**Chapter 19: InterDyne: A Simulation Method for Exploring Emergent Behavior Deriving From Interaction Dynamics**

Author(s):

\*C. D. Clack (University College London, [clack@cs.ucl.ac.uk](mailto:clack@cs.ucl.ac.uk))

L. Carlos-Sandberg (University College London, [leo.carlos-sandberg.16@ucl.ac.uk](mailto:leo.carlos-sandberg.16@ucl.ac.uk))

**Abstract**

Emergent behaviour in Systems of Systems may arise from the behaviours of the component systems, or from the dynamics of interaction between those component systems, or both.  Here we present the InterDyne simulator, and our simulation method for exploring emergent behaviour that derives from interaction dynamics, and in particular for exploring emergent behaviour that derives from coupling and feedback loops between coupled component systems. We begin with a motivation of interaction dynamics, and end with a case study in financial markets.

**19.1 Introduction**

Emergent Behaviour in Systems of Systems may derive from a variety of different low-level behaviours including (i) behaviour *within* subsystems; and (ii) behaviour *between* subsystems. We are interested in the latter – in the way that emergent behaviour can arise in a System of Systems as a result of the dynamics of interaction between subsystems.

Our leading case study for interaction dynamics in a System of Systems is the financial markets, which exhibit a range of complex and undesirable emergent behaviour. For example, in the US markets “Flash Crash” of May 6th 2010 (CFTC and SEC 2010) market prices became disconnected from rational valuations, and irrational prices and frenzied trading in one market rapidly spread to many other markets, causing extreme price crashes and spikes. For the US financial markets this was “one of the most turbulent periods in their history” (Kirilenko et al. 2014).

The financial markets are an interesting subject for study from a System of Systems perspective since they have a highly complex and interactive structure. Each individual market (such as the equities market, the commodities market, the derivatives market and the foreign exchange market) is itself a complex System of Systems comprising a network of entities such as traders, brokers, dealers, inter-broker dealers, exchanges, clearing houses, investment funds, retail investors, and so on. Each of these markets works in a slightly different way (for example, equities are traded on one or more public exchanges, whereas there is no central exchange for foreign exchange and instead there is a network of foreign exchange dealers quoting slightly different prices).

These markets are further interlinked by traders who operate in multiple markets simultaneously. For example, a trader might manage risky trades in one market by entering into risk-reducing trades in other markets. Further linkage is provided by traders who exploit arbitrage opportunities (such as transient inconsistencies in price) between markets.

**19.1.1 Emergent Behaviour Deriving from Interaction Dynamics**

It is well known in the field of physics that very complex behaviour can emerge from the *interactions* between objects, even between a small number of simple components. Consider, for example, the three-body problem (Sundman 1912) the behaviour of coupled oscillators (Wilberforce 1896), and feedback oscillation in electronics (Barkhausen 1935). An everyday example of undesirable behaviour arising from interaction occurs when the loudspeaker output of an audio amplification system interacts with the microphone input; the interaction causes a feedback loop, resulting in a high-pitched oscillation.

With all of the above examples, the high-level emergent behaviour arises from the interaction between the subsystems, not from the subsystems themselves (other than the subsystems possessing properties that permit them to become engaged in interaction – such as mass, momentum, inertia, energy, and the ability to translate sound into electricity and vice versa). In each example, if the interaction were removed, the high-level behaviour would disappear: for example, a single body if not acted upon by a force is stationary or moves with constant velocity (Newton 1687), a single oscillator alone exhibits simple movement, and an audio amplification system does not normally emit a high-pitched oscillation where the microphone is separated from the loudspeaker.

Within physics, interactions might be physical collisions or the effect experienced by a component when it is acted upon by a field, such as an electric, magnetic or gravitational field. Within the financial markets, trading by humans has almost completely been replaced by computer-directed trading, with computer algorithms making decisions on what, and when, to trade; thus, in the financial markets interactions primarily occur between computer programs and the medium for such interactions is the passing of messages between computers (such as a computer in an investment bank sending an order to a computer at a stock exchange).

Day and Huang (Day and Huang 1990) have demonstrated how interactions between two simple but different trading strategies and a market-maker can cause complex emergent features of stock market prices such as alternating periods of rising (“bull market”) and falling (“bear market”) with sudden switching between the two at irregular intervals. Further, Lyons (Lyons 1997) has shown how a feedback loop can emerge between foreign exchange dealers, causing them to repeatedly transfer inventory between themselves. Our aim is to develop a framework for modelling and analysing emergent behaviour that arises from the dynamics of interaction, and in the context of our case study to analyse behaviour that may increase risk to the stability of the financial markets.

**19.1.2 Feedback Loops**

A good example of complex interaction dynamics in the financial markets is the creation of feedback loops, where a subsystem observes in its input some value that derives from its own output. For example, falling market prices can cause traders to sell (to prevent further loss) and this selling causes prices to fall further (which provokes further selling).

Some feedback loops may have a benign effect (we call these *stabilising feedback loops*) whereas others may be malignant (we call these *destablilising feedback loops*). Control systems routinely use stabilising feedback loops to ensure an output signal stays close to a desired reference signal – see for example (Skogestad and Postlethwaite 2007, Zames 1966, Horowitz 1959).

The process by which an output value is transformed into an observed input value may be complex and transitive (i.e. it may involve intermediate processing by one or more other subsystems); this is illustrated in Figure 19.1 below, where interactions between four subsystems are depicted by directional arrows – one direct feedback loop is depicted by an arrow from subsystem 1 to itself, and one indirect feedback loop can be discovered by following the arrows from subsystem 2 to subsystem 3, to subsystem 4, and back to subsystem 2. A practical example of a direct feedback loop would be a trading algorithm that issues orders to a stock exchange where the size and price of those orders is based on the previously issued order (or perhaps on a history of previously issued orders). Alternatively, the size and price of an issued order may depend on the current size of the trader’s inventory – when each order is executed at the exchange this will cause the trader’s inventory to be changed, and this will affect the next order issued. Notice how in these two examples the feedback loop does not exist at a single point in time but rather *across* time – the trader’s inventory is affected by orders filled at a previous time, which are due to orders issued at a time before that, and whose sizes were affected by the size of the trader’s inventory at a further previous time. The fact that a feedback loop extends across time does not make it less real (nor, potentially, less powerful) but it can make it more difficult to analyse.

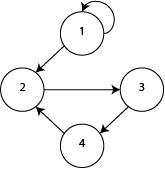


Figure 19.1 - Four subsystems, with one direct feedback loop and one indirect feedback loop.

When constructing a System of Systems model of a financial market, the behaviour of the subsystems will be specified and initial conditions will be set. In such a model, feedback loops may exist *ab initio*, and may potentially be discoverable by analysis of the specification and initial conditions. We call this a *static analysis* of the specification, because it does not require the behaviour of the subsystems to be traced (simulated) in time. Different kinds of static analysis may be required to identify different kinds of feedback loop – for example, identification of a circular loan relationship between banks might require a different analysis to identification of a feedback effect on the calculation of order sizes for trading.

Alternatively, feedback loops might be created dynamically as a result of the changing behaviour of the subsystems specified in the model (for example, a new loan relationship may create a feedback loop where there was none previously). The role of the content of messages should not be underestimated, since coupling between subsystems may for example depend upon the sizes of transactions and whether one subsystem is buying or selling. Similarly, dynamic coupling effects may involve several subsystems, and feedback effects may depend upon repeated patterns of messages occurring between several subsystems.

To identify and understand such dynamically-created feedback loops requires analysis of the dynamic behaviour of the different subsystems in the model – for example, it requires the values of different functions and state variables to be traced from one time step to the next, and it may require inspection of the history of messages (including their data content) sent from one subsystem to another. We call this a *dynamic analysis* of the model.[[1]](#footnote-1)

As part of dynamic analysis, we may also wish to investigate whether feedback loops increase or decrease in size or effect (that is, their effect on various – typically high-level - properties of the model) and whether such feedback loops split, merge or disappear.

For a given model of a System of Systems, including its specification and initial conditions, we define a *static feedback loop* as one that always exists, with unchanging size and effect. We expect that a static feedback loop is potentially discoverable by a static analysis of the model.[[2]](#footnote-2) By contrast we define a *dynamic feedback loop* to be any feedback loop that is not static: i.e. it may be transient, may have changing size or effect, and might not be detectable via static analysis. We expect that the identification of a dynamic feedback loop would require a dynamic analysis of the model, and we anticipate that dynamic feedback loops might be highly unpredictable and difficult to identify, analyse and understand.

**19.2 Research Focus**

Our initial interest in the financial markets derived from reading the reports of the U.S. Commodity Futures Trading Commission (CFTC) and the U.S. Securities and Exchange Commission (SEC) into the “Flash Crash” of May 6th 2010 (CFTC and SEC 2010), coupled with early observations from academics (Easley et al. 2011) and from industry practitioners (Nanex 2010a). We noticed that much of the approach and reasoning that was employed in the analyses immediately following the Flash Crash did not take into account the System of Systems nature of the financial markets; the issue of emergent behaviour was not discussed, and although the evidence presented in the CFTC/SEC report described several feedback loops we felt that there was insufficient attention paid to the market impact of low-level dynamic interaction, including these feedback loops. Our research therefore seeks to improve the conceptual understanding of financial markets as a System of Systems with emergent behaviour arising from interaction dynamics.

**19.2.1 Fine-grained Microstructure Approach**

To investigate the interaction dynamics involved in the Flash Crash, it is necessary to model message-passing at a very fine-grained level. For example, it is necessary to model the timing of the arrival of trader’s orders at an exchange, because the sequence in which orders are processed by the exchange might support or defeat a feedback loop. It is necessary not only to model message-passing but also to run experiments to collect and observe the *precise orderings* of messages and their content. Furthermore, several reports (CFTC and SEC 2010, Nanex 2010b) have mentioned the existence of communication delays and it is therefore necessary to model the effects of varying delays. Thus, our task has been to model the detailed microstructure of the financial markets at the level of the passing between entities of individual messages such as orders, confirmations and market data.

