ESL: An Enterprise Simulation Language

Double-blind submission

Abstract

Computational support for organisational decision-making is an emerging field where challenges arise from characteristic features such as complexity, emergent socio-technical behaviour, collaborative behaviour, and dependencies between objectives. Current approaches are mostly either human-centric or are based on technologies such as spreadsheets and equational models such as *stock and flow*. A number of researchers have proposed using *agents* to model aspects of organisations and thereby support decision-making, but there is no widely accepted approach. This paper develops existing work in this area by proposing a conceptual model for organisational decision-making and introduces an actor-based language called ESL for simulation of the concepts. The approach is demonstrated via a the implementation of a synthetic example and evaluated using a real-world example from industry.

Categories and Subject Descriptors I.6.2 [*Simulation Languages*]

Keywords enterprise modelling, multi-agent simulation.

1. Introduction

Organisational decision-making is a hard problem and we have few tools to help understand and address the area. The existing tools do not adequately represent the characteristic features of the problem and this has led to a proposal for using agent-based approaches as described in section 2.

Agents seem to offer a better basis for the characteristic features of an organisation, however there is no conceptualisation of agent-based systems that is appropriate for decision-making; section 3 analyses the problem domain and presents such a conceptual model that has been used by the authors as the basis of real-word case studies including that described in this paper.

We lack appropriate models for organisational decision-making that can help motivate the development of supporting technologies. Given that the problem domain is large and variegated, it makes sense to divide and conquer by seeking specific scenarios. A typical example relates to how commercial advantage can be achieved within a market that is created when multiple customers offer up opportunities for competitive bid by service providors. A framework for modelling this situation is described in section 4.

Our proposed approach is to view such a situation through the lens of the conceptual model and then to represent the resulting concepts in a technology that is suitable for simulation. In order to achieve this we require an appropriately expressive technology that supports translation of the problem domain concepts into agent-based solution domain patterns. We have found that the Actor Model of computation provides a suitable metaphor for such an encoding and have designed a language called ESL to support this approach whose implementation is described in section 5.

The contribution of this work is to propose a conceptual model for agent-based simulation of organisations and to provide a language for a suitable computational implementation platform. These contributions are provided in sections 3 and 5 and demonstrated in terms of a synthetic framework for bidding in section 6. Evaluation of the approach is provided by a real-world case study taken from the software services domain operating within Tata Consultancy Services as described in section 7.

2. Related Work

Enterprise Modelling (EM) [22] aims to reduce dependence on human experts for organisational decision making. However, the languages capable of modelling all relevant aspects, *e.g.*, the aspects suggested by Zachman in [27], lack support for automated analysis. In contrast, languages providing automated support for qualitative and/or quantitative analyses are capable of modelling only one aspect of enterprise [2]. Therefore, a decision maker is forced to construct a toolchain involving a large spectrum of EM tools [6] which is a challenging task.

The general industry practice is to follow a refinement-based method, *e.g.*, [17], for organizational decision making that is guided by separation of concerns driven by goal analysis, their dependencies, the means of achieving the goals and their qualitative and quantitative differences. Such analysis raises questions relating to the *levers* and *measures* that are used to undertake decision-making and ultimately differentiate between alternatives [8, 17].

Languages such as Archimate [13], EEML [14] and IEM [4] enable the specification of multiple aspects in an integrated manner. Archimate visualizes an enterprise along three aspects: *structural*, *behavioural* and *information*, and three levels: *business*, *application* and *technology*. IEM visu-

alises an enterprise along two aspects: *information* and *process*. These structured representations help to improve documentation quality but lack precise analytical semantics necessary for decision-making. As a result, these frameworks are vulnerable to multiple interpretations and rely heavily on human analysis.

Modelling languages such as BPMN [24], i* [26] and stock-n-flow (SnF) [16] are machine processable but support modelling of one aspect only. For example, the process aspect can be modelled using BPMN, high level goals and objectives can be modelled using i*, and high level dynamics can be modelled using SnF. Amongst these machine processable languages, i* supports only qualitative analysis whereas BPMN and SnF support only quantitative analysis.

There are language-specific peculiarities and limitations too: SnF modelling tools such as iThink[1] and Simantics[2] come with a rich simulation machinery supporting what-if simulation, however, the language is best suited for creating generic models which explode in size when specialised; several BPMN tools such as ARIS[3] and Bizagi[4] support what-if simulation but only in terms of time and behavioural resource parameters. Moreover, the three languages being paradigmatically diverse, it is difficult to integrate individual specifications in a meaningful manner [1, 7, 10], for example, the support available for specifying relationships across these aspects in AnyLogic[5] is little more than setting up navigation links from one specification to the other.

There is limited reported work on integration of multiple languages where each language supprts a different aspect of an organisation. The Unified Enterprise Modelling Language (UEML) [22] initiative aims to integrate existing Enterprise Modelling Languages using a meta-modelling framework. This is an ongoing initiative [18] with the first version of the UEML demonstrating integration of IEM, EEML and GRAI [9] supported by MOOGO[6] and METIS[7].

We aim to support decision-making by taking a *computational approach* to organisational modelling [23]. This builds on existing approaches described above that identify the key conceptual features and previous work by the authors (citations suppressed). We believe that the Actor Model of computation [12] is a suitable basis because it can support adaptive and emergent behaviour suitable for socio-technical systems [3, 11]. Our aim is that the actor model can be used to develop a multi-agent systems based approach to organisational simulation and decision-making [5, 15, 21].

