_f1 (t-1)) (25+((j_f1 (t-1))*(i_f1 (t-1))) + (j_f2 (t-1)))
j_f2 0 = k+q
j_f2 t = cond (t<3) 27 ((j_f1 (t-1)) + 27)
k_f1 0 = 0
k_f1 t = k + q + (j_f1 (t-1))
```

Figure 17: *Example of an experiment written in the bespoke recurrence relation language, using a set of four recurrence relations.*

5.1 Infinite List Output

In InterDyne each agent outputs an infinite list containing the outputs of that at each time step. Here each recurrence relation is changed to output its values as an infinite list in time.

For this to be achieved each recurrence relation is wrapped into a parent function, denoted by adding *_list* to the end of the relations name. This parent function returns a sub function labeled *_createlist*, function takes in a function and a time marker and returns an infinite list containing the output of that function at each time. *_createlist* does this by recursively calling itself adding one time step to its input each time, the value of the function applied to the current time step is added to the front of this list using :.

Since the recurrence relations are now contained within parent functions (that do not take time as an input) which are outputting infinite lists, to obtain a value at a certain time step these list have to be indexed into. This indexing using time now replaces the previous method of calling a relation with time as an argument. Indexing is done with the bang operator (!), this applied to a list and followed by a value will return the numbered item in the list. When numbering the list zero is considered to be the first item so $[1, 2, 3]!1$ would return a value of 2. Therefore all calls within other functions are changed to this new notation as well as the call in main for the experiment. Once these changes have been applied Fig. 17 will be translated into Fig. 18.

```
main
i_f1_list ! 3

init
q = 4
k = 16

where
i_f1_list = _createlist i_f1 0
            where
            _createlist f t = (f t):(_createlist f (t+1))
            i_f1 0 = q
            i_f1 t = (i_f1_list ! (t-1) ) + (j_f1_list ! (t-2))
j_f1_list = _createlist j_f1 0
            where
            _createlist f t = (f t):(_createlist f (t+1))
            j_f1 0 = k
            j_f1 t = cond (t<0) (k_f1_list ! (t-1))
                          (25 + ((j_f1_list ! (t-1)) * (i_f1_list ! (t-1))) + (
    j_f2_list ! (t-1)))
j_f2_list = _createlist j_f2 0
            where
            _createlist f t = (f t):(_createlist f (t+1))
            j_f2 0 = k+q
            j_f2 t = cond (t<3) 27 ((j_f1_list ! (t-1)) + 27)
k_f1_list = _createlist k_f1 0
            where
            _createlist f t = (f t):(_createlist f (t+1))
            k_f1 0 = 0
            k_f1 t = k + q + (j_f1_list ! (t-1))
```

Figure 18: *Figure 17 translated to contain relations that produce infinite lists throughout time and functionality added for relations to index into these lists.*

5.2 Grouping into Agents

InterDyne contains agents which are the entities that interact, here agents are created by grouping together recurrence relations in the same assigned group.

This is done by creating a wrapper function which contains the previously created *_list* functions. Each wrapper, which is labeled as *group_wrapper*, contains the *_list* functions assigned to the same group. These wrappers output an infinite list, each item within this list is another list containing the outputs of the internal *_list* functions at that time step. Therefore a wrapper containing two functions, *a_list* and *b_list*, would produce an infinite list of the form,
[[*a_list*!0, *b_list*!0], [*a_list*!1, *b_list*!1], .., [*a_list*!∞, *b_list*!∞]]. This is done using the internal function *transpose*, this function takes a list of lists and returns a list containing lists, where the first list is a list of the heads of the initial lists and the next list is a list of the heads of the remainder of the initial list and so on. An example of the use of transpose is *transpose* $[[1,2,3],[4,5,6]]$, which returns $[[1,4],[2,5],[3,6]]$.

Since the *_list* functions are now contained within a wrapper function calls to these functions must now be calls to the wrapper function which they are part of. The exception to this rule is for functions that are part of the same wrapper, these can still make direct calls to *_list* functions that share the same group. Since the wrapper outputs an infinite list in time, first this list must be indexed to the right time step and then this list must be indexed for the right internal function. This will change the notation of all relation calls by other recurrence relations and the recurrence relation call listed in *main*. These changes will translate the example in Fig. 18 to that shown in Fig. 19.

```
main
(i_wrapper ! 3) ! 0

init
q = 4
k = 16

where
i_wrapper = (transpose [i_f1_list])
            where
            i_f1_list = _createlist i_f1 0
                        where
                        _createlist f t = (f t):(_createlist f (t+1))
                        i_f1 0 = q
                        i_f1 t = (i_f1_list ! (t-1) ) + (((j_wrapper ! (t-2)) ! 0)
j_wrapper = (transpose [j_f1_list , j_f2_list])
            where
            j_f1_list = _createlist j_f1 0
                        where
                        _createlist f t = (f t):(_createlist f (t+1))
                        j_f1 0 = k
                        j_f1 t = cond (t<0) ((k_wrapper ! (t-1)) ! 0)
                                            (25 + ((j_wrapper ! (t-1)) ! 0) * ((i_wrapper ! (t
    -1)) ! 0) + (j_f2_list ! (t-1)))
            j_f2_list = _createlist j_f2 0
                        where
                        _createlist f t = (f t):(_createlist f (t+1))
                        j_f2 0 = k+q
                        j_f2 t = cond (t<3) 27 ((j_f1_list ! (t-1)) + 27)
k_wrapper = (transpose [k_f1_list])
            where
            k_f1_list = _createlist k_f1 0
                        where
                        _createlist f t = (f t):(_createlist f (t+1))
                        k_f1 0 = 0
                        k_f1 t = k + q + ((j_wrapper ! (t-1)) ! 0)
```

Figure 19: .

5.3 A Global Output List

In InterDyne a simulator harness is used to mediate the messages sent between different agents, here the first step for the creation of this harness will be added in the form of a global output list.

This is done by creating a function, *s_outputs*, that collects the outputs from all the agents and returns an infinite list containing these outputs. Because InterDyne has a circler structure for its message passing through time, requiring the agents to take a message at the beginning of every time step, a initial set of messages is added to the front of this list. These initial values are set as zero to represent a null value, but to still give an agent a message to consume on the first time step.

This new list of all agents outputs can now be used to refer to values produced by other agents. The list produced by *s_outputs* is an infinite list in time, containing lists for each time step which contain a list for each agent containing all the values for that time step produced by that agent.

Translating Fig. 19 to contain this global output list and to index into it for retrieving values from other agents produces Fig. 20.

