InterDyne User Manual

DRAFT - version 1.3

Christopher D. Clack[*]

Department of Computer Science, University College London

14 October 2016

Abstract

InterDyne is an agent-based simulator written in the programming language Haskell. The name "InterDyne" derives from the topic of Interaction Dynamics, and InterDyne aims to provide support for exploration of Interaction Dynamics in complex systems.

Although InterDyne has been used mostly to explore models of financial markets, it is a general-purpose tool that can be used to investigate Interaction Dynamics in a variety of fields.

This User Manual provides a basic intoduction to InterDyne and how it is used. More advanced features of InterDyne are described in other reports.

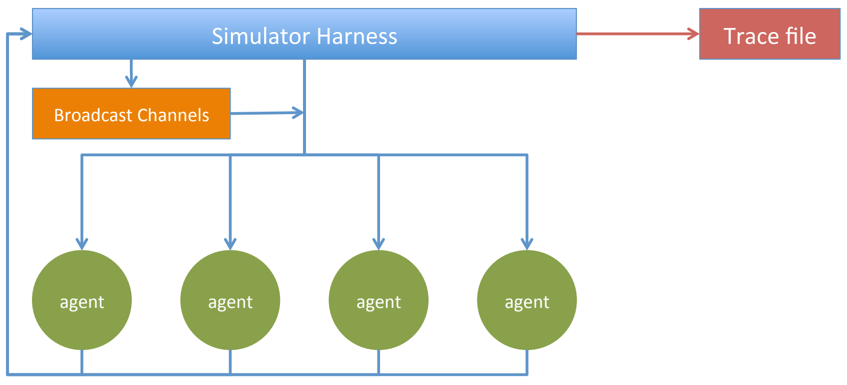
[*]Email: `clack@cs.ucl.ac.uk`

Contents

1 Basic InterDyne Structure

An InterDyne simulation comprises one or more agents that pass messages to each other via a "Simulator Harness". The designer of an InterDyne simulation can control the topology of permissible messaging routes (this will be explained later), and as a result a subsystem can be represented as a group of agents with private communication and a lead agent that communicates with other agents that are not part of the subsystem. Many such subsystems can exist in an InterDyne simulation.

Figure 1 illustrates a very basic InterDyne configuration. The primary output from InterDyne is a trace file; this can be used for post-hoc analysis of messages (including their timings and content) and any other information that a designer wishes to record. A post-hoc analysis of the output data can be incorporated into a spreadsheet, from which tables and graphs can be used to visualise the behaviour of the simulation, and especially to explore the antecedents of emergent behaviour.



2 Running a Simulation

InterDyne is run by executing a function called “sim”, which must be passed the following arguments:

- The number of time steps for the simulation.
- A list of runtime arguments: (key, value) pairs that are passed to every agent in the system.
- A list of information about each agent: a 2-tuple containing the name of the function that implements the agent, and a list of broadcast channel identifiers (see later) to which the agent will listen.

Each agent has an “agent ID” that is derived from its position in the list of agent information (the third argument to “sim”). For example, the first agent in the list has agent ID 1. Agent ID 0 is reserved for the “simulator harness” function that controls messaging between agents, controls the passage of time during simulation, and sends output to the trace file. The agent IDs are used to specify the source and destination of all one-to-one messages between agents.

Agents are pure functions that consume a (potentially infinite) list of message input packets and produce a (potentially infinite) list of message output packets. Any input or output should be intermediated by the simulator harness. At each time step an agent both consumes one input packet and generates one output packet.

Input and output packets differ:

- An 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.
- An output packet is simpler tha an input packet — it just contains zero or more messages.

As shall be explained later, messages have a rich variety of types (since InterDyne’s purpose is to focus on messaging as the mechanism of interaction).

An agent only receives a direct message if its agent ID is in the recipient field of that message. Agents do not see messages sent to other agents. Furthermore, an agent only receives a broadcast messages if it is sent to a broadcast channels to which this agent has subscribed. Broadcast messages support sending the same message from one sending agent to a large number of receiving agents; the recipients can be defined and changed in the set-up of the experiment rather than in the agent code.

3 Time

InterDyne is a discrete-time simulator. Experiments are executed for a given number of time steps (the first argument passed to the “sim” function). The period of real time that corresponds to each time step is not specified; this is a matter for the experimenter, according to the requirements of the model and its simulation. For some experiments (e.g. for a behaviour that is known or assumed to be rate-independent) it may not be necessary to link a time step to a precise period of real time.