[1] `iseesystems.com/Softwares/Business/ithinkSoftware.aspx`

[2] `sysdyn.simantics.org/`

[3] `ariscommunity.com/university/downloads/aris-business-architect`

[4] `www.bizagi.com/`

[5] `anylogic.com/`

[6] `moogo.de`

[7] `enterprise-architecture.info/Images/ComputasMetis/Metisoverview.htm`

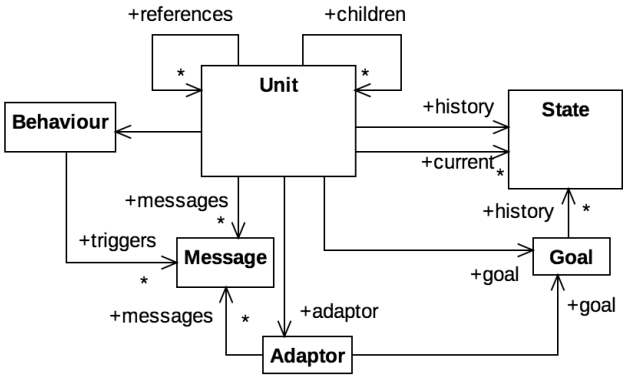

Figure 1: Conceptual Model

There are several languages that use the actor model of computation [20] many of which address the issue of easing the engineering challenge of developing concurrent software. Our motivation is slightly different: to provide an expressive basis for organisational simulation. ESL has the notion of time built-in to its executive since this seems to be key to co-ordination of units within a simulation.

3. Conceptual Model

A variety of Enterprise Modeling (EM) languages exist that provide information-capture and analysis support. The majority of these languages can be traced to the Zachman framework that is defined in terms of *why*, *what*, *how*, *who*, *when* and *where*. Many languages (*e.g.*, Archimate) provide a rich collection of modelling concepts for these concepts, however previous work (citations suppressed) has proposed that for the purposes of computational simulation and analysis the features can be reduced to a much smaller set of concepts.

Our aim is to view an organisation as a socio-technical system where the intrinsic complex emergent behaviour is achieved by representing parts of the organisation as goal-driven autonomous self-aware adaptive units. This section describes the conceptual model that underpins the approach and that is implemented using the language ESL later in the paper.

Figure 1 shows the proposed conceptual model. An organisation is represented as a collection of units. A unit is used to represent anything that can act or be acted upon within an organisation and may correspond to a person, a resource, a department, a self-organised collection of individuals, *etc*. Units have internal state which can change over time giving rise to a history of states. Units may be composite and therefore have children, for example a department contains employees. Units may reference each other and there is an expectation that there is some notion of control whereby contained information can be designated *public* or *private*.

A unit is autonomous and therefore has behaviour that acts on the unit's state. Unit interaction is message-based

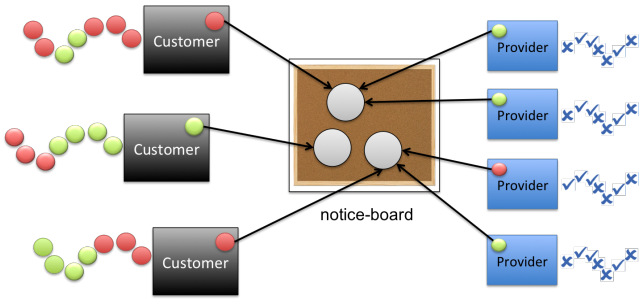

Figure 2: Competitive Bidding

with the assumption that messages can be both synchronous and asynchronous.

Units have a goal that drives their behaviour. This is an important feature of many aspects of an organisation viewed as a socio-technical system. A goal is a predicate over the history of a unit and it is therefore possible to determine at any given time whether a unit is satisfying its goal. A unit is free to dynamically change its goal, and, since units may be composite, a goal may relate to multiple units.

Units are self-adaptive, meaning that they can influence their behaviour through introspection. The conceptual model associates a unit with an adaptor that monitors the unit's goal. If the adaptor detects that the unit is not on target to achieve its goal then adaptation occurs via self-directed messages. Simulation is supported by augmenting figure 1 with:

measures A simulation is performed in order to support decision-making in terms of goals, for example: whether a goal is achieved, or the comparison of different configurations with respect to the same goal, or whether one goal is in some way more attractive than another. A goal is a predicate over unit histories and therefore a simulation measure can be conceptualised as a goal.

levers Multiple simulations are performed in order to compare different configurations of an organisation. Differences may include: unit configurations, behavioural traits, or adaptation. Levers are a conceptualisation of variations and generalise figure 1 in the sense of Software Product Lines [19].

4. Decision Making: 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 providers lose the bid.

Figure 2 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) try to observe 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.

The situation shown in figure 2 is clearly very simple, but it establishes a basic framework within which competitive bidding can be investigated through the use of simulation. This paper will show how the framework, as described, can be modelled and simulated.

Figure 1 shows the conceptual model for our approach. The competitive bidding example can be viewed in terms of these concepts as follows. Customers, providers, notice-boards and opportunities to tender are units. All units respond to messages from a global clock that drives the simulation. Customers use the time messages to schedule opportunities. Providers use the time messages to regularly inspect the notice-board for opportunities. An opportunity uses time messages to award itself to a provider whose offering matches a requirement and to inform the customer of the outcome. The histories of the units are as shown in figure 2 and the goal of a provider is to win as many opportunities as possible.

In the simple case we may be using 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 adapter in order to maximise the number of wins.

