Translating from Recurrence Relations to an Agent-Based Representation

Leo Carlos-Sandberg
Supervisor: Dr Christopher D. Clack

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MRes in Financial Computing
of the
University College London.

Department of Computer Science
University College London

August 20, 2017

1

I, Leo Carlos-Sandberg, confirm that the work presented in this dissertation is my own. Where information has been derived from other sources, I confirm that this has been indicated in the dissertation. The report may be freely copied and distributed provided the source is explicitly acknowledged.

This student is registered as dyslexic / dyspraxic with UCL Student Disability Services.

Please refer to Guidance for Academic Staff: Assessment of the work of students with specific learning difficulties – Dyslexia / Dyspraxia

http://www.ucl.ac.uk/disability/info-for-staff/dyslexia-marking-guidelines

Abstract

This dissertation investigates a process to increase the acceptance of agent-based models by translation from recurrence relation models. The work presented in this dissertation is designed to combat two main aspects that hinder the up take in agent-based models: a lack of a model-wide formalism, and a lack of familiarity to those unaccustomed to the technique.

The importance of this work relates in particular to the field of economics, where agent-based models are yet to become widely accepted. This lack of acceptance leads to an absence of work based on agent-based models in top economics journals. This creates a problem for modelling complex systems and emergent behaviour, as agent-based models are in many cases a natural modelling technique for these systems, due in part to their message passing behaviour.

The process presented in this dissertation is to create an agent-based model equivalent to a recurrence relation model. Recurrence relation models are both widely accepted and have a model-wide formalisation. Creating an equivalence between these two models allows for the recurrence relation model to give the agent-based model both a formalisation and a familiar representation, for economists.

This work comprises the creation of a bespoke recurrence relation language, the design of step-by-step translation between the models, and a numeric testing of the equivalence of these models.

The bespoke recurrence relation language presented in this work allows for an experiment involving complex and interacting systems to be laid out in a clear manner, using recurrence relations.

A design for an eight-step translation that turns an experiment written in the bespoke recurrence relation language into an agent-based model is given.

The agent-based model created in this translation is shown to be numerically equivalent to the initial recurrence relation model, for a number of starting values. Each step in the translation process is also shown to be numerically equivalent, for these same values.

The work presented in this dissertation contributes to the field by adding a method for both formalising and expressing an agent-based model using a recurrence relation model. This work gives a method, which can allow for agent-based models to become more widely accepted within the field of economics, giving broader visibility to a powerful tool for investigating complex financial markets.

Contents

1   Introduction

Complex systems have been an area of interest for a long time in many fields such as physics, biology and chemistry [1]. More recently this is true in the fields of economics and finance [2], especially as the financial markets have become increasingly automated and interconnected. Of particular interest is emergent behaviour and emergence in complex systems, these have been defined as "a property of a system is emergent; if it is not a property of any fundamental element, and emergence is the appearance of emergent properties and structures on a higher level of organization or complexity" by Fromm as well as others [3–5].

Due to its nature emergent behaviour can often be damaging to the system in which it occurs. As such an area of research within the study of complex systems in economics is focussed on regulation [6], with the aim of being able to introduce regulations to stop future occurrences of damaging emergence, such as possible flash crashes. A flash crash is a sudden drop and then recovery in the price of securities or commodities, this maybe the result automated trading [7]. A prominent example of this is the flash crash of 2010, which is thought to have, at least in part, been caused by high-frequency traders [8,9].

Emergent behaviour in markets 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 smoothly through time) or discrete (where the system takes quantised steps through time, with each step advancing the system).

For automated traders, and high-frequency traders in particular, the latter approach is best used to describe them. This is due to the fact that computers trade in discrete time, and because of the speeds involved in most of these trades this quantisation of time becomes very important [10].

For discrete time systems two particular modelling techniques that can be used are recurrence relations and agent-based models. These two techniques model systems in different ways, with recurrence relations being analytic and agent-based models being dynamic. Recurrence relations take a more mathematical approach modelling a system as a set of equations that interact through function calls between different equations. Agent-based models on the other hand model a system as a set of agents that interact through messages sent between them.

Both of these techniques have been used extensively in a large range of fields for modelling complex systems [11], however within the field of finance use of agent-based models is predominantly missing from top economics journals [12,13].[1] This absence of agent-based models can be explained by two factors: the lack of a formal definition for the model as a whole, and the technique itself being unfamiliar and not resonating with many researchers in economics [13].

The lack of widespread acceptance of agent-based models within economics can cause an issue for the development of work in this field on complex interacting systems and emergent behaviour. Agent-based models are well known for their ability to model these kinds of systems, often being considered the natural way for describing systems composed of independent entities [14]. They allow these systems to be modelled in such a way that facilitates the exploration of aggregate properties, interaction dynamics, and emergent behaviour. The modularity of their design allows for complexity of agents to be easily tuned, as well as for the models to be easily expanded [14]. Agent-based models in many fields are considered very powerful and well suited to computational modelling of complex systems, with a number of advantages over other modelling techniques [12, 15–18].

The other popular modelling technique for complex systems, recurrence relation models, has a system-wide formalisation, and is widely accepted and well used within economics [19–21]. This motivates this dissertations investigation of a correctness preserving transformation from a

---

[1]This is discussed further in Section 2.2.

recurrence relation model to an equivalent agent-based model.

The recurrence relation model gives the agent-based model a system-wide formalism and the translation seeks to make agent-based models more familiar to those experienced with recurrence relations.

This translation is from a recurrence relation model to an agent-based model, as opposed to the reverse. This trajectory is taken as the aim is to familiarise those already accustomed to the former with the latter, hence it makes sense to start with the recognised model. Facilitating them in taking advantage of agent-based models ability to investigate message passing, something, which is difficult in recurrence relation models.[2]

This translation has a number of fundamental challenges associated with it, which will be discussed in Section 3.3, but include:

- The two paradigms of the models are substantially different.

- How are function calls related to message passing?

- How can the idea of agents having an infinite list of their values through out time be derived from the recurrence relations?

- How can the idea of private and public information be introduced, and what data should fall into each category?

- How can time limited messages be introduced?

- How can output messages containing message information be introduced?

- How can the recurrence relations be split in a way that does not splinter the model?

The translation process in this dissertation takes a step-by-step approach, with each step gradually translating the model. This allows for these issues to be tackled in a more modular fashion. This also makes the equivalence of the two models more apparent, if each small step is equivalent then so too must be the models, assuming the transitivity of the equivalence.

The objectives of the project this dissertation describes are as follows:

- The creation bespoke recurrence relation language and formal definition of its syntax.

- The design of a step-by-step translation process, turning a model written in the bespoke recurrence relation language into an agent-based model.

- The validation and testing of this step-by-step translation.

This dissertation presents both a bespoke recurrence relation language and the design of the steps needed to translate a system expressed in this language into an agent-based model. Though not within the initial project scope, also presented is the first three steps of implementing this translation process into an automated program. These steps take the form of a lexer, parser, and first translation step.

The agent-based model translated to is a reduced form of the InterDyne simulator. InterDyne was chosen in particular due to it being designed to investigate emergent behaviour within the financial markets, this software was also a convenient choice due to access to the source code and local expertise.

The creation of this translation process contributes to the field by giving a method whereby economists can become familiar with agent-based models, as well as giving a formalisation to the agent-based models. This has the potential to increase the uptake in use of agent-based models and to increase the acceptance of work already done using agent-based models.

---

[2]Translation in the opposite direction is discussed in Section 7.1.

The benefits of this work are timely, matching with the increased interest in agent-based models within finance, and with the interest by regulatory bodies in interaction and emergent phenomenon [22–24].

The dissertation layout is as follows. In Section 2 a background is given, covering: emergent behaviour generally and in the financial markets, a short assessment of the literature and description of both recurrence relations and agent-based models, and an in-depth description of InterDyne. In Section 3 a description and analysis of the problem is given, covering: a detailed description of the problem this dissertation addresses, a review of similar work in tackling this problem, and a description of the challenges with the translation approach taken. In Section 4 the bespoke recurrence relation is given, along with its syntax and presentation. In Section 5 the design of the translation is given, with an example system translated from the recurrence relations to an agent-based model. In Section 6 the translation is discussed, with: comparisons to the desired agent-based model given, numerical testing of the equivalence of the steps given, and additional work outside the scope of the project presenting the first stages in implementing this translation given. In Section 7 the dissertation is concluded and summarised with notes for further work given.

2 Background

This section covers a description of emergent behaviour, feedback loops as a type of this behaviour, and their appearance in the financial markets. Recurrence relations and agent-based models are introduced as a method for modelling emergent behaviour, and 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 a description of its design will also be given.

2.1 Emergent Behaviour

Emergent behaviour is a term used to describe a macro-level behaviour of a system that is not obvious from analysis of the micro-level behaviour of the system; more formally this is a behaviour that can not be predicted through analysis of any single component of a system [25].

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 [22].

A simple way to demonstrate emergence is in the Game of Life [26], which is an example of a 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. Whether 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 fewer 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. A 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 [27], 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.

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 [28].

Though this example deals with a physical phenomenon, interaction based emergence is present in many different systems. For example interactions 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 [29]. The financial markets can be thought of as a complex system,

9

with interaction dynamics, hence one can posit that the markets would exhibit emergence. This is true and the financial markets have been seen to exhibit a large selection of emergent behaviour, such as the formation of patterns, bubbles and crashes [3]. 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 [30].

Feedback loops are a prominent type of emergent behaviour that can occur from interaction dynamics. Feedback loops occur 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.
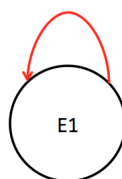
Figure 1: *Simple feedback loop with an entity supplying its input from its output.*

Feedback loops can encompass multiple entities, in Fig. 2 a feedback loop is shown that encloses two different entities. An 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.
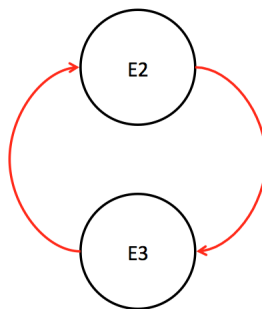

Figure 2: *Feedback loop between two entities, with each output being transformed by the other entity before becoming an input.*

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 examples 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.

10

Though feedback loops are often assumed to be a negative property, some can be stabilising due to a benign effect [31].

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 analyse, 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.

Flash crashes are a notable form of emergence behaviour, due to feedback loops, that are considered damaging. These cause the price of a security or commodity to drop very rapidly, becoming disconnected from its fundamental, before then recovering [32]. A particularly famous flash crash is that of 2010, in which the E-mini S&P 500 equity futures market dropped in price by more then 5%, before rebounding to close to its original price [8,32]. 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 [33].

Research done by [9,34] 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 [35]. 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 [36].

The feedback loop of "Hot Potato" trading, is when inventory imbalance is repeatedly passed between HFT market makers. A market maker is a trader who is required to have both a bid and an 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 from 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 sell off an amount of inventory to return back into their normal trading region. This inventory now sold by the market maker can be bought by another market maker perhaps causing them to exceed their limits and sell, this process is "Hot Potato" trading and can in theory be cyclic and continue indefinitely [37].

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 [38].

2.2 Methods for Modelling Emergent Behaviour in Finance

There are many different methods available for modelling systems that may exhibit emergent behaviour, such as: 1-period, 2-period and multi-period models, probabilistic models (e.g. hidden Markov models), ordinary differential equations and partial differential equations [39–42]. This dissertation 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 [20, 21, 43].

Recurrence relations have seen wide spread use in economics appearing in a large selection of journals, including numerous times in top journals [44–51]. This wide spread use of the technique implies the acceptance of recurrence relations within financial modelling.

Agent-based models are also used widely in economics and finance [52, 53]. However these papers are noticeably absent from top economics journals (with some exceptions, such as [54] and [55]) and tend to be published in the Journal of Economic Behavior & Organization and the Journal of Economic Development & Control [12, 13]. 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 [56–58].

Here these two modelling techniques will be described.

2.2.1 Recurrence Relations

Wimp [59] describes recurrence relations as follows, 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.

The most simplistic form of a recurrence relation is one where the next term depends only on the immediately preceding term. If the nth 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 [60].

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 recurrence relation for defining a sequence may also depending upon some parameter, α, in its simplest case this can be written as,

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

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:

- A mathematical formalisation of the whole system being modelled is given when using recurrence relations due to three factors: the way in which they model the whole system as an entity, the static view point they give to the system, and the mathematical stye which they take [61].

- 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.

A downside to this modelling technique comes with complex equations. Equations such as those making function calls within conditional statements, can be become increasingly less amenable to static analysis for dependencies. This is a problem similar to determining whether arguments will be evaluated, and makes tracking the flow of information, especially dynamically, very difficult [62]. This also leads to challenges in in tracking the individual behaviour of recurrence relations or groups of recurrence relations [63].

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 than 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 [64].

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 [65], herding among traders [66], and fundamentalists [67].

Agent-based models are particularly suited to systems that, 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 an obvious visual design of the system, such as that shown in Fig. 3.
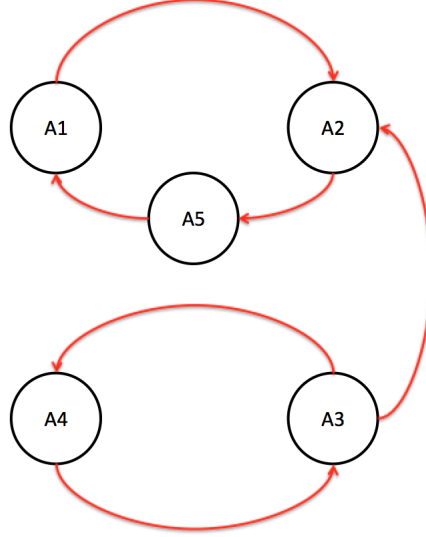

Figure 3: *Visualisation of the allowed interactions between five different agents, represented by black circles, with red lines representing the message paths.*

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 an agent to be considered an agent [68, 69]:

- 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 is 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 agent's state varies over time, representing variables associated with its current position.

- An agent's 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 system's 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 [14].

Due to the set up of this model 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 a bottom-up approach in this model, allows for a more dynamic view of a system to be achieved; messages can be more easily tracked through-out the system's 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 often be added with little or 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 [56].

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 [70], it is a general-purpose simulator for exploring emergent behaviour and interaction dynamics within complex non-equilibrium systems.

InterDyne's 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 only sent on an integer time step.

Messages in InterDyne are 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,[3] but does store the messages and their order which can be used in post analyse.

---

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

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, and 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.[4]

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 recorded for post processing.

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 [8].

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

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

---

[4]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.

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 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.
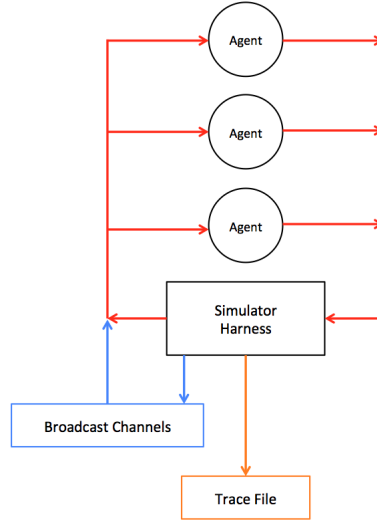
Figure 4: *Structure of an InterDyne simulation containing three agents. The messages sent by the agents to the harness and from the harness are coloured red. The trace file and messages sent to the trace file are coloured orange. The messages sent from the harness to the broadcast channels and then back to the agents are coloured blue [71].*

InterDyne has an 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 their 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 an 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

17

the maximum delay the argument ($Arg(Str\ "maxDelay", 10))$) is used, where 10 is the maximum delay. To add the list of delays the argument ($DelayArg(Str\ "DelayArg", delay)$) 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 the following 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, for example $Message\ (1, 2)\ data$, where $data$ is the information being sent from agent 1 to agent 2

A broadcast message, for example $Broadcastmessage\ (3, 1)\ broadcastdata$, can be received by agents subscribed to its channel, in this case 1. All agent subscriptions to broadcast channels 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 take 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 than what has already been calculated. In achieving this an agent must at each time step, read in a message (even if it is then not used) and produce a list of messages (even if this is an 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 typically, 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 their content. An agent written following this design could have a form similar to that of the agent show in Fig. 5. This agent's wrapper recursively calls itself on the inbound message list, consuming the head of the list[5] and producing a list of outbound messages using a logic function.

```
wrapper st args ((t, msgs, bcasts) : rest) myid = [m] : (wrapper st args rest myid)
                                              where
                                                  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.*

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

This has the type shown in Fig. 6, this type illustrates the functions input of a number of steps and runtime arguments (with agents and the broadcast channels they are subscribed to).

---

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

The output from this function can be seen to be of type $IO()$, this is writing the trace information to file.

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

Figure 6: *Type definition of the function sim.*

The simulator harness is embodied in a function called sim, this function contains code associated with a number of administrative operations. For the purpose of this project a reduced specification of the function sim will be shown. This specification captures all of the core functionality for managing message passing for agents and driving evaluation of the simulation. This reduced InterDyne sim harness can be seen in Fig. 7, with its function type shown in Fig. 8. This reduced function is shown outputting a single integer value, that is the first message sent by the first agent at time step x. This is just for illustration purposes since the normal output of sim is a trace file and is not being shown here. In the reduced type definition it can be seen that the output is now and integer instead of an $IO()$ type, the list passed to the sim that contains the subscription channels is now of unknown type as no channels are passed.

```
sim x runtime_args agentinfo = (getargval.hd.getmsgargs) (((snd (outputs!x))!0)!0)
                               where
                               outputs             = (0,[]):timed_transposed_msgs
                               timed_transposed_msgs = zip [1..] transposed_msgs
                               transposed_msgs     = transpose allmessages
                               allmessages         = map apply_to_args (zip [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 7: *Reduced InterDyne simulator. This sim function outputs, the value of the first messages from the first agent at the final time step.*

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

Figure 8: *Type definition of the reduced function sim.*

3 Description and Analysis of the Problem

Investigating emergent behaviour originating from interaction dynamics, within a complex system, relies 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 [72].