```
main
((s_outputs!(3+1))!0)!0

init
q = 4
k = 16

where
s_outputs  = niloutputs : (transpose [i_wrapper, j_wrapper, k_wrapper])
             where
             niloutputs = [[nullvalue], [nullvalue, nullvalue], [nullvalue]]
             nullvalue = 0
i_wrapper = (transpose [i_f1_list])
            where
            i_f1_list = _createlist i_f1 0
                        where
                        _createlist f t = (f t):(_createlist f (t+1))
                        i_f1 0 = q
                        i_f1 t = (i_f1_list ! (t-1) ) + j_f1_2last_timestep)
                                 where
                                 j_f1_2last_timestep = (((outputs!(t-1))!1)!0)
j_wrapper = (transpose [j_f1_list, j_f2_list])
            where
            j_f1_list = _createlist j_f1 0
                        where
                        _createlist f t = (f t):(_createlist f (t+1))
                        j_f1 0 = k
                        j_f1 t = cond (t<0) k_f1_last_timestep
                                      (25 + ((j_f1_list ! (t-1)) * i_f1_last_timestep) +
    (j_f2_list ! (t-1)))
                                 where
                                 i_f1_last_timestep = (((outputs!t)!0)!0)
                                 k_f1_last_timestep = (((outputs!t)!2)!0)
            j_f2_list = _createlist j_f2 0
                        where
                        _createlist f t = (f t):(_createlist f (t+1))
                        j_f2 0 = k+q
                        j_f2 t = cond (t<3) 27 ((j_f1_list ! (t-1)) + 27)
k_wrapper = (transpose [k_f1_list])
            where
            k_f1_list = _createlist k_f1 0
                        where
                        _createlist f t = (f t):(_createlist f (t+1))
                        k_f1 0 = 0
                        k_f1 t = k + q + j_f1_last_timestep
                                 where
                                 j_f1_last_timestep = (((outputs!t)!1)!0)
```

Figure 20: *Figure 19 translated to contain the global output list, s_outputs, and to use this list to retrieve values from other agents.*

5.4 Agent Input Lists

In InterDyne an agent is only able to access information sent to it in the form of messages. These messages take one time step to arrive at the agent and hence have the effect of only allowing the agent to use information sent to it one time step ago (if a message is sent at $t-1$ the agent will receive it for use at t). Here the beginnings of message passing will be added, by allowing each wrapper to take inputs that contain the outputs list and then internally forcing the wrappers to only use the information in this list from the previous time step.

For this functionality to be added a number of aspects from the previous translation need to be altered, these are: the output list, the wrappers, and the creation of a new type of wrapper.

The output list, *s_outputs*, will still produce an infinite list in time containing the outputs of all the wrappers, however to do this it now has to give inputs to the wrapper functions. Each wrapper function will receive two inputs, the outputs list and an initial time value. The initial time value is zero, this initialises recursion within the wrappers that will be discussed later. This will lead to *s_outputs* similar to that shown in Fig. 21.

```
s_outputs  = niloutputs : (transpose [i_wrapper s_outputs 0, j_wrapper s_outputs 0,
    k_wrapper s_outputs 0])
```

Figure 21: *Example of part of the s_outputs form after addition of inputs for the wrapper functions.*

The wrapper functions now takes two inputs, a list of all the outputs from all the wrappers (*inputs*) and a time stamp (t). When first called by *s_outputs* this list is complete and the time stamp is equal to zero. The head of the *inputs* list contains just the information produced during the last time step, in this case this is the *niloutputs*, therefore to restrict the wrapper to just use the information from the last time step only the head can be used. However has the time steps increase this will no longer be true, so to create the output list required from the wrapper, the wrapper recursively calls its self using the tail of the inputs and increasing the time stamp by one. This means on the next call the head of the list will represent the allowed information for that time step, as this will be the head of the tail of the list, and so on as the time increases. The output list created by the agent now achieved by producing a list containing the values of the logic for that time step and adding this to the front of the recursive list. An wrapper written in this way would have a similar for to that shown in Fig. 22

```
i_wrapper inputs t = outbound_messages_this_timestep : (i_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = hd inputs
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [(logic1 t), (logic2 t)]
                     logic1 t = function (inbound_messages_this_timestep)
                     logic2 t = function (inbound_messages_this_timestep)
```

Figure 22: *Example of part of a wrapper function form after addition of inputs for the wrapper functions.*

As mentioned a wrapper now only has access to information sent at the previous time step, $t-1$, however some logic requires the use of information from time steps prior to this, such as $t-2$. Given a wrapper access to this information can be done in two ways, either the wrapper can receive the information and then store it internally until use or the information can be sent to another wrapper which then sends it to the correct wrapper for the time step of use. The second method, though it introduces a new wrapper function, has a the benefiting of allowing dynamic analyses of message passing to show all the information available of any set time step to any wrapper, hence this method is the one used. This new wrapper will be named in the format $functionname\_delay$, where $functionname$ is the name for the recurrence relation value that it is delaying, such $jf1$. This wrapper will follow the same rules as any other wrapper but will containing very minimal internal logic. This wrapper will use one input from another wrapper, this will be the for the value which it is delaying. This wrapper will then output a value for each time step it is delaying for, so for a call for a value at $t-2$ the wrapper will output two values at each time step. The first value in the output list will be the value of just output by the other wrapper and the last value will be the value for the previous time step. This wrapper can be seen to work by constantly replacing the first value with the current value of the outputting wrapper and then moving the previous value to the next position in the list, when the value gets to the last position

in the list it is read by the originally intended wrapper. A delay wrapper will have a form similar to that shown in Fig. 23. Since this wrapper is treated the same as any other, it will also be added to the outputs lists and be part of the inputs.

```
jf1_delay inputs t = outbound_messages_this_timestep : (jf1_delay future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = hd inputs
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [(jf1_current), (jf1_delayed)]
                     jf1_current = value from initial wrapper
                     jf1_delayed = previous jf1_current value
```

Figure 23: *Example of a delay wrapper that repeatedly sends its self a value till it passes this on to the intended recipient.*

In this translation all information is passed as an input to the wrapper so even internal functions are referred to through the input list, this list infinite lists head is used at each time step. The head of this is another list with each item referring to a different wrapper, with the first wrapper being item zero, this list is therefore indexed to return the values associated with that wrapper. Once indexed to specify a wrapper another list is returned that contains the values from each logic function within the wrapper, again with the first logic function being item zero. This double indexing of the head allows for the value of any logic function from the previous time step to be accessed. Translating Fig. 20 in this way will produce Fig. 24.