Only integer multiple of a time step can be expressed in the simulation. A time step is typically set to be the smallest required resolution for the experiment.

4 Agents

Each InterDyne agent can be modelled in a different style and with a different level of detail — all that matters is that it complies with the required type for an agent function and that at each time step it always consumes exactly one input packet and outputs exactly one output packet. The input packets and output packets are list items.

An agent is not required to use any input data (it can be discarded), but the input packet must be read. At each time step an agent must only read the single input packet that is at the start of its list of input packets — it must not attempt to read the next input packet (because that would be interpreted as an attempt to look forward in time, which is not permitted). If the input packet is empty, this means that in the previous time step no other agent sent a message to this agent. Alternatively, an input packet might contain just one message, or many messages.

If an agent does not have any message to output in a time step then it must output an empty output packet. If an agent needs to output several messages in one time step then those messages must be incorporated into a single output 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 can achieve this by generating an output packet containing a special empty message called a "Hiaton". [1]

An InterDyne simulation may comprise many agents defined at different levels of detail. For example, one agent might generate outputs dependent on a statistical distribution, whereas a different agent might be a detailed implementation of a complex algorithm with considerable internal complexity.

Although very simple agents do not need to do this, most InterDyne agents are written as two functions:

1. A "wrapper" function that manages the reading of inbound messages, the generating of outbound messages, and the update of local state, and

2. A "logic" function that is called by the wrapper function and which calculates the messages to be sent.

The "wrapper" function is the primary function for this agent, and it is the function whose name is provided to "sim" in the list of agent information. The "logic" function is merely a subsidiary function and normally has restricted scope so that it can only be called by the wrapper function.

[1] The name Hiaton is widely believed to be due to Wadge and Ashcroft.

The following code example illustrates an agent wrapper function called `agent1` that performs no actions: it makes no use of the input packet and creates an empty output packet. It does not use a logic function.

```
agent1 state args (packet:rest) myid = []:(agent1 state args rest myid)
```

The agent wrapper function `agent1` is recursively defined and loops once per time step. To the left of the equals sign is the name of the function and formal parameter names for its arguments, whereas to the right of the equals sign is the output list of packets. The output is a list whose items are themselves lists. Since the function produces no output messages the first item in its output list is the empty list `[]`.[2] The remainder of the output items are given by the output from the recursive call to `agent1`. The arguments to `agent1` (to the left of the equals sign) are:

- `state`. This provides local state for the wrapper function, but in this example it is not used and is passed to the recursive call unchanged.
- `args`. This is the list of runtime arguments for the simulator as a whole (these are passed to every agent function).
- `(packet:rest)`. This is not a single argument name but a Haskell pattern: the argument as a whole is a list, and the pattern binds the formal parameter name `packet` to the first item in the list, and the name `rest` to the remainder of the list without the first item. This is the list of input packets. Only the name `rest` is passed to the recursive call of `agent1`, and therefore at the next time step (the next loop of the recursion) the function `agent1` will read in the next input packet in this list.
- `myid`. This is the agent ID for this agent. It must always be passed on unchanged to the recursive call.

If an agent function wishes to inspect the contents of the input packet, the pattern `(packet:rest)` can be extended so that it gives names to the individual components of the input packet. An input packet is always a 3-tuple comprising time, a list of messages, and a list of broadcast messages. Hence the code might be written:

```
agent1 state args ((t, msgs, bcasts):rest) myid = ...
```

In the above example `t` is the current time step, `msgs` is alist of all one-to-one messages sent to this agent to be received in this time step, and `bcasts` is a list of all broadcast messages available at this time step on all the broadcast channels to which this agent is subscribed.

[2]If the agent wished to output messages, the empty list `[]` would be replaced by a list of messages that could include one-to-one messages and broadcast messages.

4.1 Passing Agent Information to `sim`

The following example shows the most basic way in which agent information is passed to the simulator harness function `sim`:

```
experiment1  =  do
                sim 100 [] agents
                where
                agents = [(agent1, []), (agent2, [])]
```

In the above example, the first argument to `sim` is `100`, indicating that the simulation should run fror 100 time steps; the second argument is the empty list `[]`, indacting that their are no runtime arguments to pass to the agents; and the final argument is the list of agent information. There are two agents whose wrapper functions are `agent1` and `agent2`, neither of which subscribes to any broadcast channels. Thus, the list of agent information `agents` contains two 2-tuples (one for each agent) and in each of those 2-tuples the second element is the empty list `[]`.