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 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 [57], is to give a formal definition of the specification of an agent-based model as a set of recurrence relations.

This section discusses how relating these two models can be achieved, reviews literature on work done formalising agent-based models, and discusses challenges related to the method presented in this dissertation.

3.1 Method for Achieving a Translation

To allow for recurrence relations to be used in formalising an agent-based model, the two paradigms must be connected. This connection must be able to demonstrate some sort of equivalence between these two models.

It has been shown informally by [17], that most agent-based models can be expressed as a set of partial recursive functions. Therefore it can be inferred that a connection between a recurrence relation model and an agent-based model can be created in the form of a translation, from one to the other.

A correctness-preserving translation between a recurrence-relation model of a system and an agent-based model of the same system provides a formalism for the agent-based model if it can be said that the recurrence-relation model defines the meaning of the agent-based model. Correctness preserving in this context is used to mean that both models representing an identical system and will produce identical results; so that both views of the system should be completely interchangeable.

An important consideration in creating a translation between the two methods is the direction in which this translation should be effected. Ideally, correctness-preserving translations would be defined in both directions and this would prove the semantic equivalence of the two models. However, defining two complete translations is beyond the scope of this project. Defining the translation from a recurrence-relation model to an equivalent agent-based model has been chosen, since it is believed that those who prefer recurrence-relation models would prefer to start with a recurrence-relation model; the agent-based model then becomes an equivalent view within which dynamic analysis of emergent behaviour can be undertaken.

The next consideration is how this translation will be achieved. Inspiration can be drawn from other works investigating the design of translations, such as [73], [74] and [75]. 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. This approach relies on the correctness of the steps being transitive; that a correctness preserving translation from state 1 to state 2, and then another from state 2 to state 3, will produce a correctness preserving translation from state 1 to state 3.[6]

A connection between these two models 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, one being that from its conception it was designed with modelling and investigate emergent behaviour with the financial markets in mind (with research existing already using it to this end [9]), 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 game theory, discrete systems, classifier agents and equation-based models [76]. 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 [72], [57] and [69], looking at this area will be reviewed and discussed, along with DEVS formalism introduced by Zeigler in [77].

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? North [76] present's a 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 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 [78] and was chosen due to its similarities to modern computer systems. This is opposed to other options including Turing machines and λ-calculus, as they are less well matched with modern computer systems.

One of North's motivations for using this technique over an equation-based model formalisation, is that he claims that a 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 economists over computer scientists.

In [72] Costa et al consider the formalisation of agent-based models by creating their implementation guided by equation-based models. 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-

---

[6]A suggested method for further work in this area is given in Section 7.1.

based model, due to its nature, took 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 issue spawns from trying to recreate the behaviour of a probabilistic model using a deterministic model.

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 agent-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 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, a more micro-level focussed equation-based model should be taken.

The next paper is [57], 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 the 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 provide 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 dissertation 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.

Macal [69] 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. Once equivalence has been proven the specification of a Forrester's model can be used to derive an agent-based model that is a correct representation of the same system.

This technique is applied to an SIR epidemic model, creating a 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 without mentioning the Discrete Event System Specification (DEVS) [77]. 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 [79]) and Dynamic Structuring DEVS (attempts to allow for dynamic behaviour [80]).

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. However though DEVS is capable of representing an agent-based model it is not necessarily the best approach. Despite the work done in [80], DEVS is still not an ideal match for models that have dynamic 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. These constraints and difficulties make it challenging to use DEVS as a formal basis for agent-based modelling [76].

3.3 Challenges with the Translation 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 counterpart, or are challenging to create a counterpart for, in an agent-based model. An example of this is modelling probabilist systems and macro-level systems, which are difficult to establish an equivalent agent-based model for. This project avoids these challenges by using micro-level recurrence relation models for the translation.

A 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 an analytic (where the system evolution is defined statically.) approach where as agent-based models take a dynamic one (where the system evolution is given in simulation).

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 respectively. 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, whereas 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 way in which it is tangible to the agent-based model is a challenge.

Recurrence relations access information directly as a value where needed; agent-based models however receive special item called a message that contains this value along with other information (e.g. the senders id). Translating between these two different ways of viewing values poses difficulties.

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.

The approach taken in this dissertation 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

This section introduces a bespoke recurrence relation model, created to be able to model complex systems with interacting elements. This recurrence relation model is used as the initial description of the system in the translation into an agent-based model and hence needs to be able to express the system being modelled both statically and dynamically.

A bespoke recurrence relation language is created for three reasons. Firstly, an automated translator needs to have a well-defined input that it can parse. Secondly, the functionality of the language needs to be restricted to keep it as simple as possible, to allow for the language to be readily understandable and easy to parse, only introducing complexity where necessary. Thirdly, the language still needs to maintain sufficient functionality to model complex systems. Creating a bespoke language offers a straightforward method satisfying these three criteria.

A set of recurrence relations can be used to statically model a interacting system, however this model does not give a time-dependent experiment required for dynamic modelling. To add time dynamics to this model the values of the recurrence relations need to be traced through time.

To allow for tracing, initial values must be provided (noting that different experiments might have different initial values), and a main expression must be defined to control demand for the evaluation of the relevant traces.

This section first gives the concrete syntax, then keywords and naming conventions, finally a well formed program in this language is described and an example is given.

4.1   Syntax

The formal concrete syntax for this bespoke recurrence relation language is defined in Backus-Naur form (BNF), shown in Fig. 9. The main aspects of this notation, in this context, are described as follows:

- Expressions of the design of language are represented inside $< >$; these are high level design aspects of the language (e.g. the simulation as a whole, or the main block defining the experiment).

- These high level expressions are given meaning by ::=, the expression on the left hand side of this must take the form given on the right hand side (e.g. in $< a >::=< b >$, $< a >$ has to take the form given by $< b >$).

- The right hand side of ::=, can have multiple options, these are split by |. This can be seen to representing an *or* operation, with the left hand side being able to be represented by any of the options given on the right hand side (e.g. in $< a >::=< b > \mid < c >$, $< a >$ can be either $< b >$ or $< c >$).

- The speech marks "" are used to denote actual writing in the language, this allows the syntax to describe how the language can be written (e.g. connecting the expression $< where >$ to the word "*where*" in the language). In this notation "" are used for strings of characters, and '' are used for single characters (e.g. "*where*" and '*w*').

- Here .. is used to denote any item that would logically appear between the left hand side and right hand side of the .., these items are separated by an | (e.g. '1'..'4' gives '1' |'2' |'3' |'4').

- The character '\n' is used to denote a new line in the language.

- Parameterised nonterminals are used to allow for recursive calls with in the syntax, these are noted by a second item with the $<>$ (e.g. $< indent\ n >::=< space >< indent\ n-1 >$, for $n = 3$ will produce $< space >< space >< indent\ 1 >$).

```
1  <simulation>              ::=  <mainblock> <newline> <initblock> <newline> <whereblock>
2  <mainblock>               ::=  <main> <newline> <recrelname> <space> <numbers>
3  <numbers>                 ::=  <number> | <number> <space> <numbers>
4  <initblock>               ::=  <init>    | <init> <newline> <valdefinitions>
5  <whereblock>              ::=  <where> | <where> <newline> <recreldefinitions>
6  <valdefinitions>          ::=  <valdef> <newline> | <valdef> <newline> <valdefinitions>
7  <valdef>                  ::=  <valname> <space> <equals> <space> <expression>
8  <valname>                 ::=  <letters>
9  <recreldefinitions>       ::=  <recreldef> <newline> | <recreldef> <newline> <
       recreldefinitions>
10 <recreldef>               ::=  <newline>
11                             |  <recrelname> <space> <t> <space> <equals> <space> <
       expression>
12                             |  <recrelname> <space> <t> <space> <args> <space> <
       equals> <space> <expression>
13 <recrelname>              ::=  <letters> <underscore> <letters>
14 <args>                    ::=  <argname> | <argname> <space> <args>
15 <argname>                 ::=  <letters> | <number> | <lparenthesis> <number> <
       rparenthesis>
16 <expression>              ::=  <simple_expression> | <simple_expression> <
       expr_whereblock>
17 <simple_expression>       ::=  <number>
18                             |  <valname>
19                             |  <argname>
20                             |  <t>
21                             |  <localvalname>
22                             |  <list>
23                             |  <lparenthesis> <simple_expression> <rparenthesis>
24                             |  <cond> <simple_expression> <simple_expression> <
       expression>
25                             |  <simple_expression> <operation> <simple_expression>
26                             |  <internop> <simple_expression>
27                             |  <recrelname> <simple_expressions>
28                             |  <localfuncname> <simple_expressions>
29 <expr_whereblock>         ::=  <newline> <indent n> <where> <newline> <localdefinitions
        n> <newline> <indent n> <endwhere> <newline>
30 <localdefinitions n>      ::=  <indent n> <localvaldef> <newline>
31                             |  <indent n> <localfuncdef> <newline>
32                             |  <indent n> <localvaldef> <newline> <localdefinitions n
       >
33                             |  <indent n> <localfuncdef> <newline> <localdefinitions
       n>
34 <localvaldef>             ::=  <localvalname> <space> <equals> <space> <expression>
35 <localfuncdef>            ::=  <localfuncname> <space> <args> <space> <equals> <space>
        <expression>
36 <simple_expressions>      ::=  <simple_expression> | <simple_expression> <space> <
       simple_expressions>
37 <localfuncname>           ::=  <underscore> <letters>
38 <localvalname>            ::=  <underscore> <letters>
39 <list>                    ::=  <lbracket> <rbracket> | <lbracket> <listitems> <rbracket
       >
40 <listitems>               ::=  <simple_expression> | <simple_expression> <comma> <space
       > <listitems>
41 <internop>                ::=  <head> | <tail> | <not>
42 <number>                  ::=  <zero> | <posnumber> | <minus> <posnumber>
43 <posnumber>               ::=  <firstdigit> <digits> | <firstdigit> <digits> <dot> <
       digits>
```

(a) *First half of the concrete BNF syntax for the bespoke recurrence relations.*

```
 1  <operation>                ::=  <plus> | <minus> | <times> | <divide> | <lessthan> | <
        greaterthan> | <equals> | <notequals> | <lessthanorequalto> | <
        greaterthanorequalto> | <cons> | <index>
 2  <indent 1>                 ::=  <space>
 3  <indent n>                 ::=  <space> <indent n−1>
 4  <letters>                  ::=  <letter> | <letter> <letters>
 5  <letter>                   ::=  <lowercaseletter> | <uppercaseletter>
 6  <digits>                   ::=  <digit> | <digit> <digits>
 7  <main>                     ::=  "main"
 8  <init>                     ::=  "init"
 9  <where>                    ::=  "where"
10  <endwhere>                 ::=  "endwhere"
11  <firstdigit>               ::=  '1' .. '9'
12  <digit>                    ::=  '0' .. '9'
13  <lowercaseletter>          ::=  'a' .. 'z'
14  <uppercaseletter>          ::=  'A' .. 'Z'
15  <equals>                   ::=  '='
16  <space>                    ::=  ' '
17  <newline>                  ::=  '\n'
18  <underscore>               ::=  '_'
19  <lparenthesis>             ::=  '('
20  <rparenthesis>             ::=  ')'
21  <lbracket>                 ::=  '['
22  <rbracket>                 ::=  ']'
23  <comma>                    ::=  ','
24  <head>                     ::=  "hd"
25  <tail>                     ::=  "tl"
26  <not>                      ::=  '~'
27  <dot>                      ::=  '.'
28  <plus>                     ::=  '+'
29  <minus>                    ::=  '−'
30  <times>                    ::=  '*'
31  <divide>                   ::=  '/'
32  <lessthan>                 ::=  '<'
33  <greaterthan>              ::=  '>'
34  <notequals>                ::=  "~="
35  <lessthanorequalto>        ::=  "<="
36  <greaterthanorequalto>     ::=  ">="
37  <cons>                     ::=  ':'
38  <index>                    ::=  '!'
39  <t>                        ::=  't'
```

(b) *Second half of the concrete BNF syntax for the bespoke recurrence relations.*

Figure 9: *Concrete BNF syntax for the bespoke recurrence relation language, split between Fig. 9a and Fig. 9b.*

A simulation written in this language takes the structure shown in Fig. 10, this structure is defined to have the same meaning as the Miranda expression shown in Fig. 11; where the BNF syntax for the items encapsulated by <> is shown in Fig. 12.

```
 1  ''main''
 2  <exp>
 3  ''init''
 4  <idefs>
 5  ''where''
 6  <fdefs>
```

Figure 10: *Layout of a simulation in bespoke recurrence relation language.*

```
1  main = <exp>
2         where
3         <idefs>
4         <fdefs>
```

Figure 11: *Miranda definition of the layout shown in Fig.10.*

```
1  <exp>      ::= <expression>
2  <idefs>    ::= <valdefinitions>
3  <fdefs>    ::= <recreldefinitions>
```

Figure 12: *BNF syntax for items shown in Fig. 10, with further definitions defined in Fig. 9.*

The syntax given in this section dictates how a simulation written in this bespoke recurrence relation must be formatted and what items are allowed to be used within this language. The functionality of this language, given by the BNF, will be described more fully in the Section 4.2. Any program that does not fit the shape described by the concrete BNF can not be considered to be a model written in the language defined in this section.

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.

Restricted names are those that cannot be used other than for their designated purpose (given in Fig. 9), 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*", unless it is used to define the main block.

- The use of the name "*init*", unless it is to define the init block.

- The use of the name "*where*", unless used to make a where statement.

- The use of the name "*endwhere*", unless used to end a where statement.

- The use of symbols, unless for their defined purpose.

A recurrence relation is 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 or other variable names with in the same where block.

The driving expression is given in a defined section of the model, showing in Section 4.3, and is written as *group_name arguments*. The *group_name* must correspond to a recurrence relation listed in the model and the *arguments* must be given as values, allowing the recurrence relation to be evaluated.

Initial conditions are listed in a special location in the layout, which will be show in Section 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 or used by any other variable (even those within where blocks).

This language allows for both *arguments* and *expressions* to be written using: lists, numbers, variable names, recurrence relation names, internal function names, mathematical operators, brackets, some inbuilt functions, and conditional statements.

Lists in this language are contained within [], items within these lists are then separated using ,.

This language allows the use of both integers and decimal numbers, with . used in decimalisation.

Variables that have been defined and are in scope can be called by simply calling their name.

Recurrence relations that are in scope can be called by calling their name with a relevant set of arguments.

Internal functions that are in scope can be called by calling their name and a relevant set of arguments, if applicable.

This language allows for the mathematical operates shown in Fig. 9.

Expression can be incased in brackets by using ().

This language allows for the use of three inbuilt functions: *hd* (returns the head of a list), *tl* (returns the tail of the list), and : (adds an item as the new head of a list).

Conditional statements with in this language are written as *cond condition expression expression*. Where if the condition is true, then the first expression is run and if not the second expression is.

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.

An experiment is written as a function call, with a certain set of arguments. Figure 13 shows a experiment that will return the value of the recurrence relation i_f1 at time 3.

```
1  i_f1  3
```

Figure 13: *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. 14. There is no defined limit to the amount of initial conditions that can be defined.

```
1  q = 4
2  k = 16
```

Figure 14: *Definition of two initial conditions, q and k.*

A simple recurrence relation can be written as shown in Fig. 15, 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.

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

Figure 15: *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. 16. 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.

```
1  i_f1  0 = q
2  i_f1  t = (i_f1  (t−1) ) + (j_f1  (t−1))
```

Figure 16: *Recurrence relation containing pattern matching for initial conditions.*

An if statement, shown in Fig. 17, 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.

```
1  j_f1  t = cond (t<0) (k) (25 + ((j_f1  (t−1)) * (i_f1  (t−1))) + (j_f2  (t−1)))
```

Figure 17: *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 18 shows the use of a where statement to add an internal variable and an internal function to a recurrence relation.

```
1  i_f1  t = a * (_sf1  t)
2          where
3          a      = 5
4          _sf1  x = x*2
5          endwhere
```

Figure 18: *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. 19.

```
1  main
2  i_f1  3
3
4  init
5  q = 4
6  k = 16
7
8  where
9  i_f1  0 = q
10 i_f1  t = ( i_f1  ( t−1) ) + ( j_f1  ( t−1))
11
12 j_f1  0 = k
13 j_f1  t = cond  ( t<0)  (k)  (25 + (( j_f1  ( t−1)) * ( i_f1  ( t−1))) + ( j_f2  ( t−1)))
14 j_f2  0 = k+q
15 j_f2  t = ( j_f1  ( t−3)) + 27
```

Figure 19: *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 a reduced form of the InterDyne simulator. This is broken down into eight individual steps, with each of these steps altering the output of the previous step to resemble more closely an InterDyne simulation.

The eight steps required for this translation are to introduce:

1. Infinite list output. Solving the issue of InterDyne agents producing infinite lists of their outputs, compared with the recurrence relation model where outputs are calculated when given an input.

2. Grouping into agents. Solving the issue of InterDyne containing agents, where the recurrence relation model a list of recurrence relations.

3. A global output list. Begins to solve the issue of InterDyne containing a harness that mediates all the messages; by creating a function to mediate the messages.

4. Agent input lists. Begins to solve the issue of agents in InterDyne taking in messages, compared with the function calls in the recurrence relation model; by giving each group of recurrence relations an input to use.

5. Targeted information passing. Continues to solve the issue of message passing; by allowing information produced by a recurrence relation to have a defined group of recurrence relations as a target.

