Actor Monitors for Adaptive Behaviour

Tony Clark
Sheffield Hallam University, UK
t.clark@shu.ac.uk

Vinay Kulkarni
Souvik Barat
Tata Consultancy Services Research, India
vinay.v.kulkarni@tcs.com,
souvik.barat@tcs.com

Balbir Barn
Middlesex University, UK
b.barn@mdx.ac.uk

ABSTRACT
This paper describes a structured approach to encoding monitors in an actor language. Within a configuration of actors, each of which publishes a history, a monitor is an independent actor that triggers an action based on patterns occurring in the actor histories. The paper defines a model of monitors using features of an actor language called ESL including time, static types and higher-order functions. An implementation of monitors is evaluated in the context of a simple case study based on competitive bidding.

Categories and Subject Descriptors
I.6.2 [Simulation Languages]

Keywords
enterprise modelling, multi-agent simulation.

1. INTRODUCTION
This paper describes a structured approach to encoding monitors in an actor language. Within a configuration of actors, each of which publishes a history, a monitor is an independent actor that triggers an action based on patterns occurring in the histories. Monitors are useful for a variety of applications including adaptation and determining opportunities for collaboration.

Our particular interest is in the use of actors and monitors to develop simulations of organisations to support decision-making that is expressed in terms of goals, measures and levers. In many cases, the complexity of an organisation means that traditional approaches to simulation models, including spreadsheets and stock-n-flow, are difficult to apply because they rely on equations that explicitly represent the relationships between goals, measures and levers. In large organisations, such equations are difficult or impossible to produce for many situations. Our hypothesis, is that it is more practical and effective to take an emergent behaviour approach to the construction of the models whereby the appropriate elements of the organisation are represented as autonomous actors. Judicious construction of such models will allow a human decision-maker to observe simulation runs, make interactive changes to simulation-levers and to infer the relationships that lead to goal maximisation.

Actor languages are appropriate for emergent behaviour because they can be used to encode autonomous individuals and collaborations. Monitors are useful in this context because it is often necessary for individual elements (or groups) in a simulation to adapt their behaviour based on changes in the environment. A requirement on such monitors is that they express temporal properties and a contribution of this paper is to propose a statically-typed actor language for monitor combinators that supports temporal features.

As part of our aim to support organisational decision-making, we have developed an actor language called ESL. The features of ESL include static typing, higher-order functions, pattern matching and time. The aim of this paper is not to provide a detailed description of ESL, but we will use ESL to introduce the idea of monitors and include descriptions of ESL features as necessary to make the paper self-contained.

The paper is structured as follows: Section 2 motivates our claim that emergent behaviour is more appropriate than standard equational modelling for organisational decision-making, motivates the development of a language ESL, and reviews similar approaches to monitors in actor and agent technologies. Section 3 introduces a very simple example that we use to illustrate our proposal for monitors. Section 4 introduces that part of ESL necessary to understand how monitors are encoded. Section 5 describes how monitors are defined in ESL and section 6 describes how they are used to implement the simple example as part of decision-making.

2. MOTIVATION AND RELATED WORK
2.1 State of the Art
Enterprise Modelling (EM) supports a range of specification, visualisation and analyses capabilities for understanding various aspects of enterprises. For example, the ArchiMate [18] supports visualization of an enterprise along three aspects namely, structural, behavioural and information, arranged in three abstraction levels namely, business, application and technology, with adequate support for establishing relationships across aspects as well as levels. Other EM specifications such as EEML [19] and UEML [29] also visualize
an enterprise along multiple aspects and layers. BPMN [30], i* [32] and stock-n-flow (SnF) [22] are amenable to automated analysis. For example, the process aspect can be analysed and simulated using BPMN, the high level goals and objectives can be evaluated using i*, and high level system dynamics can be simulated using Stock-and-Flow (SnF) tools such as iThink1. Examination of existing EM reveals some interesting observations. The EM techniques capable of specifying all the relevant aspects of enterprise lack support for automated analysis (e.g., Archimate and UEML). Essentially they are not suitable for supporting an ‘apply levers - observe measure- evaluate goals’ loop. Languages capable of automated analysis only cater for a subset of the relevant aspects for decision-making (e.g., BPMN, i*, and SnF). Thus, as individuals, they are not adequate for decision making activities.