We believe that the framework can form the basis of a wide range of variations and increasingly sophisticated models. For example, the customer requirements are given as red or green, however in a real-world situation the requirements will have a range of properties that govern whether the bid

```
exp  ::= var                          variables
     |   num                          numbers
     |   bool                         booleans
     |   str                          strings
     |   self                         active actor
     |   null                         undefined
     |   new name (exp*)              create actor
     |   become name (exp*)           change behaviour
     |   exp op exp                   binary exp
     |   not exp                      negation
     |   λ(name*) exp                 λ-abstraction
     |   let bind* in exp             local bindings
     |   letrec bind* in exp          local recursion
     |   case exp* arm*               pattern matching
     |   for patt in exp { exp }      looping
     |   { exp* }                     block
     |   if exp then exp else exp     conditional
     |   [ exp* ]                     list
     |   exp(exp*)                    application
     |   Name(exp*)                   term
     |   exp ← exp                    message
     |   name := exp                  update
     |   exp . name                   name reference

bind ::= name = exp                   value binding
     |   name(patt*) = exp when exp   λ-binding
     |   act name(name*) {            behaviour def
            export name*              interface
            bind*                     locals
            → exp                     initial action
            arm*                      behaviour
         }

arm  ::= patt* → exp                  guarded exp

patt ::= name                         variables
     |   num                          numeric pattern
     |   bool                         boolean pattern
     |   str                          string pattern
     |   _                            wildcard
     |   patt : patt                  cons pair
     |   [ patt* ]                    list
     |   Name(patt*)                  term pattern
```

Figure 3: ESL Syntax

from a provider is acceptable compared to that of another provider. The information available to a provider in terms of the behavioural history of a customer and of its competitors can be made more detailed. Other aspects such as negotiation and collaboration can be introduced allowing, for example, multiple providers to work together in order to maximise the chance of collectively winning a bid.

5. ESL: An Enterprise Simulation Language

Section 3 describes a conceptual model for an approach to supporting decision-making in organisations through simulation. Given the nature of such a simulation we have proposed an actor-based implementation technology because of its intrinsic autonomy and ability to support auto-adaptation. We envisage a technology platform that can support domain-specific abstractions that has led to the definition of a simple and powerful actor-based, functional language called ESL (Enterprise Simulation Language).

The syntax of ESL is shown in figure 3. A sub-language of ESL is a dynamically-typed ML-like functional language with side-effects and pattern matching over simple data, lists and terms. Familiarity with such a sub-language is assumed and the rest of this section provides an overview of the actor-based features and the architecture of ESL.

An actor *behaviour* is the equivalent of a class in Java. For example, the following behaviour counts time:

```
act counter {
  t = 0
  Tick(_) → t := t + 1
}
```

All actors receive regular clicks from a global clock that drives the simulation. The behaviour `counter` has a local variable called `t` that is incremented each time a message is processed that matches the pattern `Time(_)`. A new actor with the `counter` behaviour is created by evaluating the expression **new** `counter` which is equivalent to the creation of a new object in Java. If the variable `t` is to be initialised differently for each new actor then the behaviour can be defined as:

```
act counter(t) { Tick(_) → t := t + 1 }
```

and the actor created as **new** `counter(0)`. Messages are sent asynchronously to actors. Typically a message is a term. The previous behaviour can be modified to allow the counter to be reset and to forward the current value of to a reception actor `r`:

```
act counter(t) {
  Tick(_) → t := t + 1;
  Reset   → t := 0;
  Get(r)  → r ← t
}
```

Each actor runs in its own thread and therefore the example above is *thread safe* in the sense that `t` cannot be modified and concurrently accessed. However, it is possible to export names from an actor:

```
act counter(t) {
  export t;
  Tick(_) → t := t + 1;
  Reset   → t := 0
}
```

allowing external actors to reference the name `t` using the . operator. ESL uses **probably** to introduce stochastic behaviour and can adapt the behaviour of an actor using **become** . For example, support an actor with behaviour `a` is implementing a unit that has a 1% chance of becoming inoperable:

```
act a { Time(_) → probably(1) become inoperable;..}

act inoperable { _ → {} }
```

ESL compiles to a virtual machine code that runs in Java[8]. 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`.

[8] link to source code suppressed.

```
1  stop := false;
2  exec() {
3    time := 0;
4    instrs := 0;
5    while(!stop) {
6      actors := copy(ACTORS);
7      clear(ACTORS);
8      for actor ∈ actors {
9        if terminated(actor)
10       then schedule(actor);
11       run(actor,MAX_INSTRS);
12     }
13     instrs := instrs + MAX_INSTRS;
14     ACTORS := ACTORS + actors;
15     if instrs > INSTRS_PER_TIME_UNIT
16     then {
17       time := time + 1;
18       instrs := 0;
19       for actor ∈ ACTORS
20         sendTime(actor,time)
21     }
22   }
23 }
```

Figure 4: The ESL Executive

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_INSTRS 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 INSTRS_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. The tick will suspend any current (non-tick related) computation on the actor's machine which will re-start when the tick has been handled.

6. Developing A Simulation

Section 3 has defined a conceptual model to be used as the basis for simulation and section 5 has defined the ESL language that is claimed to be suitable as an implementation technology for simulations. This section uses the approach and associated technology to develop a simulation for the simple scenario described in section 4. 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 6.2 that are used in an ESL-based simulation in section 6.3 producing results in section 6.4.

