The Construction and Interrogation of Actor Based Simulation Histories

Tony Clark[1], Vinay Kulkarni[3], Balbir S. Barn[2], and Souvik Barat[3]

[1] Sheffield Hallam University, UK
[2] Middlesex University, UK
[3] Tata Consultancy Services Research, India

Abstract Large socio-technical systems are complex to comprehend in their entirety because information exchanges between system components lend an emergent nature to the overall system behaviour. Although Individual system component behaviour may be known at the outset, such components may exhibit uncertainty and further exacerbate issues of a priori prediction of the overall system behaviour. Multi-agent systems and the use of simulation is a possible recourse in such situations however, simulation results need to be correctly interpreted so as to nudge the overall system behaviour towards a desired objective. We propose a solution wherein the system is modelled as a set of actors exchanging messages, a simulation engine producing execution trace for an actor as its history, and a querying mechanism to identify patterns that may span across individual actor histories to ascertain property of the overall system. The proposed solution is evaluated using a representative sample from real life.

1 Introduction

Organisations and socio-technical systems can be simulated using Multi-Agent Systems [5,10,9] where such a model of an organisation is constructed in terms of independent goal-directed agents that concurrently engage in tasks, both independently and collaboratively forming an executable model that produce simulation results representing the organisation. A key aspect is analysis of simulation outputs [6]. An important reason for using agents for simulation is that the systems of interest are complex, for example because they involve socio-technical features [8]. As a result, the simulations exhibit emergent behaviour that must be analysed in a variety of initially unforeseen ways in order to construct models that explain features of interest. Such analysis may be used to validate the simulation model by comparing it against observable truth, may be used to sense-check the results of the simulation, or may be used to guide modifications to the simulation model so that it better meets the original requirements.

Our work on simulation has led to the design of a simulation workbench built around an actor language [11] called ESL [4]. The language ESL is used to construct agent-based simulation models that are run to produce histories. Each history contains a sequence of events produced by the behaviour of the agents in the simulation and thereby captures their emergent behaviour. A history is a temporal database of facts describing the states of, and communications between, agents in the simulation.

The challenge is to provide a suitable technology that allows the temporal database to be interrogated in order to support the sense-making described above. We describe an approach to the construction and interrogation of simulation histories. History construction is achieved by extending the standard operational actor model of computation [7] in order to capture temporal events during simulation execution. History interrogation is achieved by extending standard logic programming with temporal operators that are defined in terms of a supplied history containing time-stamped events.

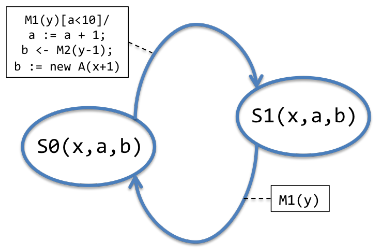
The approach is defined in terms of meta-interpreters for both construction, in section 2, and interrogation, in section 3. The approach has been implemented by extending the ESL language with history and interrogation features; the implementation is briefly described in section 4 and used to implement and interrogate the simulation of a retail shop.

2 History Construction

An agent-based simulation model consists of agents, each of which has local knowledge, goals and behaviour. Such a model can be operationalised in terms of the actor model of computation whereby each actor has an independent thread of control, has a private state and communicates with other actors via asynchronous messages to either update a local variable or change behaviour.

ESL is a text-based language that compiles to an actor-based VM. Actor behaviour can be represented using state machines as shown in figure 1 where figure 1(a) is a state machine representation of the ESL actor definition in figure 1(b). The rest of this paper will use state machine representation for actors with transitions between states labelled `M[Q]/A` where `M` is a message, `Q` is a guard condition, and `A` is an action. Note that not all ESL programs can be represented as state machines, however this is sufficient for the behaviours considered in this paper.

Figure 2 shows a model of actor program states that is suitable to define the construction of histories. The model is expressed as a collection



(a) Actor Machine

```
act name(x:Int)::B {
  export a;            // public interface.
  a::Int = 0;          // public variable.
  b::A = new A(x);     // initialised local.
  s::State = S0;       // state variable.
  M1(y::Int)           // message handler.
    when s = S0 and
         a < 10        // message guard.
  → {                  // message action.
    a := a + 1;        // update variable.
    s := S1;           // change state.
    b ← M2(y-1);       // send message.
    b := new A(x+1)    // actor creation.
  }
  M1(y::Int) when s = S1 →
    s := s0
}
```