The multi-modelling and co-simulation based frameworks, such as AnyLogic [7] and DEVs [8], are possibly the best alternative among EM techniques to address the needs of decision making, however the inadequate support to specify and analyse socio-technical characteristics such as autonomous, uncertainty, temporal behaviour and adaptability make them ineffective for a large class of organisational decision making problems.

Approaches based on Bayesian Networks or Linear Programming are also found less effective for a class of decision making problems where the significant dynamism, uncertainty, adaptation and emergent behaviour are involved.

2.2 Proposal: Actors and Monitors

Our observation is that existing methods (outlined above) rely on a-priori knowledge of organisational behaviour. The complexity, socio-technical [21], and stochastic characteristics of organisations leads to such behaviour being emergent. This has led us to propose an approach to decision-making based on behaviours that emerge from organisational models encoded as interacting agents [5, 20].

The Actor model of computation [1] has been identified as a possible basis for simulation. We have reviewed a number of existing actor languages with respect requirements for organisational simulation and decision-making. For example, Erlang [3], SALSA [28], AmbientTalk [27], and Kilim [26] provide strong safety guarantees and data-race freedom using strict encapsulation and pure asynchronous communication. The actor frameworks (e.g., ActorFoundry [4], Scala Actors [16], Akka [2]) have flexibility to use the expressive power of the underlying languages such as Java and Scala.

Modelling an organisation as a collection of communicating actors involves autonomous behaviour of individuals. Monitors can be used to influence the behaviour of individuals. The concept of monitor that observes the behaviour of a system and detects the violations of safety properties is well studied area [15]. The early work on monitors focused on off-line monitoring wherein the historical data or program trace is analysed off-line [13] to detect the anomaly. In contrast the recent research trend on monitors is primarily focusing on online monitoring that validates the observed system execution against system specification dynamically as preventive and corrective measures.

Research challenges include: (1) the use of an appropriate foundational model for specification (this can be mapped to the research related to Monitoring-Oriented Programming); (2) an efficient implementation. Existing monitor technologies are largely based on temporal logic as an underlying model for monitor specification. For example, the past-time linear temporal logic (PTLTL) is recommended as specification language in [17], the PTLTL is further extended with call and returns in [23], the Eagle logic (where the temporal logic is extended with chop operator) is proposed in [6], and Meta Event Definition Language (a propositional temporal logic of events and conditions) is used in Monitoring and Checking (MaC) toolset [25]. In contrast, the Extended Regular Expression and Context Free Grammar based language is proposed and implemented in [10].

From an implementation dimension, online monitoring technologies are implemented either by adopting an inline approach [24] or a centralised approach [14]. A centralized approach is employed to the system under observation using either synchronous [11] mode or asynchronous mode [12]. In general, the inline monitors are computationally efficient as they have access to all system variables, whereas the centralised monitors are better in other dimensions as they enable clear separation of concerns (and thus less error-prone), facilitate distributed computation, and exhibit compositional characteristics.

We propose a light-weight variant of monitor technology that can evaluate agent goals (similar to monitoring safety properties) and decide appropriate system adaptation. A restricted form of centralized asynchronous monitor is implemented in Akka [2] to realise the monitoring behaviour of supervisor actors of a hierarchical actor system. Currently, a prototype of asynchronous centralised monitor implementation is presented in [10].

Our proposed implementation also implements centralized asynchronous monitors. A key difference is that the Akka implementation monitors the occurrence of events within its sub-actors and uses a fixed set of operations such as stop actor, suspend actor as an adaptation strategy. In contrast our proposed implementation evaluates the historical data to produce adaptation, effectively achieving a combination of [10] with the adaptation strategy presented in [9] augmented with temporal features.

In all cases, we have identified areas where we feel that an existing language does not, or may not, support all the requirements for organisational decision-making as outlined above. This has led to the development of the actor language ESL which is introduced in this paper. ESL supports many of the features we believe to be useful for organisational simulation such as time, static-typing, stochastic behaviour, and the production and visualisation of simulation traces. Although it is still in development, all the examples shown in this paper are based on ESL executions.

3. EXAMPLE: COMPETITIVE BIDDING

Commercial organisations often participate in a market for products and services. Customers have requirements for the products that they make available as an invitation to tender. Providers are given a period of time within which they can produce a description of how they meet the requirements and what the cost will be. At the end of the bidding period, a customer evaluates bids using their private criteria. One of the providers (the winner) is chosen and all other

1www.isesystems.com/store/products/ithink.aspx

2http://fsl.cs.illinois.edu/index.php/Monitoring-Oriented_Programming
providers lose the bid.

