Constructing and Interrogating Actor Histories

Tony Clark

Sheffield Hallam University, UK

Vinay Kulkarni

Tata Consultancy Services Research, India

Souvik Barat

Tata Consultancy Services Research, India

Balbir Barn

Middlesex University, UK

Abstract Complex systems, such as organizations, can be represented as executable simulation models using actor-based languages. Decision-making can be supported by system simulation so that different configurations provide a basis for *what-if* analysis. Actor-based models are expressed in terms of large numbers of concurrent actors that communicate using asynchronous messages leading to complex non-deterministic behaviour. This chapter addresses the problem of analyzing the results of model executions and proposes a general approach that can be added to any actor-based system. The approach uses a logic programming language with temporal extensions to query execution traces. The approach has been implemented and is shown to support a representative system model.

Introduction

Organizations and systems can be simulated using Multi-Agent Systems [17, 33, 34]. This approach builds a model of an organisation in terms of independent goaldirected agents that concurrently engage in tasks, both independently and collaboratively. Collections of such agents form an executable model that produces results. Fishwick [18] notes the key features of computer simulation to be modelling, execution and analysis of output. An important reason for using agents for simulation is that the systems of interest are complex, for example because they involve socio-technical features [32]. As noted in [31]: *humans use patterns to order the world and make sense of things in complex situations*, and it follows that pattern-based analysis may be used to analyze an agent-based simulation model. This chapter addresses the problem of how to create and analyze agent-based simulations.

Our work on simulation for decision support [4, 5, 6, 15, 29, 30] has led to the design of a simulation workbench built around an actor language [36] called ESL [14]. 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 actors in the simulation and thereby captures their emergent behaviour. A history is a temporal database of facts describing the states of, and communications between, actors in the simulation.

The research question that we seek to investigate is: *what general-purpose mechanism can be devised to generate actor histories and then analyze them using temporal queries*? *Where possible the mechanism should be applicable to existing actor-based technologies and use standard query languages with minimal extensions*. We take a design-based approach to this research by taking an existing technology and implementing extensions that support the production and analysis of actor-histories.

The current state of the practice of analysis of simulation results is predominantly based on the visualization and human interpretation. We propose a programmatic approach to the construction and interrogation of simulation histories. History construction is achieved by extending the standard operational actor model of computation [16, 21] 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.

Our contribution is a pair of general-purpose languages for the construction and subsequent interrogation of agent-based execution histories. In both cases conventional computational models are extended with novel mechanisms for histories: an interpreter for actor languages is extended with primitives for history production and a Prolog meta-interpreter is extended to support history interrogation.

The proposed approach is evaluated in terms of its completeness, viability and validity. Completeness follows from the universality of the actor model of computation, from our claim that our actor interpreter generates all key computational events, and from our claim that the query language can express all queries of interest. Viability is demonstrated by our implementation of a simulation workbench and validity is demonstrated by showing how the implementation supports the construction and interrogation of a representative simulation. The conclusion discusses threats to validity, how we plan to address them, and outlines next steps.

Related Work

The use of Multi-Agent Systems (MAS) for system simulation has been explored by a number of researchers, for example in [10, 13 20, 39, 39], where agent simulation models range from collection of numerical equations to sophisticated behaviours encoded using a BDI-based approach. Researchers have developed approaches for the definition and analysis of simulation properties. In [10], Bosse *et al.* present a generic language for the formal specification and analysis of dynamic properties of MAS that supports the specification of both qualitative and quantitative features, and therefore subsumes specification languages based on differential equations. However, this is not an executable language like that presented in this chapter. It has been specialized for simulation and has produced the LEADSTO language [11] that is a declarative order-sorted temporal language where time is described by real numbers and where properties are modelled as direct temporal dependencies between 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 [9]. It provides mathematical machinery to validate a predefined property over simulation trace. However, there is no support for temporal logic operators.

Sukthankar and Sycara propose an algorithm to recognize team behaviour from spacio-temporal traces of individual agent behaviours using dynamic programming techniques [40], but do not address general behavioural properties arising from simulations. 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 [41].

The tool described in [43] produces static diagrams of agent communication topologies using a *society* tool. The authors support off-line video-style replay facilities with forward and backward video modes as a powerful debugging aid. However there is no programmatic mechanism for interrogating the histories.

Temporal logics have been used to specify the behaviour of MAS [12] and to analyze the specification for properties using theorem proving or model checking. Our approach uses a similar collection of temporal operators, however we are applying the behaviour specifications *post-hoc* in order to investigate whether a given behaviour took place, rather than to express required behaviour or to analyze properties such as consistency *etc*.

The need to analyze agent-based simulations is related to the field of agent-based system testing. As noted in [42] *attempting to obtain assurance of a system's correctness by testing the system as a whole is not feasible* and *there is, at present, no practical way of assuring that they will behave appropriately in all possible situations*. Our approach is intended to be a pragmatic partial solution that is used selectively in collaboration with a domain expert. Queries can be used to test whether properties exist in particular histories, and could help scope the use of more formal static methods.