**19.2.2 Discrete Time**

One challenge in modelling the interaction behaviour of financial markets is the linking of the extremely fast behaviour of discrete-time computers (and the automated trading and matching algorithms that they run) with the comparatively slow human observation of market behaviour.

When viewed at human timescales, the financial markets may appear to operate in continuous time. However, at a fine-grained level of detail all computer operations are effected in discrete time dictated by the change in voltage of a system clock (a chip that emits an extremely precise square-wave oscillating voltage – for example, a 2.7GHz chip oscillates 2,700,000,000 times per second). The passing of messages between two computers is a communication between two discrete-time systems linked by a transmission system; the transmission system itself typically comprises a sequence of cables and intermediary devices, where the intermediary devices may operate in discrete time yet the cables do not operate in discrete time. Despite the continuous-time nature of some parts of the transmission system, a message will only be received by a computer at a discrete time determined by the receiving computer’s system clock (a message arriving earlier will not be processed until the next triggering edge of the clock voltage).

Discrete time is also used in coarse-grained models, where in different models an individual time step may represent an intraday period, a day, or a month. An important characteristic of discrete-time models is that each time step represents an equal amount of time; the extent of each time step may be defined (e.g. as a number of seconds, minutes, hours or days), though in some models (e.g. (Huang et al. 2012)) the extent is left unstated in the model and may then be instantiated in a numerical simulation.

**19.3 Method**

Our approach is to model in discrete time at the finest level of detail appropriate for an experiment, and then to simulate that model for sufficiently many time steps to explore the high-level behaviour of interest (this naturally requires computational efficiency). For our purposes, the time gaps between events are just as important as the events themselves, since for example excessive time gaps may represent information delays that have been suggested by some observers to be contributory to emergent behaviour such as the Flash Crash (CFTC and SEC 2010, Nanex 2010b). Thus, we require a discrete-time model rather than a discrete-event model; furthermore, our interest is focused on the *interaction* between entities rather than on the entities themselves, and we require the ability to observe these interactions in detail.

We have considered several different approaches to modelling financial markets in discrete time. At first we explored using a process calculus (Milner 1999) to model explicit communication between subsystems (Rötzer 2012), but we found this frustrating since the very low level of specifying communication steps hindered expression of the higher-level algorithms that initiate or respond to that communication. What we sought for exploring interaction dynamics was a technique that occupied a “Goldilocks position” that neither models at a level of abstraction that is too high (eg probabilistic modelling) nor at a level that is too low (eg process calculus).

An agent-based model (Chen et al. 2010) provides an attractive fit to modelling interaction dynamics in a System of Systems, since each subsystem can be modelled by a separate agent or by a group of agents with varying levels of detail, with private communication within a group; furthermore, messaging and discrete time are handled naturally, and numerical solutions can be obtained for large problems that are analytically intractable. However, we were initially wary of this approach because of the limitations that cause agent-based models to be not widely accepted by economists, as described for example by Gould et al (Gould et al. 2013) and Leombruni and Richiardi (Leombruni and Richiardi 2005). For our purposes, the most important limitations appear to be:

* Although each agent is fully specified, the model as a whole may lack a formal definition.
* It can be difficult to track how a specified input parameter affects the output, and parameter-estimation may be achieved in a way that is not representative of all possible outcomes the model can produce.
* Finding a set of agent rules that produces a specific high-level behaviour provides no guarantee that it is the only set of rules to do so.

Leombruni and Richiardi (Leombruni and Richiardi 2005) resolved the first of these problems by providing a formal specification of an agent-based model in terms of a set of recurrence relations. This has the added advantage that the recurrence relations are easily understandable by non-programmers. They also showed how to ameliorate the second problem by analysis of the sensitivity of outcomes to parameter selections. The third problem remains, and is shared by many models,[[3]](#footnote-3) yet such models are of great value in hypothesis formulation where they can disprove a statement of the form “This high-level behaviour cannot occur”.

**19.3.1 Recurrence Relations**

Leombruni and Richiardi (Leombruni and Richiardi 2005) have shown how a discrete-time agent-based model of the dynamic microstructure of financial markets can be expressed as a set of recurrence relations.[[4]](#footnote-4) In their formulation, each of *n* agents is well described by a state variable given by *xi,t* where *i* is the agent identity (*i* ∈ *1, ... , n*) and *t* is time, and where each *xi,t+1* is describable by a recurrence relation as follows:

*xi,t+1 = fi(xi,t, x-i,t; αi)*

In the above equation, the value *xi,t+1* at time *t+1* may depend on its previous value *xi,t* at time *t*. The state evolution function *fi()* may be different for each agent, and each agent may have a bespoke parameter *αi.* The parameter *x-i,t* refers to the states at time *t* of all agents other than *i*. Note that the recurrence relation describes a state at time *t+1* only in terms of states at a *previous* time *t* (since it is not possible to know future events) and that initial values *xi,0* are assumed to exist.

Macro-level properties of the model can always be solved by iteratively solving each term *xi,t*. This provides a formal definition of the model that is accessible by domain experts, that expressly indicates the dependency of one subsystem on another (i.e. if *x3,t* occurs in the definition of *x1,t+1* then we say that *x1,t+1* depends on *x3,t*) and that is amenable to static analysis (e.g. it would be possible to perform an automatic dependency analysis). It is also possible to reason about information delays if we generalise the model to *xi,t+1 = fi(xi,t, xi,-t, x-i,t, x-i,-t; αi)* where the subscript *–t* refers to all times previous to *t*. Thus, it is for example possible to express a dependency of *x1,t+1* on *x3,t-4*.

Although recurrence relations are naturally recursive, this does not mean that destabilising feedback loops are “hard-wired” into a model that uses recurrence relations. Typically, if a relation is recursive then it will depend on its own value at a previous time step (e.g. *xi,t+1* may depend on *xi,t*), which need not be destabilising. However, if *xi,t+1* were to depend on *xi,t+1* then this would be a destabilising feedback and during a static analysis we would hope to be able to detect all cases where a state variable *might* depend upon its own value at the same time step.

There are many benefits that accrue from the use of a set of recurrence relations to model market behaviour, and such a model is very amenable to static analysis. However, a set of recurrence relations is not as well suited to dynamic analysis:

* although dependencies are explicit, viewing interaction as the sending and receiving of messages is less explicit;
* although an individual recurrence relation provides a time history of values for a given subsystem we do not have a time history of messages between two selected subsystems;

For the analysis of interaction dynamics, we wish to investigate not only the history of the states of an agent (i.e. the sequence *xi,0, xi,1, …*  for every agent *i*) but also the history of messages between every pair of agents. For example, if *x1,t+1* depends on its previous state at time *t* and on the state of agent *x2* at time *t-3* (i.e. the dependency set is *{x1,t, x2,t-3}*) then the history of dependencies at each time step (*t=1*, *t=2*, *t=3*, *t=4*, *t=5*, …) for *x1* is given by the sequence *{x1,0, x2,-2}, {x1,1, x2,-1}, {x1,2, x2,0}, {x1,3, x2,1}, {x1,4, x2,2}, …* and if we say *xi,t* is undefined for *t<1* (i.e. no dependency exists) we have: *{}, {x1,1}, {x1,2}, {x1,3, x2,1}, {x1,4, x2,2}, …*

Finally we note that where the function *fi()* contains conditional statements, e.g. where the state *xi,t+1* is conditioned on some value *xj,t*, then the set of dependencies may be conditional and in the general case the dependency set becomes more difficult to analyse.[[5]](#footnote-5)

By contrast, with an agent-based simulator it is typically a simple matter to investigate the interaction dynamics by observing actual messages transmitted from one agent to another (the absence of a message does not imply absence of dependency, but the presence of a message does imply dependency).

**19.3.2 Two Views**

Ideally we would like to perform both static analysis of a System of Systems model expressed as a set of recurrence relations and dynamic analysis of an agent-based model of the same System of Systems. We consider these to be two complementary views of the same System of Systems. Eventually, we would like to be able to express the System of Systems as a set of recurrence relations and then generate the agent-based model automatically; if this could be done in a way that preserves the semantics of the specification, then we would be confident that the static and dynamic analyses are being performed on models of the same System of Systems.[[6]](#footnote-6)

A numerical simulator for an agent-based model of interaction dynamics for System of Systems has been developed and will be presented in the following section. This simulator (called “InterDyne”) has been operational for some time and has been used to model a variety of financial Systems of Systems and to explore emergent behaviour in those systems.

We have also performed static analyses on specifications of financial Systems of Systems expressed as a set of recurrence relations. Creating the link between the two views is the focus of current research, and we are exploring the approach of using a sequence of correctness-preserving transformations that would make incremental modifications starting with the set of recurrence relations and ending with agent code that can be used in the InterDyne simulator.

**19.4 The InterDyne Simulator**

InterDyne is an agent-based simulator for the exploration of interaction dynamics in complex Systems of Systems. Although InterDyne is a general-purpose simulator that can be used in a variety of fields it has primarily been used to investigate models of financial markets.

The structure of an InterDyne simulation is straightforward, with one or more agents passing messages via a “Simulator Harness”, as illustrated in Figure 19.2, and with each subsystem in the System of Systems model being represented by a single agent or by a collection of agents. The primary output from InterDyne is a trace file, suitable for post-hoc analysis and interactive visualisation to explore the antecedents of emergent behaviour.

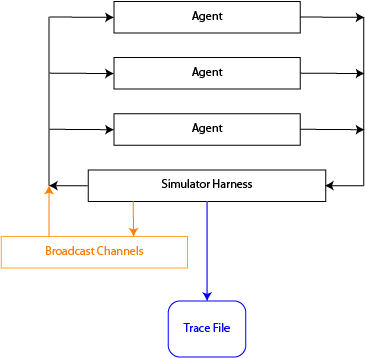


Figure 19.2 – InterDyne simulation with three agents.

InterDyne is run by executing the function "sim" applied to appropriate arguments (see Figure 19.3).  The arguments include:

* the number of time steps for the simulation,
* a list of runtime arguments: these are (key, value) pairs that are made available to every agent in the system, and
* a list of information about each agent: namely, a two-tuple containing the name of the function that implements the agent, and a list of broadcast channel identifiers (see Section 19.4.3) to which the agent will listen.