Figure 1 shows a diagram containing customers, providers and a global notice-board that contains the public invitations to tender. In this simple scenario, each customer represents their internal requirements as a colour: either red or green. When the tender is made public on the notice-board, the providers do not know the private colour that the customer requires. Each provider chooses to bid red or green. After a period of time the customer inspects the bids and chooses one that matches their required colour. Each provider is shown with a history of bid-successes and bid-failures.

Although each provider does not know the required colour of the published tender, they can inspect the history of each customer in terms of their required colours over time. It is reasonable to expect that a customer repeatedly issues invitations of a particular colour and that there is a small probability that this colour will change at any time.

In such a situation we may take the role of a provider and aim to determine a strategy that will improve our chance of winning a bid. In this case the strategic options are the levers: (1) simply take a random chance; (2) stick with the same colour over time; (3) monitor the behaviour of one or more customers via their history and predict the next requirement. Given that competitors may be using similar strategies, it is attractive to use a simulation-based approach to measure the outcomes for each option.

In the simple case we use the simulation from the perspective of a distinguished provider p in order to determine a bidding strategy. Therefore the simulation measure is the number of wins by p and the levers of the simulation vary the provider’s behaviour in order to maximise the number of wins.

4. ESL

ESL and its associated development and run-time environment is a language that has been designed to support our thesis that simulation and emergent behaviour can be used to support organisational decision-making. ESL together with a supporting toolset is currently in development. All ESL code in this paper is taken from running examples.

4.1 Syntax

The syntax of ESL is shown in figure 2. It is statically typed and includes parametric polymorphism, algebraic types and recursive types. Types start with an upper-case letter. An ESL program is a collection of mutually recursive bindings. The variables Var and var start with upper-case and lower-case letters respectively. Pattern matching is used in arms that occur in case-expressions and message handling rules. Uncertainty is supported by probably(p) x y that evaluates x in p\(x\) of cases, otherwise it evaluates q.

4.2 Evaluation

A minimal ESL application defines a single behaviour called main, for example:

```
1 type Main = Act ( Time (Int) );
2 act main : Main {
3   Time (100) → stopAll();
4   Time (\(m\) : Int) → {};
5 };
```

An ESL application is driven by time messages. The listing defines a behaviour type (line 1) for any actor that can process a message of the form Time(n) where n is an integer. In this case, the main behaviour defines two message handling rules. When an actor processes a message it tries each of the rules in turn and fires the first rule that matches. The rule on line 3 matches at time 100 and calls the system function stopAll() which will halt the application. Otherwise, nothing happens (line 4).

ESL Messages are sent asynchronously between actors.
An actor that is at rest selects a new message and processes it in a thread that is independent of other actor threads. When the thread terminates, the actor is ready to process the next message. Consider the concurrent processing of factorials:

```scala
type Customer = Act( Value[Int], Time[Int] );
type Fact = Act( Get[Int, Customer] );
act fact : Fact ( Get(0, : n : Customer) → c ← Value(1) ;
Get(n : Int, c : Customer) →
{ new fact ← Get(n : 1, new cust(n, c)) }
);
act cust(n : Int, c : Customer) : Customer {
Value(n : Int) → c ← Value(n : + );
Time(n : Int) → ()
};
act main : Customer {
Value(n : Int) → print[Int](n);
Time(n : Int) when n < 15 → new fact ← Get(3, self)
}
```

The `main` actor sends a request for `!3` to a new factorial actor at times 1 and 15. Instead of using traditional stack-based recursion, the `fact` actor dynamically creates a new `act` actor to process the recursive call and a `cust` actor to handle the initial call and a new recursive call. This allows multiple requests on the original `fact` actor to be processed concurrently as shown in figure 3 (generated by ESL) where the nodes are actors and the edges are labelled with `[t]m` where `t` is the time when the message `m` is sent. The trace shows that 14 calculations of `!3` are processed concurrently eventually leading back to the `main` actor.