6. A simulator harness. Continues to solve the issue of InterDyne containing a harness; by grouping the functionality for producing the global output list, and the sorting of inputs for the recurrence relation groups, into a single function.

7. Runtime arguments. Increases the produced agent-based models similarity to InterDyne; by adding a number of arguments to be passed to the harness and between the harness and agents, and by changing the structure to be more comparable with an InterDyne simulation.

8. Message communication. Solves the issue of message passing in InterDyne; by adding messages that correspond to those found within InterDyne. Adding these messages also allows for the harness to be restructured to correspond to that of a reduced InterDyne simulation.

These steps are conducted in a way, so that each step builds upon the translation made by its predecessor; as such the order in which these steps are organised can not be rearranged. This order also allows for the bespoke recurrence relation language to evolve into an agent-based model in a comprehensible manner; with each steps translation building on the last and hence increasing the overall complexity in small increments.

These particular translation steps have been chosen for this translation following two requirements; each step should change the target state of the previous step (with the first step changing the initial bespoke recurrence relation language) in a small enough manner that it is easy to follow, and the target state produced by each step should be a functional model (a model that can be translated into Miranda code and executed to produce an output). Additionally each step was chosen to introduce either a single larger piece of functionality to the model or a number of smaller pieces.

Though in this design eight steps have been chosen, this is by no means the only formulation that can be taken. Some of the steps described below can be translated in more steps, and some of the steps can be combined to create a large jump in the translation. However the

steps chosen follow the requirements listed above and create a translation process that should be formulated in an digestible manner; increasing the number of steps would increase the length of the translation most likely without much improving its clearness, and decreasing the number of steps, though shortening the translation, would most likely make following it more challenging.

These eight steps translate the bespoke recurrence relation model into that of an Inter-Dyne 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.[7]

Another translation step could be added to translate the final target state into the Haskell syntax, matching that of InterDyne. However this step is not necessary in creating an agent-based model that can be seen to be equivalent to a reduced version of InterDyne.

These steps will be shown in this section by translating an example written in the bespoke recurrence relation language, shown in Fig. 20. 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 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.

```
1  main
2  i_f1  3
3
4  init
5  q = 4
6  k = 16
7
8  where
9  i_f1  0 = q
10 i_f1  t = (i_f1  (t−1)) + (j_f1  (t−2))
11 j_f1  (−1) = k
12 j_f1  0 = k
13 j_f1  t = cond (t<0) (k_f1  (t−1)) (25+((j_f1  (t−1))*(i_f1  (t−1))) + (j_f2  (t−1)))
14 j_f2  0 = k+q
15 j_f2  t = cond (t<3) 27 ((j_f1  (t−1)) + 27)
16 k_f1  0 = 0
17 k_f1  t = k + q + (j_f1  (t−1))
```

Figure 20: *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 agent 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 an infinite list of the outputs of its sub recurrence relation outputs, with each item in the list being the value for this relation at each time step. This is done using the sub function *_createlist*, this function takes in a function and a time marker. These arguments are then used to add the value of the function

---

[7]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.

at that time maker to the head of a least, where the next value is created by recursively calling _createlist but with the next time step, as shown in Fig. 21.

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

Figure 21: *_createlist function, used to create an infinite list in time of a recurrence relation.*

Since the recurrence relations are now contained within parent functions (that do not take time as an input) that 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. 20 will be translated into Fig. 22.

```
1  main
2  i_f1_list  !  3
3
4  init
5  q  =  4
6  k  =  16
7
8  where
9  i_f1_list  =  _createlist  i_f1  0
10            where
11            _createlist  f  t  =  (f  t):(_createlist  f  (t+1))
12            i_f1  0  =  q
13            i_f1  1  =  (i_f1_list  !  0 )  +  k
14            i_f1  t  =  (i_f1_list  !  (t−1) )  +  (j_f1_list  !  (t−2))
15            endwhere
16  j_f1_list  =  _createlist  j_f1  0
17            where
18            _createlist  f  t  =  (f  t):(_createlist  f  (t+1))
19            j_f1  0  =  k
20            j_f1  t  =  cond  (t<0)  (k_f1_list  !  (t−1))
21                              (25  +  ((j_f1_list  !  (t−1))  *  (i_f1_list  !  (t−1)))  +  (
        j_f2_list  !  (t−1)))
22            endwhere
23  j_f2_list  =  _createlist  j_f2  0
24            where
25            _createlist  f  t  =  (f  t):(_createlist  f  (t+1))
26            j_f2  0  =  k+q
27            j_f2  t  =  cond  (t<3)  27  ((j_f1_list  !  (t−1))  +  27)
28            endwhere
29  k_f1_list  =  _createlist  k_f1  0
30            where
31            _createlist  f  t  =  (f  t):(_createlist  f  (t+1))
32            k_f1  0  =  0
33            k_f1  t  =  k  +  q  +  (j_f1_list  !  (t−1))
34            endwhere
```

Figure 22: *Figure 20 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. For example, to return the value of the first recurrence relation in a group called *g* at time *t*, the call ((*g_wrapper*!*t*)!0) would be used.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. 22 to that shown in Fig. 23.

```
 1  main
 2  (i_wrapper ! 3) ! 0
 3
 4  init
 5  q = 4
 6  k = 16
 7
 8  where
 9  i_wrapper = (transpose [i_f1_list])
10              where
11              i_f1_list = _createlist i_f1 0
12                            where
13                            _createlist f t = (f t):(_createlist f (t+1))
14                            i_f1 0 = q
15                            i_f1 1 = (i_f1_list ! 0 ) + k
16                            i_f1 t = (i_f1_list ! (t−1) ) + (((j_wrapper ! (t−2)) ! 0)
17                            endwhere
18              endwhere
19  j_wrapper = (transpose [j_f1_list , j_f2_list])
20              where
21              j_f1_list = _createlist j_f1 0
22                            where
23                            _createlist f t = (f t):(_createlist f (t+1))
24                            j_f1 0 = k
25                            j_f1 t = cond (t<0) ((k_wrapper ! (t−1)) ! 0)
26                                          (25 + ((j_wrapper ! (t−1)) ! 0) * ((i_wrapper
     ! (t−1)) ! 0) + (j_f2_list ! (t−1)))
27                            endwhere
28              j_f2_list = _createlist j_f2 0
29                            where
30                            _createlist f t = (f t):(_createlist f (t+1))
31                            j_f2 0 = k+q
32                            j_f2 t = cond (t<3) 27 ((j_f1_list ! (t−1)) + 27)
33                            endwhere
34              endwhere
35  k_wrapper = (transpose [k_f1_list])
36              where
37              k_f1_list = _createlist k_f1 0
38                            where
39                            _createlist f t = (f t):(_createlist f (t+1))
40                            k_f1 0 = 0
41                            k_f1 t = k + q + ((j_wrapper ! (t−1)) ! 0)
42                            endwhere
43              endwhere
```

Figure 23: *Translation of Fig. 22 to allow for grouping of recurrence relations into agents.*

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 list of the simulation outputs, *s_outputs*, that collects the outputs from all the agents and returns an infinite list containing these outputs, the structure of this function is shown in Fig.24. 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. The simulation outputs list, with the addition of null values is shown in Fig. 25.

```
1  s_outputs   = (transpose [i_wrapper, j_wrapper, k_wrapper])
```

Figure 24: *An infinite list in time containing the values of every recurrence relation at each time step.*

```
1  s_outputs   = niloutputs : (transpose [ i wrapper , j wrapper , k wrapper ])
2                where
3                niloutputs = [[ nullvalue ] , [ nullvalue , nullvalue ] , [ nullvalue
   ]]
4                nullvalue = 0
5                endwhere
```

Figure 25: *An infinite list in time containing the values of every recurrence relation at each time step, with null values at the start of the list.*

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. Since this list contains null values at the head, to get a value for a time step of $t - 1$, the list needs to be indexed by t.

Translating Fig. 23 to contain this global output list and to index into it for retrieving values from other agents produces Fig. 26.

```
1   main
2   ((s_outputs!(3+1))!0)!0
3
4   init
5   q = 4
6   k = 16
7
8   where
9   s_outputs  = niloutputs : (transpose [i_wrapper, j_wrapper, k_wrapper])
10                where
11                niloutputs = [[nullvalue], [nullvalue, nullvalue], [nullvalue]]
12                nullvalue = 0
13                endwhere
14  i_wrapper = (transpose [i_f1_list])
15                where
16                i_f1_list = _createlist i_f1 0
17                            where
18                            _createlist f t = (f t):(_createlist f (t+1))
19                            i_f1 0 = q
20                            i_f1 1 = (i_f1_list ! 0 ) + k
21                            i_f1 t = (i_f1_list ! (t-1) ) + j_f1_2last_timestep)
22                                    where
23                                    j_f1_2last_timestep = (((outputs!(t-1))!1)!0)
24                                    endwhere
25                            endwhere
26                endwhere
27  j_wrapper = (transpose [j_f1_list, j_f2_list])
28              where
29              j_f1_list = _createlist j_f1 0
30                          where
31                          _createlist f t = (f t):(_createlist f (t+1))
32                          j_f1 0 = k
33                          j_f1 t = cond (t<0) k_f1_last_timestep
34                                        (25 + ((j_f1_list ! (t-1)) *
        i_f1_last_timestep) + (j_f2_list ! (t-1)))
35                                    where
36                                    i_f1_last_timestep = (((outputs!t)!0)!0)
37                                    k_f1_last_timestep = (((outputs!t)!2)!0)
38                                    endwhere
39                          endwhere
40              j_f2_list = _createlist j_f2 0
41                          where
42                          _createlist f t = (f t):(_createlist f (t+1))
43                          j_f2 0 = k+q
44                          j_f2 t = cond (t<3) 27 ((j_f1_list ! (t-1)) + 27)
45                          endwhere
46              endwhere
47  k_wrapper = (transpose [k_f1_list])
48              where
49              k_f1_list = _createlist k_f1 0
50                          where
51                          _createlist f t = (f t):(_createlist f (t+1))
52                          k_f1 0 = 0
53                          k_f1 t = k + q + j_f1_last_timestep
54                                    where
55                                    j_f1_last_timestep = (((outputs!t)!1)!0)
56                                    endwhere
57                          endwhere
58              endwhere
```

Figure 26: *Figure 23 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.

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. 27.

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

Figure 27: *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 as 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 is 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. A wrapper written in this way would have a similar form to that shown in Fig. 28

```
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                          future_messages                 = tl inputs
5                          outbound_messages_this_timestep = [(logic1 t), (logic2 t)]
6                          logic1 t = function (inbound_messages_this_timestep)
7                          logic2 t = function (inbound_messages_this_timestep)
8                          endwhere
```

Figure 28: *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 $t-2$. Allowing a wrapper to access this information can be done by delaying its arrival at the wrapper. This can be done by sending a message through a delay agent, this agent will then delay the value until the time it needs to be accessed by the wrapper requiring it.

A delay agent is a new wrapper named in the format $functionname_delay$, where $functionname$ is the name for the recurrence relation value that it is delaying, such $jf1$. Use of this delay agent

requires reverse engineering of the model, the agent needs to know what value is being delayed and for how long, and the wrapper requiring the delay now needs to use the value returned by the delay agent, at the current time step, instead of the wrapper that produces the initial value.

The delay agent needs to have access to the initial values of the logic it is delaying. The wrapper then takes in the value that it is delaying and then outputs it. This process delays a value to $t-2$, as any value the delay agent reads in is already from $t-1$ and when another wrapper reads the value from the delay agent this will be delayed another time step to $t-2$. For delays greater then $t-2$ the delay agent will output two or more values and read in two or more values, with the output list being values delay in the following manner, $[.., t-4, t-3, t-2]$. The head of this list is read by the wrapper requiring the delayed value, all other items in this list are read by the delay agent, which moves each item up the list and adds the current value of the value being delayed to the end of the list.

A delay agent, for delaying a value to $t-2$, will have a form similar to that shown in Fig. 29. Since this agent is treated the same as any other wrapper, it will also be added to the outputs lists and be part of the inputs.

```
1  jf1 delay inputs t = outbound messages this timestep : (jf1 delay future messages (
       t+1))
2                        where
3                        inbound messages this timestep = hd inputs
4                        future messages = tl inputs outbound messages this timestep =
       [( jf1 delayed t) ]
5                        jf1 delayed 0 = k
6                        jf1 delayed t = (( inbound messages this timestep !1) !0)
7                        endwhere
```

Figure 29: *Example of a delay wrapper that delays the first value from the second wrapper, so that a value from $t-2$ is received at t.*

In this translation all information is passed as an input to the wrapper so even internal functions are referred to through the input list. The list head is taken at each time step and contains a list where each item is the output by each wrapper defined in the experiment, with delay wrappers added to the end of this list.[8] Each output by a wrappers is a list of the values produced by that wrapper at the previous time step. To access a required value this input list must be indexed twice, first return the values from the correct wrapper and then to return the correct value outputted by that wrapper. Translating Fig. 26 in this way will produce Fig. 30.

---

[8]If there are multiple delay wrappers, reverse engineering is used to make sure that the wrappers are referencing the correct delay agents.

```
1  main
2  ((s_outputs!(3+1))!0)!0
3
4  init
5  q = 4
6  k = 16
7
8  where
9  s_outputs  = niloutputs : (transpose [i_wrapper s_outputs 0, j_wrapper s_outputs 0,
        k_wrapper s_outputs 0, jf1_delay s_outputs 0 ])
10               where
11               niloutputs = [[ nullvalue], [ nullvalue, nullvalue], [ nullvalue], [
      nullvalue ]]
12               nullvalue = 0
13               endwhere
14 i_wrapper inputs t = outbound_messages_this_timestep : (i_wrapper future_messages (
      t+1))
15                    where
16                    inbound_messages_this_timestep  = hd inputs
17                    future_messages                 = tl inputs
18                    outbound_messages_this_timestep = [( i_f1  t)]
19                    i_f1  0 = q
20                    i_f1  t = cond  (t<0) 0
21                    (  ((inbound_messages_this_timestep!0)!0)  +((
      inbound_messages_this_timestep!3)!0))
22                    endwhere
```

(a) From Fig. 30, main, init, the outputs list, and i wrapper.

```
 1 j_wrapper inputs t = outbound_messages_this_timestep : (j_wrapper future_messages (
     t+1))
 2                              where
 3                              inbound_messages_this_timestep  = hd inputs
 4                              future_messages                 = tl inputs
 5                              outbound_messages_this_timestep = [(j_f1 t), (j_f2 t)]
 6                              j_f1 0 = k
 7                              j_f1 t = cond (t<0) k_f1_last_timestep
 8                                       (25 + (j_f1_last_timestep * i_f1_last_timestep)+
     j_f2_last_timestep)
 9                                  where
10                                  i_f1_last_timestep = ((inbound_messages_this_timestep
     !0)!0)
11                                  j_f1_last_timestep = ((inbound_messages_this_timestep
     !1)!0)
12                                  j_f2_last_timestep = ((inbound_messages_this_timestep
     !1)!1)
13                                  k_f1_last_timestep = ((inbound_messages_this_timestep
     !2)!0)
14                                  endwhere
15                              j_f2 0 = k+q
16                              j_f2 t = cond (t<3) 27 (j_f1_last_timestep + 27)
17                                  where
18                                  j_f1_last_timestep = ((inbound_messages_this_timestep
     !1)!0)
19                                  endwhere
20                              endwhere
21 k_wrapper inputs t = outbound_messages_this_timestep : (k_wrapper future_messages (
     t+1))
22                              where
23                              inbound_messages_this_timestep  = hd inputs
24                              future_messages                 = tl inputs
25                              outbound_messages_this_timestep = [k_f1 t]
26                              k_f1 0 = 0
27                              k_f1 t = k + q + j_f1_last_timestep
28                                  where
29                                  j_f1_last_timestep = ((inbound_messages_this_timestep
     !1)!0)
30                                  endwhere
31                              endwhere
32 jf1_delay inputs t = outbound_messages_this_timestep : (jf1_delay future_messages (
     t+1))
33                              where
34                              inbound_messages_this_timestep  = hd inputs
35                              future_messages                 = tl inputs
36                              outbound_messages_this_timestep = [(jf1_delayed t)]
37                              jf1_delayed 0 = k
38                              jf1_delayed t = ((inbound_messages_this_timestep!1)!0)
39                              endwhere
```

(b) From Fig. 30, *j* and *k* wrapper functions.

Figure 30: *Translation of Fig. 26 to allow wrappers to access information through input values, and to limit the information given to only that from the previous time step. Figure 30a shows the experiment, the outputs list and the i wrapper, and Fig. 30b shows the j and k wrappers.*

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 then 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 meant for it.