(b) ESL Actor

Figure 1. Example Actor Behaviour

```
type Id        = Int;
type Time      = Int;
type Behaviour = [Handler(Str,[Command])];
data Command   = Send(Id,Str) | Update(Str) | Block([Command]) | Become(Str) | New(Str);
type Queue     = [Message(Str)];
type Actor     = Machine(Id,[Command],Behaviour,Queue,Time);
type DB        = [Fact(Time,Command)];
type ESL       = State(Set{Actor},DB,Time);
```

Figure 2. An Abstract Model of Actor Programs

of ESL type definitions. All details relating to how variables are managed within an actor has been elided and can be understood in terms of standard operational models of programming languages. The term `State(a,db,t)` represents the state of an executing actor configuration where `a` is a set of actors, `db` is a history database, and `t` is the current time. We are interested in specifying how `db` is constructed through the conventional operational semantics of actors. This is achieved by defining a single-step operational semantics: `s2=step(s1)` where system state `s1` performs an execution step in order to become state `s2`. The complete execution of a system can be constructed by repeated application of `step`.

It is useful in simulations to be able to refer to global time via a clock. This can be used to schedule future computation or to allow actors to perform joint actions. To support the notion of global time, each actor in our operational model receives a regular `Time` message where each global time unit is measured in machine instructions. This mechanism seems to be fair and, although is not related to real-time, provides a basis for time that is useful in a simulation. To support this, each actor has an instruction count that, when reached, halts the actor. When all actors have been halted, global time is increased, and a message is sent to all actors.

A history database `db` is created from an initial configuration of actors `a` by repeated application of `step` until a terminal state is achieved such that all actors are exhausted and have no pending messages, `db=run(a)`:

```
isTerminal :: (ESL) → Bool;
isTerminal(State(a,_,_)) = allTerminated(a);

allTerminated :: (Set{Actor}) → Bool;
allTerminated(set{})       = true;
allTerminated(set{a | as}) = isTerminated(a) and allTerminated(as);

isTerminated::(Actor) → Bool;
isTerminated(Machine(_,[],_,[],_)) = true;
isTerminated(_)                    = false;

run::([Actor]) → DB;
run(a) = repeat(step,State(a,[],0),isTerminal)
```

Summarising, a history is a collection of facts of the form `Fact(t,f)` where `t` is a timestamp and `f` is a term representing an actor execution step. The next section describes how the histories are interrogated using a temporal extension to standard relational programming.

3 Interrogation of Histories

Simulations consist of many autonomous agents with independent behaviour and motivation. Consequently, system behaviour is difficult to predict. Furthermore, the highly concurrent nature of the actor model of computation makes the simulation difficult to instrument in order to detect situations of interest. Therefore, we propose the construction of simulation histories as a suitable approach to simulation interrogation. Given such a history we would like to construct queries that determine whether particular relationships exist, where the relationships are defined in terms of the key features of actor computation. Logic programming, as exemplified by Prolog, would seem to be an ideal candidate for the construction of such queries, however standard Prolog does not provide intrinsic support for expressing the temporal features of the histories that are generated in section 2.

This section describes a variation on standard Prolog that incorporates both temporal features and simulation histories. The logic programming language is presented as a meta-interpreter written in a nontemporal subset of itself that is a statically typed version of Prolog:

```
append[T] :: ([T],[T],[T]);
append[T]([],l,l) ← !;
append[T]([x|l1],l2,[x|l3]) ←
  append[T](l1,l2,l3);

length[T] :: ([T],Int);
length[T]([],0) ← !;
length[T]([h|t],n) ←
  length[T](t,m), n := m + 1;

member[T] :: (T,[T]);
member[T](x,[x|_]);
member[T](x,[_|l]) ←
  member[T](x,l);
```

```
subset[T] :: ([T],[T]);
subset[T]([],[]);
subset[T]([x|l],[x|s]) ←
  subset[T](l,s);
subset[T](l,[_|s]) ←
  subset[T](l,s);

lookup[V] :: (Str,V,[Bind(Str,V)]);
lookup[V](n,v,[Bind(n,v) | _]);
lookup[V](n,v,[_|env]) ←
  lookup[V](n,v,env);
```

The examples above are standard Prolog rules that have been elaborated with static type information that is checked by the ESL Workbench before execution. The rules `length` and `member` use parametric polymorphism over the type `T` of elements in a list. The rule `lookup` is parametric with respect to the type of the bindings in the environment list.