Each agent is uniquely identified by its position in that list of agent information - the first agent has identifier 1, the second has identifier 2 and so on.  Identifier 0 is reserved for the simulator “harness" function that mediates messaging, controls the passage of time during InterDyne simulation, and sends output to a trace file.  The agent identifiers are used to specify the source and destination of all one-to-one messages.

Each agent receives a sequence of packets containing messages and outputs a sequence of packets containing messages; at each time step an agent both consumes one input packet and generates one output packet. Each output packet contains zero or more messages (of varying type – see Section 19.4.3), and each input packet contains three items: (i) the current time, (ii) a collection of zero or more direct messages, and (iii) a collection of zero or more broadcast messages. Each agent only receives direct messages where its agent identifier is in the recipient field of the message (i.e. agents do not see messages sent to other agents), and only receives broadcast messages sent to broadcast channels to which this agent has subscribed. Broadcast messages support sending the same message to a large number of other agents, such that the recipients can be defined and changed in the set-up of the experiment rather than in the agent code.

### **19.4.1 InterDyne Time**

InterDyne operates in discrete time, and simulation experiments are executed for a defined number of time steps. InterDyne does not specify what period of real time corresponds to each time step; this is a matter for the experimenter to define, according to the requirements of the model and its simulation, and a time step is typically set to be the smallest required resolution with all other times being integer multiples of that time. The experimenter may also choose not to give such a definition; for example, where the behaviour being investigated is known to be or assumed to be rate-independent.

### **19.4.2 InterDyne Agents**

InterDyne supports a heterogeneous set of communicating agents, and each agent can be modelled with a different level of detail. For example a System of Systems model may comprise two agents: one that generates outputs dependent on a statistical distribution together with a second agent that is a thorough implementation of a complex trading algorithm with considerable internal complexity.

At each time step an agent must consume one inbound message packet and must generate one outbound message packet.  Each packet may contain multiple messages and inbound packets are different to outbound packets (see later example). If an agent does not have any messages to receive or send at a given time step then it will either receive or generate an empty packet.  Optionally an agent may distinguish between an output packet that is a empty by mistake and an output packet that is a empty by design - it does this by generating an output packet containing the distinguished empty message called a "Hiaton".[[7]](#footnote-7)

InterDyne agents are typically (but not always) written in two parts: (i) a "wrapper" function that manages the consumption of inbound messages, the generation of outbound messages, and the update of local state, and (ii) a "logic" function that is called by the wrapper function and which calculates the messages to be sent.  The “wrapper” function is the agent function, and the “logic” function is a subsidiary function with restricted scope so that it can only be invoked by the wrapper function.

Here is a simple agent wrapper that does nothing, written in the programming language Haskell:[[8]](#footnote-8) at each time step it reads an inbound item, and creates an empty outbound item (the inbound item is not used).  An inbound packet is represented as a Haskell three-tuple (discussed below), and a sequence of packets is represented as a list of packets. This example wrapper function does not call a logic function:

f st args ((t, msgs, bcasts):rest) myid = []:(f st args rest myid)

The agent function “f” is recursively defined and loops once per time step.  A typical Haskell syntax is to write the name of the function, followed by names for its arguments, to the left of the equals sign: to the right of the equals sign is the expression that calculates the value returned by the function. In this case, the function “f” takes four arguments: st is a local state variable (in this example it is never inspected and never changed), args is a copy of the runtime arguments (every agent is passed a copy of all the runtime arguments), and the penultimate argument is the list of inbound packets written as a structural pattern so that individual parts can be named: the first available inbound packet is a 3-tuple and the remainder of the list (the packets that will be available in the future) is given the name “rest”; the components of the first inbound packet are (i) the current time “t”; (ii) a list “msgs” of all one-to-one messages sent to this agent to be received in this time step; and (iii) a list “bcasts” of all broadcast messages available at this time step on all the broadcast channels (see Section 19.4.3) to which this agent is subscribed. The last argument myid is the identifier of this agent (which should never be changed).

The output of the function is a Haskell list of outbound packets, one per time step, and each outbound packet is a list of messages (including both one-to-one messages and broadcast messages). Thus, the output comprises a first list item (an outbound packet – in this case the empty list “[]”) connected via the Haskell operator “:” to the remainder of the list items (given by the result of the recursive call). In the recursive call the three-tuple “(t, msgs, bcasts)” is missing from the third argument (only the tail of the list, called “rest” in this example, is used), and so in the next recursive invocation the function f will observe at the front of the list the inbound packet corresponding to the following time step.

exampleExperiment1   
  = do  
    sim 60 [] agents  
    where  
    agents = [ (traderWrapper, []),  
               (brokerWrapper, []),  
               (exchangeWrapper, []) ]

exampleExperiment2   
  = do  
    sim 60 [convert] (map snd agents)  
    where  
 convert = generateAgentBimapArg agents  
    agents = [ ("Trader",  (traderWrapper, [1] )),  
               ("Broker",  (brokerWrapper, [3] )),  
               ("Exchange",(exchangeWrapper, [2,3])) ]

**Figure 19.3 – Two simple code examples for running an InterDyne simulation. Both examples run for 60 time steps and have three agents. The second example differs in two ways: it uses a convert function so that agents can be referenced by name rather than number, and the agents themselves subscribe to broadcast channels.**

Figure 19.3 illustrates two versions of the function sim running an InterDyne simulation with three agents. In both examples the simulator runs for 60 time steps. The second argument is a list of runtime arguments - empty for the first example, and the single argument convert in the second example. The argument convert is an application of the library function generateAgentBimapArg to myagents which results in a runtime argument containing a function that converts a name to an agent identifier and vice versa.[[9]](#footnote-9)  The third argument is a list of agents and broadcast channels on which they will listen – in the first example there are no subscriptions.[[10]](#footnote-10)  In the second example the first agent subscribes to broadcast channel 1, the second subscribes to channel 3, and the third subscribes to channels 2 and 3.

These examples omit many details – for example they do not give the definitions for the agent functions traderWrapper, brokerWrapper and exchangeWrapper; they do not illustrate how to define an output file for the results; nor how to use names instead of integers for broadcast channels; nor how to specify the communications topology and the delays that should be applied to each communication link (see Section 19.4.4).  They do however indicate the parsimonious style that can be achieved when using InterDyne.

### **19.4.3 InterDyne Interaction and Messages**

InterDyne supports message-based communication between agents as the only form of interaction. Messages are directed (a message is sent from one agent to one or more others), and where two agents need to send messages to each other simultaneously this is modelled as two separate directed messages. InterDyne supports a wide range of messages from the very simple (for which examples are given below) to the very complex (for example, detailed messages between subsystems of a financial system using the industry-standard Financial Information eXchange protocol).  The messaging system is of central importance to InterDyne since messages are the medium for interaction between subsystems.

This communication can be one-to-one (a private message from one agent to another) or one-to-many (a broadcast message which can be read by all other agents). Both of these messaging types are needed in modelling financial markets, for example an exchange would send a broadcast message to all members updating them on the latest executed trades, whereas a trader would send a private message to an exchange to place an order. Broadcast messages are directed to a broadcast channel, and agents subscribe to zero or more broadcast channels at the start of a simulation.

All messages comprise (i) a tag to indicate the type of message being sent; (ii) a pair of integers that indicate either the identifiers of the sending and receiving agents (for one-to-one messages) or the sending agent identifier and the receiving broadcast channel identifier (for broadcast messages); and (iii) the message data.  The message types (and associated tags) include both generic types and a few domain-specific types. Examples include:

* Message (1, 2) data - a one-to-one message, from agent 1 to agent 2, typically sending a list of (key, value) pairs
* Ordermessage (3, 4) data - a domain-specific one-to-one message, from agent 3 to agent 4, sending data that represents an order (typically sent to a subsystem that models an exchange)
* Datamessage (3,1) data - a one-to-one message, from agent 3 to agent 1, where the data component is a string
* Debugmessage (1, 4) data - a one-to-one message, from agent 1 to agent 4, where the data component is a string (for debugging purposes)
* Broadcastmessage (3, 1) broadcastdata - a broadcast message, sent from agent 3 to broadcast channel 1, containing data that will be received by all agents that have previously subscribed to broadcast channel 1

If a message is sent to agent identifier 0 it receives special treatment.  Agent identifier 0 is reserved for the simulator harness, and a message sent to identifier 0 is printed to an output file.  This is the primary mechanism for defining the output of the simulator.  Currently the main output file is the "trace file" that records all messages sent to identifier 0 except for those with type Datamessage - these messages are instead output to a comma-separated file, and this provides a mechanism for structured output that can be viewed as a spreadsheet.

Here is a simple agent wrapper that sends a debug message to the output trace file (via the simulator harness) at every time step.  It does not call a logic function:

f st args ((t, msgs, bcasts) : rest) myid  
     =  [m]  : (f st args rest myid)  
        where  
        m = Debugmessage (myid,0) ("Debug "++(show t))

### **19.4.4 InterDyne Delays and Topology**

An important aspect of interaction dynamics is the role played by communication delays. In our discrete-time models there is always a minimum communication delay in that we assume a message sent at time step *t* will not be received until time step *t+1*. Where greater time delays are required, these could be implemented in a variety of ways; for example, an agent could implement code that explicitly puts some or all messages into a delay queue prior to processing, or explicit delay agents could be used. Having explored a number of different mechanisms, we have chosen to extend InterDyne to support (but not mandate) the use of delay information that is passed to the simulator as an optional runtime argument.

The delay information provides a unique delay value (as an integer multiple of time steps) for each directed communication path between two agents. A specified delay applies to any messages using that interaction path, whether the message is defined as being one-to-one or one-to-many. Since the communication paths are directed, asymmetric delays can be specified between two agents: for example, messages sent from agent A to agent B may have a greater delay than messages sent from agent B to agent A. Passing the delay information as a runtime argument provides a uniform mechanism where the amounts of delay between different agents can be varied systematically without requiring agent code to be changed.