The changes required to the output list function ($s\_outputs$) is purely related to the *nullvalue* aspect of it. The *nullvalue*, which were previously zero now have 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 wrapper id and the value it wishes to send. The 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. 31.

```
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)
6                        endwhere
```

Figure 31: *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). Here logic1 is just used to show where the logic would be and that this produces the output values, as well as what inputs this takes in.*

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. 32, 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.

```
1  i_wrapper inputs t = outbound_messages_this_timestep : (i_wrapper future_messages (
       t+1))
2                          where
3                          inbound_messages_this_timestep  = map f (hd inputs)
4                                                              where
5                                                              f xs = map g xs
6                                                              g xs = filter h xs
7                                                              h (a,b,c) = ((b=1) \/ (a,b,c
       ) = (0,0,0))
8                                                            endwhere
9                          future_messages                 = tl inputs
10                         outbound_messages_this_timestep = [ [messages]]
11                         logic t = function (inbound_messages_this_timestep)
12                         endwhere
```

Figure 32: *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 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 *thd3.hd* is used, this is equivalent to *thd3(hd input)*, where the command *thd3* returns the third item in a three item tuple. Translating Fig. 30 with these steps will lead to Fig. 33.

```
 1  main
 2  (thd3.hd)  (((s_outputs!(3+1))!0)!0)
 3
 4  init
 5  q = 4
 6  k = 16
 7
 8  where
 9  s_outputs   = niloutputs :  transpose [ i_wrapper s_outputs 0 ,j_wrapper s_outputs 0
        ,k_wrapper s_outputs 0,  jf1_delay s_outputs 0   ]
10               where
11               niloutputs = [[newnullvalue], [newnullvalue , newnullvalue], [
     newnullvalue], [newnullvalue]]
12               nullvalue = [(0 ,0 ,0)]
13               endwhere
14  i_wrapper  inputs  t = outbound_messages_this_timestep : (i_wrapper future_messages (
     t+1))
15                    where
16                    inbound_messages_this_timestep  = map f (hd inputs)
17                                                 where
18                                                 f  xs = map g  xs
19                                                 g  xs = filter h  xs
20                                                 h (a ,b ,c) = ((b=1) \/ (a ,b ,c
     ) = (0 ,0 ,0))
21                                                 endwhere
22                    future_messages                = tl inputs
23                    outbound_messages_this_timestep = [ [ (1 ,1 , res ) ,(1 ,2 , res ) ]]
24                                                 where
25                                                 res = i_f1 t
26                                                 endwhere
27                    i_f1  0 = q
28                    i_f1  t = cond (t<0) 0 (i_f1_last_timestep +
     j_f1_2last_timestep)
29                           where
30                           i_f1_last_timestep  = (thd3.hd) ((
     inbound_messages_this_timestep!0)!0)
31                           j_f1_2last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!3)!0)
32                           endwhere
33                    endwhere
```

(a) From Fig. 33, main, init, the outputs list and *i_wrapper*.

```
 1  j_wrapper inputs t = outbound_messages_this_timestep : (j_wrapper future_messages (
        t+1))
 2                          where
 3                          inbound_messages_this_timestep  = map f (hd inputs)
 4                                                              where
 5                                                              f xs = map g xs
 6                                                              g xs = filter h xs
 7                                                              h (a,b,c) = ((b=2) \/ (a,b,c
        ) = (0,0,0))
 8                                                              endwhere
 9                          future_messages                 = tl inputs
10                          outbound_messages_this_timestep = [ [ (2,4,res1),(2,2,res1)
        ,(2,3,res1)],[(2,2,res2)]]
11                                                              where
12                                                              res1 = j_f1 t
13                                                              res2 = j_f2 t
14                                                              endwhere
15                          j_f1  0 = k
16                          j_f1  t = cond (t<0) k_f1_last_timestep
17                                  (25 + (j_f1_last_timestep * i_f1_last_timestep)+
        j_f2_last_timestep)
18                                      where
19                                      i_f1_last_timestep = (thd3.hd) ((
        inbound_messages_this_timestep!0)!0)
20                                      j_f1_last_timestep = (thd3.hd) ((
        inbound_messages_this_timestep!1)!0)
21                                      j_f2_last_timestep = (thd3.hd) ((
        inbound_messages_this_timestep!1)!1)
22                                      k_f1_last_timestep = (thd3.hd) ((
        inbound_messages_this_timestep!2)!0)
23                                          endwhere
24                          j_f2  0 = k+q
25                          j_f2  t = cond (t<3) 27 (j_f1_last_timestep + 27)
26                                      where
27                                      j_f1_last_timestep = (thd3.hd) ((
        inbound_messages_this_timestep!1)!0)
28                                          endwhere
29                          endwhere
```

(b) From Fig. 33, *j_wrapper*.

```
1  k_wrapper inputs t = outbound_messages_this_timestep : (k_wrapper future_messages (
       t+1))
2                            where
3                            inbound_messages_this_timestep  = map f (hd inputs)
4                                                               where
5                                                               f xs = map g xs
6                                                               g xs = filter h xs
7                                                               h (a,b,c) = ((b=3) \/ (a,b,c
       ) = (0,0,0))
8                                                               endwhere
9                            future_messages                 = tl inputs
10                           outbound_messages_this_timestep = [[(3,2,k_f1 t)]]
11                           k_f1 0 = 0
12                           k_f1 t = k + q + j_f1_last_timestep
13                                    where
14                                    j_f1_last_timestep = (thd3.hd) ((
       inbound_messages_this_timestep!1)!0)
15                                    endwhere
16                           endwhere
17 jf1_delay inputs t = outbound_messages_this_timestep : (jf1_delay future_messages (
       t+1))
18                           where
19                           inbound_messages_this_timestep  = map f (hd inputs)
20                                                               where
21                                                               f xs = map g xs
22                                                               g xs = filter h xs
23                                                               h (a,b,c) = ((b=4) \/ (a,b,c
       ) = (0,0,0))
24                                                               endwhere
25                           future_messages                 = tl inputs
26                           outbound_messages_this_timestep = [[(4, 1, jf1_delayed t)]]
27                           jf1_delayed 0 = k
28                           jf1_delayed t = (thd3.hd) ((inbound_messages_this_timestep!1)
       !0)
29                           endwhere
```

(c) From Fig. 33, $k\_wrapper$ and $jf1\_delay$.

Figure 33: *Translation of Fig. 30 to allow messages to be sent as a tuple in the form (from, to, value) and for wrappers to sort these messages as to only used ones addressed to them. Figure 33a shows the experiment, the outputs list and the i wrapper, Fig. 33b shows the wrapper function for j, and Fig. 33c shows the wrapper function for k and the wrapper function for the delay agent $jf1\_delay$.*

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 by 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.

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 taken 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. 34.

```
1  h_sim  x  =  outputfunction  (x)
2            where
3            newnullvalue  =  [(0,0,0)]
4            niloutputs    =  listfuction  (newnullvalue)
5            outputs       =  niloutputs  :  redactedoutputs
6            redactedoutputs  =  transpose  [  i_wrapper  i_inputs  0  ,j_wrapper  j_inputs  0]
7                                where
8                                i_inputs    =  map  (map  (map  (filter  (h  1))))  outputs
9                                j_inputs    =  map  (map  (map  (filter  (h  2))))  outputs
10                               h  x  (a,b,c)  =  ((b=x)  \/  (a,b,c)  =  (0,0,0))
11                               endwhere
12           endwhere
```

Figure 34: *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. The function listfuction represents the null value list, which uses the value newnullvalue.*

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 *inputs* list to access the messages at the current time step.

Translating Fig. 33 to contain a harness function with this functionality will result in the experiment shown in Fig. 35.

```
 1  main
 2  h_sim (3+1)
 3
 4  init
 5  q = 4
 6  k = 16
 7
 8  where
 9  h_sim x = (thd3.hd) (((outputs!x)!0)!0)
10          where
11          newnullvalue = [(0,0,0)]
12          niloutputs   = [[newnullvalue], [newnullvalue, newnullvalue], [
    newnullvalue], [newnullvalue]]
13          outputs      = niloutputs : redactedoutputs
14          redactedoutputs = transpose [ i_wrapper i_inputs 0 ,j_wrapper j_inputs 0
    ,k_wrapper k_inputs 0, jf1_delay jf1_inputs 0 ]
15                          where
16                          i_inputs   = map (map (map (filter (h 1)))) outputs
17                          j_inputs   = map (map (map (filter (h 2)))) outputs
18                          k_inputs   = map (map (map (filter (h 3)))) outputs
19                          jf1_inputs = map (map (map (filter (h 4)))) outputs
20                          h x (a,b,c) = ((b=x) \/ (a,b,c) = (0,0,0))
21                          endwhere
22          endwhere
```

(a) From Fig. 35, main, init, and the sim wrapper.

```
 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                        future_messages                 = tl inputs
 5                        outbound_messages_this_timestep = [ [ (1,1,res),(1,2,res)]]
 6                                                          where
 7                                                          res = i_f1 t
 8                                                          endwhere
 9                        i_f1 0 = q
10                        i_f1 t = cond (t<0) 0 (i_f1_last_timestep +
     j_f1_2last_timestep)
11                                where
12                                i_f1_last_timestep  = (thd3.hd) ((
     inbound_messages_this_timestep!0)!0)
13                                j_f1_2last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!3)!0)
14                                endwhere
15                        endwhere
16 j_wrapper inputs t = outbound_messages_this_timestep : (j_wrapper future_messages (
     t+1))
17                        where
18                        inbound_messages_this_timestep  = hd inputs
19                        future_messages                 = tl inputs
20                        outbound_messages_this_timestep = [ [ (2,4,res1),(2,2,res1)
     ,(2,3,res1)],[(2,2,res2)]]
21                                                          where
22                                                          res1 = j_f1 t
23                                                          res2 = j_f2 t
24                                                          endwhere
25                        j_f1 0 = k
26                        j_f1 t = cond (t<0) k_f1_last_timestep
27                                      (25 + (j_f1_last_timestep * i_f1_last_timestep)+
     j_f2_last_timestep)
28                                where
29                                i_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!0)!0)
30                                j_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!1)!0)
31                                j_f2_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!1)!1)
32                                k_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!2)!0)
33                                endwhere
34                        j_f2 0 = k+q
35                        j_f2 t = cond (t<3) 27 (j_f1_last_timestep + 27)
36                                where
37                                j_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!1)!0)
38                                endwhere
39                        endwhere
```

(b) From Fig. 35, *i_wrapper* and *j_wrapper*.

```
1  k_wrapper inputs t = outbound_messages_this_timestep : (k_wrapper future_messages (
       t+1))
2                             where
3                             inbound_messages_this_timestep  = hd inputs
4                             future_messages                 = tl inputs
5                             outbound_messages_this_timestep = [[(3,2,k_f1 t)]]
6                             k_f1 0 = 0
7                             k_f1 t = k + q + j_f1_last_timestep
8                                  where
9                                  j_f1_last_timestep = (thd3.hd) ((
       inbound_messages_this_timestep!1)!0)
10                                 endwhere
11                            endwhere
12 jf1_delay inputs t = outbound_messages_this_timestep : (jf1_delay future_messages (
       t+1))
13                            where
14                            inbound_messages_this_timestep  = hd inputs
15                            future_messages                 = tl inputs
16                            outbound_messages_this_timestep = [[(4,1,jf1_delayed t)]]
17                            jf1_delayed 0 = k
18                            jf1_delayed t = (thd3.hd) ((inbound_messages_this_timestep!1)
       !0)
19                            endwhere
```

(c) From Fig. 35, *k_wrapper* and *jf1_delay*.

Figure 35: *Translation of Fig. 33 to contain a harness function that deals with message sorting and passing, as well as outputting the experiment value. Figure 35a shows the experiment and sim wrapper, Fig. 35b shows the wrapper functions for groups i and j, and Fig. 35c shows the wrapper function for k and the wrapper function for the delay agent jf1_delay.*

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.

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 $zip2$ 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. 36.

```
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))
12                                     endwhere
13                      endwhere
```

Figure 36: *Example of how the harness function h_sim is formatted when using runtime arguments. The function listfuction represents the null value list, which uses the value newnullvalue.*

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 mark the sender in outbound messages. A wrapper function with these changes takes the form shown in Fig. 37.

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

Figure 37: *Example of how a wrapper function is formatted when passing the arguments: localstate, args, inputs and id. Here logic1 is just used to show where the logic would be and that this produces the output values, as well as what inputs this takes in.*

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. 35 using these steps results in the experiment shown in Fig. 38.

```
 1  main
 2  h_sim  (3+1)  runtime_args  agentinfo
 3
 4  where
 5  runtime_args  =  [(Arg ("q",4)), (Arg ("k", 16))]
 6  agentinfo  =  [(i_wrapper,[]), (j_wrapper, []), (k_wrapper, []), (jf1_delay, [])]
 7  h_sim  x  args  agents  =  (thd3.hd) (((snd (outputs!x))!0)!0)
 8                              where
 9                              newnullvalue  =  [(0,0,0)]
10                              niloutputs  =  [[newnullvalue], [newnullvalue, newnullvalue], [
    newnullvalue], [newnullvalue]]
11                              outputs  =  zip2  [0..]  (niloutputs : (redactedoutputs))
12                              redactedoutputs  =  transpose  (map apply_to_args  (zip2 [1..]
    agents))
13                                      where
14                                      apply_to_args (agentid, (f,
    broadcastsubscriptions))  =  f  Emptyagentstate  args  (myoutputs agentid)  agentid
15                                      myoutputs  id  =  map (f id) outputs
16                                      f  x  (t, xs)  =  (t, map (map (filter (h x)))  xs)
17                                      h  x  (a,b,c)  =  ((b=x) \/ (a,b,c) = (0,0,0))
18                                          endwhere
19                              endwhere
20  i_wrapper  localstate  args  inputs  id  =  outbound_messages_this_timestep : (i_wrapper
    localstate args future_messages id)
21                                              where
22                                              q  =  (getargval.hd) (filter ((="q").getargstr)
     args)
23                                              k  =  (getargval.hd) (filter ((="k").getargstr)
     args)
24                                              inbound_messages_this_timestep  =  snd (hd
    inputs)
25                                              t                                =  fst (hd
    inputs)
26                                              future_messages                  =  tl inputs
27                                              outbound_messages_this_timestep = [ [ (id,1,
    res),(id,2,res)]]
28                                                      where
29                                                      res = i_f1
    t
30                                                      endwhere
31                                              i_f1  0  =  q
32                                              i_f1  t  =  cond  (t<0)  0  (i_f1_last_timestep +
    j_f1_2last_timestep)
33                                                      where
34                                                      i_f1_last_timestep = (thd3.hd) ((
    inbound_messages_this_timestep!0)!0)
35                                                      j_f1_2last_timestep = (thd3.hd) ((
    inbound_messages_this_timestep!3)!0)
36                                                      endwhere
37                                              endwhere
```

(a) From Fig. 38, main, init, the sim wrapper, and the i wrapper.

```
1  j_wrapper localstate args inputs id = outbound_messages_this_timestep : (j_wrapper
       localstate args future_messages id)
2                                          where
3                                          q = (getargval.hd) (filter ((="q").getargstr)
     args)
4                                          k = (getargval.hd) (filter ((="k").getargstr)
     args)
5                                          inbound_messages_this_timestep  = snd (hd
     inputs)
6                                          t                               = fst (hd
     inputs)
7                                          future_messages                 = tl inputs
8                                          outbound_messages_this_timestep = [ [ (id,4,
     res1),(id,2,res1),(id,3,res1)],[ (id,2,res2)]]
9                                                              where
10                                                             res1 = j_f1
     t
11                                                             res2 = j_f2
     t
12                                                             endwhere
13                                       j_f1  0 = k
14                                       j_f1  t = cond (t<0) k_f1_last_timestep
15                                                     (25 + (j_f1_last_timestep *
     i_f1_last_timestep)+j_f2_last_timestep)
16                                                 where
17                                                 i_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!0)!0)
18                                                 j_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!1)!0)
19                                                 j_f2_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!1)!1)
20                                                 k_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!2)!0)
21                                                 endwhere
22                                       j_f2  0 = k+q
23                                       j_f2  t = cond (t<3) 27 (j_f1_last_timestep +
     27)
24                                                 where
25                                                 j_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!1)!0)
26                                                 endwhere
27                                     endwhere
```

(b) From Fig. 38, *j_wrapper*.

```
1  k_wrapper localstate args inputs id = outbound_messages_this_timestep : (k_wrapper
       localstate args future_messages id)
2                                              where
3                                              q = (getargval.hd) (filter ((="q").getargstr)
    args)
4                                              k = (getargval.hd) (filter ((="k").getargstr)
    args)
5                                              inbound_messages_this_timestep  = snd (hd
    inputs)
6                                              t                               = fst (hd
    inputs)
7                                              future_messages                 = tl inputs
8                                              outbound_messages_this_timestep = [[(id,2,
    k_f1 t)]]
9                                              k_f1 0 = 0
10                                             k_f1 t = k + q + j_f1_last_timestep
11                                                     where
12                                                     j_f1_last_timestep = (thd3.hd) ((
    inbound_messages_this_timestep!1)!0)
13                                                     endwhere
14                                             endwhere
15 jf1_delay localstate args inputs id = outbound_messages_this_timestep : (jf1_delay
       localstate args future_messages id)
16                                             where
17                                             inbound_messages_this_timestep  = snd (hd
    inputs)
18                                             future_messages                 = tl inputs
19                                             t                               = fst (hd
    inputs)
20                                             outbound_messages_this_timestep = [[(id,1,
    jf1_delayed t)]]
21                                             k = (getargval.hd) (filter ((="k").getargstr)
    args)
22                                             jf1_delayed 0 = k
23                                             jf1_delayed t = (thd3.hd) ((
    inbound_messages_this_timestep!1)!0)
24                                             endwhere
```

(c) From Fig. 38, *k_wrapper* and *jf1_delay*.

Figure 38: *Translation of Fig. 35 to contain args and agents as an input to the harness, and to contain localstate, args, inputs and id as inputs for the wrapper functions. Figure 38a shows the experiment, sim wrapper and i wrapper, Fig. 38b shows the wrapper functions for group j, and Fig. 38c shows the wrapper function for k and the wrapper function for the delay agent jf1_delay.*

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.

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 $Message\ (from, to)\ [(Arg("fun\_name", value))]$. 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, the arguments from these messages are then turned into a list, which is in turn filtered to return the value in the argument associated with the function name of interest. The functions $getmsgfrom$ and $getmsgargs$ are used, these return, the from ids and the arguments, respectively. The function $concat$ is also used, this turns a list of lists into a list. An example of this process is shown in Fig. 39.