The next example (below) also runs for 100 time steps and contains a single runtime argument `convert`. This argument provides each agent with a bi-directional mapping between agent names and agent IDs: it does this by applying the InterDyne library function `generateAgentBimapArg` to a list of extended agent information. The extended information is given by `agentinfo`, where we have provided example agent names "Trader" and "Exchange".

```
experiment2  =  do
                sim 100       [convert] agents
                where
                agents     =  [(agent1,[]), (agent2,[])]
                convert    =  generateAgentBimapArg agentinfo
                agentinfo  =  [("Trader", (agent1, [])),
                               ("Exchange",(agent2, []))]
```

Because the list of agent informaton used by `generateAgentBimapArg` must be in the same order as the list `agents` provided to the function `sim`, a common style generates `agents` from `agentinfo` as shown below:

```
experiment3  =  do
                sim 100       [convert] agents
                where
                agents     =  map snd agentinfo
                convert    =  generateAgentBimapArg agentinfo
                agentinfo  =  [("Trader", (agent1, [])),
                               ("Exchange",(agent2, []))]
```

If the agents wish to subscribe to listen to broadcast channels, the numeric ID of each broadcast channel is contained in a list associated with the agent. This is illustrated in the following example where agent1 subscribes to channel 1 and agent2 subscribes to channels 1 and 3:

```
experiment4  =  do
                sim 100       [convert] agents
                where
                agents     =  map snd agentinfo
                convert    =  generateAgentBimapArg agentinfo
                agentinfo  =  [("Trader", (agent1, [1])),
                               ("Exchange",(agent2, [1,3]))]
```

5 The Trace File

InterDyne stores output in a trace file. The experiment designer can pass a special runtime argument to tell InterDyne in which folder this file should be stored.

The runtime argument `(Arg (String foldername, 9989793425))` uses the "magic number" 9989793425 and causes the string `foldername` to be used as the name of the directory in which to store the trace file. It also ensures that the trace file uses the name of the directory. For example, given the runtime argument `(Arg (String "testoutput", 9989793425))` the trace file would be saved as `./testoutput/testoutput-trace`.

This somewhat idiosyncratic mechanism may be changed in the future.

6 Interaction and Messages

The only available form of interaction between InterDyne agents is the passing of messages, and the simulator harness acts as an intermediary for all messages. A message is sent from one agent to one or more other agents (more than one recipient is supported through the use of broadcast messages). If an agent wishes to send the same message to two private recipients (i.e. not via a broadcast message) then two separate messages must be sent (this can be done in the same time step). Where two agents engage in bidirectional communication, this is achieved with a sequence of one-to-one messages.

InterDyne supports a wide range of messages from the very simple (see below) to the very complex (for example, support has been developed for modelling FIX messages based on the industry-standard Financial Information eXchange protocol).

Both one-to-one and broadcast messages are needed in modelling complex systems. For example, in the financial markets an exchange might send a broadcast message to all members updating them on the latest executed trades, whereas a trader might send a private message to an exchange to place an order.

Broadcast messages are sent to a numbered broadcast channel, and agents subscribe to zero or more broadcast channels. The subscriptions are set out at the start of the simulation and cannot change during a simulator run (though an agent is not forced to read the content of the broadcasts to which it is subscribed).

All messages comprise the following parts:

- A tag to indicate the type of message being sent.
- A tuple of two integers that indicate either (i) the agent IDs of the sending and receiving agents (for one-to-one messages) or (ii) the sending agent ID and the receiving broadcast channel identifier (for broadcast messages).
- The message data.

Example message type (an tags) are shown below:

- `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.

A message sent to agent ID 0 receives special treatment: it indicates that the message is being sent to the simulator harness, not to another agent. Messages sent to the simulator harness are printed to an output file (this is the main mechanism for specifying the output from the simulator). Currently all messages sent to the simulator harness are recorded in the trade file *except* those with type `Datamessage`, which are instead sent to a comma-separated file (this provides a structured form of output that can sometimes be useful — it is a first step in exploring better mechanisms for visualising the simulator output).

The following example illustrates a simple agent wrapper function that sends a debug message to the trace file via the simulator harness at every time step:

```
agent1 state args (packet:rest) myid
       =  [message]:(agent1 state args rest myid)
          where
          message = Debugmessage (myid,0) "Debug"
```

7 Delays and Topology

Communication delay is an important aspect when exploring interaction dynamics. With InterDyne there is always a minimum communication delay of one time step: a message sent at time step *t* will be received at time step *t+1*.