This approach also provides an opportunity to define the topology of connections between InterDyne agents: if messages are not permitted from Agent A to Agent B then the communication delay for that path could be set either to an abort error message or to a delay that is longer than the expected length (in time steps) of the experiment.[[11]](#footnote-11) This defines the interaction topology of the simulation as a directed graph, with the agents being the nodes of that graph and the communication paths being the edges.

The interaction topology can be used as a validation tool (to test a complex experiment to determine whether a subsystem is sending messages as expected), but normally is used as part of the mapping from model to simulation, to reflect a semantic interpretation of the subsystems of the modelled System. For example, as illustrated in Figure 19.4 an investment bank subsystem might be modelled as two agents – a sales agent and a trader agent; the sales agent may communicate with several client subsystems and pass client orders to the trader agent for execution, and the trader agent would send orders to a stock-exchange subsystem, yet the sales agent may be prohibited from sending/receiving messages directly to/from the exchange, and the trader may be prohibited from sending/receiving messages directly to/from the clients. This provides a way to express a degree of hierarchy of systems and subsystems in an overall System of Systems.[[12]](#footnote-12)



**Figure 19.4**

To define delays and interaction topology, two runtime arguments must be passed to the sim function (both arguments must be present, or none):

1. the name of a function that takes two agent identifiers (integers that uniquely specify the start point and end point of an interaction) and returns an integer delay which is taken to be a number of timesteps; and
2. the maximum delay in the system.

The user has complete freedom to define the delay function, including sending trace messages and raising errors if an attempt is made to send a message on a path that is not permitted.  The simulator harness (agent identifier 0) detects the presence of these two runtime arguments and, if they are present, uses the function to calculate the required delay for every sent message (agents normally ignore these two runtime arguments). In the current implementation the delay function cannot be conditioned on other factors such as time or local state, and can therefore only implement static delays (which, of course, could be changed for different runs of an experiment). To implement dynamically-varying delays the user would create a delay agent to act as an intermediary on a given communication path.

exampleExperiment   
= do  
  sim 60 args agents  
  where  
  args = [ (Arg (Str "maxDelay", maxDelay)),  
             (DelayArg (Str "DelayArg", delay)) ]  
 maxDelay = 3  
 delay x y = if ((x = 1) && (y = 2)) then 1 else  
 if (x = 2) then 0 else  
 if ((x = 3) && (y = 2)) then 3 else  
  error (“bad message: from ”++show x++” to ”++show y)  
 agents = [ (traderWrapper, [1] ),  
            (brokerWrapper, [3] ),  
            (exchangeWrapper, [2,3]) ]

**Figure 19.5**

Figure 19.5 gives a simplified example for three agents: runtime arguments are (key, value) pairs: the maximum delay argument has a string key “maxDelay” and value maxDelay, and is passed as a runtime argument using the tag Arg (a Haskell constructor), whereas the delay function has the key string “DelayArg” and uses the Haskell constructor DelayArg. The maximum delay is a whole number of time steps – in this case 3.

After setting the delays as shown above, one-to-one messages are automatically delayed by the stated number of time steps (additional to the minimum communication delay of one time step), and broadcast messages are split into separate messages for each recipient and delayed by the amount stated for each communication path (these “routed broadcasts” are received in the list of normal messages in an agent’s input).

### **19.4.5 InterDyne Determinism**

InterDyne supports both deterministic and non-deterministic simulations. Unless specified otherwise by the user, an InterDyne simulation will be deterministic and provide the same results every time it is run with the same initial values (this can be helpful in determining causal pathways). The user may alternatively express non-determinism in two ways: by including non-determinism in the code for an individual agent, and/or by instructing InterDyne to randomise the order in which multiple inter-agent messages are received at each time step (the alternative is that where multiple messages are received in one time step they are processed in order according to their agent identifier). Randomising message arrival is effective where an agent receives more than one message from another agent in one time step, and can be useful to remove suspected systematic bias when exploring dynamic behaviour (for example, a feedback loop might be manifest if a subsystem representing a stock exchange always processes orders from one trader subsystem first, yet may disappear when this systematic bias is removed). The re-ordering of messages is managed in the same way every time the simulator is run and is therefore repeatable (if different behaviour is required, different randomisation seeds can be used on each run).

**19.5 Case Study**

To illustrate the use of InterDyne to explore interaction dynamics, we investigate the emergent phenomenon of “Hot Potato” trading, as observed for example in Foreign Exchange (Lyons 1997) and Futures (CFTC and SEC 2010, Kirilenko et al. 2014) markets.

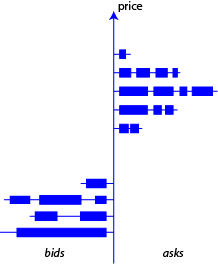
We define Hot Potato trading as the *repeated* passing of inventory imbalances (i.e. trading) between dealers (market-makers) (Lyons 1997), and we define Hot Potato Instability as the repeated passing of inventory imbalances between dealers when market supply and demand are zero. Hot Potato trading starts when a dealer with excess inventory sells to another dealer: if that causes the second dealer to have excess inventory, the second dealer sells to another dealer; if the inventory returns to the first dealer, a cycle is created and repeats until non-dealer trades remedy the imbalances. Hot Potato trading may be prompted by repeated market supply or demand: by contrast Hot Potato Instability may have been initially triggered by market supply or demand but thereafter repeats without further input from the market (it is thereafter a self-exciting oscillation).

We start with a simple problem (we call it “the *n-dealer* problem” for Hot Potato trading on a limit order book): given *n* dealers (market makers) who, being averse to inventory risk, employ a threshold strategy to avoid excessively high or low inventory, and each of which submits orders to an exchange running a limit order book, in the context of a market with supply *s* and demand *d*, under what circumstances can those dealers be compelled to enter into Hot Potato Instability?[[13]](#footnote-13)

To explore this problem, we consider a System of Systems comprising the following subsystems, each implemented as a single agent:[[14]](#footnote-14)

* A subsystem that implements the behaviour of a financial exchange operating a central limit order book (see Figure 19.6), e.g. for the trading of equities (shares) in a single company.
* *n* subsystems, each of which implements the behaviour of a dealer that both buys and sells equities and makes a profit from the difference between the buy price and the sell price. Each dealer will implement an inventory threshold policy that controls the types, sizes and prices of orders sent to the exchange.

Market supply and demand prior to the Hot Potato behaviour are implemented by proxy – i.e. by the setting up of initial inventories for the dealers. So the research question is whether it is possible to trigger Hot Potato Instability (which requires no further market supply or demand) by some prior pattern of market behaviour that sets the dealers’ inventories in such a way that the dealers enter into unstable trading between themselves.



**Figure 19.6: a central limit order book maintains a data structure containing previously received “limit orders”. These will be either “bids” (orders to buy a stated quantity at a stated price) or “asks” (orders to sell a stated quantity at a stated price). The data structure is ordered by price, and for each price there is a queue of received limit orders, sorted by arrival time. When a market order is received it will state whether to “buy” or “sell” a stated quantity – the price will be the best obtainable. A “buy” is matched against the lowest-priced “ask” and a trade is executed at that price and for the quantity of the smaller of the “buy” and “ask” quantities – any remaining “ask” quantity stays in position in the queue, and any remaining “buy” quantity is matched against the next “ask” in the queue (or against the first “ask” in the queue at the next highest price). This process is mirrored when a “sell” market order is matched against “bid” limit orders.**

The interaction topology of this System of Systems is defined such that the exchange can send and receive messages to and from all dealers (and vice versa) but none of the dealers can interact with each other directly.

Each dealer must implement a threshold policy for inventory control. Dealers are typically subject to multiple risks, with the two predominant risks being inventory risk (for example, if too high an inventory were kept then its value might suddenly diminish if the market price were suddenly to drop) and adverse selection risk (for example, that a trader who sells to the dealer might have better information than the dealer and might know that the market price is about to drop). In the *n-dealer* problem only one type of risk is considered – inventory risk. Huang et al (Huang et al. 2012) have shown that an optimal way for a Foreign Exchange dealer to manage inventory risk is to use a dynamic threshold, where the inventory must not be permitted to exceed a threshold value that may vary according to dynamic factors such as the size of the spread (the difference between buy and sell prices). When inventory does not exceed a given threshold, the dealer submits both a “bid” quote and an “ask” quote and profits from the spread when both orders are executed. By contrast, if inventory exceeds a given threshold a market order is issued to another dealer for an amount that is exactly equal to the amount that exceeds the threshold.

The Huang et al model assumes a dealer market where prices are constrained by a dominant dealer, and so there is no control of the prices or sizes of the quotes issued by the dealer whose inventory is optimized. By contrast, in our model we have an order-book market and we will provide more subtle inventory control by allowing the market maker to vary both the prices and sizes of its “bid” and “ask” limit orders; for example, as inventory approaches (but is not in excess of) the upper threshold, the market maker may make the size and price of its “bid” orders unattractive to sellers whilst making the size and price of its “ask” orders more attractive to buyers (the combined effect of which should be to reduce the dealer’s inventory).

Given the greater control afforded to market makers in an order-book market, it is not obvious how Hot Potato behaviour could be triggered. Given that a market maker whose inventory is just below the upper threshold could opt to issue no bid orders at all[[15]](#footnote-15) or could issue a bid for a price that is below other bids and therefore is unlikely to be executed, how could such a market maker be induced to buy inventory that it does not want to buy?

**19.5.1 Time**

Our model assumes that all subsystems have synchronised clocks with no time drift between subsystems. All communications delays are assumed to be equal.

We let the quantum of time be the total amount of time taken for data to be accessed (communicated), for calculations to be made, and for a new datum to be generated: in this simple model we assume that this is equal for all subsystems interacting via any communication path.

At the start of time *t*, a subsystem can access the values of other subsystems at time *t-x*, where *x>=1*, and generates its own datum for time *t*. This datum will then be available to other subsystems at time *t+y*, where *y>=1*.

**19.5.2 Static Analysis**

In this section we show how static analysis can be performed on a recurrence-relation model of the defined System of Systems.