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

Figure 39: *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. 40. 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.

```
1  h_sim x runtime_args agentinfo = (getargval.hd.getmsgargs) (((snd (outputs!x))!0)
      !0)
2                                          where
3                                          outputs = (0,[]):timed_transposed_msgs
4                                          timed_transposed_msgs = zip2 [1..] transposed_msgs
5                                          transposed_msgs = transpose allmessages
6                                          allmessages = map apply_to_args (zip2 [1..]
      agentinfo)
7                                              where
8                                              apply_to_args (agentid, (agentfn,
      broadcastsubscriptions)) = agentfn Emptyagentstate runtime_args (redacted_msgs
      agentid outputs) agentid
9                                              redacted_msgs id outputs = map (f id
      ) outputs
10                                             f x (time, xs)                = (time
      , concat (map (filter (h x)) xs))
11                                             h x (Message (f,t) args)     = (t=x)
12                                             endwhere
13                                      endwhere
```

Figure 40: *Simulator harness for translation comparable to InterDynes simulator harness.*

After these translations are applied to the experiment in Fig. 41 they create the experiment shown in Fig. 41.

```
1  main
2  h_sim (3+1) runtime_args agentinfo
3
4  where
5  runtime_args = [(Arg ("q",4)), (Arg ("k", 16))]
6  agentinfo = [(i_wrapper,[]), (j_wrapper, []), (k_wrapper, []), (jf1_delay, [])]
7  h_sim x runtime_args agentinfo = (getargval.hd.getmsgargs) (((snd (outputs!x))!0)
      !0)
8                                          where
9                                          outputs = (0,[]):timed_transposed_msgs
10                                         timed_transposed_msgs = zip2 [1..] transposed_msgs
11                                         transposed_msgs = transpose allmessages
12                                         allmessages = map apply_to_args (zip2 [1..]
      agentinfo)
13                                             where
14                                             apply_to_args (agentid, (agentfn,
      broadcastsubscriptions)) = agentfn Emptyagentstate runtime_args (redacted_msgs
      agentid outputs) agentid
15                                             redacted_msgs id outputs = map (f id
      ) outputs
16                                             f x (time, xs)                = (time,
      concat (map (filter (h x)) xs))
17                                             h x (Message (f,t) args)     = (t=x)
18                                             endwhere
19                                     endwhere
```

(a) From Fig. 41, main, init, and the sim wrapper.

```
1  i_wrapper localstate args inputs id = outbound_messages_this_timestep : (i_wrapper
       localstate args future_messages id)
2                                          where
3                                          q = (getargval.hd) (filter ((="q").getargstr)
    args)
4                                          k = (getargval.hd) (filter ((="k").getargstr)
    args)
5                                          inbound_messages_this_timestep  = snd (hd
    inputs)
6                                          t                               = fst (hd
    inputs)
7                                          future_messages                 = tl inputs
8                                          outbound_messages_this_timestep = [ Message (
    id,1)  [(Arg ("i_f1",res))],Message (id,2)  [(Arg ("i_f1",res))]]
9                                                                  where
10                                                                 res = i_f1
    t
11                                                                 endwhere
12                                 i_f1  0 = q
13                                 i_f1  t = cond (t<0) 0 (i_f1_last_timestep +
    j_f1_last_timestep)
14                                          where
15                                          msgs_from_i        = filter ((=1).
    getmsgfrom) inbound_messages_this_timestep
16                                          args_from_i        = concat (map
    getmsgargs msgs_from_i)
17                                          i_f1_last_timestep = (getargval.hd)
    (filter ((="i_f1").getargstr) args_from_i)
18                                          msgs_from_jd       = filter ((=4).
    getmsgfrom) inbound_messages_this_timestep
19                                          args_from_jd       = concat (map
    getmsgargs msgs_from_j)
20                                          j_f1_2last_timestep = (getargval.hd)
     (filter ((="j_f1d").getargstr) args_from_jd)
21                                          endwhere
22                                 endwhere
```

(b) From Fig. 41, *i_wrapper*.

```
1  j_wrapper localstate args inputs id = outbound_messages_this_timestep : (j_wrapper
      localstate args future_messages id)
2                                           where
3                                           q = (getargval.hd) (filter ((="q").getargstr)
    args)
4                                           k = (getargval.hd) (filter ((="k").getargstr)
    args)
5                                           inbound_messages_this_timestep = snd (hd
   inputs)
6                                           t                              = fst (hd
   inputs)
7                                           future_messages                = tl inputs
8                                           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))]]
9                                                                     where
10                                                                    res1 = j_f1
   t
11                                                                    res2 = j_f2
   t
12                                                                    endwhere
13                                          j_f1 0 = k
14                                          j_f1 t = cond (t<0) k_f1_last_timestep
15                                                   (25 + (j_f1_last_timestep *
   i_f1_last_timestep)+j_f2_last_timestep)
16                                                   where
17                                                   msgs_from_i        = filter ((=1).
   getmsgfrom) inbound_messages_this_timestep
18                                                   args_from_i        = concat (map
   getmsgargs msgs_from_i)
19                                                   i_f1_last_timestep = (getargval.hd)
   (filter ((="i_f1").getargstr) args_from_i)
20                                                   msgs_from_j        = filter ((=2).
   getmsgfrom) inbound_messages_this_timestep
21                                                   args_from_j        = concat (map
   getmsgargs msgs_from_j)
22                                                   j_f1_last_timestep = (getargval.hd)
   (filter ((="j_f1").getargstr) args_from_j)
23                                                   j_f2_last_timestep = (getargval.hd)
   (filter ((="j_f2").getargstr) args_from_j)
24                                                   msgs_from_k        = filter ((=3).
   getmsgfrom) inbound_messages_this_timestep
25                                                   args_from_k        = concat (map
   getmsgargs msgs_from_k)
26                                                   k_f1_last_timestep = (getargval.hd)
   (filter ((="k_f1").getargstr) args_from_j)
27                                                   endwhere
28                                          j_f2 0 = k+q
29                                          j_f2 t = cond (t<3) 27 (j_f1_last_timestep +
   27)
30                                                   where
31                                                   msgs_from_j        = filter ((=2).
   getmsgfrom) inbound_messages_this_timestep
32                                                   args_from_j        = concat (map
   getmsgargs msgs_from_j)
33                                                   j_f1_last_timestep = (getargval.hd)
   (filter ((="j_f1").getargstr) args_from_j)
34                                                   endwhere
35                                          endwhere
```

(c) From Fig. 41, *j_wrapper*.

```
1  k_wrapper localstate args inputs id = outbound_messages_this_timestep : (k_wrapper
       localstate args future_messages id)
2                                          where
3                                          q = (getargval.hd) (filter ((="q").getargstr)
    args)
4                                          k = (getargval.hd) (filter ((="k").getargstr)
    args)
5                                          inbound_messages_this_timestep  = snd (hd
   inputs)
6                                          t                               = fst (hd
   inputs)
7                                          future_messages                = tl inputs
8                                          outbound_messages_this_timestep = [Message (
   id,2) [(Arg ("k_f1",k_f1 t))]]
9                                          k_f1 0 = 0
10                                         k_f1 t = k + q + j_f1_last_timestep
11                                             where
12                                             msgs_from_j        = filter ((=2).
   getmsgfrom) inbound_messages_this_timestep
13                                             args_from_j        = concat (map
   getmsgargs msgs_from_j)
14                                             j_f1_last_timestep = (getargval.hd)
   (filter ((="j_f1").getargstr) args_from_j)
15                                             endwhere
16                                         endwhere
17 jf1_delay localstate args inputs id = outbound_messages_this_timestep : (jf1_delay
       localstate args future_messages id)
18                                         where
19                                         k = (getargval.hd) (filter ((="k").getargstr)
    args)
20                                         inbound_messages_this_timestep  = snd (hd
   inputs)
21                                         t                               = fst (hd
   inputs)
22                                         future_messages                = tl inputs
23                                         outbound_messages_this_timestep = [Message (
   id, 1) [(Arg ("j_f1d", jf1_delayed t))]]
24                                         msgs_from_j        = filter ((=2).getmsgfrom)
    inbound_messages_this_timestep
25                                         args_from_j        = concat (map getmsgargs
   msgs_from_j)
26                                         jf1_delayed 0     = k
27                                         jf1_delayed t     = (getargval.hd) (filter
   ((="j_f1").getargstr) args_from_j)
28                                         endwhere
```

(d) From Fig. 41, *k_wrapper* and *jf1_delay*.

Figure 41: *Translation of Fig. 38 to contain the Message constructor and have a harness function comparable to that of InterDynes. Figure 41a shows the experiment and sim wrapper, Fig. 41b shows the wrapper function for groups i, Fig. 41c shows the wrapper function for j, and Fig. 41d shows the wrapper function for k and the wrapper function for the delay agent jf1_delay.*

6   Validation and Testing

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 a reduced-functionality version 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 target state of the translation to a reduced-functionality version of 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

This section attempts to validate whether the final output of the translation is a valid InterDyne simulation. It does this by:

1. A visual inspection of the output syntax and comparison with InterDyne syntax

2. Comparison of the type of the output's harness function with the type of InterDyne's harness function

3. Comparison of the type of each output agent function with the type of an InterDyne agent function

The reduced-functionality version of InterDyne being compared to has a harness of the form shown in Fig 42, a harness type shown in Fig. 43, and a agent type shown in Fig. 44.

```
1  sim x runtime_args agentinfo = (getargval.hd.getmsgargs) (((snd (outputs!x))!0)!0)
2                                  where
3                                  outputs                = (0,[]):timed_transposed_msgs
4                                  timed_transposed_msgs = zip [1..] transposed_msgs
5                                  transposed_msgs        = transpose allmessages
6                                  allmessages            = map apply_to_args (zip [1..]
       agentinfo)
7                                                          where
8                                                          apply_to_args (agentid, (
       agentfn,broadcastsubscriptions)) = agentfn Emptyagentstate runtime_args (
       redacted_msgs agentid outputs) agentid
9                                                          redacted_msgs id outputs =
       map (f id) outputs
10                                                         f x (time, xs)            = (
       time, concat (map (filter (h x)) xs))
11                                                         h x (Message (f,t) args) = (
       t==x)
```

Figure 42: *Harness for a reduced-functionality version of InterDyne.*

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

Figure 43: *Harness type for a reduced-functionality version of InterDyne.*

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

Figure 44: *Agent type for a reduced-functionality version of InterDyne.*

Looking at the simulator harness for InterDyne in Fig. 42 and comparing this to the simulator harness in the final target stat in Fig. 40, one can see that these two harness are identical in form and very similar in syntax. The harness produced in the final target state of the translation has not yet been converted into Haskell (the language in which InterDyne is written), this explains the difference in syntax between the two harnesses.

The type of the harness produced by the translation process is show in Fig. 45, as can be seen this type adheres exactly to the type of InterDyne harness shown in Fig. 43.

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

Figure 45: *Type of the harness in the final target state of the translation.*

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, and not the form. The type of an agent created in this translation is shown in Fig. 46, comparing this to the type of an InterDyne agent, shown in Fig. 44, it can be seen that these types exactly match.

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

Figure 46: *Type of the agents in the final target state of the translation.*

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 the correctness in these steps is investigated from a extensionality view point; this view states that two objects are equal if they have the same external behaviour [81].

For this translation the output produced by running the target state of each step is compared. Following the logic of extensionality, if these outputs are equal then so to are the target states that produced them and by association the translation.

This numerical equivalence is assessed by translating each step into its Miranda equivalent and then executing the produced program. This conversion was done using a translation program (shown in Appendix A), the output of this program were then edited to match the Miranda syntax and to allow for an input value of t to replace the static value of 3 used in the

translation. These programs were then compiled into one file (shown in Appendix A) and executed for the time steps (0, 1, 2, 3 and 4).

Table 1 shows the results from these runs, as can been seen all steps produced the same output. This shows, at least for the values tested, there exists a numerical equivalence between each target state of the steps of the translation and hence between the initial recurrence relation model and the final agent-based model.

Table 1: *The output value of the initial recurrence relation model and each translation step at the five different time steps (0, 1, 2, 3, 4, 5).*

Input	Initial	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
0	4	4	4	4	4	4	4	4	4
1	20	20	20	20	20	20	20	20	20
2	36	36	36	36	36	36	36	36	36
3	145	145	145	145	145	145	145	145	145
4	2377	2377	2377	2377	2377	2377	2377	2377	2377

This equivalence suggests that each step is correct and that the entire translation produces an agent-based model that numerically represents the initial recurrence relation model.

6.3   Automated Translation Implementation

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? Implementing this automation is outside the scope of this project, however the first three steps have been achieved and are discussed below (this work is shown in Appendix B).

The three aspects of implementation discussed are, the lexer, the parser, and the first translation step. These steps are written in the Miranda functional language and allows for a text file contain an experiment written in the bespoke language to be translated into the first translation step.

The selected text file is read in by the lexer, this tokenises the list of characters using the lexemes show Fig.47. These lexemes represent each item that is allowed as part of the bespoke recurrence relation language, these can be keywords or symbols. Tokenising these items adds meaning to them, for example a list of characters could be identified as a variable or as a recurrence relation, the lexemes hold this information.

```
1 lexeme::= Init | Var [char] | Recrel [char] [char] | Cond | Main | Fwhere | Optl | Ophd | Intfunc [
     char] | Num num | Opcon | Opbng | Opdiv | Optim | Opmin | Opplus | Rbckt | Lbckt | Rlbkt | Llbkt |
     Opequ | Opne | Oplt | Opgt | Oplet | Opget | Endwhere
```

Figure 47: *Definition of allowed lexemes for tokenising a list of characters.*

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 a 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 spacing, 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 to represents this experiment, shown in Fig. 48.

```
 1 simulation              ::=  Simulation mainblock initblock whereblock
 2 mainblock               ::=  Main recrelname [num]
 3 recrelname              ::=  Recrelname [char]
 4 initblock               ::=  Init [valdef]
 5 whereblock              ::=  Where [recreldef]
 6 valdef                  ::=  Valdef valname expression
 7 valname                 ::=  Valnamedef [char]
 8 recreldef               ::=  Recreldef agentname funcname char [argname] expression
 9 agentname               ::=  Agentnamedef [char]
10 funcname                ::=  Funcnamedef [char]
11 argname                 ::=  Argnamedef [char] | Argnumdef num
12 expression              ::=  SimpleExpression simple_expression | WhereExpression
        simple_expression expr_whereblock
13 simple_expression       ::=  Number num | Valname [char] | Argname [char] | Time char |
        Localvalname [char]
14                              | List [listitem] | Bracketed simple_expression
15                              | Conditional simple_expression simple_expression
        simple_expression
16                              | Binaryop simple_expression operation simple_expression
17                              | Unaryop internop simple_expression
18                              | Application recrelname [simple_expression]
19                              | Localapplication localfuncname [simple_expression]
20 expr_whereblock         ::=  Whereblock [localdefinition]
21 localdefinition         ::=  Localvaldef localvalname expression | Localfuncdef
        localfuncname [argname] expression
22 localfuncname           ::=  Localfuncnamedef [char]
23 localvalname            ::=  Localvalnamedef [char]
24 listitem                ::=  Listitem simple_expression
25 internop                ::=  Head | Tail | Not
26 operation               ::=  Plus | Minus | Times | Divide | Lessthan | Greaterthan |
        Equals | Notequals | Lessthanorequalto
27                              | Greaterthanorequalto | Cons | Index
```

Figure 48: *Parse tree, written in Miranda, to store the structure and data of the passed 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 translation 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.

This implementation can be seen to be syntactically complete as the parse tree shown above matches exactly with the concrete syntax given in Fig. 9. This means that there is no case in the concrete syntax which can not exist in the parse tree definition. Semantic completeness of this implementation is beyond the scope of this project, but would define what is not allowed because it does not make sense in the language despite being syntactically correct, such as a recurrence relation name that has no definition (but is correctly formed).

7 Summary and Conclusion

This project has looked at the use of recurrence relation models and agent-based models within 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. This provides the ability to investigate message passing dynamics, which is something difficult to achieve with recurrence relation models. However in economics agent-based, unlike recurrence relation models, very rarely appear in top journals and in general are 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 work presented in this dissertation combats two main criticisms of agent-based models: their lack of a system-wide formalism and their lack of familiarity to those not accustomed to this method of modelling. This is done by adding a system-wide formalism to an agent-based model as well creating a process to allow for agent-based models to become more transparent to those unfamiliar with them.

A formalism is specified for the agent-based model in the form of an equivalent recurrence relation model, with the transparency given by a step-by-step (with each step being equivalent) translation from this recurrence relation model to the agent-based model.

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

The created bespoke recurrence relation language allows for experiments involving interacting recurrence relations to be fully specified. The language its self is defined with both a concrete and abstract BNF form, as well as details on creating well formed programs within it. The concrete BNF defines what symbols are allowed within this language, and abstract BNF defines the structure this language is allowed to take.

A design for an eight step translation process was created, to translate experiments defined by the bespoke recurrence relations into an agent-based model that takes a reduced form of an InterDyne simulation. This design changes a few small aspects of the model at each step, gradually reshaping the recurrence relation model into an agent-based model, the eighth step hence representing an agent-based model. This process was demonstrated by hand using an example experiment written in the bespoke recurrence relation and translating this into an agent-based model, showing its structure at each step.

Each model presented at each of these steps during the translation, were rewritten as Miranda programs that code be executed for different time steps. These Miranda versions of the steps were then run for a number of steps to establish their numerical equivalency, with each step being shown to return the same output for a given time input. The initial recurrence relation model was also rewritten as a Miranda program and executed for the time time inputs, this program also returned the same values as each of the steps.

The project this dissertation describes, scope was to create both a bespoke recurrence relation language and the design for a step-by-step translation from this language to an agent-based model. Both these aims were achieved and additionally the first three steps of implementing this translation as an automated program have been accomplished. These steps were the creation, in the Miranda language, of a lexer, parser and the first translation step.

This implementation of the first steps, shows the method and potential for implementing the entire translational process. Though this implementation does not improve upon the concept of this project, it does allow for the translation processes to be achieved fair quicker.