```
main
((s_outputs!(3+1))!0)!0

init
q = 4
k = 16

where
s_outputs  = niloutputs : (transpose [i_wrapper s_outputs 0, j_wrapper s_outputs 0,
    k_wrapper s_outputs 0, jf1_delay s_outputs 0 ])
             where
             niloutputs = [[nullvalue], [nullvalue, nullvalue], [nullvalue], [nullvalue,
     nullvalue]]
             nullvalue = 0
i_wrapper inputs t = outbound_messages_this_timestep : (i_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = hd inputs
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [(i_f1 t)]
                     i_f1 0 = q
                     i_f1 t = cond (t<0) 0
                              ( ((inbound_messages_this_timestep!0)!0) +((
    inbound_messages_this_timestep!3)!1))
j_wrapper inputs t = outbound_messages_this_timestep : (j_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = hd inputs
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [(j_f1 t), (j_f2 t)]
                     j_f1 0 = k
                     j_f1 t = cond (t<0) k_f1_last_timestep
                              (25 + (j_f1_last_timestep * i_f1_last_timestep)+
    j_f2_last_timestep)
                              where
                              i_f1_last_timestep = ((inbound_messages_this_timestep!0)
    !0)
                              j_f1_last_timestep = ((inbound_messages_this_timestep!1)
    !0)
                              j_f2_last_timestep = ((inbound_messages_this_timestep!1)
    !1)
                              k_f1_last_timestep = ((inbound_messages_this_timestep!2)
    !0)
                     j_f2 0 = k+q
                     j_f2 t = cond (t<3) 27 (j_f1_last_timestep + 27)
                              where
                              j_f1_last_timestep = ((inbound_messages_this_timestep!1)
    !0)
k_wrapper inputs t = outbound_messages_this_timestep : (k_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = hd inputs
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [k_f1 t]
                     k_f1 0 = 0
                     k_f1 t = k + q + j_f1_last_timestep
                              where
                              j_f1_last_timestep = ((inbound_messages_this_timestep!1)
    !0)
jf1_delay inputs t = outbound_messages_this_timestep : (jf1_delay future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = hd inputs
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [(jf1_current), (jf1_delayed)]
                     jf1_current = ((inbound_messages_this_timestep!1)!0)
                     jf1_delayed = ((inbound_messages_this_timestep!3)!0)
```

Figure 24: *Translation of Fig. 20 to allow wrappers to access information through input values, and to limit the information given to only that from the previous time step.*

5.5 Targeted Information Passing

The next step in creating message passing is to have the wrappers send messages addressed to each other instead of just producing a blanket output for any other wrapper. In InterDyne a message has three main components, these are a sender, a receiver and a value. In this setup a wrapper can only access a value if it is sent directly to it, therefore if more then one wrapper uses a value than that many messages need to be sent. Here these three item messages will be added along with the functionality for a wrapper to distinguish which messages are for it.

To add this functionality features of the previous translation need to be changed, these are: The main function call, the wrapper functions and output list.

The changes required to the output list function (*s_outputs*) is purely related to the *nullvalue* aspect of it. The *nullvalue*, which was previously zero now has to be a list of three value messages. These three items will be stored in a tuple, in the format ($from$, *to*, *value*), for the *nullvalue* this the message list will be a single message containing zeros, [(0, 0, 0)].

The wrapper functions now output messages in the same form, so a message from the first wrapper function will contain its id which is one the targeted id and the value it wishes to send. Allow wrappers id's can be found by looking at where that wrapper appears in the order of wrappers, with the first wrapper having id one and so on. The the id of the receiving wrapper is found by analysing the call functions made by each wrapper to discern, which wrappers they are drawing information from. Once this is found messages can be created in these wrappers to send the information to the wrappers that require it. This reverse engineering as it were can be used to deduce what message need to be sent to what wrappers. An example of the form for sending a messages from a wrapper can be seen in Fig. 25.

```
1 i_wrapper inputs t = outbound_messages_this_timestep : (i_wrapper future_messages (t+1))
2                      where
3                      inbound_messages_this_timestep  = hd inputs
4                      outbound_messages_this_timestep = [ [ (1,1,logic1),(1,2,logic1)]]
5                      logic1 t = function (inbound_messages_this_timestep)
```

Figure 25: *Example of part of a wrapper that is sending two messages containing the same value. These messages come from this wrapper, which is the first wrapper (1), and are sent: to itself (1), and to the second wrapper (2).*

Though a wrapper receives all messages sent it should only use those that are sent directly to it. To do this the wrapper sorts the inbound messages to return only those with its id as the target. The sorting function can be seen in Fig. 26, this takes the head of the list, so the current time step and then uses the function *map* with the input f on it. This function applies the the function f to each item in the list it is passed and then returns the new list, for example *map hd* $[[1,2],[3,4]] = [1,3]$. The function f in this case then maps the function g across all the items in the lists that it is mapped across. The function g then uses a function called $filter$ takes a condition and a list and returns a list of all elements that matched the condition. In this case the condition is that the message tuple is sent this this wrapper ($id = 1$) or () that it is a null message from the *s_outputs* function.

```
i_wrapper inputs t = outbound_messages_this_timestep : (i_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = map f (hd inputs)
                                                       where
                                                       f xs = map g xs
                                                       g xs = filter h xs
                                                       h (a,b,c) = ((b=1) \/ (a,b,c) =
    (0,0,0))
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [ [messages]]
                     logic t = function (inbound_messages_this_timestep)
```

Figure 26: *Example of part of a wrapper that is used to sort the input off all messages and only return the ones directed at this wrapper.*

Since the format that the information is passed in has changed the experiment function in main must change as well as any calls to *inbound_messages_this_timestep* within wrappers. These calls now must access the first index to the correct message (this is again found through reverse engineering), then take the head of this list as it will be a list just containing one message and then take the third item from the tuple which is the value. To take the head and the third item the command *thd*3.*hd* is used, this is equivalent to *thd*3(*hd input*), where the command *thd*3 returns the third item in a three item tuple. Translating Fig. 24 with these steps will lead to Fig. 27.