The rules shown above do not reference simulation history facts. In order to support rules over histories we introduce new rule features based on temporal logic. The following data type `Body` describes the elements that can occur in a rule body:

```
data Value =  Term(Str,[Value]) | Var(Str) | I(Int) | S(Str);

data Body = Call(Str,[Value]) | Is(Str,Value) | Start | End | Next([Body]) | Prev([Body])
          | Always([Body]) | Eventually([Body]) | Past([Body]) | Forall([Body],Value,Value
     );
```

The terms `Call` and `Is` represent standard Prolog body elements; all other elements are extensions to standard Prolog. The extensions all relate to *current time* which is the time-stamp associated with the facts in the history:

```
type Time  = Int;
type Entry = Fact(Time,Str,[Value]);
type DB    = [Entry];
```

Elements `Start` and `End` are satisfied when the current time is `0` and the end of the history respectively. An element `Next(es)` is satisfied when the elements `es` are satisfied in current time $+1$, similarly `Prev(es)` in current time -1. An element `Always(es)` is satisfied when the elements `es` are satisfied at all times from now, similarly `Past(es)` all times before now. Element `Eventually(es)` is satisfied when `es` are satisfied at some time in the future.

Figure 3 defines a meta-interpreter for the history query language. Given a query `q(v1,...,vn)`, a program `prog`, a database `db` and a history end time `t`, the query is satisfied when `call(0,t,db,'q',[v1,...,vn],prog)` is satisfied with respect to the definitions given in figure 3.

The meta-interpreter is based on a standard operational semantics for Prolog that is extended with features to process the supplied database

```
type Prog = [Rule(Str,[Value],[Body])];
type Env = [Bind(Str,Value)];

rule::(Str,Rule(Str,[Value],[Body]),Prog);
rule(n,Rule(n,as,body),[Rule(n,as,body)|prog]);
rule(n,r,[_|prog]) ← rule(n,r,prog);

call ::(Time,Time,DB,Str,[Value],Prog);
call(time,eot,db,n,vs,prog) ←
  member[Entry](Fact(time,n,vs),db);
call(time,eot,db,n,vs,prog) ←
  rule(n,Rule(n,as,body),prog),
  length[Value](vs,l),
  length[Value](as,l),
  matchs(as,vs,[],vars),
  trys(time,eot,db,body,vars,_,prog);

eval :: (Value,Env,Value);
eval(Term('+',[left,right]),env,I(i)) ←
  eval(left,env,lv), eval(right,env,rv),
  i := lv + rv;
eval(Var(n),env,v) ← lookup[Value](n,v,env);
eval(I(i),env,I(i));
eval(S(s),env,S(s));

matchs :: ([Value],[Value],Env,Env);
matchs([],[],env,env);
matchs([a|as],[v|vs],in,out) ←
  match(a,v,in,in'),
  matchs(as,vs,in',out);

match :: (Value,Value,Env,Env);
match(Term(n,vs),Term(n,vs'),in,out) ←
  matchs(vs,vs',in,out);
match(Var(n),v,e,e) ← lookup[Value](n,v,e);
match(Var(n),v,env,[Bind(n,v) | env]);
match(I(i),I(i),env,env);
match(S(s),S(s),env,env);

derefs :: ([Value],[Value],Env,Env);
derefs([],[],env,env);
derefs([v|vs],[v'|vs'],in,out) ←
  deref(v,v',in,in'),
  derefs(vs,vs',in',out);
```

```
deref :: (Value,Value,Env,Env);
deref(Term(n,vs),Term(n,vs'),in,out) ←
  derefs(vs,vs',in,out);
deref(Var(n),v,e,e) ← lookup[Value](n,v,e),!;
deref(Var(n),v,env,[Bind(n,v) | env]);
deref(I(n),I(n),env,env);
deref(S(s),S(s),env,env);

trys :: (Time,Time,DB,[Body],Env,Env,Prog);
trys(_,_,_,[],env,env,prog);
trys(time,eot,db,[e|es],in,out,prog) ←
  try(time,eot,db,e,in,in',prog),
  trys(time,eot,db,es,in',out,prog);

try :: (Time,Time,DB,Body,Env,Env,Prog);
try(t,eot,db,Call(n,vs),in,out,p) ←
  derefs(vs,vs',in,out),call(t,eot,db,n,vs',p);
try(eot,eot,db,End,env,env,prog);
try(0,_,db,Start,env,env,prog);
try(eot,eot,db,Next(es),env,env,prog) ← !, false;
try(time,eot,db,Next(es),in,out,prog) ←
  time' := time + 1;
  trys(time',eot,db,es,in,out,prog);
try(0,eot,db,Prev(es),env,env,prog) ← !, false;
try(t,eot,db,Prev(es),in,out,prog) ←
  t' := t - 1, trys(t',eot,db,es,in,out,prog);
try(eot,eot,db,Always(es),env,out,prog) ← !;
try(time,eot,db,Always(es),in,out,prog) ←
  trys(time,eot,db,es,in,in',prog),
  time' := time + 1,
  try(time',eot,db,Always(es),in',out,prog);
try(eot,eot,db,Eventually(es),in,out,prog) ←
  !, false;
try(time,eot,db,Eventually(es),in,out,prog) ←
  trys(time,eot,db,es,in,out,prog);
try(time,eot,db,Eventually(es),in,out,prog) ←
  time' := time + 1,
  try(time',eot,db,Eventually(es),in,out,prog);
try(0,eot,db,Past(es),in,out,prog) ← !, false;
try(time,eot,db,Past(es),in,out,prog) ←
  trys(time,eot,db,es,in,out,prog);
try(time,eot,db,Past(es),in,out,prog) ←
  time' := time - 1,
  try(time',eot,db,Past(es),in,out,prog);
try(time,eot,db,Is(n,exp),env,env,prog) ←
  eval(exp,env,v), lookup[Value](n,v,env);
```