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

$$T = \{r, g\}$$
$$B_o, B_a, B_b, B, \Delta : [\Sigma]$$
$$\_ \oplus \_ : [\Sigma] \times [\Sigma] \to [\Sigma]$$
$$\mathit{offer}, \mathit{award} : [\Sigma] \to \{t, f\}$$

$$B_o = \{\epsilon + [\{c^{(o,t)}\}^a] + r \mid c \in C, o \in O, t \in T, \epsilon \in \emptyset^*, r \in B_o\}$$
$$B_a = \{\epsilon + [\{p^{(o)}\}] + r \mid p \in P, o \in O, \epsilon \in \emptyset^*, r \in B_a\}$$
$$B_b = \{\epsilon + b + r \mid p \in P, o \in O, t \in T, b \in \{p_{(o,t)}\}^*, \epsilon \in \emptyset^*, r \in B_b\}$$
$$B = \{t_1 \oplus t_2 \oplus t_3 \mid t_1 \in B_o, t_2 \in B_a, t_3 \in B_b\}$$
$$\sigma_1 : t_1 \oplus \sigma_2 : t_2 = \sigma_1 \cup \sigma_2 : (t_1 \oplus t_2)$$
$$\Delta = \{b \mid b \in B,\ \mathit{offer}(b) \wedge\ \mathit{award}(b)\}$$

Figure 5: Specification of Bidding Behaviours Δ

```
p,q ::= ϵ        always satisfied.
      | !e       always satisfied and performs e.
      | □ p      p is satisfied at all times.
      | μ n.p[n] q=p[q/n] and q is satisfied.
      | n        a name bound by a surrounding μ.
      | p;q      p and q are satisfied.
      | p+q      p or q are satisfied.
      | ○ p      the history is _:h and h satisfies p.
      | ?c       c(x) and the history is x:h.
```

Figure 6: History Formulas

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 5 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 of traces B_o contains all possible offerings $c^{(o,t)}$ by customer c, of offering o with colour t and defines them to be available for a clock-ticks. The set of traces B_a contains awards $p^{(o)}$ of offering o to provider p. Traces B_b 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.

6.2 Monitors

The conceptual model described in section 3 uses monitors to determine whether an actor in an organisational simulation is currently on target to achieve its goal, and uses adaptors to change the actor's behaviour when the goal is likely to be missed. This section shows how ESL can represent a monitor/adapter as an abstraction that uses history formulas to specify goals.

A monitor must regularly check the history of an actor. In the case of the example from section 4, a provider may

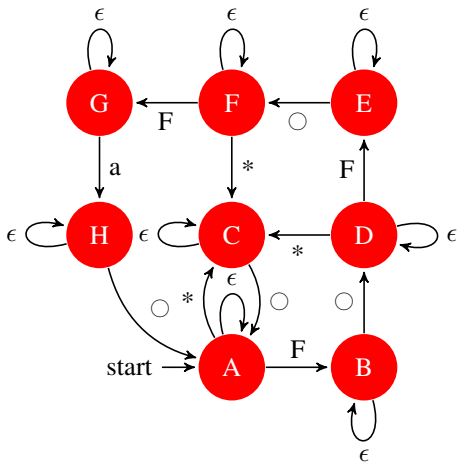

Figure 7: A Monitor State Machine

choose to continually check whether they are repeatedly losing bids and then adapt their behaviour accordingly. Figure 6 shows a language that can be used to express predicates over the history of an actor. 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 `checkFs` that causes action `a` to be performed every time a sequence of 3 fails, `FFF`, is detected:

```
anF(F)           = true
anF(_)           = false
anything(_)      = true
threeF(command) = ?anF ; ○(?anF ; ○(?anF ; !command))
checkFs(command)= □(threeF(command) + ?anything)
```

The predicates `anF` and `anything` are defined to detect the appropriate state elements. The history predicate `threeF` uses the operator `_;_` to compose three `F` detectors one after another in the history. The operator ○ is used to advance through the history. Finally, the history predicate `checkFs` combines the three `F` detector with an alternative detector `?anything` that skips a state value. The monitor `p+q` checks `p` first, if `p` fails then `q` is checked. The monitor `□ p` continually checks `p` throughout the history, so `□(p+q)` will check a series of `p`'s with `q` checked each time a `p` fails.

The history of an actor is produced incrementally over time. Therefore an expression written in the language defined in figure 6 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 7 shows the machine corresponding to `checkFs`. Each transition is triggered by a clock-tick, the labels on the transitions are: ϵ when no history is available; `F` occurs when the next state element in the history is an `F`; ○ occurs when

```
act ϵ {
  Monitor(a,i,s,f) →
    s ← Monitor(a,i,self,f)
}

act !(command) {
  Monitor(a,i,s,f) → {
    command();
    s ← Monitor(a,i,new ϵ,f)
  }
}

act p ; q {
  Monitor(a,i,s,f) →
    p ← Monitor(a,i,new q + s,f)
}

act p + q {
  Monitor(a,i,s,f) →
    p ← Monitor(a,i,s,λ() q ← Monitor(a,i,s,f))
}

□ p = new μ(λ(q) new(p ; new ○(□ q)))

act ○ p {
  Monitor(a,i,s,f) →
    p ← Monitor(a,i+1,s,f)
}

act ? pred {
  Monitor(a,i,s,f) →
    become activate(pred,a,i,s,f)
}

act activate(pred,actor,i,succ,fail) {
  Monitor(_,_,_,_) → {};
  Time(_) →
    if length(actor.history) > i
    then {
      become ? pred;
      if pred(nth(actor.history,i))
      then succ ← Monitor(actor,i,new ϵ,fail)
      else fail()
    } else {}
}

act μ g {
  Monitor(a,c,s,f) → g(new μ g) ← Monitor(a,c,s,f)
}
```

Figure 8: ESL Monitor Behaviours

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 `F`.