This translation process allows for recurrence relations of first order and higher, with

delay agents created to allow for the passing of the required delayed information. This translation also allows for recurrence relations to have more inputs then just time, with these inputs being passed to the final agents as arguments.

The numerical equivalence between each step and between the initial recurrence relation model and final agent-based model, implies the correctness of this translation process and the equivalence of the two models.

The step-by-step nature of the translation allows for it to be smoothly followed and hence be more comprehensible to a reader. This allows those with knowledge of recurrence relation models to gain a deeper understanding of the formation and structure of agent-based models.

This translation also gives an numerically equivalent agent-based model for a given recurrence relation model, allowing this recurrence relation model to act as the system wide formalism for the produced agent-based model.

The achievements of this project are hence:

- The creation bespoke recurrence relation language and formal definition of its syntax.

- The design of a step-by-step translation process, turning a model written in the bespoke recurrence relation language into an agent-based model.

- The validation and testing of this step-by-step translation.

- The creation of the lexer, parser, and first translational step, in a program designed to automate the translation process.

The work presented in this dissertation gives a method that can be used to give a system-wide formalisation to an agent-based model as well as giving a method that allows for one familiar with recurrence relation models to increase their understanding of this technique. This allows for an increased understanding of agent-based models in economics and gives potential of an increase in the use of these models and acceptance of work based on them.

7.1   Discussion and Further work

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

Though the implementation of delays explained given in this dissertation, is correct in preserving the meaning of the origin, for large simulations it could result in the creation of many more agents. This might produce a high over head for a simulation, due to the number of agents and messages, making it difficult to run the final agent-based model. A final step would be to convert from this agent delay method into another one allowed by InterDyne, where the delay agents are incorporated into the simulator harness.

The the correctness of each step, and of the two models, has been argued informally and numerically in this dissertation, however this has not been formally proved. A formal proof of the correctness of each step and hence the two models, is mostly readily achieved by translating each step into lambda calculus and then proving the correctness in this form.

The first three steps of implementation have been achieved in this project, however there are a remaining eight steps to be created. Seven of these steps are the remaining translational steps, for the implementation of these the parse tree storing the model will have to be altered as the translation progresses to allow for new items, such as tuples and types. The final step is to create a function that translates from the parse into a text file that lays out the models in a way that is accommodating to a user, this is will produce the layouts of the form shown during the translation in this dissertation.

Though the translating from a recurrence relation model to an agent-based model fits the aims of this dissertation, further work could be to create a reverse translation and implementing it. This translation would allow a given agent-based model to be translated in a recurrence relation model.

Implementing and combining both of these translation would give a method for readily viewing a system as either a recurrence relation model or an agent-based model, as is required. This method would be of particular use when wanting to preform both static and dynamic analysis to a system, as the most amenable view can be chosen and used.

Appendix A: Program for Numerical Testing

User Manual

Here the code that translates from the target states in the design steps shown in this dissertation is given, along with the code used to numerically test the target states.

This code is written in Miranda, therefore to run it Miranda must be installed; installation instructions and downloads for Miranda can be found through the following link: http://miranda.org.uk/.

The first code shown is for translating the target states in the translation steps into their Miranda form. This can be run after loading the file (using *mira file.m*) by running the command *tomira "import.m" "output.m"*, where the file names given are examples.

Though this code correctly translates the target states into their Miranda form, there are still a number of issues stop this form from then being run. The target states use some syntax that is different in Miranda, as well as some internal functions and types that need to be defined in Miranda.

To run numerical testing the outputs from this code were compiled into one program and edited to run in Miranda, definitions were also given to internal functions and types requiring them. Each step was given the name *main* with the step number appended to the end, for example the result from the first translation step is called *main*1. To run the numerical test the command *testvalue* is used and to test the numerical equivalence of these tests the command *testequiv* is used.

Translation from Design to Miranda

```
1  tomira infile outfile = [Tofile outfile result]
2                           where
3                           input = read infile
4
5                           result = "main = "++blk1++"\n"++"where\n"++blk2++"\n"++blk3
6                           rest0        = losemain input
7                           (blk1, rest1) = findblk1 rest0 []
8                           (blk2, blk3)  = findblk2 rest1 []
9
10                          losemain ('m':'a':'i':'n':xs) = xs
11
12                          findblk1 ('i':'n':'i':'t':xs) blk1 = (blk1, xs)
13                          findblk1 (x:xs)               blk1 = findblk1 xs (blk1++[x
   ])
14
15                          findblk2 ('w':'h':'e':'r':'e':xs) blk2 = (blk2, xs)
16                          findblk2 (x:xs)                   blk2 = findblk2 xs (blk2
   ++[x])
```

Program for Numerical Testing

```
1
2
3
4
5
6  eqivtest x = (main0 x = main1 x = main2 x = main3 x = main4 x = main5 x = main6 x =
       main7 x = main8 x)
7  valuelist x = [main0 x, main1 x, main2 x, main3 x, main4 x, main5 x, main6 x, main7
       x, main8 x]
8
9  testvalue = [(valuelist 0), (valuelist 1), (valuelist 2), (valuelist 3), (valuelist
       4)]
```

```
10  testequiv = [(eqivtest 0), (eqivtest 1), (eqivtest 2), (eqivtest 3), (eqivtest 4)]
11
12  || This Miranda script tests various stages of the translation from
13  || recurrence relations to an agent-based formulation
14  ||
15  || _____
16  ||
17
18  str == [char]
19  double == num
20  int == num
21  lob == [char]      || this is a placeholder - it is not used yet, and should be more
        structured
22
23  || Later (in main6 onwards) we will use arg_t,  agentstate_t, msg_t etc. as follows
        :
24
25  arg_t ::= EmptyArg | Arg (str, double)
26
27  getargstr EmptyArg = error "Can't get argstring from empty arg"
28  getargstr (Arg (s,n)) = s
29
30  getargval EmptyArg = error "Can't get argval from empty arg"
31  getargval (Arg (s,n)) = n
32
33
34  agentstate_t ::= Emptyagentstate | Agentstate (double, double, int, lob)
35
36  msg_t ::= Hiaton | Message (int,int) [arg_t] | Datamessage (int,int) [char]
37  getmsgfrom   Hiaton                   = 0          || Essentially: the sim harness is
        the from/to of all Hiatons
38  getmsgfrom (Message (f,t) args)      = f
39  getmsgfrom (Datamessage (f,t) text) = f
40
41  getmsgto      Hiaton                   = 0          || Essentially: the sim harness is
        the from/to of all Hiatons
42  getmsgto     (Message (f,t) args)     = t
43  getmsgto     (Datamessage (f,t) text) = t
44
45  getmsgargs   Hiaton                   = []
46  getmsgargs (Message (f,t) args)      = args
47  getmsgargs (Datamessage (f,t) text) = error "Datamessage does not have data of type
        arg_t"
48
49  ||agent_t == agentstate_t -> [arg_t] -> [(num, [msg_t])] -> num -> [[msg_t]]
50
51
52
53  || We need functions to get the components of a 3-tuple:
54
55  fst3 (a,b,c) = a
56  snd3 (a,b,c) = b
57  thd3 (a,b,c) = c
58
59
60  || Miranda doesn't have a distfix conditional "if ... then ... else ..."
61  || so we'll create something similar - a ternary conditional called "cond"
62
63  cond x y z = y, if x
64             = z, otherwise
65
66  || We also use the following function:
67
68  s_createlist     f    t = (f t):(s_createlist f (t+1))
```

```
69
70  || ——————————————————————
71
72
73
74  || ═══════════
75  ||  Original
76
77
78  main0 x
79        = i_f1 x
80          where
81          q = 4
82          k = 16
83          i_f1  0 = q
84          i_f1  t = (i_f1  (t−1)) + (j_f1  (t−2))
85          j_f1  (−1) = k
86          j_f1  0 = k
87          j_f1  t = cond (t<0) (k_f1  (t−1))
88                    (25+((j_f1  (t−1))*(i_f1  (t−1))) + (j_f2  (t−1)))
89          j_f2  0 = k+q
90          j_f2  t = cond (t<3) 27 ((j_f1  (t−1)) + 27)
91          k_f1  0 = 0
92          k_f1  t = k + q + (j_f1  (t−1))
93
94
95
96
97  || ═══════════
98  ||  Step 1
99
100 main1 x
101       = i_f1_list ! x
102         where
103         q = 4
104         k = 16
105         i_f1_list = s_createlist i_f1 0
106                       where
107                       i_f1  0 = q
108                       i_f1  1 = (i_f1_list ! 0 ) + k
109                       i_f1  t = (i_f1_list ! (t−1) ) + (j_f1_list ! (t−2))
110         j_f1_list = s_createlist j_f1 0
111                       where
112                       j_f1  0 = k
113                       j_f1  t = cond (t<0) (k_f1_list ! (t−1))
114                                 (25 + ((j_f1_list ! (t−1)) * (i_f1_list ! (t−1))) + (
115    j_f2_list ! (t−1)))
115         j_f2_list = s_createlist j_f2 0
116                       where
117                       j_f2  0 = k+q
118                       j_f2  t = cond (t<3) 27 ((j_f1_list ! (t−1)) + 27)
119         k_f1_list = s_createlist k_f1 0
120                       where
121                       k_f1  0 = 0
122                       k_f1  t = k + q + (j_f1_list ! (t−1))
123
124
125
126 || ═══════════
127 ||  Step 2
128
129 main2 x
130       = (i_wrapper ! x) ! 0
131         where
```

71

```
132          q = 4
133          k = 16
134          i_wrapper = ( transpose [ i_f1_list ] )
135                          where
136                          i_f1_list = s_createlist i_f1 0
137                                      where
138                                      i_f1  0 = q
139                                      i_f1  1 = ( i_f1_list ! 0 ) + k
140                                      i_f1  t = ( i_f1_list !(t-1)) + (( j_wrapper !(t-2))!0)
141          j_wrapper = ( transpose [ j_f1_list , j_f2_list ] )
142                          where
143                          j_f1_list = s_createlist j_f1 0
144                                      where
145                                      j_f1  0 = k
146                                      j_f1  t = cond  ( t <0)  (( k_wrapper !(t-1))!0)
147                                                      (25 + (( j_f1_list !(t-1)) * ((
    i_wrapper !(t-1))!0))
148                                                      + ( j_f2_list !(t-1)))
149                          j_f2_list = s_createlist j_f2 0
150                                      where
151                                      j_f2  0 = k+q
152                                      j_f2  t = cond  ( t <3)  27  (( j_f1_list ! (t-1)) + 27)
153          k_wrapper = ( transpose [ k_f1_list ] )
154                          where
155                          k_f1_list = s_createlist k_f1 0
156                                      where
157                                      k_f1  0 = 0
158                                      k_f1  t = k + q + (( j_wrapper !(t-1)) ! 0)
159
160
161
162 || ==========
163 || Step 3
164
165 main3 x
166      = (( outputs !(x+1))!0)!0
167          where
168          q = 4
169          k = 16
170          nullvalue = 0
171          niloutputs = [[ nullvalue ], [ nullvalue , nullvalue ], [ nullvalue ]]
172          outputs     = niloutputs : ( transpose [ i_wrapper , j_wrapper , k_wrapper ])
173          i_wrapper   = transpose [ i_f1_list ]
174                       where
175                       i_f1_list = s_createlist i_f1 0
176                       i_f1  0 = q
177                       i_f1  1 = ( i_f1_list ! 0 ) + k
178                       i_f1  t = cond  ( t <0)  0  (( i_f1_list !(t-1)) + j_f1_2last_timestep
    )
179                                  where
180                                  j_f1_2last_timestep = ((( outputs !(t-1))!1)!0)
181          j_wrapper   = transpose [ j_f1_list , j_f2_list ]
182                       where
183                       j_f1_list = s_createlist j_f1 0
184                                   where
185                                   j_f1  0 = k
186                                   j_f1  t = cond  ( t <0)  k_f1_last_timestep
187                                                   (25 + (( j_f1_list ! (t-1)) *
    i_f1_last_timestep )
188                                                   + ( j_f2_list ! (t-1)))
189                                              where
190                                              i_f1_last_timestep = ((( outputs !t)!0)!0)
191                                              k_f1_last_timestep = ((( outputs !t)!2)!0)
192                       j_f2_list = s_createlist j_f2 0
```

72

```
193                                           where
194                                           j_f2  0 = k+q
195                                           j_f2  t = cond  (t<3)  27  (( j_f1_list  !  (t−1)) + 27)
196           k_wrapper   = transpose  [ k_f1_list ]
197                         where
198                         k_f1_list = s_createlist  k_f1  0
199                                       where
200                                       k_f1  0 = 0
201                                       k_f1  t = k + q + j_f1_last_timestep
202                                               where
203                                               j_f1_last_timestep = (((outputs!t)!1)!0)
204
205

|| ══════════
|| Step  4

main4  x
       = ((outputs!(x+1))!0)!0
         where
         q = 4
         k = 16
         nullvalue = 0
         niloutputs = [[nullvalue], [nullvalue, nullvalue], [nullvalue], [nullvalue
     ]]
         outputs = niloutputs : transpose [ i_wrapper outputs 0 ,j_wrapper outputs 0
     ,k_wrapper outputs 0,jf1_delay outputs 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)!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 = [( 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
```

73

```
245                                         inbound_messages_this_timestep  = hd inputs
246                                         future_messages                 = tl inputs
247                                         outbound_messages_this_timestep = [k_f1 t]
248                                         k_f1  0 = 0
249                                         k_f1  t = k + q + j_f1_last_timestep
250                                                    where
251                                                    j_f1_last_timestep = ((
     inbound_messages_this_timestep!1)!0)
252         jf1_delay inputs t = outbound_messages_this_timestep : (jf1_delay
     future_messages (t+1))
253                                         where
254                                         inbound_messages_this_timestep  = hd inputs
255                                         future_messages                 = tl inputs
256                                         outbound_messages_this_timestep = [(jf1_delayed t)]
257                                         jf1_delayed 0 = k
258                                         jf1_delayed t = ((inbound_messages_this_timestep!1)!0)
259
260
261
262
263  || ═══════════
264  || Step 5
265
266
267  main5 x
268       = (thd3.hd) (((outputs!(x+1))!0)!0)
269         where
270         q = 4
271         k = 16
272         newnullvalue = [(0,0,0)]
273         niloutputs = [[newnullvalue], [newnullvalue, newnullvalue], [newnullvalue],
     [newnullvalue]]
274         outputs = niloutputs : transpose [ i_wrapper outputs 0 ,j_wrapper outputs 0
     ,k_wrapper outputs 0, jf1_delay outputs 0]
275         i_wrapper inputs t = outbound_messages_this_timestep : (i_wrapper
     future_messages (t+1))
276                                 where
277                                 inbound_messages_this_timestep  = map f (hd inputs)
278                                                                     where
279                                                                     f xs = map g xs
280                                                                     g xs = filter h xs
281                                                                     h (a,b,c) = ((b=1)
     \/ (a,b,c) = (0,0,0))
282                                 future_messages                 = tl inputs
283                                 outbound_messages_this_timestep = [ [ (1,1,res)   ||
     from i to i
284                                                                     ,(1,2,res)] ||
     from i to j
285                                                                    ]
286                                                                 where
287                                                                 res = i_f1 t
288                                 i_f1  0 = q
289                                 i_f1  t = cond (t<0) 0 (i_f1_last_timestep +
     j_f1_2last_timestep)
290                                            where
291                                            i_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!0)!0)
292                                            j_f1_2last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!3)!0)
293         j_wrapper inputs t = outbound_messages_this_timestep : (j_wrapper
     future_messages (t+1))
294                                 where
295                                 inbound_messages_this_timestep  = map f (hd inputs)
296                                                                     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)   ||
      from  j  to  i
                                                                   ,(2,2,res1)    ||
      from  j  to  j
                                                                   ,(2,3,res1)]  ||
      from  j  to  k
                                                                   ,[(2,2,res2)]    ||
      from  j  to  j
                                                                                    ]
                                                                     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
                                 outbound_messages_this_timestep = [[(3,2,k_f1  t)]]   ||
       From  k  to  j  only
                                 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))
                                 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=4)
      \/  (a,b,c) = (0,0,0))
                                 future_messages              = tl  inputs
                                 outbound_messages_this_timestep = [[(4,1,jf1_delayed  t
      )]]
```

```
343                                     jf1_delayed  0 = k
344                                     jf1_delayed  t = (thd3.hd) ((
       inbound_messages_this_timestep!1)!0)
345
346
347
348
349
350
351  || ==========
352  ||  Step  6
353
354  main6 x
355       = sim  (x+1)
356          where
357          q = 4
358          k = 16
359          sim  x = (thd3.hd) (((outputs!x)!0)!0)
360                   where
361                   newnullvalue = [(0,0,0)]
362                   niloutputs = [[newnullvalue], [newnullvalue, newnullvalue], [
       newnullvalue], [newnullvalue]]
363                   outputs = niloutputs : redactedoutputs
364                   redactedoutputs
365                    = transpose [ i_wrapper i_outputs 0 ,j_wrapper j_outputs 0 ,
       k_wrapper k_outputs 0, jf1_delay jf1_inputs 0 ]
366                     where
367                     i_outputs   = map (map (map (filter (h 1)))) outputs
368                     j_outputs   = map (map (map (filter (h 2)))) outputs
369                     k_outputs   = map (map (map (filter (h 3)))) outputs
370                     jf1_inputs  = map (map (map (filter (h 4)))) outputs
371               h x (a,b,c) = ((b=x) \/ (a,b,c) = (0,0,0))
372           i_wrapper inputs t = outbound_messages_this_timestep : (i_wrapper
       future_messages (t+1))
373                               where
374                               inbound_messages_this_timestep  = hd inputs
375                               future_messages               = tl inputs
376                               outbound_messages_this_timestep = [ [ (1,1,res)  ||
       from i to i
377                                                                    ,(1,2,res)] ||
       from i to j
378                                                                  ]
379                                                                  where
380                                                                  res = i_f1 t
381                               i_f1  0 = q
382                               i_f1  t = cond (t<0) 0 (i_f1_last_timestep +
       j_f1_2last_timestep)
383                                         where
384                                         i_f1_last_timestep = (thd3.hd) ((
       inbound_messages_this_timestep!0)!0)
385                                         j_f1_2last_timestep = (thd3.hd) ((
       inbound_messages_this_timestep!3)!0)
386           j_wrapper inputs t = outbound_messages_this_timestep : (j_wrapper
       future_messages (t+1))
387                               where
388                               inbound_messages_this_timestep  = hd inputs
389                               future_messages               = tl inputs
390                               outbound_messages_this_timestep = [ [ (2,4,res1)  ||
       from j to i
391                                                                    ,(2,2,res1)  ||
       from j to j
392                                                                    ,(2,3,res1)] ||
       from j to k
393                                                                  ,[(2,2,res2)]  ||
```