Figure 3. Meta-Interpreter For History Query Language

(the definition of `Forall` is omitted, but is consistent with standard Prolog). The rule `call` is used to process a body element of the form `Call(n,vs)` where `n` is the name of a fact and `vs` are the arguments. Conventional Prolog processes such a call using the definition of `call` defined on lines 11 – 16 where a rule named `n` with an appropriate arity is found in the program and is supplied with the argument values using `matchs`.

Figure 3 extends conventional Prolog rule calling by allowing the fact to be present in the history *at the current time* (lines 9-10). Therefore, the facts in the history become added to the facts that can be conventionally deduced using the rules.

The semantics of the additional types of body elements are processed by the `try` rule (lines 89–115) by modifying the value of the current time appropriately, For example, the rule for `Next` (lines 91–94) fails if the end of the history has been reached, otherwise it attempts to satisfy the elements `es` after incrementing the current time by `1`.

An example rule is `customers` where `customers(cs)` is satisfied when `cs` is a list of all the customer actor identifiers in the history:

```
1 customers :: ([Int]);
2 customers([]) ← end, !;
3 customers(cs) ←
4   forall[actor(a,'customer',_)](a,cs'), next[customers(cs'')],
5   append[Int](cs',cs'',cs);
```

Line 2 defines that there can be no customers if we are at the end of the history. Lines 3–5 define how to extract the customer identifiers from this point in the history: line 4 uses **forall** to match all database facts of the form `actor(a,'customer',_)` where this fact has been added to the database when a new customer actor is created. Then, **next** is used to advance the time so that `cs''` are all the customer actors from this point onwards.

4 Evaluation

We have described an approach to constructing and interrogating agent-based simulation histories. History construction is defined by identifying the key operational features of the actor model of computation and then extending an actor interpreter with a database of temporal facts, each of which is based on a key operational feature. History interrogation is performed by extending a standard Prolog interpreter with language features for processing the facts in a temporal database. The approach has been implemented within ESL where we have extend the ESL VM to produce histories and then integrated a query language, a Prolog VM extended with features to process histories (as defined in section 3).

This section provides an overview of the implementation, describes a simulation that produces a history, and shows how the query language can interrogate the history.

4.1 Implementation

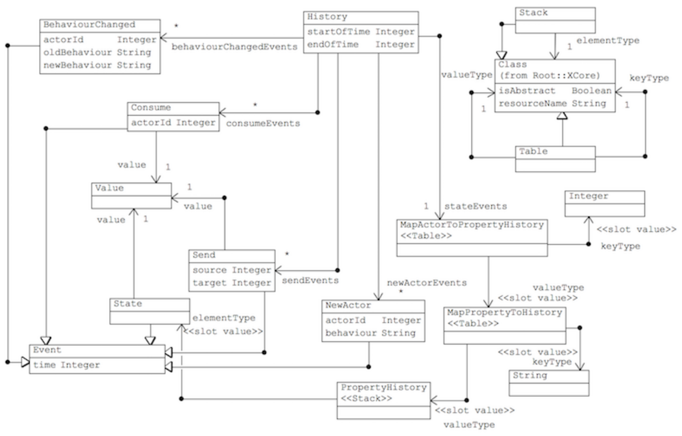


Figure 4. History Implementation

ESL compiles to a virtual machine that is similar to the Java VM where actors are the equivalent of objects. The ESL VM manages a history that is automatically updated when actors are created, change behaviour, updated and send messages. Given the scale of simulations, it is important that the history is represented as efficiently as possible, both for construction and interrogation. Figure 4 shows the Java classes that are used to implement the history.