A monitor state machine is implemented as an ESL actor whose behaviour supports two messages. The first message `Time(t)` drives the state machine. The second message `Monitor(a,c,s,f)` activates the monitor and contains the monitored actor `a`, an integer `i` 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 `m ← Monitor(a,i,s;s',f)` then at some future clock-tick, if `m` is able to consume the `i`th element from `a.history` then `s ← Monitor(a,i+1,s',f)` otherwise if `m` rejects the `i`th element then `f()` tries an alternative.

The monitor behaviours are defined in figure 8, note that where the clock-tick handler is `Time(_) → {}` it is omitted. The simplest monitor has the behaviour ϵ that directly acti-

vates the success continuation without modifying any of the supplied data. An action ! is similar to ϵ except that it activates the supplied command. A sequenced monitor `p;q` consists of two component monitors `p` and `q`. When activated, the sequence tries `p` adding `q` to the success continuation. Alternative `p+q` activates `p` and adds `q` to the fail continuation. The monitor $\bigcirc$ `p` increments the count `i` thereby skipping the next element of the history.

The monitor `? pred` changes its behaviour when it is activated in order to check the history of the actor `a`. If there is an `i`th element in `actor.history` then the monitor applies `pred` to check whether the indexed history element satisfies the predicate. If so then the successor is activated, otherwise the monitor fails.

Recursive expressions are created using μ `g`. An example of recursion is used in the implementation of $\Box$ where $\Box$ `p` is unrolled to become `p ; ○(p ; ○(p ; ○(p ; ...)))`.

6.3 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` that models the generation of opportunities, `opportunity` that models the bidding and awarding process for opportunities, a notice-board `nb`, `provider` that models the creation of bids and 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:

```
1  opp(Red) = Green;
2  opp(Green) = Red;
3  act customer(colour,pcChange,frequency,avail) {
4    next = 0;
5    offer()=nb ←Add(new opportunity(self,colour,avail))
6    Time(_) when next=frequency → {
7      next := 0;
8      probably(90) {
9        offer();
10       probably(pcChange) colour := opp(colour)
11     }
12   };
13   Time(_)      →  { next := next + 1 };
14   Done(colour) → history := history + [colour]
15 };
```

A customer detects clock-ticks on lines 7 and 14. Every `frequency` clock-ticks (line 7) there is a 90% chance (line 9) that an offer is created (line 10) and a `pcChange` chance that the colour changes. A new opportunity is created in line 6 by sending an `Add` message to the global notice-board `nb`.

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

```
1  act opportunity(customer,colour,available) {
2    export customer;
3    tryAward(Bid(_,v),true)  = { v ← Failed; true };
4    tryAward(Bid(c,v),false) = { v ← Award;  true }
5                               when c = colour;
6    tryAward(Bid(_,v),false) = { v ← Failed; false };
7    terminate() = {
8      nb ← Remove(self);
9      customer ← Done(colour);
10     kill(self)
11   };
12   bids = []
13   b=Bid(colour,vendor) when available > 0 →
14     bids := b:bids;
15   Bid(_,vendor)  → vendor ← Failed;
16   Time(_) when (available = 0) and (bids <> []) → {
17     let won = false
18     in for b in shuffle(bids) do
19          won := tryAward(b,won);
20     terminate()
21   };
22   Time(_) when (available = 0) → terminate();
23   Time(_) → available := available - 1
24 };
```

An opportunity receives bids from providers (13-15). If the opportunity is available then the bid is saved (14) otherwise the provider has missed the deadline and is informed that the bid has failed (15). Clock-ticks reduce the availability count (23), when the opportunity closes and there are bids (16), the bids are taken in a random order (18) and awarded (3-6). An opportunity is terminated by removing it from the global notice-board (8) and informing the customer that the colour was processed (9). The actor is killed (10) meaning that is can perform no further computation.

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

```
1  act provider(colour) {
2    export history;
3    history = [];
4    bidding = false;
5    record(x) = history := history + [x];
6    bid(o:_) = {
7      o ← Bid(getColour(history,o.customer),self);
8      bidding := true
9    };
10   getColour([],c) = colour;
11   getColour(Changed(c,customer):_,customer) = c;
12   getColour(_:h,c) = getColour(h,c)
13   Time(_) when (nb.data <> []) and (bidding=false) →
14     bid(shuffle(nb.data));
15   Award → { record(S); bidding := false };
16   Failed → { record(F); bidding := false };
17   Change(customer) →
18     record(Changed(opp(colour),customer));
19   Change(_) → {};
20   Time(_) → {}
21 };
```

A provider checks the opportunities on the notice-board and makes a bid (13-14,6-9) based on the history (3). A monitor (defined below) adds terms of the form `Changed(colour,customer)` to the history of a provider (17,18) which is subsequently used to determine the colour of a bid (10-12) for that customer. The opportunity informs a provider of the outcome of a bid (15,16).