### 4.3 Simple Monitors

Consider the definition of a simple monitor for a class of behaviours `M[T]` where each actor in the class has a value of type `T`. A monitor observes a monitored object until a given condition is satisfied at which point it fires an action provided by the monitored object by sending it a message. The type of a monitored actor is:

```scala
type M[T] = Act(
export value : T ;
Time[Int] )
// The value to be monitored.
// Time drives the application.
Action()
// An action available to others.
);
```

An example monitored actor manages a single integer value that is incremented or decremented randomly each time unit. The cell prints a message when the action is performed:

```scala
act cell : M[Int] {
export value ;
value : Int = 0
Action → print[String]('Action Performed');
Time(n : Int) →
probably(50) : Int {
  value := value + 1
} else value := value - 1
}
```

A simple monitor is constructed using a predicate that is checked against the exported value. When the predicate is satisfied, an action message is sent to the monitored actor:

```scala
act monitor[T] (p : T) : Act[Int, M[T]] = Main {
Time(n : Int) when p(n.value) →
m ← Action;
Time(n : Int) → ()
};
```

The monitor `m` is applied to the monitored cell `c` and invokes the action when the value is increased to 10:

```scala
c : M[Int] = new cell;
m : Main = new monitor[Int](λ(v : Int) : Bool v > 10, c);
```

### 4.4 ESL Execution

ESL compiles to a virtual machine code that runs in Java. Each actor is its own machine and thereby runs its own thread of control. Figure 4 shows the ESL executive that controls the pool of actors. When the executive is called, the global pool `ACTORS` contains at least one actor that starts the simulation. Global time and the current instruction count are initialised (lines 3 and 4) before entering the main loop at line 4; the loop continues until one of the actors executes a system call to change the variable `stop`.

Lines 6 – 7 copy the global pool `ACTORS` so that freshly created actors do not start until the next iteration. If an actor’s thread of control has terminated (line 9) then a new thread is created on the actor’s VM by scheduling the next message (line 10) if it is available.

The executive schedules each actor for `MAX_INS` VM instructions. This ensures that all actors are treated fairly. Once each actor has been scheduled, the existing actors are merged with any freshly created actors (line 14).

The executive measures time in terms of VM instructions. Each clock-time in the simulation consists of `INSTS_PER_ TIME_UNIT` instructions performed on each actor. When actors need to be informed of a clock-tick (line 15), global time is incremented (line 17), the instruction counter is reset (line 18) and all actors are sent a clock-tick message.

### 5. MONITORING HISTORY

The previous section described a simple state monitor which is a special case of a monitor that regularly checks the history of an actor. In the case of the example from section 3, a provider may choose to continually check whether they...
are repeatedly losing bids and then adapt their behaviour accordingly. A more sophisticated mechanism for encoding monitors is required that handles temporal features. This section defines a simple language, outlines its semantics.

### 5.1 History Formulas

Figure 5 shows a language that can be used to express predicates over the history of an actor. In each case the language construct is defined on the left and a definition of satisfaction is given on the right. Satisfaction of a formula is defined with respect to a history, for example seq(p,q) is satisfied with respect to a history h1+h2 when the prefix h1 satisfies p and the suffix h2 satisfies q. The language is to be viewed operationally because an action may cause a side-effect. The difference between alt and xor is that alt waits to determine that p is not satisfied before trying q whereas xor expects one of the sub-formulas to fail and tries both at the same time.

A history can be thought of as a list of public state information and as such each history formula is defined as being satisfied in terms of a list of data. For example, suppose that we want to express a history formula \( \text{fff} \) that causes action a to be performed every time a sequence of 3 fails, 000, is detected:

\[
\begin{align*}
\text{anF}(0) & : \text{Bool} = \text{true} \\
\text{anF}(1) & : \text{Bool} = \text{false} \\
\text{anF}(n) & : \text{Bool} = \text{not}(\text{anF}(n-1)) \\
\text{any}(n: \text{Int}) & : \text{Bool} = \text{anF}(n) \text{ or } \text{aT}(n) \\
\text{pxn}(\text{T})(n: \text{Int}, p: \text{T}) & : \text{Bool} = \text{anF}(n) \\
\text{seq}(p, q) & : \text{Bool} = \text{seq}(\text{pxn}(\text{T})(n-1, p), q) \\
\text{fff} & : \text{Mtr}([\text{Int}]) = \\
& \text{always}([\text{Int}]) \text{seq}(\text{pxn}(\text{[Int]})(3, \text{anF}), \text{is}([\text{Int}])(\text{any}))
\end{align*}
\]