**19.5.2.1 The exchange subsystem**

We present recurrence relations to define the dynamic behaviour of the two order books of bids and asks respectively (*ob\_bids* and *ob\_asks*), of the market orders that have been executed (*xsells* and *xbuys*), and of the limit orders that have been executed (*xbids* and *xasks*). These recurrence relations assume appropriate definitions of operators “⊕”, “∅“, “⊗” and “§” as explained below:

* *(x* ⊕ *y)* incorporates limit orders contained in “*y*” into the order book “*x*” using the correct price-time sequence (Gould et al. 2013). The result is an order book.
* (*x* ∅ *y*) takes the market orders contained in “*y*” and calculates which limit orders in the order book “*x*” should be executed against those market orders and should therefore be removed from “*x*”. The result is an order book.
* (*x* ⊗ *y*) calculates which of the market orders contained in “*y*” will be executed against the limit orders held in order book “*x*”. The result is a set of market orders, and if either *x* or *y* are empty or undefined the result is the empty set *{}*.
* (*x* § *y*) calculates which of the limit orders in the order book “*x*” will be executed against the market orders held in “*y*”. The result is a set of limit orders, and if either *x* or *y* are empty or undefined the result is the empty set *{}*.

The empty order book is given by *ob\_empty*. To simplify the model, we assume that all limit orders are good for only one time step – if they are not executed within one time step of being added to the order book they are discarded (this requires the dealers to issue new limit orders at each time step).

Given the above operators, Figure 19.7 illustrates the required recurrence relations. Notice that *xsellst*, for example, depends on *ob\_bidst* which in turn depends on *lo\_bidst-1*: in calculating the set of executed sells at the start of time *t*, the exchange uses the bids that were issued at the start of the previous timestep *t-1*. The market orders *mo\_sells* and *mo\_buys* and limit orders *lo\_bids* and *lo\_asks* are the combined orders that are submitted to the exchange by the dealers.

*ob\_bidst = (ob\_empty* ⊕ *lo\_bidst-1)* ∅ *mo\_sellst-1*

*ob\_askst = (ob\_empty* ⊕ *lo\_askst-1)* ∅ *mo\_buyst-1*

*xsellst = ob\_bidst* ⊗ *mo\_sellst-1*

*xbuyst = ob\_askst* ⊗ *mo\_buyst-1*

*xbidst  = ob\_bidst* § *mo\_sellst-1*

*xaskst = ob\_askst* § *mo\_buyst-1*

*mo\_sellst = (∪i dealersellsi,t), for i ∈ {1..n}*

*mo\_buyst = (∪i dealerbuysi,t), for i ∈ {1..n}*

*lo\_bidst = (∪i dealerbidsi,t), for i ∈ {1..n}*

*lo\_askst = (∪i dealerasksi,t), for i ∈ {1..n}*

**Figure 19.7**

*dealerbidsi,t = {(bidpricei,t, bidsizei,t)}, if (lthreshi,t < invi,t < uthreshi,t) and even(t)  
 = {}, otherwise*

*dealerasksi,t = {(askpricei,t, asksizei,t)}, if (lthreshi,t < invi,t < uthreshi,t) and even(t)  
 = {}, otherwise*

*dealersellsi,t = {(invi,t – uthreshi,t)}, if (invi,t > uthreshi,t) and even(t)  
 = {}, otherwise*

*dealerbuysi,t = {(lthreshi,t – invi,t)}, if (invi,t < lthreshi,t) and even(t)  
 = {}, otherwise*

*invi,t  = invi,t-1 +xbuysizesi,t-1 – xsellsizesi,t-1+xbidsizesi,t-1 – xasksizesi,t-1, if even(t)  
 = invi,t-1, otherwise*

*xbuysizesi,t = ψi(xbuyst)*

*xsellsizesi,t =ψi(xsellst)*

*xbidsizesi,t = ψi(xbidst)*

*xasksizesi,t = ψi(xaskst)*

**Figure 19.8**

**19.5.2.2 The dealer subsystems**

To complete this very simple model Figure 19.8 provides definitions for the individual limit and market orders issued by the dealers (where *lthresh* is the lower threshold and *uthresh* is the upper threshold).

Orders are only issued on even time steps because it takes 2 time steps for inventory to be updated as a result of the execution of a previous order. Consequently:

*invi,t = invi,t-1 if odd(t)* (19.1)

The above definitions assume that a dealer maintains inventory between two thresholds (upper and lower), as described in (Huang et al. 2012). We assume that the upper and lower threshold will vary dynamically, but we do not define them further. The quantities *dealersells* and *dealerbuys* are similarly taken from (Huang et al. 2012). The quantities *bidprice*, *bidsize*, *askprice* and *asksize* will vary according to inventory and perhaps also according to other factors (e.g. to avoid adverse selection risk), but for the moment are not defined further. The function *ψi( )* takes a set of executed orders, extracts the subset that were originally issued by dealer *i*, and then extracts and sums the executed sizes within that subset (i.e. it calculates the inventory impact – the number of assets bought or sold).

From inspection of dependencies in this model we can determine the following:

* A feedback dependency involving a single dealer: a dealer’s inventory at time t depends on its inventory at a previous time (since the latter determines the orders previously sent to the exchange, from which trades are executed, potentially causing inventory to change). Of course, dealers would not issue orders if they could not influence future inventory!
* A feedback dependency involving two dealers: for example in a 2-dealer system *inv1,t* depends on *xsellsizes1,t-1*, which depends on *xsellst-1*, which depends on *ob\_bidst-2*, which depends on *lo\_bidst-2*, which depends on *dealerbidst-2*, which depends on *inv2,t-2*, and in the same fashion *inv2,t-2* depends on *inv1,t-4*

These feedback dependencies are hard-wired into the static model. They (and other hard-wired dependencies) are easy to see in the recurrence relations, and although no value depends on the same value at the same time, these dependencies have the potential to become destabilising.

But is the system is actually stable or unstable? We perform a static analysis of the recurrence relations to gain some initial insight.

Analysing the values of the recurrence relations requires us to be more specific about certain values, for example the definitions of *bidsize* and *asksize*, and the starting inventories for the dealers. Arbitrarily, for the purposes of example, we will use the variables *UTi,t* and *LTi,t* to denote the upper and lower thresholds, *BPi,t* and *APi,t* will denote the bid and ask prices for dealer *i* – we assume that *max(BPi,t) < min(APi,t)* throughout so that limit orders never execute against each other (moreover, we require that market orders never execute against each other). The bid and ask sizes can be any value that would not, if executed in full, cause the inventory to exceed *UTi,t* or *LTi,t*. Thus the upperbound for *bidsize* is given by:

*bidsizei,t < UTi,t* *– invi,t* (19.2)

and the upperbound for *asksize* is given by

*asksizei,t < inventoryi,t – LTi,t*. *(19.3)*

The lowerbound for both *bidsize* and *asksize* is zero.

*Dealers issuing limit orders*

Let there be one or more dealers having inventories between the two thresholds (therefore issuing bid and ask limit orders). These dealers will not initially issue any buy or sell orders. Thus, we can ask “what is the maximum possible inventory for such a dealer?”. The inventory equation can be expanded as illustrated in Figure 19.9.

*MAX(invi,t )   
 = invi,t-1 +MAX(xbidsizesi,t-1 – xasksizesi,t-1), if even(t)  
 = invi,t-1 +MAX(xbidsizesi,t-1) (maximised if no asks are executed)  
 = invi,t-1 +MAX(ψi(xbidst-1))   
 = invi,t-1 +MAX(ψi(ob\_bidst-1* § *mo\_sellst-2))   
 = invi,t-1 +MAX(ψi(((ob\_empty* ⊕ *lo\_bidst-2)* ∅ *mo\_sellst-2)*§ *mo\_sellst-2))   
 = invi,t-1 +MAX(ψi(lo\_bidst-2)) (maximised if all bids are executed)  
 = invi,t-1 +MAX(ψi((∪i dealerbidsi,t-2), for i ∈ {1..n}))   
 = invi,t-1 +MAX(ψi(dealerbidsi,t-2)) (ψi only considers bids for this dealer)  
 = invi,t-1 +MAX(bidsizei,t-2) (ψI  returns the total size of the bids)  
 = invi,t-1 +( UTi,t* *– invi,t-2 – 1)*  *(from (19.2))*  
 *= UTi,t* *– 1 + (invi,t-1 - invi,t-2)  
 = UTi,t* *– 1 (even(t) implies invi,t-1 = invi,t-2)*

**Figure 19.9**

Thus, those dealers that start with an inventory below the upper threshold will never equal or exceed the upper threshold. By a similar expansion it is possible to show neither will they ever equal or exceed the lower threshold.

*Dealers issuing market orders*

Let there be one or more dealers having initial inventories exceeding a threshold. These dealers will not initially issue any bid or ask orders. Thus, we can ask “what is the maximum possible inventory for a dealer whose initial inventory is below the lower threshold?”. The inventory equation can be utilised and expanded as illustrated above, with the result that any dealer whose inventory starts below the lower threshold can raise its inventory at most to the value of the lower threshold (at which point it issues no orders at all). A similar finding holds (mirrored) for those dealers whose inventories start above the upper threshold. Furthermore, any dealer whose inventory starts below the lower threshold only issues buy orders and can never decrease inventory further (unless the lower threshold itself changes): similarly, those with starting inventories above the upper threshold can never increase their inventory (unless the upper threshold changes).

**19.5.3 Dynamic analysis using InterDyne**

Static analysis of a recurrence relation model of the case study indicates that despite the existence of feedback dependencies the market is stable: if all dealers start with inventories between the thresholds they will never exceed either threshold and therefore Hot Potato trading will not occur. Furthermore, if there were some way for a dealer’s inventory to exceed a threshold its inventory would monotonically move towards the nearest threshold as fast as possible, so Hot Potato Instability will not occur.

We will now illustrate the importance of dynamic analysis by exploring ways in which an apparently stable (and self-stabilising) model can nevertheless be induced to exhibit Hot Potato Instability.