A monitor is created for each provider and customer pair. The predicate `hasTail` checks whether the three most recent opportunities produced by a supplied customer are the opposite colour of that for provider bids:

```
change(p,c) = λ() p ← Change(c)
```

```
  when hasTail(c.history,three(opp(p.colour)))

checkFs(change(p,c)) ← Monitor(p,0,new ϵ,λ() {})
```

The entire simulation is implemented as an ESL actor with the following behaviour:

```
act bidding(pc,d,P,C,results,monitors) {                          1
  theirMonitor = head(head(monitors));                            2
  myMonitor    = tail(head(monitors));                            3
  cs = [ new customer(Green,pc,4,4) | c ← 1..C ];                 4
  vs = [ new provider(Red) | v ← 1..P ];                          5
  ms = [ theirMonitor(v,c) | v ← vs, c ← cs ];                     6
  v  = let v = new vendor(Red)                                    7
       in {                                                       8
         for c in cs do myMonitor(v,c);                           9
         v                                                        10
       };                                                         11
  killAll() = {                                                   12
        for c in cs do kill(c);                                   13
        for v in vs do kill(v);                                   14
        for m in ms do kill(m);                                   15
        kill(v);                                                  16
  }                                                               17
  → print('Start Bidding');                                       18
  Time(t) when t = d → {                                          19
    if pc > 5                                                     20
    then                                                          21
      if tail(monitors) = []                                      22
      then { stopAll(); print(results) }                          23
      else {                                                      24
        killAll(v);                                               25
        resetTime(0);                                             26
        become bidding(0,d,P,C,[],tail(monitors))                 27
      }                                                           28
    else {                                                        29
      results := results + [count(S,v.history)];                  30
      killAll();                                                  31
      resetTime(0);                                               32
      become bidding(pc + 1,d,P,C,results,monitors)               33
    }                                                             34
  };                                                              35
  Time(_) → {}                                                    36
};                                                                37
```

The behaviour `bidding` is supplied with: `pc` the percentage likelihood of each customer changing their colour,`d` the duration of the simulation in clock-ticks`P` the number of providers other than ourselves, `C` the number of customers, `results` a list of numbers that represent our successes for each iteration of the simulation, `goals` a list of pairs of functions `m1:m2` where each function is applied to a vendor and a customer and created a monitor for that vendor with respect to the customer.

A `bidding` actor creates `C` customers with a `pc` chance of changing colour from the initial `Green` (line 4). The providers created on line 5 are all of the opposite colour and have monitors created on line 6 for each customer. The vendor that is designated *ourselves* is created on lines 7–11 and have a monitor created for each customer.

Each clock-tick the customer, vendor and monitor actors will behave autonomously. The simulation detects when the duration `d` has been reached on line 19. The aim is to run the simulation for different values of `pc` up to 5 and for different combinations of monitor creation. If the limit has been reached (line 20) then we stop when there are no more monitor combinations to try (line 22). Otherwise, if the limit has been reached and there are outstanding monitor combinations we reset (25–26) and try the rest of the monitors (line 27). If the value of `pc` has not reached 5 then the results are extended (line 30), the simulation is reset (lines 31–32) and the simulation is re-run with the next value for `pc`.

The bidding behaviour is used to generate results. We need 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:

```
act random(provider,customer) {
  Time(_) → {
    colour := opp(colour);
    vendor ← Change(customer)
  }
};
```

The monitors will be created dynamically in the simulation so appropriate functions are created that can be called as required:

```
randomMonitor(p,c)   = new random(p,c);
noMonitor(p,c)       = new inoperable;
adaptiveMonitor(p,c) =
  checkFs(change(p,c)) ← Monitor(p,0,new ϵ,λ() {})
monitors = [randomMonitor,noMonitor,adaptiveMonitor];
```

Finally, an ESL program must define a behaviour called `main` that has the same function as the class called `Main` in Java. The bidding simulation is to be run for `1000` clock-ticks with `10` providers and `2` customers:

```
act main {
  → new bidding(0,1000,10,2,[], me * them);
  Time(_) → {}
}
```

6.4 Results

Figure 9 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 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 9 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

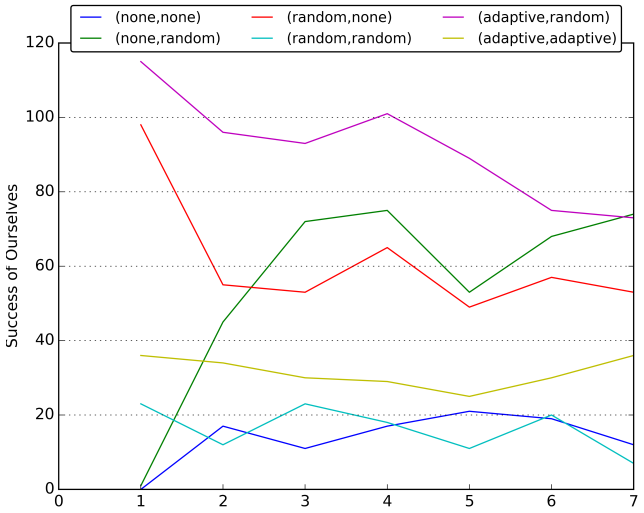

Figure 9: Results

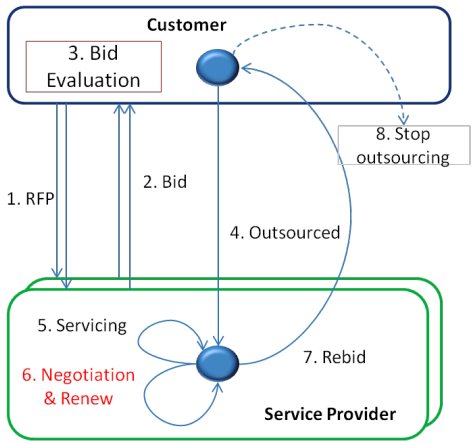

Figure 10: BPO State Machine

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. Evaluation

Section 4 has described a synthetic case study used to demonstrate a simulation-based approach to decision-making in terms of the conceptual model given in section 3 and the ESL language in 6.3. We have used the approach on a real-world industrial business processing outsourcing (BPO) case study as described in this section.