Once constructed, a history is interrogated by the ESL query language that compiles to a VM which is based on a standard Warren Abstract Machine [14].

In a WAM, Choice-points are represented as pointers into a frame on a *fail stack*. The ESL query language VM extended the fail stack with a new form of fail-frame that indexes the history. For example, a query `actor (a,b,t)` relates to actor creation events where `a` is the unique identifier, `b` is the behaviour, and `t` is the time, any of which may be logic variables. Therefore the query potentially represents a choice point. The VM creates a fail-frame corresponding to the choice point and indexes the 0-th fact that matches the supplied values, if this subsequently fails by returning to the fail-frame, then the index is incremented. A history is any Java object that implements the following interface which is used by the call and fail mechanism within the VM:

```
public interface DB {
  boolean hasFact(String name,int arity,int time,int index,Machine machine);
  Object getFactArg(String name,int arity,int time,int index,int argNumber,Machine machine
      );
  int endOfTime();
}
```

The method `hasFact` is used by the query VM when performing a call to determine whether the hitsory database contains a fact corresponding to the call. The first time this occurs the supplied `index` is `0`. If this succeeds then the VM constructs a fail-frame that captures the name, arity, time and the next index: `1`. Each time the fail-frame is used for backtracking, the index is incremented. The method `getFactArg` is used to extract the individual fact arguments and subsequently unify them in the query VM.

4.2 Case Study Application

In this paper we use a case study that is based on existing work on agent-based organisation simulations [12] and as such is seen as a reference example. The case study will be used to demonstrate the construction of an agent model that produces a history and the subsequent interrogation via a query. The query is chosen to demonstrate the utility of logic programming using the ESL query language and will also be analysed in terms of its efficiency based on the implementation described in the previous section.

Case study description: A shop provides stock on the shop-floor. Customers enter the shop and may browse until they either leave, seek help or decide on a purchase. Items must be purchased at tills and multiple customers are serviced via a queue. Shop assistants may be on the shop-floor, helping a customer or may service a till. A queueing customer can only make a purchase when they reach the head of a queue at a serviced till. A customer who waits too long at an unserviced till, or for whom help is not available, will become unhappy and leave the shop. The shop would like to minimise unhappiness. In addition the shop owner has noticed that stock is going missing. A criminal gang is known to be operating in the area. Typically they operate by engaging all the assistants in a shop whilst one of the gang members leaves the shop without paying for the goods.

An ESL simulation can be modelled using structure and behaviour diagrams which are variations on class and state machine diagrams respectively. The structure of the shop simulation is shown in figure 5 in