It is straightforward to recast the previous recurrence relation model as an InterDyne simulation:

* One InterDyne agent will represent the exchange subsystem.
* *n* InterDyne agents will represent the *n* dealer subsystems.

To explore the progression of this System of Systems as it moves from stable behaviour into unstable behaviour (we will explain below how this is possible), we can also add one or more trading agents to provide market supply and demand. Starting inventories can be passed to the dealer subsystems as runtime arguments.

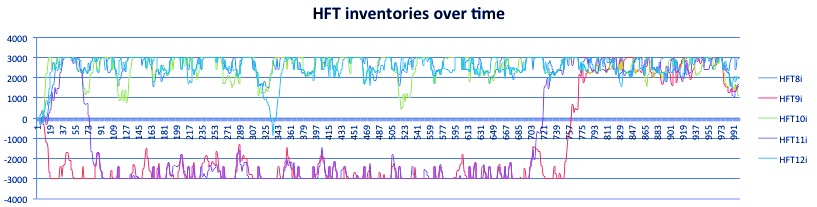
Figure 19.10 provides example code for a System of Systems that comprises seven subsystems: an exchange, five dealers, a noise trader (that randomly issues buys and sells), three seller traders (issuing only sell orders) and three buyer traders (issuing only buy orders). The runtime argument “inventory” provides the same starting inventory for all the dealers (in this example, 0). No subsystem subscribes to a broadcast channel and therefore any broadcast messages will not be received by any subsystem.

newExperiment   
= do  
  sim 1000 args agents  
  where  
  args = [ (Arg (Str “inventory”, 0)) ]  
 agents = (replicate 1 (exchangeWrapper, [])) ++  
 (replicate 5 (dealerWrapper, [])) ++  
 (replicate 1 (noiseTraderWrapper, [])) ++  
 (replicate 3 (sellerWrapper, [])) ++  
 (replicate 3 (buyerWrapper, []))

**Figure 19.10. A financial market System of Systems comprising an exchange, five dealers, a “noise trader”, three sellers and three buyers.[[16]](#footnote-16)**

The exchange subsystem will maintain *ob\_bids* and *ob\_asks* as internal state items, receive *lo\_bids*, *lo\_asks*, *mo\_buys* and *mo\_sells* as inbound messages, and output *xbids*, *xasks*, *xbuys* and *xsells* as messages. Each dealer subsystem will maintain *invi,t*, *lthresh* and *uthresh* internal state items, receive *xbids*, *xasks*, *xbuys* and *xsells* as inbound messages from the exchange, and output *lo\_bids*, *lo\_asks*, *mo\_buys* and *mo\_sells* messages to the exchange. The function *ψi( )* is partially implemented by the exchange because for example it only sends *xbids* and *xsells* messages to the two dealers involved in each trade.

In our previous static analysis the complexity of the dealer code was hidden – for example, the analysis simply referred to the calculated upper and lower constraints for limit order sizes. In the dynamic analysis we have the opportunity be more specific, for example to provide an accurate implementation of the Huang et al algorithm previously discussed. However, initially we are interested only in whether it is possible to induce a stable algorithm to become unstable and so we implement code that always issues the maximum sized orders according to the constraints used in the static analysis this may be considered a “stress test”). Furthermore, to reduce complexity, we initially set static inventory thresholds. Figure 19.11 (from (Court 2013) with permission) is a plot of the five dealer inventories in a simulation with inventory thresholds 3,000 and -3,000, and illustrates (as predicted by the static analysis) that the dealer inventories never exceed either threshold. The inventories tend to stay close to the thresholds due to the extreme “stress test” size calculations for limit orders.



**Figure 19.11 Dealer inventories**

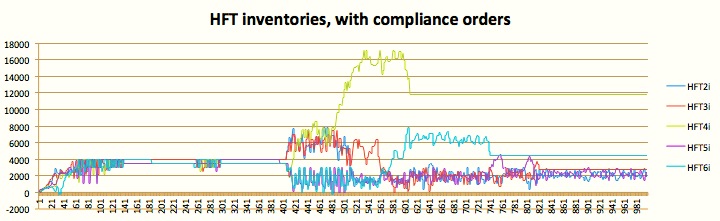
To explore how such a system might be induced to become unstable, we systematically change elements of the experiment, one at a time, and visualise time-varying data such as inventories, order sizes, message sources and destinations, and sequences of messages. There are two aspects to this investigation: (i) how to cause a dealer to exceed its inventory threshold, and (ii) once that has been accomplished, how to cause two or more dealers to be recruited into a behaviour of repeated inter-dealer trading.

**19.5.3.1 Exceeding the Inventory Thresholds**

For an inventory threshold to be exceeded, either the inventory must be forced to cross the threshold, or the threshold must be forced to cross the inventory. The former corresponds to loss of inventory control, and the latter corresponds to a dynamically recalculated threshold.

Huang et al (Huang et al. 2012) discuss inventory control in the context of a currency exchange market and derive an optimal inventory-control strategy using thresholds that assumes thresholds will be temporarily exceeded because a currency dealer has little control over market supply and demand and must execute inbound market orders. Dealers in an order-book market have more control (as previously discussed) but if a dealer prioritises a “market-maker rebate” (see Section 19.5) over inventory control then a bid and an ask will always be submitted to the exchange even when execution of such limit orders would cause a threshold to be exceeded. Furthermore we note that Huang et al’s thresholds are varied dynamically and either threshold might be reduced at any time: if the inventory were very close to a threshold before that threshold was reduced, then at the next time step the inventory might exceed the threshold even though the inventory did not change.[[17]](#footnote-17)

The dealer code for the InterDyne simulation can be modified so that dealers always issue bids and asks of given size regardless of inventory. Figure 19.12 (from (Court 2013) with permission) shows that in this scenario all dealer inventories will exceed the upper threshold.



**Figure 19.12 Inventories for five dealers with inventory thresholds of ± 3000 and issuing bid and ask orders with minimum sizes of 1000, together with a trader that issues sell orders of size 500 for each of the first 500 timesteps.**

Further exploration shows that if there is both supply and demand, even if net supply is zero, then some dealers can exceed the upper threshold while others exceed the lower threshold. It is straightforward to vary the thresholds during a simulation and explore the circumstances under which this can cause inventory to exceed a threshold, though we do not present the details here.

**19.5.3.2 Inducing Hot Potato Instability**

Although we have identified two ways in which a dealer’s inventory might exceed an inventory threshold, understanding Hot Potato Instability is more problematic.

* If one dealer were to exceed an inventory threshold, it would issue a market order and that market order would be executed against one or more limit orders placed by the other dealers. If the market order were very large, it would be executed against many smaller limit orders, and the excess inventory would be distributed across the other dealers, thereby reducing the likelihood that another dealer would, as a result, be caused to exceed its inventory threshold.
* If the first dealer’s excess inventory were offloaded, and if no other dealer were caused to exceed a threshold, then there would be no further market orders and no Hot Potato trading.
* Alternatively, if *all* dealers were somehow induced to exceed a threshold, they would all issue market orders. In the absence of any market participants issuing limit orders there would be no trades executed and no Hot Potato trading. However, if the dealers were issuing fixed-size limit orders even when inventory exceeds the thresholds, then Hot Potato behaviour could occur.
* To achieve Hot Potato trading it is necessary to match a market order from one dealer with a limit order from another dealer, i.e. it is necessary for a group of dealers either (i) to continue issuing limit orders when inventory exceeds a threshold (so that trading continues even when *all* dealers exceed a threshold), or (ii) to oscillate between issuing limit orders and issuing market orders, and to do so in synchronised anti-correlation (so that some dealers always issue limit orders at the times that other dealers issue market orders).

Closer inspection of Figure 19.12 provides evidence of both Hot Potato trading and Hot Potato Instability. In the first 50 time steps the inventories of all five dealers increase (non-monotonically), after which point the inventories start to oscillate above and below the upper threshold (set to 3,000). From approximately time steps 140 to 260 the inventories are stationary: all exceed the upper threshold and either no trading is done or in every time step the sizes bought equal the sizes sold (amounts bought will be due to the fixed-size bids continuing to be issued). Following a brief session of trading and oscillating inventories (roughly time steps 260 to 300) the inventories are once again stationary until about time step 400, at which point the inventories start to oscillate with large amplitude (two dealers’ inventories drop to below the threshold, while three continue to have excessive inventory). At time step 500 the trader stops issuing sell orders, and from time step 500 to the end of the simulation the dealers are only trading with themselves.