An outsourcing deal starts with a customer RFP inviting bids for a business process. An interested service provider's bid includes the number of full time employees (FTEs) to be deployed and billing rate (BR), its *capability ranking* (CR) via an independent agency, track record, any particular strengths such as number of capable employees, domain knowledge, *etc*. At the end of the bidding phase the customer evaluates all bids and identifies a service provider for outsourcing. Soft issues such as familiarity with the processes being outsourced, rapport with the service provider *etc.*, also play a part in selection. It is common observation that BPO contracts come up for renewal after 3-5 years where the contract may be renewed on modified terms (typically advantageous to the customer) or may go for rebidding. The factors that influence renewal include rate reductions, number and degree of escalations, quality perceptions. Figure 10 shows a representative state transition diagram of BPO business.

The BPO environment is modelled as a set of customer (C) units, business processes (BP) units and service provider (SP) units. The goals of customer units are to save costs and improve effectiveness; the goals of service provider units are to increase revenue, realisation (*i.e.*, revenue per FTE per hour) and size in terms of FTE numbers; the goals of business process units is to optimise its operational cost. All units encapsulate their characteristics as data. For example, service providers have data representing CR, BR, FTE count, market influence and delivery excellence expressed as a fixed value, a range of values or a probabilistic distribution. For example, CR is one of: *contender*, *challenger*, *visionary*, or *leader*, BR is a range (*e.g.*, $8 to $10 per hour per FTE) and quality is a probability distribution describing confidence of delivering *excellent*, *normal* and *below normal* services.

BPO events have a certain frequency and are stochastic in nature. When the customer raises an RFP, all interested service providers respond by selecting values from their characteristics variables. The bid evaluation function is a weighted aggregate of the various elements of RFP response and a random value to capture effect of inherent uncertainty. The service provider with the lowest computed value of a bid wins the outsourcing deal which gets executed by selected service provider. The decision to renew existing contract is modelled on similar lines. New processes may emerge as candidates for outsourcing are no longer needed. The existence of customers and service providers may also change with the time.

Units observe the visible history of other actors while reacting to an event and try to adapt if its goals are not achievable. The service providers are equipped with two negotiation levers: FTE productivity and billing rate. A BPO service provider must make decisions that address market share, internal growth, *etc*. In the interest of space we focus on question: With the current strategy how much market share will a provider capture and how much revenue opportunity will be lost? Other service providers are considered as competitors and the modelled provider will be referred to as SP.

The case study was represented using ESL and simulated for 10 years with goals for revenue, FTE numbers and realisation shown in blue, green and dotted red in figure 11.

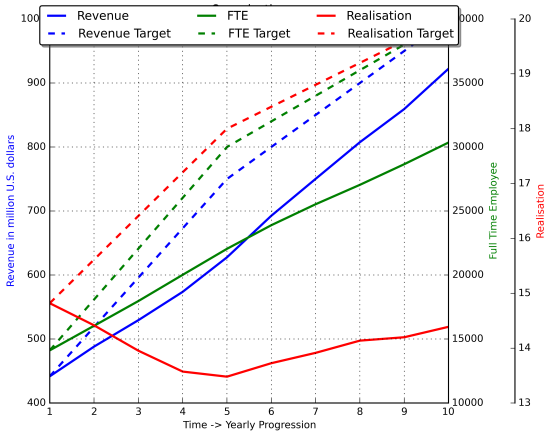

Figure 11: BPO Goal Measurement

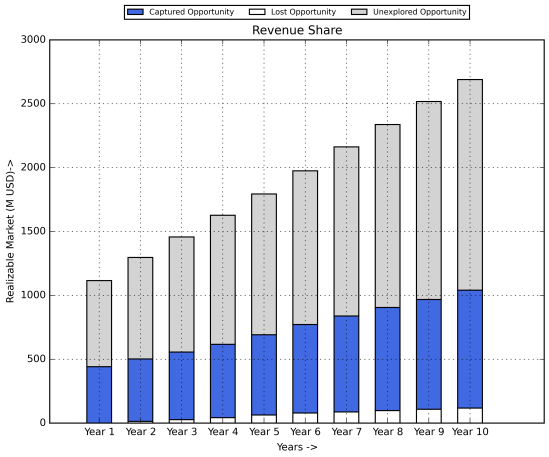

Figure 12: BPO Revenue Share

The SP revenue share is shown in figure 12 where the current SP revenue is $441.66M with a 60% unexplored market where grey represents the revenue of the competitors and white at the bottom of the bar indicates the SP revenue loss. SP revenue increases, the number of deployed FTEs is increases, but SP realisation factor decreases from 14.9 to 14.3 as shown using in blue, green and red lines in figure 11. After 10 years the total revenue increases from $1115.88M to $2571.44M, the SP revenue increases from $441.66M to $922.28M but market share is reduced from 40% to 35% of total revenue (figure 12). The graph in figure 12 also indicates a little revenue loss due to the loss in renewal process.

The graphs indicate many interesting aspects about SP: it is negotiating well for deal renewal, however there is a significant scope for improving revenue share for FTE and FTE realisation. This could be improved by winning more deals and by increasing the billing rate. However, winning more deals without compromising the billing rate is hard problem. The options such as increasing productivity with better resources and market influence should be exploited for an optimum solution and the ESL-based simulation can be used to explore the options.