Using temporal operators to construct queries over databases is a standard approach. Queries can be encoded in logic [7] or in SQL extensions [3], although as noted in [26]: *Much of real-life data is temporal in nature, and there is an increasing application demand for temporal models and operations in databases. Nevertheless, SQL:2011 has only recently overcome a decade-long standstill on standardizing temporal features. As a result, few database systems provide any temporal support, and even those only have limited expressiveness and poor performance*. A logic provides increased expressive power over an SQL-like language at the cost of requiring a theorem prover or a model checker with the associated scalability issues. Our approach, using logic-programming, aims to be more expressive than SQL whilst addressing scalability.

Managing temporal data is becoming increasingly important for many applications [25, 28]. Our work is related to process mining from the event logs that are created by enterprise systems [35] where queries can be formulated in terms of a temporal logic and applied to data produced by monitoring real business systems. Other researchers have proposed adding temporal operators to query languages in order to process knowledge bases [8]. We have extended these approaches in the context of simulation histories by showing how to encode them in an operational query language.

The nature of agent-based systems leads to high levels of concurrency with an associated challenge regarding system analysis when behaviour is not as expected. As reported in [27]: *in order to locate the cause of such behaviour, it is essential to explain how and why it is generated* [22]. If histories are linked to source code then queries can be used as part of an interactive debugger, extending the approach described in [27]. In conclusion, there have been a number of approaches in the literature that analyze actor-based systems, some of them are based on histories, but none provide the expressive power of the language defined in this chapter. Furthermore, we show how any actor-based language can be extended to produce histories that are suitable for interrogation by queries written in an extended logic programming language.

Actors and Histories

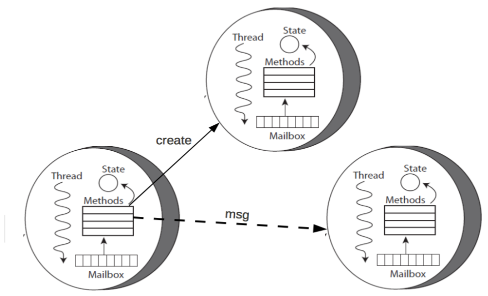
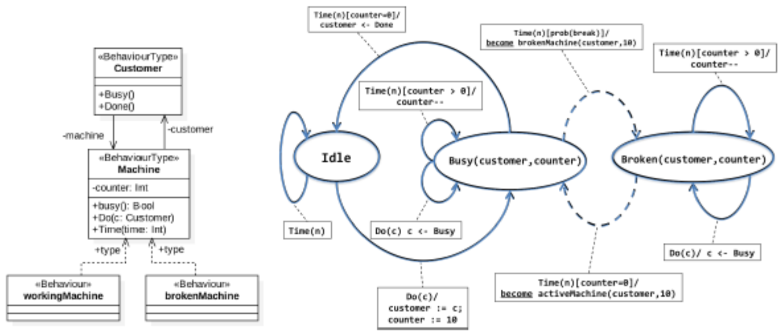


Figure 1: Actor Model of Computation [24]

An agent-based simulation model consists of agents, each of which has local knowledge, goals and behaviour. Such a model can be operationalized 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 as shown in **Fig. 1**. For the purposes of this chapter we conflate the terms *agent* and *actor*.

The key features of the actor computation model are [1, 2, 24]: (1) the creation of new actors; (2) sending asynchronous messages; (3) updating a local variable; (4) changing behaviour. The latter allows an actor to adapt by changing the way in which it responds to messages. The state of an actor can be represented in terms of its local variable storage (including references to other actors), its thread of execution, and its message queue. Execution proceeds independently at each actor by



(a) Actor Type Structure (b) Actor Behaviour

0	1	2	3	...	7	...
New(activeMachine,0) Update(0,customer,null) Update(0,counter,0) New(customer,1) Update(1,machine,Ref(0))	Send(1,0,Do,Ref(1))	Consume(0,Do,Ref(1)) Update(0,customer,Ref(1)) Update(0,counter,10)	Consume(0,Time(3)) Update(0,counter,9)	...	Consume(0,Time(7)) Become(0,brokenMachine) Update(0,counter,10)	...

(c) Execution History

Figure 2: Features of Actor Behaviour

selecting the next message on the queue, using the message to index a suitable handler in the actor's behaviour description, and proceeding to execute the handler on the actor's thread. When the execution terminates, it repeats the process by selecting the next message.