```
main
(thd3.hd) (((s_outputs!(3+1))!0)!0)

init
q = 4
k = 16

where
s_outputs  = niloutputs :  transpose [ i_wrapper s_outputs 0 ,j_wrapper s_outputs 0 , k_wrapper s_outputs 0, jf1_delay s_outputs 0  ]
             where
             niloutputs = [[newnullvalue], [newnullvalue, newnullvalue], [newnullvalue], [newnullvalue, newnullvalue]]
             nullvalue = [(0,0,0)]
i_wrapper inputs t = outbound_messages_this_timestep : (i_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = map f (hd inputs)
                                                       where
                                                       f xs = map g xs
                                                       g xs = filter h xs
                                                       h (a,b,c) = ((b=1) \/ (a,b,c) = (0,0,0))
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [ [ (1,1,res),(1,2,res)]]
                                                       where
                                                       res = i_f1 t
                     i_f1 0 = q
                     i_f1 t = cond (t<0) 0 (i_f1_last_timestep + j_f1_2last_timestep)
                              where
                              i_f1_last_timestep  = (thd3.hd) ((inbound_messages_this_timestep!0)!0)
                              j_f1_2last_timestep = (thd3.hd) ((inbound_messages_this_timestep!3)!0)
j_wrapper inputs t = outbound_messages_this_timestep : (j_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = map f (hd inputs)
                                                       where
                                                       f xs = map g xs
                                                       g xs = filter h xs
                                                       h (a,b,c) = ((b=2) \/ (a,b,c) = (0,0,0))
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [ [ (2,4,res1),(2,2,res1),(2,3,res1)],[(2,2,res2)]]
                                                       where
                                                       res1 = j_f1 t
                                                       res2 = j_f2 t
                     j_f1 0 = k
                     j_f1 t = cond (t<0) k_f1_last_timestep
                              (25 + (j_f1_last_timestep * i_f1_last_timestep)+ j_f2_last_timestep)
                              where
                              i_f1_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!0)!0)
                              j_f1_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!1)!0)
                              j_f2_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!1)!1)
                              k_f1_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!2)!0)
                     j_f2 0 = k+q
                     j_f2 t = cond (t<3) 27 (j_f1_last_timestep + 27)
                              where
                              j_f1_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!1)!0)
k_wrapper inputs t = outbound_messages_this_timestep : (k_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = map f (hd inputs)
                                                       where
                                                       f xs = map g xs
                                                       g xs = filter h xs
                                                       h (a,b,c) = ((b=3) \/ (a,b,c) = (0,0,0))
                     future_messages                 = tl inputs
```

5.6 A Simulator Harness

As mentioned earlier InterDyne has a harness that mediates the message passing between the agents, this harness covers aspects such as driving the simulation be requiring an output from the agents, receiving and redistributing the messages and redacting the inputs to each agent so that the only receive the messages sent to them. Here a harness is added that takes control of this functionality.

To add this harness features of the previous translation need to be changed, these are: The main function call, the wrapper functions, deleting of the output list and creating a harness function.

The harness function, called *h_sim*, takes one input, which is time. The return of this function is the value of i_f1 at the time step given to the harness function, this value is found by indexing into the output list. The output list itself is now created within the harness function, along with its first set of messages *niloutputs*. The rest of the list is now *redactedoutputs*, this contains the same values as the prior rest of the list, however in this formulation each wrapper is only passed the messages directed to it. This is because the harness has token over the role of sorting the messages, from the wrapper functions. Hence the input values labeled as *group_inputs* contain just the messages sent to the specific wrapper. An example of how a harness is formatted is shown in Fig. 28.

```
h_sim x = outputfunction (x)
          where
          newnullvalue = [(0,0,0)]
          niloutputs   = listfuction (newnullvalue)
          outputs      = niloutputs : redactedoutputs
          redactedoutputs = transpose [ i_wrapper i_inputs 0 ,j_wrapper j_inputs 0]
                            where
                            i_inputs   = map (map (map (filter (h 1)))) outputs
                            j_inputs   = map (map (map (filter (h 2)))) outputs
                            h x (a,b,c) = ((b=x) \/ (a,b,c) = (0,0,0))
```

Figure 28: *Example of how the harness function h_sim is formatted for two wrappers. This function produces the output for the system and in the process mediates the passing of messages between wrappers, only allowing a wrapper to receive messages sent to it.*

Now that the *h_sim* can return the experiment value main can be changed to call this function at the correct time step, since the output list starts with a null value the time step required is $3 + 1$.

Since the *h_sim* sorts the inputs to the wrapper functions so that they are only receiving messages sent to them, the wrappers no longer need to deal with this internally. The *inbound_messages_this_timestep* in the wrapper now simply takes the head of the list to access the messages at the current time step.

Translating Fig. 27 to contain a harness function with this functionality will result in the experiment shown in Fig. 29.

```
main
h_sim (3+1)

init
q = 4
k = 16

where
h_sim x = (thd3.hd) (((outputs!x)!0)!0)
          where
          newnullvalue = [(0,0,0)]
          niloutputs   = [[newnullvalue], [newnullvalue, newnullvalue], [newnullvalue], [newnullvalue, newnullvalue]]
          outputs      = niloutputs : redactedoutputs
          redactedoutputs = transpose [ i_wrapper i_inputs 0 ,j_wrapper j_inputs 0 , k_wrapper k_inputs 0, jf1_delay jf1_inputs 0 ]
                            where
                            i_inputs   = map (map (map (filter (h 1)))) outputs
                            j_inputs   = map (map (map (filter (h 2)))) outputs
                            k_inputs   = map (map (map (filter (h 3)))) outputs
                            jf1_inputs = map (map (map (filter (h 4)))) outputs
                            h x (a,b,c) = ((b=x) \/ (a,b,c) = (0,0,0))
i_wrapper inputs t = outbound_messages_this_timestep : (i_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = hd inputs
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [ [ (1,1,res),(1,2,res)]]
                                                       where
                                                       res = i_f1 t
                     i_f1 0 = q
                     i_f1 t = cond (t<0) 0 (i_f1_last_timestep + j_f1_last_timestep)
                              where
                              i_f1_last_timestep  = (thd3.hd) ((inbound_messages_this_timestep!0)!0)
                              j_f1_2last_timestep = (thd3.hd) ((inbound_messages_this_timestep!3)!0)
j_wrapper inputs t = outbound_messages_this_timestep : (j_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = hd inputs
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [ [ (2,4,res1),(2,2,res1),(2,3,res1)],[(2,2,res2)]]
                                                       where
                                                       res1 = j_f1 t
                                                       res2 = j_f2 t
                     j_f1 0 = k
                     j_f1 t = cond (t<0) k_f1_last_timestep
                                   (25 + (j_f1_last_timestep * i_f1_last_timestep)+ j_f2_last_timestep)
                                   where
                                   i_f1_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!0)!0)
                                   j_f1_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!1)!0)
                                   j_f2_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!1)!1)
                                   k_f1_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!2)!0)
                     j_f2 0 = k+q
                     j_f2 t = cond (t<3) 27 (j_f1_last_timestep + 27)
                              where
                              j_f1_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!1)!0)
k_wrapper inputs t = outbound_messages_this_timestep : (k_wrapper future_messages (t+1))
                     where
                     inbound_messages_this_timestep  = hd inputs
                     future_messages                 = tl inputs
                     outbound_messages_this_timestep = [[(3,2,k_f1 t)]]
                     k_f1 0 = 0
                     k_f1 t = k + q + j_f1_last_timestep
                              where
                              j_f1_last_timestep = (thd3.hd) ((inbound_messages_this_timestep!1)!0)
jf1_delay inputs t = outbound_messages_this_timestep : (jf1_delay future_messages (t+1))
```

5.7 Runtime Arguments

At the current stage of this translation a form of agent-based model has been created, this agent-based model however is still missing a number of features that InterDyne has. These features include: passing the agent id to each wrapper, having input messages and time as a tuple, incrementing time within the harness, passing local state augments to the wrappers, passing run time arguments to the harness, and passing agent information to the harness. Here these features are added to the translation.

To achieve adding these features the harness, wrappers, initial conditions and main have to be changed.

The harness *h_sim* now takes three arguments, the first is still time followed by the run time arguments and then the agent information. The run time arguments, labeled as *runtime_args*, now take the place of the initial conditions defined under *init*, the list of runtime arguments define both q and k with their respective arguments. To signify an argument a data type is used, signified with the constructor *Arg*, the initial condition q would hence be given as *Arg* ("q", 4). The agent information argument, labeled as $agentinfo$, is a list containing tuples, there is one tuple for each wrapper and the tuple contains the wrappers name and an empty list, e.g. (*i_wrapper*, []). The empty list in this tuple is for broadcast channels and would be used to list the channels that this agent is subscribed to, but since no broadcast messages are being implemented this list is here to make this translation closer to a InterDyne formulation.

The harness now also bunches messages sent to each agent with a time stamp, this is done using the *zip*2 function. This function takes to list add combine each consecutive items in the list into tuples, for example $zip2\ [1,2,3]\ [4,5,6] = [(1,4), (2,5), (3,6)]$. This function is also used to pass the agent id to each agent, with the first agent being id equal one. The harness also passes the agents the arguments and a local state (which is unused), as well as adding the functionality for broadcast subscriptions, which are also unused. These changes create a *h_sim* function that is very similar to the simulator harness within InterDyne, a harness of this form is shown in Fig. 30.