The predicates \( \text{anF} \) and \( \text{any} \) are defined to detect the appropriate state elements. The history formula \( \text{fff} \) uses the operator \( \text{seq} \) to compose three \( \text{F} \) detectors one after another in the history. The operator \( \text{next} \) is used to advance through the history. Finally, the history predicate \( \text{fff} \) combines the three \( \text{F} \) detector with an alternative detector \( \text{is}(\text{any}) \) that skips a state value. The monitor \( \text{alt}(p, q) \) checks p first, if p fails then q is checked, so line 13 will use three \( \text{F} \)'s as a guard on the action a, if the guard fails then the head of the history is skipped. The monitor \( \text{always}(p) \) continually checks p throughout the history.

The history of an actor is produced incrementally over time. Therefore an expression written in the language defined in figure 5 must continually monitor the actor’s history. The expression can be thought of as a state machine whose nodes correspond to monitor states and whose transitions consume parts of actor histories. Each transition is triggered by a clock-tick and can proceed when there is some history to consume, otherwise the machine must stay in its current state and try again when the next tick occurs.

Figure 6 shows the machine corresponding to \( \text{fff} \). Each transition is triggered by a clock-tick, the labels on the transitions are: \( \epsilon \) when no history is available; \( \text{F} \) occurs when the next state element in the history is an \( \text{F} \); \( \text{C} \) occurs when there is at least one element at the head of the history and causes the element to be consumed; \( * \) denotes the situation when the next state element in the history is anything but \( \text{F} \).

### 5.2 Monitor Types

A monitor for histories over type \( \text{T} \) is an actor of type \( \text{Mtr}[\text{T}] \):

\[
\begin{align*}
\text{type Fail} & = () \rightarrow \text{Void} \\
\text{type Mtr}[\text{T}] & = \text{Act} \{ \\
& \text{export history} : [\text{T}]; \\
& \text{Time}(\text{Int}) \} \\
\text{type Mtr}[\text{T}] & = \text{rec M. Act} \{ \\
& \text{Check}(\text{Mtr}([\text{Int}], \text{Int}, \text{Fail})); \\
& \text{Time}(\text{Int}) \}
\end{align*}
\]

whose behaviour supports two messages: \( \text{Time(t)} \) drives the state machine; \( \text{Mtr}(a,c,s,f) \) activates the monitor and contains a monitored actor a, an integer c that indexes the next element of a.history, a monitor s that is used as a success continuation, and a function f that is used as a fail continuation. The key idea is that if \( \text{m} \leftarrow \text{Mtr}(a,c+1,s',f) \) then at some future clock-tick, if \( \text{m} \) is able to consume the \( c \)th element from a.history then \( \text{n} \leftarrow \text{Mtr}(a,c+1,s',f) \) otherwise if \( \text{m} \) rejects the \( c \)th element then \( f() \) tries an alternative.

### 5.3 ESL Monitors

The monitor behaviours are defined in figure 7, note that where the clock-tick handler is \( \text{Time(n: Int)} \rightarrow () \) it is omitted. The simplest monitor has the behaviour nothing that directly activates the success continuation without modifying any of the supplied data. Since the nothing behaviour is always the same, it makes sense to create a distinguished actor idle.

An action \( \text{action(c)} \) performs the command c when it is activated. A sequence monitor \( \text{seq}(p,q) \) will be satisfied when p is satisfied followed by q. Note that p may have consumed some of the history and so q is supplied as part of the success continuation to p.

There are two forms of alternatives: \( \text{alt}(p,q) \), if p is satisfied then so is \( \text{alt}(p,q) \) and q is ignored, otherwise q is tried.

\[
\begin{align*}
p, q & : = \text{nothing} \\
& \text{nothing}(h) \Rightarrow \text{always}(p)(h1+b2+\ldots) \\
& \text{rec}(\lambda(n)p) q(h) \text{ where } q = (\text{q/n})p \Rightarrow q(\text{rec}(\lambda(n)p)) \\
& \text{seq}(p,q) \Rightarrow \text{seq}(p,q)(h1+b2) \\
& \text{alt}(p,q) \Rightarrow \text{alt}(p,q)(h1+b2) \\
& \text{xor}(p,q) \Rightarrow \text{xor}(p,q)(h1+b2) \\
& \text{next}(p) \Rightarrow \text{next}(p)(a:h) \\
& \text{is(c)} \Rightarrow \text{is(c)}(x:h) \\
& \text{action(a)} \Rightarrow \text{action(a)}(h) \\
& \text{split}(p,q) \Rightarrow \text{split}(p,q)(\text{zip}(h1,b2))
\end{align*}
\]
The alternative xor(p,q) assumes that only one of p and q will be satisfied and tries them concurrently.