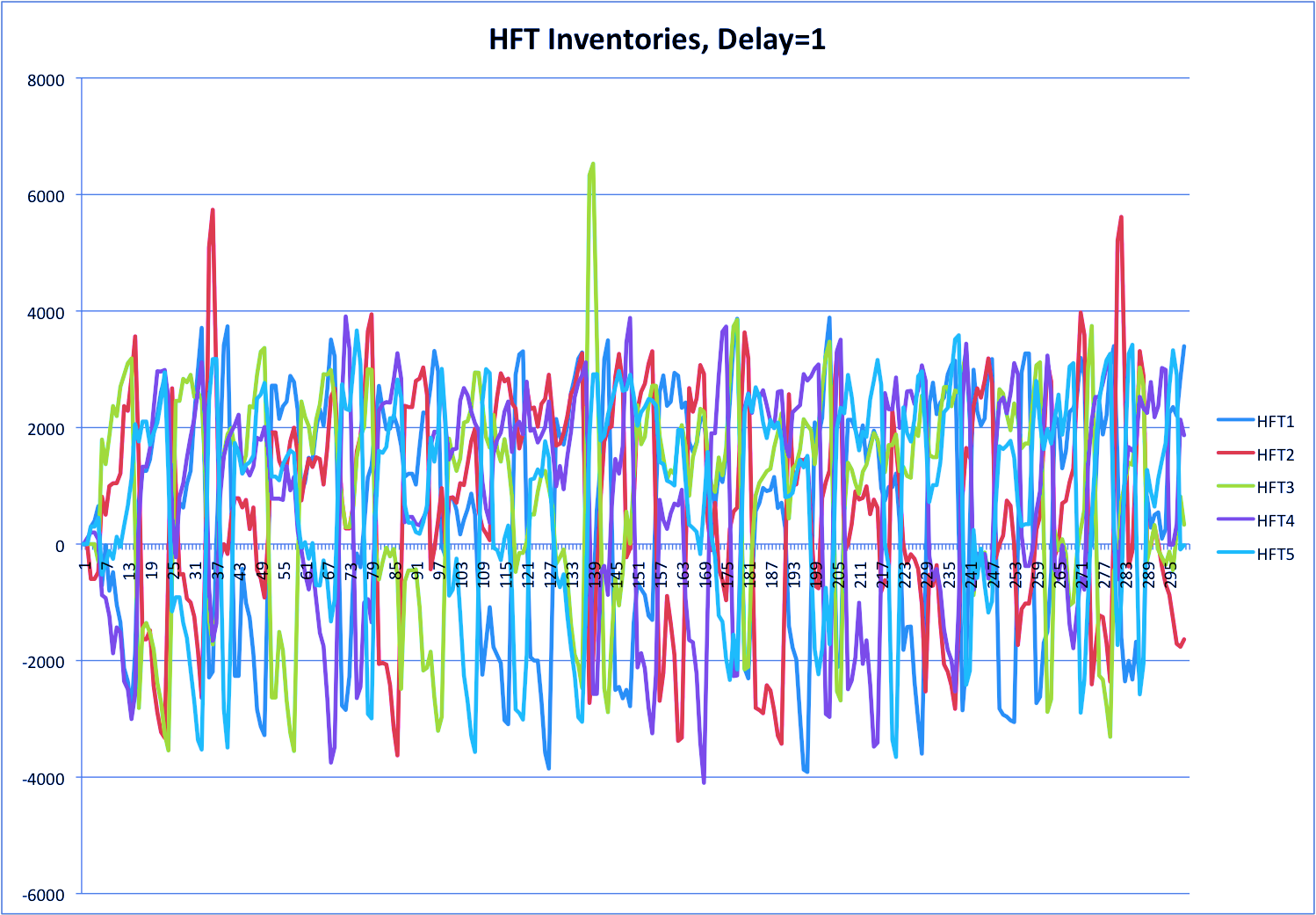
This corresponds to our definition of Hot Potato Instability, and dynamic analysis reveals a complex pattern of behaviour that could be further explored with a micro-analysis of the message history.

**19.5.3.2 Hot Potato Instability triggered by information delay**

An important element in the 2010 Flash Crash was the presence of substantial information delay, due to the technology infrastructure becoming overwhelmed with data (CFTC and SEC 2010, Kirilenko et al. 2014, Nanex 2010a, Nanex 2010b). As part of our hypothesis formulation, we explore information delay to see whether it might play a role in triggering Hot Potato trading.

In real life such communication delays are typically unknown for some initial period of time. During this period, dealers and other traders will continue to operate on the assumption that incoming data is timely (i.e. that it has not been delayed). In InterDyne we simply revert to the first experiment (Section 19.5.3) and set an extra delay of one time step (Section 19.4.4) for all messages from the exchange to the dealers.

Figure 19.13 (from (Court 2013) with permission) illustrates how such Hot Potato Instability (induced by a communication delay of just 1 time step) can persist. The figure plots inventories for five homogeneous dealers over 300 time steps. The instability is triggered by a seller trader that issues sell orders of size 100 for each of the first 40 time steps.



**Figure 19.13**



**Figure 19.14: Inventory changes for a market with five homogeneous dealers. The shaded zone is a stable zone within the inventory thresholds.**

Figure 19.14 shows the dynamic inventories of five homogeneous dealers when a market exhibits an information delay of one additional time step from exchange to dealer. To determine whether this is due to message selection bias the facility for randomising message arrival times has been activated (the effect persists regardless of order arrival time). For this example, two of the dealers have an initial inventory that exceeds the upper threshold, two start with inventories below the lower threshold, and the last starts with an initial inventory of zero. Inventories for the first 100 time steps are shown.

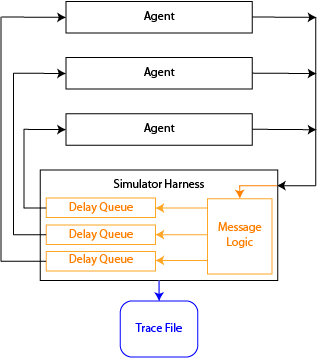
In roughly the first 25 steps all dealers trade among themselves causing periodic jumps to excessive inventory and back (due to information delay). These jumps are undesired because the market orders typically incur financial loss (a dealer loses the spread on each market order trade); the dealers therefore try to avoid those jumps by restraining their limit orders when their inventories approach the upper and lower thresholds (UL and LL). In the remaining 75 steps three out of five dealers manage to stabilize their inventories near the upper threshold. At an inventory of exactly UL-1 they do not issue any bids, and if there are no delayed executions in the pipeline they cannot thereafter exceed the upper threshold. However, the other two dealers remain coupled in a feedback loop and continue to trade with each other. This leads to an infinite oscillation, where the two dealers repeatedly exchange the same inventory. This could create a false impression of continuous market liquidity.

**19.5.3.3 Understanding Hot Potato Instability**

Figures 19.12, 19.13 and 19.14 illustrate the complexity of Hot Potato Instability, with a broad range of dynamic behaviour. At first sight the behaviour may appear chaotic, yet exploration of different initial conditions (such as initial inventories) and other factors such as the length of delays and details of the order size equations has helped us to identify several patterns of behaviour.

We aim to understand this emergent behaviour sufficiently to suggest proposals for reengineering the financial markets infrastructure: to avoid cases of undesirable oscillations where that is possible, and to detect other cases so that remedial intervention can be applied.

Dynamic analysis of the interaction dynamics is essential, and an essential component of that is the ability to observe message-passing histories. With InterDyne it is possible to trace full information about (i) all messages sent by an agent; (ii) all messages received by an agent; and (iii) all messages held in a delay queue awaiting delivery to an agent. The latter is possible because of the way that InterDyne implements delays – the simulator harness receives all messages output by agents, determines their destinations, and constructs queues of messages awaiting delivery to each agent. Although the user-defined delay function (Section 19.4.4) provides delay information for every possible combination of sender and receiver of messages, the simulator harness only needs to maintain *n* delay queues for *n* agents, as illustrated in Figure 19.15.



**Figure 19.15**

As an example of how observation of messages can be useful, consider a simple 2-dealer system: we might wish to investigate the effects of changes in the size of sell orders that a dealer issues when inventory exceeds the upper threshold. One option discussed previously is to sell sufficient inventory to revert inventory (in the best case) to the upper threshold. Another option is to always sell an amount of UT, where UT is the value of the upper threshold: if the inventory were only just above the threshold, this would in the best case revert inventory to be very close to zero (a neutral position if the thresholds are symmetrically arranged above and below zero). By investigating the time history of message-passing given in the InterDyne trace file we can see which policy might be more stable in the presence of communication delays.

As a concrete example, assume a delay of one time step in trade confirmations reaching the dealers, a static upper threshold of 1,000 for all dealers, and a two-dealer system with starting inventories -998 and 1998. Assume that bids are issued at the “stress test” size of (999-invt) and compare the effect of a sell size of (i) invt – 1,000; versus (ii) 1,000. With a sell size of (*invt – 1,000)* the traced message histories are illustrated in Table 19.1: at each time step the inventories of the two dealers are recorded (following receipt of trade confirmations in that time step), the next two columns give the sizes of any orders issued in that time step (here we only consider bids and sells: a positive size is a bid, a negative size is a sell), the fifth column indicates the size of a sent trade confirmation message, the sixth column shows a message in a delay queue (in fact there are two confirmation messages – one for each dealer), and the final column indicates that a trade confirmation has been received (by both dealers) in that time step. This example leads to stable inventories (i.e. not exceeding the upper threshold) at time step 5.

|  |  |
| --- | --- |
|  |  |

Table 19.1: Tracing message histories for sell size (*invt – 1,000)*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Time | **Inv1** | **Inv2** | **Message Order1** | **Message Order2** | **Executed** | **Messages Delayed** | **Messages Received** |
| 0 | -998 | 1998 | 1997 | -998 |  |  |  |
| 1 | -998 | 1998 |  |  | 998 |  |  |
| 2 | -998 | 1998 | 1997 | -998 |  | 998 |  |
| 3 | 0 | 1000 |  |  | 998 |  | 998 |
| 4 | 0 | 1000 | 999 | 0 |  | 998 |  |
| 5 | 998 | 2 |  |  |  |  | 998 |
| 6 | 998 | 2 |  |  |  |  |  |

By contrast consider Table 19.2 where sell sizes are always 1,000. In this case the 2-dealer System of Systems enters a loop were the system state repeats every 12 time steps. If a system state is repeated when InterDyne is operating in deterministic mode then such a loop will continue forever (i.e. until the end of the simulation).

|  |  |
| --- | --- |
|  |  |

Table 19.2: Tracing message histories for sell size (*1,000)*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Time | **Inv1** | **Inv2** | **Message Order1** | **Message Order2** | **Executed** | **Messages Delayed** | **Messages Received** |
| 0 | -998 | 1998 | 1997 | -1000 |  |  |  |
| 1 | -998 | 1998 |  |  | 1000 |  |  |
| 2 | -998 | 1998 | 1997 | -1000 |  | 1000 |  |
| 3 | 2 | 998 |  |  | 1000 |  | 1000 |
| 4 | 2 | 998 |  |  |  | 1000 |  |
| 5 | 1002 | -2 |  |  |  |  | 1000 |
| 6 | 1002 | -2 | -1000 | 1001 |  |  |  |
| 7 | 1002 | -2 |  |  | 1000 |  |  |
| 8 | 1002 | -2 | -1000 | 1001 |  | 1000 |  |
| 9 | 2 | 998 |  |  | 1000 |  | 1000 |
| 10 | 2 | 998 | 997 | 1 |  | 1000 |  |
| 11 | -998 | 1998 |  |  |  |  | 1000 |
| 12 | -998 | 1998 | 1997 | -1000 |  |  |  |

**19.5 Chapter Summary**

In this chapter we have introduced the InterDyne simulator and our simulation method for exploring Systems of Systems and identifying the antecedents of undesirable emergent behaviour that derives from interaction dynamics such as coupling and feedback loops. InterDyne is an agent-based simulator that differs from other agent-based technology in (i) its focus on supporting experimental observation of message-passing interaction, including broadcast messages, a broad range of message types, and a very flexible treatment of agent-to-agent communication delays; and (ii) our “two-view” approach where the semantics of the agent-based model are paired with a recurrence-relation model (the latter supports static analysis, whereas the former supports dynamic analysis). A current research topic is to establish a formal equivalence of the two semantics.