```
1  h_sim x args agents = (thd3.hd) (((snd (outputs!x))!0)!0)
2                        where
3                        newnullvalue    = [(0,0,0)]
4                        niloutputs      = listfuction (newnullvalue)
5                        outputs         = zip2 [0..] (niloutputs : (redactedoutputs))
6                        redactedoutputs = transpose (map apply_to_args (zip2 [1..] agents)
     )
7                                          where
8                                          apply_to_args (agentid, (f,
     broadcastsubscriptions)) = f Emptyagentstate args (myoutputs agentid) agentid
9                                          myoutputs id = map (f id) outputs
10                                    f x (t, xs)  = (t, map (map (filter (h x))) xs)
11                                    h x (a,b,c)  = ((b=x) \/ (a,b,c) = (0,0,0))
```

Figure 30: *Example of how the harness function h_sim is formatted when using runtime arguments.*

The wrapper functions four inputs: the local state (*localstate*), the arguments (*args*), the input messages and time step (*inputs*), and the agent id (*id*). The arguments are then read in the wrapper using *getargstr* which returns the argument name when applied and *getargval* which returns the argument value when applied. The input list is unpacked using *fst* and *snd*, these return the first (time) and second (messages) items in a tuple respectively. The id passed to the wrapper is now used to make the sender in outbound messages. A wrapper function with these changes takes the form shown in Fig. 31.

```
i_wrapper localstate args inputs id = outbound_messages_this_timestep : (i_wrapper
    localstate args future_messages id)
                                      where
                                      q = (getargval.hd) (filter ((="q").getargstr) args
    )
                                      k = (getargval.hd) (filter ((="k").getargstr) args
    )
                                      inbound_messages_this_timestep  = snd (hd inputs)
                                      t                               = fst (hd inputs)
                                      future_messages                 = tl inputs
                                      outbound_messages_this_timestep = [ [ (id,1,logic)
    ,(id,2,logic)]]
                                      logic t = function (inbound messages this timestep
    )
```

Figure 31: *Example of how a wrapper function is formatted when passing the arguments: localstate, args, inputs and id.*

Since the inputs to the harness function have changed the call in *main* must also change to include these arguments, the addition of runtime arguments has also replaced the *init* section of the experiment. Translating Fig. 29 using these steps results in the experiment shown in Fig. 32.