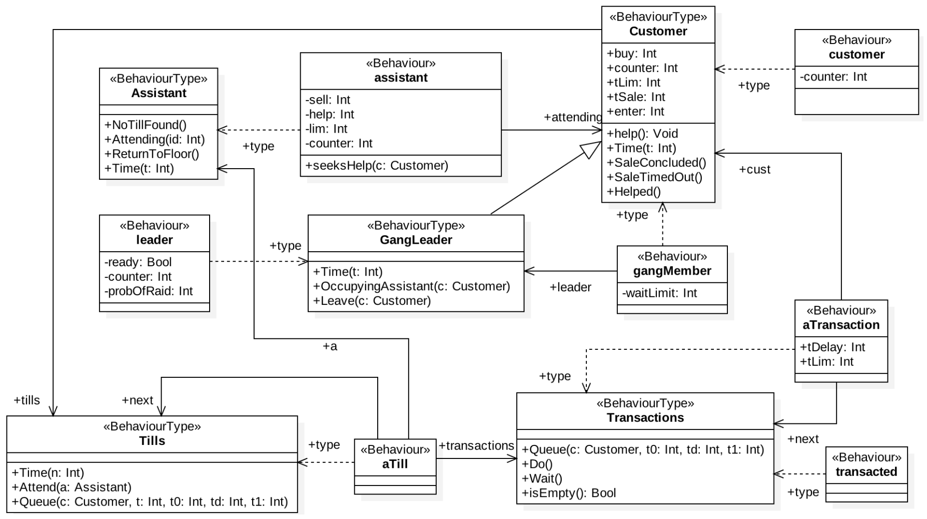


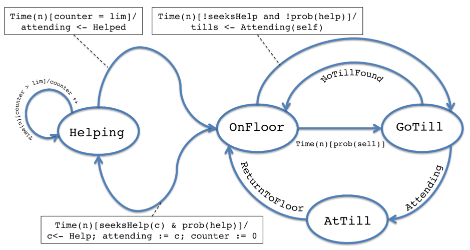
Figure 5. Structure of Shop Actors

which we make the distinction between behaviour types and behaviours. The difference between these two concepts corresponds to the difference between software module types and modules in that a behaviour type is an interface definition and a behaviour provides an implementation of the interface. There may be many actors that are instances of the same behaviour.

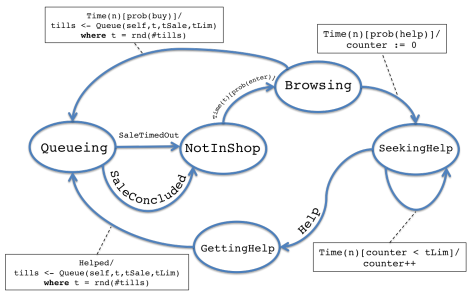
The model shown in figure 6 organises a shop as a collection of customers, assistants and tills. Each till is an actor that is associate with a sequence of transactions, also represented as actors. Each transaction may be serviced by sending it a `Do` message, or may be delayed, by sending it a `Wait` message. When a transaction is successfully terminated the corresponding actor will change its behaviour to `transacted` which means that it acts as a proxy for the next transaction in the queue.

In addition to normal customers, the simulation represents a special type of customer called `gangMember`. These unscrupulous actors are organised by a gang leader to occupy sales assistants in order that the leader can steal merchandise. We will not explicitly consider the behaviour of the gang leader, however we will be interested in examining the simulation history in order to detect potential gang member behaviour.

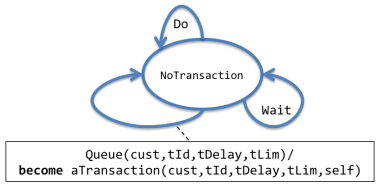
Figure 6 shows the behaviour definitions for the shop simulation actors. The simulation is driven by `Time` messages and all actors implement a `Time` transition for all states; the empty transitions arising from `Time` are omitted. Time is used in figure 6(g) to show how customers waiting at a till can time out and ultimately leave the shop. The value supplied to a



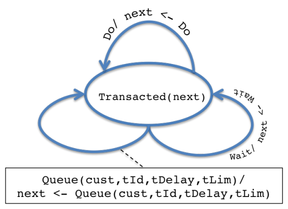
(a) Assistant Behaviour



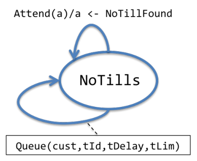
(b) Customer Behaviour


Do
NoTransaction
Wait
Queue(cust,tId,tDelay,tLim)/
become aTransaction(cust,tId,tDelay,tLim,self)

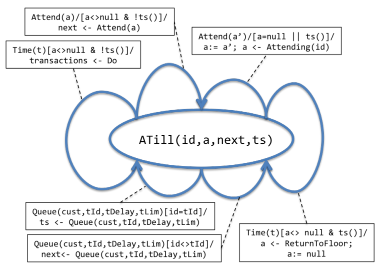
(c) No Transactions Behaviour


Do/ next <- Do
Transacted(next)
Wait/ next <- Wait
Queue(cust,tId,tDelay,tLim)/
next <- Queue(cust,tId,tDelay,tLim)

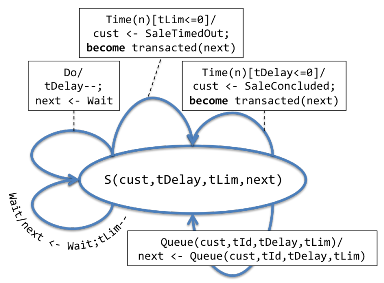
(d) Transacted Behaviour



(e) No Tills Behaviour


Attend(a)/[a<>null & !ts()]/
next <- Attend(a)
Time(t)[a<>null & !ts()]/
transactions <- Do
Attend(a')/[a=null || ts()]/
a:= a'; a <- Attending(id)
ATill(id,a,next,ts)
Queue(cust,tId,tDelay,tLim)[id=tId]/
ts <- Queue(cust,tId,tDelay,tLim)
Queue(cust,tId,tDelay,tLim)[id<>tId]/
next<- Queue(cust,tId,tDelay,tLim)
Time(t)[a<> null & ts()]/
a <- ReturnToFloor;
a:= null