InterDyne supports investigation of time-varying interactions between subsystems, to explore substantially more complex Systems of Systems that may not be amenable to static analysis, and to encompass a greater range of real-life behaviour. InterDyne also supports deterministic and non-deterministic experiments, with randomised message arrival at each agent (to demonstrate that emergent behaviour is not an artefact of hard-wired bias in arrival times) and heterogeneous agent strategies (to demonstrate that emergent behaviour is not simply due to a resonance between homogeneous algorithms). With InterDyne (as with other simulators for Systems of Systems) we can explore questions such as “how is a given behaviour triggered?”, “under what circumstances might a repeated behaviour continue forever?”, “under what circumstances might it naturally resolve?”. Although the antecedents of emergent behaviour need not be unique, and may not be applicable to other configurations of a given System of Systems, they often serve to falsify a negative hypothesis in stress testing (e.g. “it is not the case that this can never happen”). Moreover, we view dynamic analysis as an important component of hypothesis formulation, to assist in clarifying hypotheses and their consequences for dynamic behaviour, and to assist in communication between domain experts.

Many of the project tools, including the InterDyne simulator and visualisation tools, are part of on-going research. The InterDyne Project team at UCL has included staff and many students, and we gratefully acknowledge their contributions. Members of the project team who have worked on various aspects of the core technology have included Vikram Bakshi, Aman Chopra, Elias Court, Richard Everett, Kyle Liu, and Dmitrijs Zaparanuks.

**Bibliography**

Barkhausen, H. 1935. *Lehrbuch der Elektronen-Röhren und ihrer technischen Anwendungen: Rückkopplung*, Hirzel.

Box, G. E. P. 1976. Science and Statistics. *Journal of the American Statistical Association* 71: 791–799.

CFTC and SEC. 2010.Findings regarding the market events of May 6, 2010.

Chan, W.K.V. and Son, Y.-J. and Macal, C.M. 2010. Agent-Based Simulation Tutorial - Simulation Of Emergent Behavior And Differences Between Agent-Based Simulation And Discrete-Event Simulation. In *Proceedings of the 2010 Winter Simulation Conference*, 135–150.

Clack, C.D. and Peyton Jones, S.L. 1985a. Generating Parallelism from Strictness Analysis. In Proceedings *Workshop on Implementation of Functional Languages,  
Programming Methodology Group.* Report 17, 92-131. Chalmers University, Sweden.

Clack C.D. and Peyton Jones, S.L. 1985b. Strictness analysis — a practical approach, *Lecture Notes in Computer Science* 201:35-49.

Court, E. 2013. The Instability of Market-Making Algorithms. MEng thesis. Department of Computer Science, University College London.

Day, R.H. and Huang, W. 1990. Bulls, bears and market sheep. *Journal of Economic Behavior & Organization* 14(3):299-329.

Easley, D. and López de Prado, M.M. and O’Hara, M. 2011. The microstructure of the “flash crash”: flow toxicity, liquidity crashes, and the probability of informed trading. *The Journal of Portfolio Management* 37.2:118-128.

Faustini A.A. 1982. An operational semantics for pure dataflow. *Lecture Notes in Computer Science* 140.

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

Horowitz, I. 1959. Fundamental theory of automatic linear feedback control systems. *IRE Transactions on Automatic Control* 4.3:5-19.

Huang, K. and Simchi-Levi, D. and Song, M. 2012. Optimal Market-Making with Risk Aversion. *Operations Research* 60(3):541–565.

Hudak, P. and Peyton Jones, S.L. and Wadler, P. et al. 1992. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. SIGPLAN Not. 27(5):1-164.

Kirilenko, A. and Kyle, A.S. and Samadi, M. and Tuzun, T. 2014. [The Flash Crash: The Impact of High Frequency Trading on an Electronic Market](http://www.cftc.gov/ucm/groups/public/@economicanalysis/documents/file/oce_flashcrash0314.pdf), Working Paper, SSRN. http://ssrn.com/abstract=1686004 (accessed April 30, 2017).

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

Lyons, R.K. 1997. A simultaneous trade model of the foreign exchange hot potato. *Journal of International Economics* 42(3-4):275–298.

Milner, R. 1999. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press.

Nanex 2010a. Nanex Flash Crash Analysis Final Conclusion. Working paper. Nanex LLC. <http://www.nanex.net/FlashCrashFinal/FlashCrashAnalysis_Theory.html> (accessed April 30, 2017).

Nanex 2010b. Nanex Flash Crash Summary Report. Working paper, Nanex LLC. <http://www.nanex.net/FlashCrashFinal/FlashCrashSummary.html> and <http://www.nanex.net/FlashCrashFinal/FlashCrashSummary_DJIADelay.html> (accessed April 30, 20170).

Newton, I. 1687. *Philosophiæ Naturalis Principia Mathematica*.

Rosu, I. 2009. A Dynamic Model of the Limit Order Book. *Review of Financial Studies* 22(11):4601–4641.

Rötzer, G. 2012. The Stochastic Pi Machine and Finance. MSc thesis, Department of Computer Science, University College London.

Skogestad, S. and Postlethwaite, I. 2007. *Multivariable feedback control: analysis and design. Vol. 2.* Wiley.

Sundman, K. 1912. Memoire sur le probleme des trois corps. *Acta Mathematica* 36:105–179.

Turing, A. 1937. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2* 42:230–265.

Turner, D.A. 1985. Functional programs as executable specifications. In Proceedings of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages, ed. C A.R. Hoare and J. C. Shepherdson. Prentice-Hall, Inc.

Wilberforce, L.R. 1896. On the vibrations of a loaded spiral spring. *Philosophical Magazine*. 38:386–392.

Zames, G. 1966. On the input-output stability of time-varying nonlinear feedback systems part one: Conditions derived using concepts of loop gain, conicity, and positivity. *IEEE transactions on automatic control* 11.2:228-238.

1. It can be difficult to derive this information from a static inspection of the specification and initial conditions. Indeed, if our models are viewed as executable specifications some properties are undecideable (Turing 1937), yet observations of the dynamics of behaviour during simulation can provide valuable insight. [↑](#footnote-ref-1)
2. Such loops may be intended or unintended. [↑](#footnote-ref-2)
3. We recall Box’s dictum that “All models are wrong; some models are useful.” (Box 1976). [↑](#footnote-ref-3)
4. Recurrence relations (and difference equations) are widely used in economics and finance – for example, see (Day and Huang 1990, Huang et al. 2012, Rosu 2009). [↑](#footnote-ref-4)
5. It is similar to the problem of determining for a function *g(x,y)* whether evaluation of that function will necessarily require evaluation of its arguments *x* and *y* (Clack and Peyton Jones 1985a, Clack and Peyton Jones 1985b). [↑](#footnote-ref-5)
6. There will also be a need to map names of entities in the agent-based model to names used in the set of recurrence relations, and vice versa, so that results in one view can easily be understood in terms of the other view. [↑](#footnote-ref-6)
7. Faustini (Faustini 1982) asserts that this term is due to W.Wadge and E.Ashcroft. [↑](#footnote-ref-7)
8. The current version of InterDyne (and therefore the agent code) is written in the programming language Haskell (Hudak et al. 1992). This aligns with our view of modelling a System of Systems as an executable specification (Turner 1985). [↑](#footnote-ref-8)
9. Messages (see Section 19.4.3) can be sent using agent identifiers or (more usefully) agent names.  To achieve the latter, the user includes in the list of runtime arguments a function that when executed will convert an agent identifier into a name or vice-versa.  Runtime arguments are available to all agents, and if desired the user can change the mapping each time sim is called. [↑](#footnote-ref-9)
10. In the second example, the Haskell function “map” takes two arguments, a function and a list, and applies the function to every item in the list. The function “snd” when applied to a structure such as “(“Trader”, (traderWrapper,[1]))” returns the second component, e.g. “(traderWrapper, [1])”. [↑](#footnote-ref-10)
11. Optionally an error message can also be sent to the trace file without aborting. [↑](#footnote-ref-11)
12. InterDyne is not fully hierarchical – an agent cannot itself be another InterDyne simulation. [↑](#footnote-ref-12)
13. The *n-dealer* problem for Hot Potato trading in an order-book market is important since this phenomenon was observed at the heart of the Flash Crash (CFTC and SEC 2010). [↑](#footnote-ref-13)
14. This System of Systems is highly simplified. In reality there would for example be a very large number of different kinds of traders and market makers, and there would be multiple exchanges, trading equities for multiple companies. Furthermore, we assume that the market makers implement identical policies, though in reality they would differ. [↑](#footnote-ref-14)
15. Although this may come at a cost: several exchanges offer a rebate to dealers who agree always to have a bid and an ask submitted to the order book, and if no bid or ask were received and exchange might remove the rebate. [↑](#footnote-ref-15)
16. The Haskell expression (replicate n x) creates a list of n items each of which is x, and the Haskell operator “++” concatenates lists. [↑](#footnote-ref-16)
17. Note that if dealers issue limit orders that might cause the inventory threshold to be exceeded then this departs from the System of Systems specification used in the static analysis and we should not expect stability. By contrast, that specification does accommodate varying thresholds – however we seem to have identified a problem with the calculation of “safe” limit order sizes, which should reference the thresholds of the next time step (though in general these cannot be known). [↑](#footnote-ref-17)