```
main
h_sim (3+1) runtime_args agentinfo

where
runtime_args = [(Arg ("q",4)), (Arg ("k", 16))]
agentinfo = [(i_wrapper,[]), (j_wrapper, []), (k_wrapper, []), (jf1_delay, [])]
h_sim x args agents = (thd3.hd) (((snd (outputs!x))!0)!0)
                      where
                      newnullvalue = [(0,0,0)]
                      niloutputs = [[newnullvalue], [newnullvalue, newnullvalue], [
    newnullvalue], [newnullvalue, newnullvalue]]
                      outputs = zip2 [0..] (niloutputs : (redactedoutputs))
                      redactedoutputs = transpose (map apply_to_args (zip2 [1..] agents)
    )
                                        where
                                        apply_to_args (agentid, (f,
    broadcastsubscriptions)) = f Emptyagentstate args (myoutputs agentid) agentid
                                        myoutputs id = map (f id) outputs
                                  f x (t, xs)  = (t, map (map (filter (h x))) xs)
                                  h x (a,b,c)  = ((b=x) \/ (a,b,c) = (0,0,0))
i_wrapper localstate args inputs id = outbound_messages_this_timestep : (i_wrapper
    localstate args future_messages id)
                                    where
                                    q = (getargval.hd) (filter ((="q").getargstr) args
    )
                                    k = (getargval.hd) (filter ((="k").getargstr) args
    )
                                    inbound_messages_this_timestep  = snd (hd inputs)
                                    t                               = fst (hd inputs)
                                    future_messages                 = tl inputs
                                    outbound_messages_this_timestep = [ [ (id,1,res),(
    id,2,res)]]
                                                                      where
                                                                      res = i_f1 t
                                    i_f1 0 = q
                                    i_f1 t = cond (t<0) 0 (i_f1_last_timestep +
    j_f1_2last_timestep)
                                             where
                                             i_f1_last_timestep = (thd3.hd) ((
    inbound_messages_this_timestep!0)!0)
                                             j_f1_2last_timestep = (thd3.hd) ((
    inbound_messages_this_timestep!3)!0)
j_wrapper localstate args inputs id = outbound_messages_this_timestep : (j_wrapper
    localstate args future_messages id)
                                    where
                                    q = (getargval.hd) (filter ((="q").getargstr) args
    )
                                    k = (getargval.hd) (filter ((="k").getargstr) args
    )
                                    inbound_messages_this_timestep  = snd (hd inputs)
                                    t                               = fst (hd inputs)
                                    future_messages                 = tl inputs
                                    outbound_messages_this_timestep = [ [ (id,4,res1)
    ,(id,2,res1),(id,3,res1)],[ (id,2,res2)]]
                                                                      where
                                                                      res1 = j_f1 t
                                                                      res2 = j_f2 t
                                    j_f1 0 = k
                                    j_f1 t = cond (t<0) k_f1_last_timestep
                                                  (25 + (j_f1_last_timestep *
    i_f1_last_timestep)+j_f2_last_timestep)
                                             where
                                             i_f1_last_timestep = (thd3.hd) ((
    inbound_messages_this_timestep!0)!0)
                                             j_f1_last_timestep = (thd3.hd) ((
    inbound_messages_this_timestep!1)!0)
                                             j_f2_last_timestep = (thd3.hd) ((
    inbound_messages_this_timestep!1)!1)
                                             k_f1_last_timestep = (thd3.hd) ((
    inbound_messages_this_timestep!2)!0)
                                    j_f2 0 = k+q
                                    j_f2 t = cond (t<3) 27 (j_f1_last_timestep + 27)
                                             where
                                             i_f1_last_timestep = (thd3.hd) ((
```

5.8 Message Communication

The last part of the translation that is needed to make this experiment comparable with InterDyne is the addition of the message type. This type is used to signal that a set of data is a message and to encompass the information of the sender, receiver and data being sent. Here the use of this message type is added creating a experiment equivalent to a reduced version of InterDyne.

To achieve this part of the translation both the wrappers and the harness will have to be changed.

The main aspect of the wrappers transformation is the addition of this type into their output messages, instead of sending a message of the form $(from, to, value)$ they will now send messages of the form $Message\ (from, to)\ [(Arg(\text{“}fun\_name\text{”}, value))]$. The key word $Message$ is the type constructor and takes two values, the first is a tuple containing the from and to information and the second is a list of argument values. Arguments are signified with the constructor Arg, which takes a tuple with the arguments name (in this case the arguments are the internal functions so name for example could be i_f1) and the value associated with that argument (so the result of i_f1 at that time step). A wrappers output at each time step will now be a list of messages written in this form.

With this new message form the assigning of message values to variables within the wrapper can be more generalised. Previously this was hard coded using reverse engineering to know where in the list the message would appear, since there were no identifiers in the messages to deduce what data it held. Now that the argument name is passed with the data this can be used to assign the data without needing to know where in the list of messages it appears. This is done by first filtering the messages to return only those from the sender of interest, then this is is turned into a list of the arguments from these messages, the new functions *getmsgfrom*, *getmsgargs* and *concat* are used these return the from ids from the messages, return the arguments from the messages and turn a list of lists into just a list, respectively. Once the list of arguments is returned this can then be filtered to return the required value using *getargstr* and *getargval* which return the name of the argument and the value of the argument respectively. An example of this process is shown in Fig. 33.

```
msgs_from_i        = filter ((=1).getmsgfrom) inbound_messages_this_timestep
args_from_i        = concat (map getmsgargs msgs_from_i)
i_f1_last_timestep = (getargval.hd) (filter ((="i_f1").getargstr) args_from_i)
```

Figure 33: *Example of the filtering process for returning message values, here the value from function i_f1 sent by wrapper one is being searched for, when it is found it will be saved to the variable $i\_f1\_last\_timestep$.*

The harness has also been altered to make it more general, allow for this new message type and make it more similar to that of the harness in InterDyne, this is shown in Fig. 34. This harness creates an output list (*outputs*) where each item is a tuple of a time stamp and all the messages sent at that time. This list is created by adding an tuple for time zero with no messages to the head of the *timed_transposed_msgs* list, this list is a list of tuples with a time stamp and all the messages at that time, after time zero (for example this list will look like $[(1, t1\_outputs), (2, t2\_outputs), ..])$. This is created by using the $zip2$ function to add a time stamp to the list containing lists of messages at each time (*transposed_msgs*). This list is created from the *allmessages* list, this is a finite list containing a tuple of an agent id and an infinite list all the messages sent, for each agent, this is created from the $agentinfo$ passed to the harness. The *apply_to_args* function is used to turn the input information of the different agents into this list of outputs.

```
h_sim x runtime_args agentinfo = (getargval.hd.getmsgargs) (((snd (outputs!x))!0)!0)
                                 where
                                 outputs = (0,[]):timed_transposed_msgs
                                 timed_transposed_msgs = zip2 [1..] transposed_msgs
                                 transposed_msgs = transpose allmessages
                                 allmessages = map apply_to_args (zip2 [1..] agentinfo)
                                               where
                                               apply_to_args (agentid, (agentfn,
    broadcastsubscriptions)) = agentfn Emptyagentstate runtime_args (redacted_msgs
    agentid outputs) agentid
                                               redacted_msgs id outputs = map (f id)
    outputs
                                               f x (time, xs)               = (time,
    concat (map (filter (h x)) xs))
                                               h x (Message (f,t) args)     = (t=x)
```

Figure 34: *Simulator harness for translation comparable to InterDynes simulator harness.*

After these translations are applied to the experiment in Fig. 35 they create the experiment shown in Fig. 35.