Recursive monitors are created using rec which is supplied with a function g whose argument is a cyclic monitor. For example:

\[
\text{rec[Int]}(\lambda(\text{fstar}::\text{Mtr[Int]}):\text{Mtr[Int]} \\
\text{seq[Int]}(\text{isF,next(fstar)}))
\]

is a monitor that will expect a history to contain an infinite sequence of 0s.

### 5.4 Example Traces

ESL provides a visualisation of execution traces that can be used to check our understanding monitor behaviours. Consider the case of a monitored actor a that produces a trace of true and false encoded as integers 1 and 0. To check that the trace is correctly formed we might construct the following monitor:

\[
isT::\text{Mtr[Int]} = \text{new is[Int](}a\text{T)} \\
isF::\text{Mtr[Int]} = \text{new is[Int](}a\text{F)} \\
\text{alwaysForT}::\text{Mtr[Int]} = \text{always}[\text{Int}](\text{new xor[Int]}(\text{isT},\text{isF}))
\]

Figure 8 shows the trace (up to time 141) produced by ESL as a result of evaluating:

\[
\text{alwaysForT} \leftarrow \text{Check(Leaf\[\text{Int}\]}(a),0,\text{idle\[\text{Int}\]},\lambda()())
\]

The actor init(138) is just the initialisation actor and can be ignored. The trace starts at main(140) which sends a Check message at time [1] to rec(145) that is responsible for setting up a loop. Each loop sends a Check message to the both(147) actor which in turn sends the message to the isT(142) and isF(143) actors. We deduce from the trace that a produces 0 as the first two elements of its history since the isF(143) actor invokes its success continuation in response to checking the history at index 0 and index 1, whereas actor isT(142) does not send any messages.

It may be necessary to check the histories of two or more actors at the same time. This is achieved by encoding the actors using the constructor Pair[T]::(MTree[T],MTree[T]) → MTree[T]. Given two actors that produce histories of 0 and 1 the following monitor checks that the first actor produces 1 and the second produces 0 on the next two occasions:

\[
\text{twice}[T](n::\text{Mtr[T]}):\text{Mtr[T]} = \text{new seq}[T](\text{m},\text{new next[T]}(\text{m})) \\
\text{tfX2}[Mtr[T]] = \text{twice}[\text{Int}](\text{new split}[\text{Int}](\text{isT},\text{isF})) \\
\text{pair[NTree[Int]]} = \text{Pair[Int]}(\text{Leaf\[Int\]}(a1),\text{Leaf\[Int\]}(a2)) \\
\text{tfX2} \leftarrow \text{Check(pair,0,\text{idle\[Int\]},\lambda()())}
\]

### 6. DEVELOPING A SIMULATION

Section 5 has defined a language for representing monitors over actors with histories. The language has been implemented as a statically type-checked collection of monitor combinators in ESL. This section uses the approach and associated technology to develop a simulation for the simple scenario described in section 3. The first step is to define a trace-based specification of the scenario in section 6.1, to define monitors for adaptive behaviour in section 5 that are
used in an ESL-based simulation in section 6.2 producing results in section 6.3.

6.1 Specification

Each ESL simulation is based on an execution model that creates a sequence of states of type $\Sigma$. The starting point for developing a simulation is a black-box model that identifies the key features of the histories that will be measured and that can be affected via levers. Refinement of the initial specification introduces detail necessary for adaptation.

Figure 10 shows a constructive approach in terms of a set of customers $C$, a set of providers $P$ and a set of offers $O$. The set $\emptyset^*$ denotes the set of all traces of arbitrary length. The set of traces $B_o$ contains all possible offerings $c^{\alpha(t)}$ by customer $c$, of offering $o$ with colour $t$ and defines them to be available for a clock-ticks. The set of traces $B_{p}$ contains awards $p^{o(t)}$ of offering $o$ to provider $p$. Traces $B_{o}$ define bids $p^{o(t)}$ by provider $p$ for opportunity $o$.