76

```
                                        from j to j
                                                                           ]
                                                                         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)]]   ||
 From k to j only
                                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))
                                where
                                inbound_messages_this_timestep  = hd inputs
                                future_messages                 = tl inputs
                                outbound_messages_this_timestep = [[(4,1,jf1_delayed t
)]]
                                jf1_delayed 0 = k
                                jf1_delayed t = (thd3.hd) ((
inbound_messages_this_timestep!1)!0)



|| ============
||  Step 7

    main7 x
      = sim (x+1) runtime_args agentinfo
        where
        runtime_args = [(Arg ("q",4)), (Arg ("k", 16))]
        agentinfo = [(i_wrapper,[]), (j_wrapper, []), (k_wrapper, []), (jf1_delay,
[])]
        sim x args agents
            = (thd3.hd) (((snd (outputs!x))!0)!0)
                where
                newnullvalue = [(0,0,0)]
                niloutputs = [[newnullvalue], [newnullvalue, newnullvalue], [
newnullvalue], [newnullvalue]]
                outputs = zip2 [0..] (niloutputs : (redactedoutputs))
```

```
443              redactedoutputs
444                = transpose (map apply_to_args (zip2 [1..] agents))
445                  where
446                  apply_to_args (agentid, (f,broadcastsubscriptions))
447                              = f Emptyagentstate args (myoutputs agentid)
     agentid
448                  myoutputs id = map (f id) outputs
449              f x (t, xs)  = (t, map (map (filter (h x))) xs)
450              h x (a,b,c)  = ((b=x) \/ (a,b,c) = (0,0,0))
451        i_wrapper localstate args inputs id
452             = outbound_messages_this_timestep : (i_wrapper localstate args
     future_messages id)
453                  where
454                  q = (getargval.hd) (filter ((="q").getargstr) args)
455                  k = (getargval.hd) (filter ((="k").getargstr) args)
456                  inbound_messages_this_timestep  = snd (hd inputs)
457                  t                               = fst (hd inputs)
458                  future_messages                 = tl inputs
459                  outbound_messages_this_timestep = [ [ (id,1,res)  || from i to i
460                                                       ,(id,2,res)] || from i to j
461                                                     ]
462                                                     where
463                                                     res = i_f1 t
464                  i_f1 0 = q
465                  i_f1 t = cond (t<0) 0 (i_f1_last_timestep + j_f1_2last_timestep)
466                          where
467                          i_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!0)!0)
468                          j_f1_2last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!3)!0)
469        j_wrapper localstate args inputs id
470             = outbound_messages_this_timestep : (j_wrapper localstate args
     future_messages id)
471                  where
472                  q = (getargval.hd) (filter ((="q").getargstr) args)
473                  k = (getargval.hd) (filter ((="k").getargstr) args)
474                  inbound_messages_this_timestep  = snd (hd inputs)
475                  t                               = fst (hd inputs)
476                  future_messages                 = tl inputs
477                  outbound_messages_this_timestep = [ [ (id,4,res1)  || from j to i
478                                                       ,(id,2,res1)  || from j to j
479                                                       ,(id,3,res1)] || from j to k
480                                                      ,[ (id,2,res2)] || from j to j
481                                                     ]
482                                                     where
483                                                     res1 = j_f1 t
484                                                     res2 = j_f2 t
485                  j_f1 0 = k
486                  j_f1 t = cond (t<0) k_f1_last_timestep
487                              (25 + (j_f1_last_timestep * i_f1_last_timestep)+
     j_f2_last_timestep)
488                          where
489                          i_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!0)!0)
490                          j_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!1)!0)
491                          j_f2_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!1)!1)
492                          k_f1_last_timestep = (thd3.hd) ((
     inbound_messages_this_timestep!2)!0)
493                  j_f2 0 = k+q
494                  j_f2 t = cond (t<3) 27 (j_f1_last_timestep + 27)
495                          where
496                          j_f1_last_timestep = (thd3.hd) ((
```

```
        inbound_messages_this_timestep !1) !0)
497     k_wrapper localstate args inputs id
498         = outbound_messages_this_timestep : (k_wrapper localstate args
    future_messages id)
499             where
500             q = (getargval.hd) (filter ((="q").getargstr) args)
501             k = (getargval.hd) (filter ((="k").getargstr) args)
502             inbound_messages_this_timestep  = snd (hd inputs)
503             t                               = fst (hd inputs)
504             future_messages                 = tl inputs
505             outbound_messages_this_timestep = [[(id,2,k_f1 t)]]   || From k to j
     only
506             k_f1  0 = 0
507             k_f1  t = k + q + j_f1_last_timestep
508                     where
509                     j_f1_last_timestep = (thd3.hd) ((
    inbound_messages_this_timestep!1)!0)
510     jf1_delay localstate args inputs id
511         = outbound_messages_this_timestep : (jf1_delay localstate args
    future_messages id)
512             where
513             inbound_messages_this_timestep  = snd (hd inputs)
514             future_messages                 = tl inputs
515             t                               = fst (hd inputs)
516             outbound_messages_this_timestep = [[(id,1,jf1_delayed t)]]
517             k = (getargval.hd) (filter ((="k").getargstr) args)
518             jf1_delayed 0 = k
519             jf1_delayed t = (thd3.hd) ((inbound_messages_this_timestep!1)!0)
520
521
522
523
524
525 || ==========
526 || Step 8
527
528 main8 x
529     = sim (x+1) runtime_args agentinfo
530         where
531         runtime_args = [(Arg ("q",4)), (Arg ("k", 16))]
532         agentinfo = [(i_wrapper,[]), (j_wrapper, []), (k_wrapper, []), (jf1_delay,
    [])]
533     sim x runtime_args agentinfo
534         = (getargval.hd.getmsgargs) (((snd (outputs!x))!0)!0)
535             where
536             outputs = (0,[]):timed_transposed_msgs
537
538             || timed_transposed_msgs is a list of outputs at each timestep,
539             || each being a tuple of a time and a list of outputs from all
    agents
540             || [(1,t1_outputs), (2,t2_outputs),...]
541
542             timed_transposed_msgs = zip2 [1..] transposed_msgs
543
544             || transposed_msgs is a list of outputs at each timestep
545             || [t1_outputs, t2, outputs, ...]
546             || each being a list of outputs from all agents
547             || [t1_outputs_agent1, t1_outputs_agent2, ...]
548             || each item of which is a list of msg_t
549
550             transposed_msgs = transpose allmessages
551
552             || allmessages is a list of outputs from each agent
553             || [agent1_output, agent2_output, ...]
```

```
                      || each of which is a list of outputs
                      || per timestep, each being a list of msg_t

                    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 :: num -> [(num, [[msg_t]])] -> [(num,[msg_t])]
                        redacted_msgs id outputs    = map (f id) outputs
                    f x (time, xs)                  = (time, concat (map (filter (h x)) xs)
    )
                    h x Hiaton                      = False
                    h x (Message (f,t) args)    = (t=x)
                        h x (Datamessage (f,t) text) = error "Datamessage not yet
    implemented"
        || i_wrapper :: agent_t
        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))] || from i to i
                                                   ,Message (id,2) [(Arg ("i_f1",
    res))] || from i to j
                                                   ]
                                                     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_j        = filter ((=4).getmsgfrom)
    inbound_messages_this_timestep
                            args_from_j        = concat (map getmsgargs msgs_from_j)
                            j_f1_last_timestep = (getargval.hd) (filter ((="j_f1d").
    getargstr) args_from_j)
        ||j_wrapper :: agent_t
        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))] || from j to i
                                                   ,Message (id,2) [(Arg ("j_f1",
    res1))]  || from j to j
                                                   ,Message (id,3) [(Arg ("j_f1",
    res1))]  || from j to k
```

```
604                                                                  ,Message (id,2) [(Arg ("j_f2",
       res2))]   || from j to j
605                                                              ]
606                                                              where
607                                                              res1 = j_f1 t
608                                                              res2 = j_f2 t
609                j_f1  0 = k
610                j_f1  t = cond (t<0) k_f1_last_timestep
611                          (25 + (j_f1_last_timestep * i_f1_last_timestep)+
       j_f2_last_timestep)
612                          where
613                          msgs_from_i        = filter ((=1).getmsgfrom)
       inbound_messages_this_timestep
614                          args_from_i        = concat (map getmsgargs msgs_from_i)
615                          i_f1_last_timestep = (getargval.hd) (filter ((="i_f1").
       getargstr) args_from_i)
616                          msgs_from_j        = filter ((=2).getmsgfrom)
       inbound_messages_this_timestep
617                          args_from_j        = concat (map getmsgargs msgs_from_j)
618                          j_f1_last_timestep = (getargval.hd) (filter ((="j_f1").
       getargstr) args_from_j)
619                          j_f2_last_timestep = (getargval.hd) (filter ((="j_f2").
       getargstr) args_from_j)
620                          msgs_from_k        = filter ((=3).getmsgfrom)
       inbound_messages_this_timestep
621                          args_from_k        = concat (map getmsgargs msgs_from_k)
622                          k_f1_last_timestep = (getargval.hd) (filter ((="k_f1").
       getargstr) args_from_j)
623                j_f2  0 = k+q
624                j_f2  t = cond (t<3) 27 (j_f1_last_timestep + 27)
625                          where
626                          msgs_from_j        = filter ((=2).getmsgfrom)
       inbound_messages_this_timestep
627                          args_from_j        = concat (map getmsgargs msgs_from_j)
628                          j_f1_last_timestep = (getargval.hd) (filter ((="j_f1").
       getargstr) args_from_j)
629       ||k_wrapper :: agent_t
630       k_wrapper localstate args inputs id
631            = outbound_messages_this_timestep : (k_wrapper localstate args
       future_messages id)
632             where
633             q = (getargval.hd) (filter ((="q").getargstr) args)
634             k = (getargval.hd) (filter ((="k").getargstr) args)
635             inbound_messages_this_timestep  = snd (hd inputs)
636             t                               = fst (hd inputs)
637             future_messages                 = tl inputs
638             outbound_messages_this_timestep = [Message (id,2) [(Arg ("k_f1",
       k_f1 t))]]   || From k to j only
639             k_f1  0 = 0
640             k_f1  t = k + q + j_f1_last_timestep
641                       where
642                       msgs_from_j        = filter ((=2).getmsgfrom)
       inbound_messages_this_timestep
643                       args_from_j        = concat (map getmsgargs msgs_from_j)
644                       j_f1_last_timestep = (getargval.hd) (filter ((="j_f1").
       getargstr) args_from_j)
645       jf1_delay localstate args inputs id
646            = outbound_messages_this_timestep : (jf1_delay localstate args
       future_messages id)
647              where
648              k = (getargval.hd) (filter ((="k").getargstr) args)
649              inbound_messages_this_timestep  = snd (hd inputs)
650              t                               = fst (hd inputs)
651              future_messages                 = tl inputs
```

81

```
652                    outbound_messages_this_timestep = [Message (id, 1) [(Arg ("j_f1d",
       jf1_delayed t))]]
653                    msgs_from_j           = filter ((=2).getmsgfrom)
       inbound_messages_this_timestep
654                    args_from_j           = concat (map getmsgargs msgs_from_j)
655                    jf1_delayed 0      = k
656                    jf1_delayed t      = (getargval.hd) (filter ((="j_f1").getargstr)
       args_from_j)
```

Appendix B: Implementation of Translation

User Manual

Here the code for the implementation of the translation described in this dissertation is given; this implementation only covers the first three aspects of the automated translation. These three steps are the lexer, parser and first translation step, shown here in one file.

This code is written in Miranda, therefore to run it Miranda must be installed; installation instructions and downloads for Miranda can be found through the following link: http://miranda.org.uk/.

Once Miranda is installed, simply run the the code as a file using *mira file.m*. Once this is done the first transaltion step can be achieved by using the command *trans_step*1 (*parser* (*lex input*)), where *input* is the character list defining the experiment to be translated (this can be created by reading in a text file).

Lexer, Parser and First Translation Step Code

```
1  ||Lexer
2  ||Reads in the file and converts it to tokens
3
4  lexeme::=Init|Var [char]|Recrel [char] [char]|Cond|Main|Fwhere|Optl|Ophd|Intfunc [
       char]|Num num|Opcon|Opbng|Opdiv|Optim|Opmin|Opplus|Rbckt|Lbckt|Rlbkt|Llbkt|
       Opequ|Opne|Oplt|Opgt|Oplet|Opget|Concomma|nl
5
6
7  lex  []               = []
8  lex  ('>':'=':xs)     = Opget:(lex xs)
9  lex  ('<':'=':xs)     = Oplet:(lex xs)
10 lex  ('>':xs)         = Opgt:(lex xs)
11 lex  ('<':xs)         = Oplt:(lex xs)
12 lex  ('~':'=':xs)     = Opne:(lex xs)
13 lex  ('=':xs)         = Opequ:(lex xs)
14 lex  ('[':xs)         = Llbkt:(lex xs)
15 lex  (']':xs)         = Rlbkt:(lex xs)
16 lex  ('(':xs)         = Lbckt:(lex xs)
17 lex  (')':xs)         = Rbckt:(lex xs)
18 lex  ('+':xs)         = Opplus:(lex xs)
19 lex  ('-':xs)         = Opmin:(lex xs)
20 lex  ('*':xs)         = Optim:(lex xs)
21 lex  ('/':xs)         = Opdiv:(lex xs)
22 lex  ('!':xs)         = Opbng:(lex xs)
23 lex  (' ':xs)         = lex xs
24 lex  ('\n':xs)        = nl:lex xs
25 lex  (',':xs)         = Concomma:(lex xs)
26 lex  (':':xs)         = Opcon:(lex xs)
27 lex  (x:xs)           = (Num (numval a)):(lex b), if (isnumber a)
28                       = (Intfunc (tl a)):(lex b), if (hd a) = '_'
29                       = Ophd:(lex b), if a=['h','d']
30                       = Optl:(lex b), if a=['t','l']
31                       = Fwhere:(lex b), if a=['w','h','e','r','e']
32                       = Main:(lex b), if a=['m', 'a', 'i', 'n']
33                       = Cond:(lex b), if a=['c','o','n','d']
34                       = (returnfunc a []):(lex b), if (isfunc a)
35                       = (Var a):(lex b), otherwise
36                         where
37                         (a,b)=f (x:xs) []
38                         f []         a = (a,[])
39                         f (' ':xs)   a = (a,xs)
40                         f (')':xs)   a = (a,(')':xs))
41                         f (',':xs)   a = (a,(',':xs))
42                         f ('+':xs)   a = (a,('+':xs))
```

83

```
43                                  f  ('-':xs)   a = (a,('-':xs))
44                                  f  ('*':xs)   a = (a,('*':xs))
45                                  f  ('/':xs)   a = (a,('/':xs))
46                                  f  ('<':xs)   a = (a,('<':xs))
47                                  f  ('>':xs)   a = (a,('>':xs))
48                                  f  ('=':xs)   a = (a,('=':xs))
49                                  f  (']':xs)   a = (a,(']':xs))
50                                  f  (':':xs)   a = (a,(':':xs))
51                                  f  ('{':xs)   a = (a,('{':xs))
52                                  f  ('}':xs)   a = (a,('}':xs))
53                                  f  ('\n':xs) a = (a,xs)
54                                  f  (x:xs)     a = f xs (a++[x])

56  isnumber x = (removeall "0123456789." (mkset x)) = []

58  removeall xs []      = []
59  removeall xs (y:ys) = removeall xs ys, if member xs y
60                      = y:(removeall xs ys), otherwise

62  isfunc  [] = False
63  isfunc  ('_':xs) = True
64  isfunc  (x:xs) = isfunc xs

66  beforescore []        a = a
67  beforescore ('_':xs) a = a
68  beforescore (x:xs)    a = beforescore xs (a++[x])

70  returnfunc []        a = Recrel a []
71  returnfunc ('_':xs) a = Recrel a xs
72  returnfunc (x:xs)    a = returnfunc xs (a++[x])

73
74
75
76
77
78
79
80
81
82
83
84
85
86  ||Parse tree
87  ||Stores the structure of the inputted file
88
89
90  simulation ::= Simulation [definition] experiment
91
92  definition ::= Function [char] [char] [argument] expression
93
94  argument::= Argument expression
95
96  experiment::= Experiment [initcon] experimentrun
97
98  initcon::= Initcon [char] expression
99
100 experimentrun::= Emptyrun|Experimentrun expression
101
102 expression::= Emptyexpression
103             |Ifthenelse expression expression expression
104             |Brackets expression
105             |List [expression]
106             |Operation expression op expression
```

```
107                    |IntFunct [char] [argument]
108                    |ExtFunct [char] [char] [argument]
109                    |IntVar [char]
110                    |Specialfunc specfunc expression
111                    |Number num
112                    |Where expression [subdefinition]
113
114  op::=  Plus
115        |Minus
116        |Multiply
117        |Divide
118        |Lessthan
119        |Greaterthan
120        |Equals
121        |Notequals
122        |Lessequ
123        |Greaterequ
124        |Listadd
125        |Bang
126
127  specfunc::=  Listhead|Listtail
128
129  subdefinition ::= IntVariable [char] expression |  IntFunction [char] [argument]
         expression
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162  ||Parser
163  ||This turns the tokens into the parse tree
164
165  parser x = Simulation (get_defs a []) (get_exprement b c)
166           Where
167           a = find_recrel x
168           b = find_intial x
169           c = find_exp x
```