```
main
h_sim (3+1) runtime_args agentinfo

where
runtime_args = [(Arg ("q",4)), (Arg ("k", 16))]
agentinfo = [(i_wrapper,[]), (j_wrapper, []), (k_wrapper, []), (jf1_delay, [])]
h_sim x runtime_args agentinfo = (getargval.hd.getmsgargs) (((snd (outputs!x))!0)!0)
                                 where
                                 outputs = (0,[]):timed_transposed_msgs
                                 timed_transposed_msgs = zip2 [1..] transposed_msgs
                                 transposed_msgs = transpose allmessages
                                 allmessages = map apply_to_args (zip2 [1..] agentinfo)
                                               where
                                               apply_to_args (agentid, (agentfn, broadcastsubscriptions)) = agentfn Emptyagentstate runtime_args (redacted_msgs agentid outputs) agentid
                                               redacted_msgs id outputs = map (f id) outputs
                                          f x (time, xs)               = (time, concat (map (filter (h x)) xs))
                                          h x (Message (f,t) args)     = (t=x)
i_wrapper localstate args inputs id = outbound_messages_this_timestep : (i_wrapper localstate args future_messages id)
                                      where
                                      q = (getargval.hd) (filter ((="q").getargstr) args)
                                      k = (getargval.hd) (filter ((="k").getargstr) args)
                                      inbound_messages_this_timestep  = snd (hd inputs)
                                      t                               = fst (hd inputs)
                                      future_messages                 = tl inputs
                                      outbound_messages_this_timestep = [ Message (id,1) [(Arg ("i_f1",res))],Message (id,2) [(Arg ("i_f1",res))]]
                                                                        where
                                                                        res = i_f1 t
                                      i_f1 0 = q
                                      i_f1 t = cond (t<0) 0 (i_f1_last_timestep + j_f1_last_timestep)
                                               where
                                               msgs_from_i        = filter ((=1).getmsgfrom) inbound_messages_this_timestep
                                               args_from_i        = concat (map getmsgargs msgs_from_i)
                                               i_f1_last_timestep = (getargval.hd) (filter ((="i_f1").getargstr) args_from_i)
                                               msgs_from_jd        = filter ((=4).getmsgfrom) inbound_messages_this_timestep
                                               args_from_jd        = concat (map getmsgargs msgs_from_j)
                                               j_f1_2last_timestep = (getargval.hd) (filter ((="j_f1d").getargstr) args_from_jd)
j_wrapper localstate args inputs id = outbound_messages_this_timestep : (j_wrapper localstate args future_messages id)
                                      where
                                      q = (getargval.hd) (filter ((="q").getargstr) args)
                                      k = (getargval.hd) (filter ((="k").getargstr) args)
                                      inbound_messages_this_timestep  = snd (hd inputs)
                                      t                               = fst (hd inputs)
                                      future_messages                 = tl inputs
                                      outbound_messages_this_timestep = [ Message (id,4) [(Arg ("j_f1",res1))],Message (id,2) [(Arg ("j_f1",res1))],Message (id,3) [(Arg ("j_f1",res1))],Message (id,2) [(Arg ("j_f2",res2))]]
                                                                        where
                                                                        res1 = j_f1 t
                                                                        res2 = j_f2 t
                                      j_f1 0 = k
                                      j_f1 t = cond (t<0) k_f1_last_timestep
                                               (25 + (j_f1_last_timestep * i_f1_last_timestep)+j_f2_last_timestep)
                                               where
                                               msgs_from_i        = filter ((=1).getmsgfrom) inbound_messages_this_timestep
```

6 Translation Considerations

Now that the design of the translation process has been described there are a few aspects relating to it that should be considered. These considerations can be formed as three questions: Is the final form of the translation a good representation of InterDyne? Does this design give a correct translation? Can this translation process be automated?

This section covers these three questions, first by comparing the final form of the translation to InterDyne, then by looking at the design and the operation of each step in the translation, and finally by showing the first steps in an automated version of this translation.

6.1 Translation Validation

Here the validity of this translation is assessed, this is done by comparing the main aspects of InterDyne to the final form of this translation. These aspects are: The design of the harness, the type of the harness, the type of the agents, and the type of messages.

The design of the simulator harness in the translation can be seen to be very similar to that of the InterDyne simulator harness by comparing Fig. 34 with Fig. 6. The translation limits the functionality of InterDyne some what, such as reducing its output to a single number and only allowing for one message type. These reductions in the functionality of InterDyne explain the difference between the two harness.

Comparing the harness types, InterDyne has the harness type shown in Fig. 36, where as the translations harness as the type sown in Fig. 37. As can be seen these types are very similar, with the translations harness type containing an empty list instead of a list of integers and the output of the translation being an integer instead of an IO output, which is explained due to the reduced nature of this model.

```
1 Int -> [Arg_t] -> [(Agent_t, [Int])] -> IO()
```

Figure 36: *Type of InterDynes harness.*

```
1 Int -> [Arg_t] -> [(Agent_t, [])] -> Int
```

Figure 37: *Type of the translations harness.*

Since an agent can be formulated in almost any manner within InterDyne, it is only worth while to compare the types of the agents between InterDyne and the translation. The type of an InterDyne agent, shown in Fig. 38, is very similar to that of an agent within the translation, shown in Fig. 39, the only difference exists on the input to an InterDyne agent containing one extra list of messages. This extra list of messages is relating to broadcast messages, which are not implemented in the translation and hence do not appear in the type of the agents. This mismatch in type is again down to the translation creating a reduced form of InterDyne.

```
1 Agentstate_t -> [Arg_t] -> [(Int, [Msg_t], [Msg_t])] -> Int -> [[Msg_t]]
```

Figure 38: *Type of InterDynes agents.*