7.1 Delays

Communication delays of more than one time step can be achieved in several ways:

- If the delay changes during the course of the simulation, an agent may implement code that explicitly puts messages into a delay queue prior to sending them as part of the output at a given time step. If communication delay gets shorter as time increases, the code will have to calculate how messages sent later (but with less delay) might overtake messages sent earlier (with long delay).
- If the delay changes during the course of a simulation, it might be advantageous to create separate delay agents that intermediate between sender and receiver. Advantages might accrue from (i) keeping this delay logic separate from the other calculations that an agent must make, and (ii) where identical delays occur between different agents, routing messages through a single delay agent (thereby avoiding duplicated code).
- Where delays are more structured, the experiment designer can utilize InterDyne's built-in mechanism for commuication delay, explained below.

InterDyne supports (but does 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 broadcast, and asymmetric delays can be specified between two agents: for example, messages sent from agent 3 to agent 4 may have a smaller delay than messages sent from agent 4 to agent 3.

The primary advantage of this built-in delay facility is that during multiple runs of an experiment the amounts of delay between different agents can be varied systematically without requiring agent code to be changed.

7.2 Topology

InterDyne's built-in mechanism for specifying communication delays approach also provides an opportunity to define the topology of connections between InterDyne agents: if messages are not permitted from agent 7 to agent 3 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. 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.

Defining the communication topology can be useful in two ways:

1. As a validation tool: to test a complex experiment to observe whether an agent is interacting (via message-passing) as expected.

2. As part of the mapping from model to simulation, to implement a semantic interpretation of the modelled system. For example, where some agents represent the internal structure of a subsystem and are only permitted to interact with one other agent it can be useful to specify this in the topology.[3] The topology definition is then kept in one place and is easy to verify, and unspecified interactions are not possible in the experiments.

In order to define communication delays and topology two runtime arguments must be passed to the function `sim` (both must be passed, or neither). Examples of the two runtime arguments are given below:

1. `(DelayArg (Str "DelayArg", delay))`: the name of the delay function. The delay function (in this case, a function with the name `delay`) takes two agent IDs (one for the start point and one for the end point of an interaction) and returns a number that is the number of timesteps of delay to be added to all communication between the stated start point and end point.

2. `(Arg (Str "maxDelay", 10))`: the maximum delay in the system (in this case, 10 time steps).

The user may define the delay function to have any functionality that adheres to the specified type, including the possibility that asking for the delay between agent X and agent Y might cause a simulator abort with an error message if that communication path is not permitted. However, in the current implementation the delay function can only implement static delays[4] (which could however be changed for different runs of an experiment).

InterDyne detects the presence of these two runtime arguments and, if they are present, the simulator harness uses the delay function to calculate the required delay (additional to the minimum communication delay of one time step) for every sent message. Broadcast messages are separated into a distinct message for each recipient and each such message is delayed for the delay amount specified for the relevant comunication path (these are called "routed broadcasts", and are received in the list of normal messages in an agent's input).

The following example illustrates how a delay function might be defined and how to pass the information to InterDyne:

[3]This might be used to express a degree of hierarchy of subsystems.

[4]The delay function does not have access to the simulator time, nor to the local agent state of the sending agent.

```
experiment5
 = do
 sim 100      [(Arg (Str "maxDelay", 10)),
               (DelayArg (Str "DelayArg", delay))] agents
 where
 agents =     [(agent1, []), (agent2, []), (agent3,[])]
 delay x y =  if ((x = 1) && (y = 2)) then 10 else
              if (x = 2) then 0 else
              if ((x = 3) && (y = 2)) then 3 else
              error ("bad message:"++show x++" to "++show y)
```

8 Determinism and Non-Determinism

An InterDyne simulation can be either deterministic or non-deterministic. By default, InterDyne is deterministic and will provide the same results every time a simulation is run with the same initial values (this can be helpful when attempting to discover the antecedents of emergent behaviour, and in finding causal pathways). In particular, if multiple messages arrive at an agent in a given time step, they will always arrive in the same order (in the list of messages) every time the simulation is run with the same initial values.

Non-determinism can be expressed as follows:

- By including non-determinism in the code for an individual agent.
- By instructing InterDyne to randomise the order in which multiple messages are received at each time step. This is only effective where an agent receives more than one message in a given time step. It can be useful to remove suspected systematic bias when exploring dynamic behaviour: for example, a feedback loop might only occur if mesages are always received in the same order and the first message is always processed before the second message. 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).