8. Conclusion

This paper has defined an approach that aims to support organisational decision-making in terms of a conceptualisation of the key features and a technology platform. The approach is based on a computational view of an organisation that is encoded using actors. We have used a simple bidding scenario to demonstrate the features, including adaptive behaviour, and have described a real-world case study that has produced results that could be used to support management decisions.

All the code in the paper has been implemented in the ESL language which runs on a VM written in Java and which we claim is a suitable basis for building abstractions for organisational simulation including ideas from multi-agent systems including collaboration and negotiation.

Further work is planned in the following areas: ESL currently lacks a static type system which will help to engineer large-scale simulations. 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 [25] 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 encode features such as negotiation and collaboration.

References

[1] Joseph Barjis. Enterprise, organization, modeling, simulation: putting pieces together. *Proceeding of EOMAS*, 2008.

[2] Joseph Barjis and Alexander Verbraeck. The relevance of modeling and simulation in enterprise and organizational study. In *Enterprise and Organizational Modeling and Simulation*, pages 15–26. Springer, 2010.

[3] Jannis Beese, Kazem Haki, and Stephan Aier. On the conceptualization of information systems as socio-technical phenomena in simulation-based research. 2015.

[4] Peter Bernus, Kai Mertins, and Günter J Schmidt. *Handbook on architectures of information systems*. Springer Science & Business Media, 2013.

[5] Luis M Camarinha-Matos and Hamideh Afsarmanesh. Virtual enterprise modeling and support infrastructures: applying multi-agent system approaches. In *Multi-agent systems and applications*, pages 335–364. Springer, 2001.

[6] 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 Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 85–90. Society for Computer Simulation International, 2015.

[7] David Chen, Guy Doumeingts, and François Vernadat. Architectures for enterprise integration and interoperability: Past, present and future. *Computers in industry*, 59(7):647–659, 2008.

[8] Richard M Cyert, James G March, et al. A behavioral theory of the firm. *Englewood Cliffs, NJ*, 2, 1963.

[9] Guy Doumeingts, David Chen, Bruno Vallespir, P Fenie, and François Marcotte. Gim (grai integrated methodology) and its evolutions-a methodology to design and specify advanced manufacturing systems. In *Proceedings of the JSPE/IFIP TC5/WG5. 3 workshop on the design of information infrastructure systems for manufacturing*, pages 101–120. North-Holland Publishing Co., 1993.

[10] Mark S Fox. Issues in enterprise modelling. In *Systems, Man and Cybernetics, 1993.'Systems Engineering in the Service of Humans', Conference Proceedings., International Conference on*, pages 86–92. IEEE, 1993.

[11] Asif Qumer Gill and Muhammad Atif Qureshi. Adaptive enterprise architecture modelling. *Journal of Software*, 10(5): 628–638, 2015.

[12] Carl Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[13] M Iacob et al. State of the art in architecture support, archimate deliverable d3. 1. enschede, the netherlands: Telematica instituut, 2003.

[14] John Krogstie. Using eeml for combined goal and process oriented modeling: A case study. In *Proceedings of EMMSAD*, page 113, 2008.

[15] Ju-Sung Lee, Tatiana Filatova, Arika Ligmann-Zielinska, Behrooz Hassani-Mahmooei, Forrest Stonedahl, Iris Lorscheid, Alexey Voinov, J Gary Polhill, Zhanli Sun, and Dawn C Parker. The complexities of agent-based modeling output analysis. *Journal of Artificial Societies and Social Simulation*, 18(4):4, 2015.

[16] Donella H Meadows and Diana Wright. *Thinking in systems: A primer*. chelsea green publishing, 2008.

[17] Henry Mintzberg, Duru Raisinghani, and Andre Theoret. The structure of" unstructured" decision processes. *Administrative science quarterly*, pages 246–275, 1976.

[18] Andreas L Opdahl and Giuseppe Berio. A roadmap for ueml. In *Enterprise Interoperability*, pages 169–178. Springer, 2007.

[19] Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin, editors. *Domain Engineering, Product Lines, Languages, and Conceptual Models*. Springer, 2013. ISBN 978-3-642-36653-6. doi: 10.1007/978-3-642-36654-3. URL `http://dx.doi.org/10.1007/978-3-642-36654-3`.

[20] Alessandro Ricci, Gul Agha, Rafael H Bordini, and Assaf Marron. Special issue on programming based on actors, agents and decentralized control. *Science of Computer Programming*, 98:117–119, 2015.

[21] Wenan Tan, Wei Xu, Fujun Yang, Lida Xu, and Chuanqun Jiang. A framework for service enterprise workflow simulation with multi-agents cooperation. *Enterprise Information Systems*, 7(4):523–542, 2013.

[22] François Vernadat. Ueml: towards a unified enterprise modelling language. *International Journal of Production Research*, 40(17):4309–4321, 2002.

[23] Justin M Weinhardt and Jeffrey B Vancouver. Computational models and organizational psychology: Opportunities abound. *Organizational Psychology Review*, 2(4):267–292, 2012.

[24] Stephen A White et al. Business process modeling notation. *Specification, BPMI. org*, 2004.

[25] Shohei Yasutake and Takuo Watanabe. Actario: A framework for reasoning about actor systems. In *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE*, 2015.

[26] Eric Yu, Markus Strohmaier, and Xiaoxue Deng. Exploring intentional modeling and analysis for enterprise architecture. In *Enterprise Distributed Object Computing Conference Workshops, 2006. EDOCW'06. 10th IEEE International*, pages 32–32. IEEE, 2006.

[27] John A Zachman. A framework for information systems architecture. *IBM systems journal*, 26(3):276–292, 1987.