```
Agentstate_t -> [Arg_t] -> [(Int, [Msg_t])] -> Int -> [[Msg_t]]
```

Figure 39: *Type of the translations agents.*

The translation has explicitly implemented the message type used in InterDyne by creating messages using the constructor $Message$. This constructor is used for simple one-to-one messages and in InterDyne there are a large range of messages for different data. The translation has only implements this single type, again show casing the reduced functionality compared to InterDyne.

This assessment of the resemblance between the final product of this translation and an InterDyne simulation, show that this process does indeed create a InterDyne simulation, which is of reduced functionality. This reduced form of InterDyne is required to allow a matching to the initial recurrence relations and to limit excess complications during the translation.

6.2 Translation Semantic Considerations

Though the translation can be seen to produce an InterDyne simulation, the question still remains about the correctness of the steps in this process. Here two arguments for the correctness of meaning of this approach will be presented.

The first argument is that each step can be seen as nothing more then a reformatting of the information and experiment already presented. With new structures such as types being nothing more then wrappers for data that is then extracted and treated as before. This reformatting creates no new information in the experiment, with reverse engineering used to create aspects that may have added new information, such as introducing messages sent from a sender to a receiver in place of a receiver calling on a sender to get the information.

The second argument relates to the functionality of each step, if each step can produce the same outputs given the same inputs then they are likely equivalent. To test the functionality each step was converted into a Miranda program, where the experiment time of 3 was replaced by an input value, which could be executed. These programs were then run using a large selection of time stamps, including zero, and the values were compared as an equality. This returned a value of $True$, showing that the outputs for each step given the same inputs were identical.

These arguments suggest that this translation has steps that are equivalent and hence that the produced version of InterDyne is an equivalent representation of the initial recurrence relation experiment.

6.3 Automated Translation Testing

Though the design of the translation has been shown, with an example being translated by hand, this is a long process to achieve for multiple experiments. The question hence arises, can this process be automated? Here it is shown that automation of this processes can be achieved, this automation is however outside the current project scope and hence only the first three steps will be done to show that it is possible. The first three steps of this automation are shown in the appendix but will be described here.

The recurrence relation experiment is first passed to the translation program as a list of characters, therefore the first step in the program is to give these characters some meaning. This is done using a lexer, this part of the program reads in the list of characters and looks for keywords or symbols to tokenise. These tokens are contained in a list of lexemes which represent all items that can be passed to this program. The lexer produces a new list tokens, with each token a defined meaning to the program, such as representing an equals sign or a variable.

When passing characters to the lexer it is assumed that they are in keeping with the rules of the bespoke recurrence relation language. The lexer has no knowledge of this language passed its

keywords and symbols and will tokenise everything according to this. If an character or string of charters is passed that does not match any of the known keywords or symbols, it will be interpreted as a variable name and saved as such. This lexer does not use spaces so all tokenising will be done strictly on the characters order and not their layout, this also allows a experiment to be passed as a single continuous line if one wished.

The next step after tokenising the input is to add a structure to this list of tokens, this is done using a parser. The parser reads in the list of tokens and knows the BNF and structure a recurrence relation experiment should take. The parser uses this knowledge of the layout and the meaning of the tokens to translate the token list into a parse tree created in custom numeric type to represents this experiment.

The parser expects for the tokens given to it to conform to the rules and layout of the bespoke recurrence relation language, this includes containing a *main* with an experiment, a *init* though this can be empty, and a *where* with a list of recurrence relations (this can be a single recurrence relation). Any formatting that does not match the formatting rules of this language will cause the parser to crash. However the actual function of the recurrence relations can be incorrect, causing black holes or other issues, but will still be parsed correctly.

Once in this structure a function can be used to translate the recurrence relation experiment into the first step in the translation (to allow for infinite list outputs). This is done by reading in this numeric type and then creating a new version of this numeric type that is translated identically apart from the aspects that need to be changed to allow for the translation step.

This translation step should never crash as the format passed to it should always be correct, with an incorrect format causing the parser to crash. This translation step also only focusses on the format of the experiment and any actual functionality could be incorrect but will still be correctly translated.

For later steps that introduce aspects not covered in the original numeric type, a new numeric type needs to be created to allow for these new aspects of functionality, such as message types. Though these first three steps are only a small aspect of an automated translation, they show that this is achievable.

7 Conclusion

This project has looked at the use of recurrence relation models and agent-based models with economics. Both these models are widely used to model complex systems in many fields, including for investigating emergent behaviour within these systems. Agent-based models in particular offer an attractive method for analysing emergent behaviour within interacting systems, due to their use of message passing. However in economics agent-based, unlike recurrence relation models, models very rarely appear in top journals and are in general not widely accepted in this field.

This dissertation presents a method designed to increase the general acceptance of agent-based models within economics. Increasing the traction of this method within economics will benefit the investigation of numerous complex systems within this field, as well as helping work already done using this technique to be better received.

The method presented in this dissertation adds a formalisation to an agent-based model as well making agent-based models more transparent to those unfamiliar with them. A formalism is specified for the agent-based model in the form of a equivalent recurrence relation model, with the transparency given by a step-by-step translation from this recurrence relation model to the agent-based model.

To achieve this, this project has created a bespoke recurrence relation language, the design of a step-by-step translation between this language and an agent-based model, and a program for implementing the initial steps of this translation.

The created bespoke recurrence relation language allows for experiments involving interacting relations to be specified to be fully specified. The language its self is specified with a given BNF form,

as well as details on creating well formed programs within it.

A design for the eight step translation from the bespoke recurrence relation language to an agent-based model known as InterDyne was created and demonstrated. This was done by translating an example system, which was written in the bespoke recurrence relation language and then rewritten at each step of the translation, till a final form comparable to a reduced InterDyne experiment was given.

The initial aim was creating the design for this translation, but not to implement this design, however the project has succeed in all of the design aims and additionally succeeded in implementing the first elements need to automate this translation process.

The two models created for the example system, the recurrence relation model and the agent-based model, are both equivalent numerically. Not only are the initial and final model equivalent numerically but so are each of the steps in the translation process.

The method presented in this dissertation both formalises and familiarises agent-based models with recurrence relations. Allowing one familiar with recurrence relations to be introduced to agent-based models and to increase their understanding of this technique.

7.1 Further Work

This project builds the foundations for a number of additional extensions to be done. Three particular extensions to note are: implementing a the full translation, proving the correctness of the translation, and creating a reverse translation.

The most obvious pieces of further work that can be undertaken is to implement the rest of this translation processes. This would involve implementing steps two to eight in a similar manner to that shown for step one.

The the correctness of each step has been argued informally in this dissertation, a formal prove of the correctness of each step can be undertaken. This is will be most readily achieved by translating each step into lambda calculus and then proving the correctness in this form.

A large extension to this project would be the creation of a translation that turns an InterDyne simulation into a model written in the bespoke recurrence relation language. This would allow for systems to be readily viewed as either a set of recurrence relations or an agent-based model, making these systems particularly amenable to both static and dynamic analysis.

8 Appendix A.

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] 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.

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

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

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

[40] 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.

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

[42] 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.

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

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

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

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

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

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

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

[50] Lee Braine and Chris Clack. The clover rewrite rules: A translation from oofp to fp. Draft, 1997.

[51] William A. Wulf. A case against the goto. In *Proceedings of the ACM Annual Conference - Volume 2*, 1972.

[52] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *ICCAD*, 1989.

[53] Michael J. North. A theoretical formalism for analyzing agent-based models. *Complex Adaptive Systems Modeling*, 2(1):3, May 2014.

[54] Antonio C. R. Costa, Fernanda M. Jeannes, and Ulisses A. Cava. Equation-based models as formal specifications of agent-based models for social simulation: Preliminary issues. In *2010 Second Brazilian Workshop on Social Simulation*, pages 119–126, 2010.

[55] Bernard Zeigler, Tag Kim, and Herbert Praehofer. *Theory of Modeling and Simulation 2nd Edition.* Academic Press, 2000.

[56] MJ North. A theoretical foundation for computional agent-based modeling. In *In 2012 World Congress on Social Simulation*, 2012.

[57] Jean-Sebastien Bolduc and Hans Vangheluwe. A modeling and simulation package for classic hierarchical devs. *MSDL, School of Computer McGill University, Tech. Rep*, 2002.

[58] Hui Shang and Gabriel Wainer. A simulation algorithim for dynamic structure devs. *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, 2006.