85

```
170
171  find_recrel  (Fwhere:y) = y
172  find_recrel  (z:y)       = find_recrel y
173
174  find_intial  (Init:y) = find_intial1  y  []
175  find_intial  (z:y)    = find_intial y
176
177  find_intial1  (Fwhere:y)  list = list
178  find_intial1  (z:y)       list = find_intial1 y  (list++[z])
179
180  find_exp  (Main:y) = find_exp1  y  []
181  find_exp  (z:y)    = find_exp y
182
183  find_exp1  (Init:y)  list = list
184  find_exp1  (z:y)     list = find_exp1 y  (list++[z])
185
186
187
188
189  get_defs  []           deflist = deflist
190  get_defs  unedefs  deflist = get_defs newundefs newdeflist
191                             where
192                             newundefs = b
193                             newdeflist = deflist ++ [a]
194                             a = gd_fd  unedefs
195
196
197  gd_fd  ((Recrel a b):xs) = Function a b args expr
198                             where
199                             (args,  rest1) = get_args  xs  []
200                             exstuff = after_equals rest1
201                             (expr,  none) = get_expr exstuff Emptyexpression
202
203
204
205  get_args  (nl:xs)        args = (args,  xs)
206  get_args  (Rbckt:xs)     args = (args,  xs)
207  get_args  (Opequ:xs)     args = (args,  xs)
208  get_args  (Concomma:xs) args = (args,  xs)
209  get_args  x              args = get_args new_x new_args
210                                  where
211                                  new_args = args++[arg]
212                                  (arg,  new_x) = get_expr x Emptyexpression
213
214
215
216
217
218
219
220  get_expr x                      Emptyexpression = (new_a,  new_xs),  if istherewhere x
221                                                   where
222                                                   (bwhere,  awhere) = wheresplitter x []
223                                                   (inter_where,  new_xs) = get_wherestate
       awhere  []
224                                                   (mid_a,  none) = get_expression bwhere
        Emptyexpression
225                                                   new_a = Where mid_a inter_where
226  get_expr  (Cond:xs)             Emptyexpression = get_expr new_xs new_a
227                                                   where
228                                                   new_a = (Ifthenelse expr1 expr2 expr3)
229                                                   (expr1,  xs1) = get_expr xs
        Emptyexpression
230                                                   (expr2,  xs2) = get_expr xs1
```

```
              Emptyexpression
231                                                    (expr3, new_xs) = get_expr xs2
              Emptyexpression
232 get_expr (Lbckt:xs)          Emptyexpression = get_expr new_xs new_a
233                                                where
234                                                new_a = Braclets expr
235                                                (expr, rest) = get_expr in_bracket
              Emptyexpression
236                                                (in_bracket, new_xs) = get_bracket xs
         []
237 get_expr (Llbkt:xs)          Emptyexpression = get_expr new_xs new_a
238                                                where
239                                                (listeditems, new_xs) = f xs []
240                                                f (Llbkt:xs) a = (a, xs)
241                                                f (Concomma:xs) a = f xs a
242                                                f x a = f nn_xs nnn_a
243                                                        where
244                                                        (mid_a, nn_xs) = get_expr x
              Emptyexpression
245                                                        nnn_a = a++[mid_a]
246                                                new_a = List listeditems
247 get_expr ((Intfunc a):xs)  Emptyexpression = get_expr new_xs new_a
248                                                where
249                                                new_a = IntFunct a args
250                                                (args, new_xs) = get_args xs []
251 get_expr ((Recrel a b):xs) Emptyexpression = get_expr new_xs new_a
252                                                where
253                                                new_a = ExtFunct a b args
254                                                (args, new_xs) = get_args xs []
255 get_expr ((Var a):xs)       Emptyexpression = get_expr new_xs new_a
256                                                where
257                                                new_a = IntVar a
258                                                new_xs = xs
259 get_expr (Ophd:xs)          Emptyexpression = get_expr new_xs new_a
260                                                where
261                                                new_a = (Specialfunc Listhead exp)
262                                                (exp, new_xs) = get_expr xs
              Emptyexpression
263 get_expr (Optl:xs)          Emptyexpression = get_expr new_xs new_a
264                                                where
265                                                new_a = (Specialfunc Listtail exp)
266                                                (exp, new_xs) = get_expr xs
              Emptyexpression
267 get_expr ((Num a):xs)       Emptyexpression = get_expr new_xs new_a
268                                                where
269                                                new_a = Number a
270                                                new_xs = xs
271 get_expr (x:xs)             a               = (a, (x:xs)), if isexprend x
272                                            = get_expr new_xs new_a, otherwise
273                                                where
274                                                new_a = (Operation a (get_op x) mid_a)
275                                                (mid_a, new_xs) = get_expr xs
              Emptyexpression
276
277
278 isexprend []            = True
279 isexprend (Opplus:xs)  = False
280 isexprend (Opmin:xs)   = False
281 isexprend (Optim:xs)   = False
282 isexprend (Opdiv:xs)   = False
283 isexprend (Oplt:xs)    = False
284 isexprend (Opgt:xs)    = False
285 isexprend (Opequ:xs)   = False
286 isexprend (Opne:xs)    = False
```

```
287  isexprend (Oplet:xs)   = False
288  isexprend (Opget:xs)   = False
289  isexprend (Opcon:xs)   = False
290  isexprend (Opbng:xs)   = False
291  isexprend x            = True
292
293  get_op Opplus = Plus
294  get_op Opmin  = Minus
295  get_op Optim  = Multiply
296  get_op Opdiv  = Divide
297  get_op Oplt   = Lessthan
298  get_op Opgt   = Greaterthan
299  get_op Opequ  = Equals
300  get_op Opne   = Notequals
301  get_op Oplet  = Lessequ
302  get_op Opget  = Greaterequ
303  get_op Opcon  = Listadd
304  get_op Opbng  = Bang
305
306
307  wheresplitter (Fwhere:xs) a = (a, xs)
308  wheresplitter (x:xs)      a = wheresplitter xs (a++[x])
309
310
311  get_wherestate []                 inwhere = (inwhere, [])
312  get_wherestate ((Recrel a b):xs)  inwhere = (inwhere, ((Recrel a b):xs))
313  get_wherestate ((Var a):xs)       inwhere = get_wherestate new_xs new_inwhere
314                                              where
315                                              new_inwhere = inwhere ++ [sub_func]
316                                              sub_func = IntVariable a expr
317                                              (expr, new_xs) = get_expr lexs
         Emptyexpression
318                                              lexs = get_aftereqs xs
319  get_wherestate ((Intfunc a):xs)   inwhere = get_wherestate new_xs new_inwhere
320                                              where
321                                              new_inwhere = inwhere ++ [sub_func]
322                                              sub_func = IntFunction a args expr
323                                              (args, rest) = get_args xs []
324                                              (expr, new_xs) = get_expr lexs
         Emptyexpression
325                                              lexs = get_aftereqs rest
326
327
328
329  get_aftereqs (Opequ:xs) = xs
330  get_aftereqs (x:xs)     = get_aftereqs xs
331
332
333
334
335  get_bracket (Lbckt:xs) a = get_bracket new_xs new_a
336                             where
337                             (new_a, new_xs) = get_innerbracket xs (a++[Lbckt])
338  get_bracket (Rbckt:xs) a = (a, xs)
339  get_bracket (x:xs)     a = get_bracket xs (a++[x])
340
341
342  get_innerbracket (Lbckt:xs) a = get_innerbracket new_xs new_a
343                                  where
344                                  (new_a, new_xs) = get_innerbracket xs (a++[Lbckt])
345  get_innerbracket (Rbckt:xs) a = ((a++[Rbckt]), xs)
346  get_innerbracket (x:xs)     a = get_innerbracket xs (a++[x])
347
348
```

```
349
350  get_exprement b c = Experiment intialcons exprun
351                      where
352                      exprun = Experimentrun exprsion
353                      (exprsion, none) = get_expr c Emptyexpression
354                      intialcons = get_inconlist b []
355
356  get_inconlist []                      itcons = itcons
357  get_inconlist ((Var a):Opequ:xs) itcons = get_inconlist new_x new_itcons
358                                            where
359                                            new_itcons = itcons ++[newit]
360                                            newit = Initcon a expr
361                                            (expr, new_x) = get_expr xs
         Emptyexpression
362
363
364
365
366
367
368
369  || first step translation
370
371
372  trans_step1 (Simulation defs expriment) = Simulation new_defs new_expirment
373                                           where
374                                           new_defs       = wrapdefs
375                                           new_expirment = callexpermnet
376                                           wrapdefs       = add_wdefs calldefs
377                                           calldefs       = add_cdefs defs []
378                                           callexpermnet = add_cexpem expriment
379
380
381
382
383
384  add_cexpem Experiment inits exprun = Experiment new_inits new_exprun
385                                      where
386                                      new_inits = add_initcalss inits []
387                                      new_exprun = add_expcalls exprun
388
389  add_initcalss []                      ninits = ninits
390  add_initcalss ((Initcon a expr):xs) ninits = add_initcalss xs new_ninits
391                                               where
392                                               new_ninits = Initcon a new_expr
393                                               new_expr = addcalls_exprions expr
394
395  add_expcalls (Experimentrun a) = Experimentrun new_a
396                                   where
397                                   new_a = addcalls_exprions a
398
399
400  add_cdefs [] ndefs = ndefs
401  add_cdefs (x:xs) ndefs = add_cdefs xs (ndefs++[new_x])
402                           where
403                           new_x = adddefcal x
404
405  adddefcal (Function a b c d) = Function a b new_c new_d
406                                 where
407                                 new_c = cargs c []
408                                 new_d = addcalls_exprions d
409
410
411
```

```
412
413
414  addcalls_exprions (Ifthenelse a b c) = Ifthenelse new_a new_b new_c
415                                            where
416                                            new_a = addcalls_exprions a
417                                            new_b = addcalls_exprions b
418                                            new_c = addcalls_exprions c
419  addcalls_exprions (Brackets a)       = Brackets new_a
420                                            where
421                                            new_a = addcalls_exprions a
422  addcalls_exprions (List a)           = List new_a
423                                            where
424                                            new_a = cargs a []
425  addcalls_exprions (Operation a b c) = Operation new_a b new_c
426                                            where
427                                            new_a = addcalls_exprions a
428                                            new_c = addcalls_exprions c
429  addcalls_exprions (IntFunct a b)     = IntFunct a new_b
430                                            where
431                                            new_b = cargs b []
432  addcalls_exprions (ExtFunct a b c)   = Operation wrap Bang fstarg
433                                            where
434                                            wrap = ExtFunct a newb newc
435                                            newb = b ++ "_list"
436                                            newc = tl (cargs c [])
437                                            fstarg = hd (cargs c [])
438  addcalls_exprions (IntVar a)         = IntVar a
439  addcalls_exprions (Specialfunc a b) = Specialfunc a new_b
440                                            where
441                                            new_b = addcalls_exprions b
442  addcalls_exprions (Number a)         = Number a
443  addcalls_exprions (Where a b)        = Where new_a new_b
444                                            where
445                                            new_a = addcalls_exprions a
446                                            new_b = callssubdefs b []
447
448  cargs []      list = list
449  cargs (x:xs)  list = cargs xs (list++[new_x])
450                         where
451                         new_x = addcalls_exprions x
452
453
454  callssubdefs []  newdefs = newdefs
455  callssubdefs ((IntVariable a b):xs) = callssubdefs xs nnewdefs
456                                            where
457                                            nnewdefs = newdefs ++ [ndef]
458                                            ndef = IntVariable a new_b
459                                            new_b = addcalls_exprions b
460  callssubdefs (( IntFunction a args b):xs) = callssubdefs xs nnewdefs
461                                                 where
462                                                 nnewdefs = newdefs ++ [ndef]
463                                                 ndef = IntFunction a new_args new_b
464                                                 new_args = cargs args []
465                                                 new_b = addcalls_exprions b
466
467
468
469
470
471
472
473  add_wdefs []  ndefs = ndefs
474  add_wdefs (x:xs) ndefs = add_wdefs nxs nnedfs
475                              where
```

```
476                              nndefs = addwrapper fx fun
477                              (id1, id2) = getdefid x
478                              fun = creatffunc (x:xs) [] (id1,id2)
479                              fx = findfx fun
480
481 getdefid (Function a b c d) = (a, b)
482
483 creatffunc ((Function a b c d):xs) defl id = creatffunc xs ndefl
484                                             where
485                                             ndefl = defl++[(Function a b c d)], (a
       ,b) = id
486                                             ndefl = defl, otherwise
487
488 findfx ((Function a b c d):xs) = Function a b c d, if (hd c) = (IntVar "t")
489
490
491
492 addwrapper (Function a b c d) fun = Function a newb newc newd
493                                        where
494                                        newb = b ++ "_list"
495                                        newc = tl c
496                                        newd = Where exp subsdefs
497                                        exp = IntFunct "createlist" [(ExtFunct a b newc
       ), (Number 0)]
498                                        subsdefs = creatldef:fun
499                                        creatldef = IntFunction "createlist" [(IntFunct
        "f"), (IntVar "t")] log
500                                        log = Operation (Brackets IntFunct "f" [IntVar
       "t"]) Listadd (IntFunct "createlist" [(IntFunct "f" []), (Brackets Operation (
       IntVar "t") Plus (Number 1))])
```

91

References

[1] M. de Laplace and P. Simon. A philosophical essay on probabilities. *New York, London: Wiley, Chapman and Hall*, 1902.

[2] John Foster. From simplistic to complex systems in economics. *Cambridge Journal of Economics*, 29, 2005.

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

[4] Gary William Flake. The computational beauty of nature. *The MIT Press*, 2000.

[5] Yaneer Bar-Yam. Dynamics of complex systems. *Addison-Wesley*, 1997.

[6] J. Doyne Farmer, M. Gallegati, C. Hommes, A. Kirman, P. Ormerod, S. Cincotti, A. Sanchez, and D. Helbing. A complex systems approach to constructing better models for managing financial markets and the economy. *The European Physical Journal Special Topics*, 214:295–324, 2012.

[7] Ian Domowitz and Benn Steil. Automation, trading costs, and the structure of the securities trading industry. In *Brookings-Wharton Papers on Financial Services*, 1999.

[8] 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.

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

[10] Robert Goodell Brown. *Smoothing, Forecasting and Prediction of Discrete Time Series*. Dover Phoenix Editions, 2004.

[11] Steven C. Bankes. Agent-based modeling: A revolution? *Proceedings of the National Academy of Sciences of the United States of America*, 99:7199–7200, 2002.

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

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

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

[15] Nawfal Naciri and Mohamed Tkiouat. Economic agent based models: Review. *International Journal of Applied Engineering Research*, 11(8):5492–5502, 2016.

[16] N. Gilbert and K.G. Troitzsch. Simulation for the social scientist. *Open University Press*, 2005.

[17] J.M. Epstein. Generative social science: Studies in agent-based computational modeling. *Princeton University Press, Princeton*, 2007.

[18] H. Van Dyke Parunak, Robert SavitRick, and L. Riolo. Agent-based modeling vs. equation-based modeling: A case study and users' guide. *International Workshop on Multi-Agent Systems and Agent-Based Simulation*, 1534, 1998.

[19] Alex Mogilner and Angelika Manhart. Agent-based modeling: case study in cleavage furrow models. *Molecular Biology of the Cell*, 2016.

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

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

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

[23] Mario A. Bertella, Felipe R. Pires, Ling Feng, and Harry Eugene Stanley. Confidence and the stock market: An agent-based approach. *PLOS ONE*, 2014.

[24] Matteo Serri, Guido Caldarelli, and Giulio Cimini. How the interbank market becomes systemically dangerous: an agent-based network model of financial distress propagation. *Journal of Network Theory in Finance*, 2017.

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

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

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

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

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

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

[31] John Comstock Doyle, Bruce A. Francis, and Allen Tannenbaum. *Feedback Control Theory*. Dover, 2009.

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

[33] 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.

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

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

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

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

[38] 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.

[39] Julien Mostard and Ruud Teunter. The newsboy problem with resalable returns: A single period model and case study. *European Journal of Operational Research*, 169:81–96, 2006.

[40] M. R. Hassan and B. Nath. Stock market forecasting using hidden markov model: a new approach. In *Intelligent Systems Design and Applications*, 2002.

[41] Syoiti Ninomiya and Nicolas Victoir. Weak approximation of stochastic differential equations and application to derivative pricing. *Applied Mathematical Finance*, 15, 2008.

[42] Mark M. Meerschaert and Charles Tadjeran. Finite difference approximations for two-sided space-fractional partial differential equations. *Applied Numerical Mathematics*, 56:80–90, 2006.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[57] 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.

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

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

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

[61] 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.

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

[63] Grazziela P. Figueredo, Peer-Olaf Siebers, Markus R. Owen, Jenna Reps, and Uwe Aickelin. Comparing stochastic differential equations and agent-based modelling and simulation for early-stage cancer. *PLOS ONE*, 2014.

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

[65] 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.

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

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

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

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

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

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

[72] 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.

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

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

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

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

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

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

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

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

[81] Davide Sangiorgi. Extensionality and intensionality of the ambient logics. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, pages 4–13. ACM, 2001.