(f) A Till Behaviour

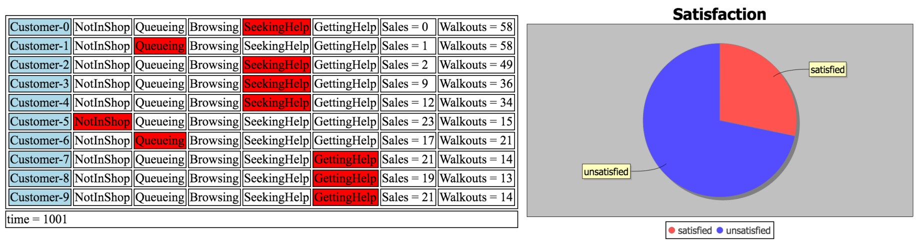


(g) A Transaction Behaviour

Figure 6. Shop Actor Behaviour

transaction is `tLim` which determines how long a customer is prepared to wait without a sale being concluded.

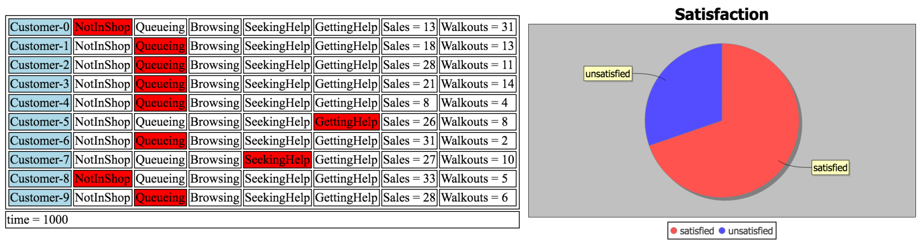
Customers who leave the shop because they have waited too long to be serviced at a till are deemed to be *unhappy*. the shop is interested in how to organise its assistants, sales and floor-walking strategies in order to minimise unhappy customers. Figure 7 shows the ESL Workbench output from two different simulation configurations. The Workbench can generate a display based on actor states during the simulation, figure 7 shows the final output. Figure 7(a) shows the result of 10 customers, 5 tills and 3 sales assistants where roughly 75% of the customers are left unhappy. The number of assistants has been increased to 5 in figure 7(b) where the situation is reversed. note that the simulation has many



Customer-0	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 0	Walkouts = 58
Customer-1	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 1	Walkouts = 58
Customer-2	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 2	Walkouts = 49
Customer-3	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 9	Walkouts = 36
Customer-4	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 12	Walkouts = 34
Customer-5	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 23	Walkouts = 15
Customer-6	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 17	Walkouts = 21
Customer-7	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 21	Walkouts = 14
Customer-8	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 19	Walkouts = 13
Customer-9	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 21	Walkouts = 14

time = 1001

(a) Shop with 10 Customers, 5 Tills and 3 Assistants



Customer-0	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 13	Walkouts = 31
Customer-1	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 18	Walkouts = 13
Customer-2	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 28	Walkouts = 11
Customer-3	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 21	Walkouts = 14
Customer-4	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 8	Walkouts = 4
Customer-5	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 26	Walkouts = 8
Customer-6	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 31	Walkouts = 2
Customer-7	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 27	Walkouts = 10
Customer-8	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 33	Walkouts = 5
Customer-9	NotInShop	Queueing	Browsing	SeekingHelp	GettingHelp	Sales = 28	Walkouts = 6

time = 1000

(b) Shop with 10 Customers, 5 Tills and 5 Assistants

Figure 7. Shop Simulation Output

random elements and therefore each run is different, but the two outputs characterise the relative differences.

5 Related Work

Simulation of multi agent systems (MAS) is quite close to the problem we are addressing. Several simulation approaches have been suggested but none seem to address the querying need. In [2], Bosse *et al.* present a generic language for the formal specification and analysis of dynamic properties of MAS. This language supports the specification of both qualitative and quantitative aspects, and therefore subsumes specification languages based on differential equations. However, this is not an executable language. In fact, it has been specialised for simulation domain leading to LEADSTO language[3]. The LEADSTO is essentially a declarative order-sorted temporal language extended with quantitative notions (like integer, and real). Time is considered linear, continuous, described by real values. Dynamics is modelled as direct temporal dependencies between state properties in successive states. Though quite useful in specifying simulations of dynamic systems, it does not provide any help in querying the resultant behaviour. Bosse *et al.* further propose a multi-agent model for mutual absorption of emotions to investigate emotion as a collective property of a group using simulation[1]. It provides mathematical machinery to validate a pre-defined property over simulation trace. However, there is no support for temporal logic operators in specifying a property.