The operator $\oplus$ freely combines traces with the expectation that predicates hold for all traces. The predicates are not defined here but are outlined as follows: offer holds for traces where each customer can offer, and each provider can bid for, at most one opportunity at any given time, and bids must be concurrent with offers. award holds for traces where an award occurs at the end of the offer period $a$ and when the colour of the bid matches that of the offer; no award can be made when no bids are present and at most one award can be made.

The goal of the bidding simulation is to achieve a behaviour that is specified by $\Delta$. ESL can be used to construct a system of interacting actors with emergent behaviour. Monitors can be used to detect when the behaviour diverges from the specification and take corrective action. Levers that control the monitors can help inform strategic decisions.

6.2 Implementation

The specification given in section 6.1 defines a sequence of system states that must be refined to produce an ESL simulation model of the system whose levers can be modified and whose goals can be measured. The refinement must produce a system that is consistent with the specification, but that may introduce more detail. The result is an implementation defined in terms of five types of behaviour customer generating opportunities, opportunity managing the bidding and awarding process for opportunities, a notice-board $ab$, provider generating bids and receiving their outcome, and a monitor for provider adaptive behaviour. This section gives the key elements of the implementation.

The customer behaviour is supplied with a colour, a percentage chance of changing the colour, the frequency at which the opportunities are created and the availability of the opportunity:

```plaintext
1 data Colour = Red | Green
2 opp(Red).Colour = Green
3 opp(Green).Colour = Red
4
type Customer = Act {
1 // a customer is a monitored actor.
2 export history:[Colour].
3 Time(Int); // time generated opportunities.
4 Done(Colour); // record opportunity completion.
5 }
7 // create opportunities coloured $c$, with frequency $freq$, availability $avail$ with a $cng$ chance of
8 // changing colour.
9 export history.
10 next:Int = 0;
11 history:[Colour] = \[
12 offer() = \text{nb} \leftarrow \text{add} \text{(new opportunity(self,c,avail))}\n13 \text{Time(a:Int) when next}=freq \rightarrow \{
14 next := next + 1;
15 \}
16 \}
17 \}
18 \}
19 \}
20 \}
21 \}
22 \}
23 \}
24 \}
25 \}
26 \}
27 \}
28 \}
29 \}
```

A customer detects clock-ticks on lines 20 and 27. Every clock-ticks (line 20) there is a 90% chance (line 22) that an offer is created (line 23) and a chance that the colour changes. A new opportunity is created in line 19 by sending an Add message to the global notice-board $ab$.

An opportunity behaviour is supplied with a customer, the colour of the opportunity and the length of time the opportunity should be available:

```plaintext
1 type Opportunity = Act
2 export customer:Customer;
3 Time(Int);
4 Bid(Colour,Provider); // check whether to make award.
5 }
```
An opportunity receives bids from providers (26-28). If the opportunity is available then the bid is saved (27) otherwise the provider has missed the deadline and is informed that the bid has failed (28). Clock-ticks reduce the availability count (45), when the opportunity closes and there are bids (29), the bids are taken in a random order (33) and awarded (34). An opportunity is terminated by removing it from the global notice-board (36) and informing the customer that the colour was processed (38). The actor is killed (37) meaning that it can perform no further computation.

A provider is created with a colour that controls the bids:

```
data result = S | F;
act provider = Act{
  export history=Result, nb
  history=history ++ [result],
  bid():
    history=history ++ [result],
    nb: bid();
  when nb = ()
    history=history ++ [result]}

// a provider initially bids using colour. It publishes
// a history of successes and failures. It can be
// instructed to change bid-colour for particular
// customers.
act colour = Color

// Colour_for the supplied customer.
act colour_for = Colour

// Colour supplier
act supplier = Supplier
```

Figure 9: Checking for F or T

Figure 10: Specification of Bidding Behaviours $\Delta$
A provider checks the opportunities on the notice-board and makes a bid (131–33, 19–22) based on the history (20). A monitor (defined below) may request that a provider changes colour for a particular customer (42–43) that is subsequently used to determine the colour of a bid (23–30) for that customer.

A bidding simulation has the following levers: the number of providers, the number of customers, the likelihood that the customer changes the colour of their opportunities, the duration of the simulation and the monitors.