Sukthankar and Sycara, in contrast, propose an algorithm to recognize team behaviour from spacio-temporal traces of individual agent behaviours using dynamic programming techniques[13]. Though quite close as regards the solution space, this result lays strong emphasis on team behaviour thus hinting collaboration. Our proposed machinery is neutral to the nature of environment - collaborative or competitive or both. Vasconcelos *et al.* present mechanisms based on first-order unification and constraint solving techniques for the detection and resolution of normative conflicts concerning adoption and removal of permissions, obligations and prohibitions in societies of agents[15]. Though we too propose Prolog based realisation for querying generated histories, our focus is more quantitative.

6 Conclusion

In this paper, we have sought to address the challenge of analysing simulations that exhibit emergent behaviour for unforseen conditions. Our basic architectural approach to this problem was to extend our actor based language ESL with a Prolog based meta interpreter that is able to execute queries over simulation histories. The extensions to ESL and the Prolog variant include new rule features based on temporal logic that reference simulation history facts that are both global across the system or local to specific actors within the system. The technology was evaluated using a reference case study. While we recognise the limitations of such an evaluation approach, the case study is sufficiently realistic in generating rich and complex simulation histories. Our long-term goal for this research is the construction of robust, scaleable, encompassing technology that complex dynamic decision making situations that deal with uncertainty and emergent behaviour.

References

1. Tibor Bosse, Rob Duell, Zulfiqar A Memon, Jan Treur, and C Natalie Van Der Wal. Multi-agent model for mutual absorption of emotions. *ECMS*, 2009:212–218, 2009.
2. Tibor Bosse, Catholijn M Jonker, Lourens Van der Meij, Alexei Sharpanskykh, and Jan Treur. Specification and verification of dynamics in cognitive agent models. In *IAT*, pages 247–254. Citeseer, 2006.
3. Tibor Bosse, Catholijn M Jonker, Lourens Van Der Meij, and Jan Treur. Leadsto: a language and environment for analysis of dynamics by simulation. In *German Conference on Multiagent System Technologies*, pages 165–178. Springer, 2005.

4. Tony Clark, Vinay Kulkarni, Souvik Barat, and Balbir Barn. Actor monitors for adaptive behaviour. In Ravi Prakash Gorthi, Santonu Sarkar, Nenad Medvidovic, Vinay Kulkarni, Atul Kumar, Padmaja Joshi, Paola Inverardi, Ashish Sureka, and Richa Sharma, editors, *Proceedings of the 10th Innovations in Software Engineering Conference, ISEC 2017, Jaipur, India, February 5-7, 2017*, pages 85–95. ACM, 2017.
5. Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 128–135. IEEE, 1998.
6. Paul A Fishwick. Computer simulation: growth through extension. *Transactions of the Society for Computer Simulation*, 14(1):13–24, 1997.
7. Carl Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.
8. Tom McDermott, William Rouse, Seymour Goodman, and Margaret Loper. Multi-level modeling of complex socio-technical systems. *Procedia Computer Science*, 16:1132–1141, 2013.
9. Geoffrey P Morgan and Kathleen M Carley. An agent-based framework for active multi-level modeling of organizations. 2016.
10. David V Pynadath and Milind Tambe. An automated teamwork infrastructure for heterogeneous software agents and humans. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):71–100, 2003.
11. 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.
12. Peer-Olaf Siebers and Uwe Aickelin. A first approach on modelling staff proactiveness in retail simulation models. *J. Artificial Societies and Social Simulation*, 14(2), 2011.
13. Gita Sukthankar and Katia Sycara. Simultaneous team assignment and behavior recognition from spatio-temporal agent traces. In *AAAI*, volume 6, pages 716–721, 2006.
14. Terrance Swift and David Scott Warren. An abstract machine for slg resolution: Definite programs. In *ILPS*, pages 633–652. Citeseer, 1994.
15. Wamberto W Vasconcelos, Martin J Kollingbaum, and Timothy J Norman. Normative conflict resolution in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 19(2):124–152, 2009.