A monitor is created for each provider and customer pair. Our levers are three different types of monitor: none that does not change a provider, random that changes a provider’s colour at random, and adaptive that changes a provider’s colour based on its failure rate:

```plaintext
none(p:Provider,c:Customer):Mtr[Int] = // Do nothing ...
idle[Int]
random(p:Provider,c:Customer):Mtr[Int] = // Randomly change the colour for the supplied // provider when bidding for an opportunity from // the supplied customer...
let getColour():Colour = probably(50):Colour
Red
else Green;
change(): Void = p :< Change(getColour(),c)
in always[Int](now action[Int](change))
adaptive(k:Colour,p:Provider,c:Customer):Mtr[Int] = // Use a guard to check whether the colour should // change. The guard checks that:
// (1) Provider p has failed 3 times; and
// (2) Customer c has offered the opposite colour 3x...
let chkOpp(v:Colour):Bool = v = opp(k) in
let isOpp:Mtr[Colour] = is[Colour](chkOpp) in
let op3:Mtr[T] = pmx[Colour](3,isOpp);
change(): Void = {
  k := opp(k);
  p :< Change(k,c)
} in
let guard = split[Int](fff,op3)
in always[Int](seq[Int](guard,action[Int](change)))
```

Figure 11 shows a configuration of actors involving two providers and a single customer. Note that both providers have their own monitor. Figure 12 shows the initial part of a trace for a situation involving a single customer and a single provider. Four opportunities are created during the simulation. The provider bids for the first three opportunities at times [5], [12] and [17]. All bids fail at which point the monitor detects the failing situation and changes the colour to Red at [21]. After the provider has been influenced by the monitor it makes a successful bid for an opportunity at time [22].

### 6.3 Results

Figure 13 shows the results of the simulation given 2 customers and two categories of service provider: competitors (10) and ourselves (1). Each category of service provider can have three different adaptor types:

- **none** No adaptive behaviour: the provider will not change the colour of the bid.
- **random** The provider changes its bid colour at random.
- **adaptive** The provider uses a monitor on all customer histories in order to change bid colour when it has failed to be awarded an opportunity 3 or more times.

The simulation was executed for 1000 clock-ticks for each configuration of service provider category and adaptor type. The number of successful bids for ourselves and the maximum number of successful bids for competitors was recorded for each configuration. Figure 13 shows results for a selection of configurations: the legend labels each result line in terms of configuration strategies (ourselves, competitors) and shows the number of successes for ourselves.

The simulation is intended to aid decision making with respect to whether a particular strategy is cost effective, and although we would need to know the cost of monitoring historical records, the results indicate that it is clearly in our favour to implement the adaptive strategy if we believe that our competitors are behaving randomly. Also, it would appear that no strategy at all is of similar value to random choice. Where all categories are behaving strategically, there is a benefit to us, however this is significantly less that when our competitors are not strategic.

### 7. CONCLUSION
This paper has proposed an approach to organisational decision-making using emergent behaviour expressed as actor models. We are in the process of developing technology to support this approach in the form of a language called ESL. Part of the approach depends on monitors that detect patterns of behaviour and influence actors by sending them messages. The contribution of the paper is to show how a language of monitor combinators can be defined in ESL and used to implement a simple case study. All the code in the paper has been implemented in the ESL language which runs on a VM written in Java and all actor configurations and message traces have been generated from EDB and visualised using GraphViz.

Further work is planned in the following areas: Although we have identified an approach to specification of a simulation given in section 6.1, no attempt has been made to verify properties of the ESL program; research in this area [31] will be investigated. ESL has been designed with the aim of expressing patterns of organisational behaviour, and with the expectation that results from Multi-Agent Systems research will be relevant to constructing organisational simulation models. We intend to investigate how to extend ESL with abstractions that are specific to organisations and to encode features such as negotiation and collaboration.

ESL is supported by a development environment called EDB that performs syntax checking and type checking. EDB also supports debugging and exporting simulation traces such as those shown in this paper. We aim to use EDB as a basis for research into debugging and visualising actor systems.

References
[8] Benjamin Camus, Christine Bourjot, and Vincent Chevrier. Combining devs with multi-agent concepts to design and simulate multi-models of complex systems (wip). In Proceedings of the Symposium on Theory of