Consider a situation where a customer processes jobs on a machine. The customer submits a job request to a machine that may subsequently result in a notification that the job has been completed, or that the machine is busy and cannot accept the job. After accepting a job, a machine may break down causing a delay. A simple actor model for this situation is shown in **Fig. 2(a)** where the types `Customer` and `Machine` and the behaviours `workingMachine` and `brokenMachine`. A type defines an interface that may be implemented by many different behaviours, and a behaviour is equivalent to a Java class that can be instantiated to produce actors. The behaviour of a machine is shown in figure Fig. 2(b) and is distributed between the two behaviour definitions: a machine initially has the behaviour `workingMachine` and is Idle. A working machine becomes `Busy` when it receives a job request, and may change behaviour to become a `brokenMachine`. When broken, the `Machine` interface is implemented differently and may become working after a period of repair.

Fig. 2(c) shows an execution history corresponding to the machine and customer actors. The simulation is driven by messages `Time(n)` which are generated at regular intervals, and the history contains the events that are produced at each time interval. The event types are: `New(b,i)` where `i` is a unique actor identifier, and b is the corresponding behaviour; `Update(i,n,v)` where i is an actor identifier, `n` is the name of a state variable, and `v` is a new value for the variable; `Send(s,t,m)` records a message m being sent from actor `s` to target `t`; `Consume(i,m)` removes message m from the head of the message queue for actor `i`; `Become(i,b)` records the change of behaviour of actor `i` to have behaviour `b`.

The machine example demonstrates typical features of actor-based simulation: time and stochastic behaviour. All actors receive `Time(n)` messages that drive the simulation and therefore events can be associated with a specific time, thereby providing an event ordering within a specific history. The event that causes a machine breakdown is dependent on a given probability: `break`, and therefore multiple runs of the same actor model can produce different histories corresponding to emergent behaviour. Analysis of a history is based on detecting patterns in the sequence of events.

```
type Id       = Int;   // Actor identifiers.
type Time      = Int; // Atomic time units.
type Behaviour = [Handler(Str,[Command])]; //Handlers.
data Command   =         // Abstract commands.
     | Send(Id,Str)     // Send a name to a target actor.
     | Update(Str)      // Change a variable.
     | Block([Command]) // Group of commands.
     | Become(Str)      // Change behaviour.
     | New(Str);        // Create a new actor.
     | Choice(Set(Command)) // Choice between alternatives.
     | Consume          // Handle next message.
type Queue     = [Message(Str)]; // Sequence of messages.
type Actor     = Machine(Id,     // Actor's unique id.
     | [Command],        // Instructions on actor thread.
     | Behaviour,        // Actor message handlers.
     | Queue,            // Pending messages.
     | Time);            // Computation steps available.
type DB  = [Fact(Time,Command)];      // A History of facts.
type ESL = State(Set{Actor},DB,Time); // The system state.
```

Figure 3: An Abstract Model of Actor Programs as an ESL Data Type

Constructing Histories

The actor model of computation has been implemented in a significant number of languages and libraries [23]. The implementations differ in terms of syntax and in the integration with other language features, however the key aspects of the actor model are common to all. This section shows how any of these implementations can be extended to produce histories by defining an actor interpreter that abstracts away all non-essential features and that has been minimally extended to produce histories. The interpreter is defined using a functional subset of ESL, which supports basic data types, simple algebraic data, lists and sets. The latter is used in conjunction with pattern matching to support non-deterministic set element selection, which is key to the *fairness* property of actor systems.

Fig. 3 shows a model of actor program states: `State(a,db,t)` is the state of an executing actor system 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

```
send::(Int,Str,Set{Actor}) -> Set{Actor}
send(a,n,set{Machine(i,cs,b,q,t) | ms}) when i=a =
set{Machine(i,cs,b,Message(n):q,t) | ms}

getHandler::(Str,Behaviour) -> [Command]
getHandler(n,[])            = [];
getHandler(n,Handler(m,cs):b) = if m=n then cs else getHandler(n,b);

newState::(Actor,Set{Actor},DB,Time) -> ESL
newState(Machine(id,[],b,[],ta),ms,db,t) = State(set{m|ms},db,t);
newState(Machine(id,[],b,Message(n):q,ta),ms,db,t) =
  State(set{Machine(id,getHandler(n,b),b,q,ta-1) | ms},Fact(t,Consume):db,t);
newState(Machine(id,Send(target,n):cs,b,q,ta),ms,db,t) =
  State(set{Machine(id,cs,b,q,ta-1) | send(target,n,ms)},
        Fact(t,Send(target,n)):db,
        t);
newState(Machine(id,Update(var):cs,b,q,ta),ms,db,t) =
  State(set{Machine(id,cs,b,q,ta-1) | ms},
        Fact(t,Update(var)):db,
        t);
newState(Machine(id,Become(var):cs,b,q,ta),ms,db,t) =
  State(set{Machine(id,cs,getBehaviour(var),q,ta-1) | ms},
        Fact(t,Become(var)):db,
        t);
newState(Machine(id,New(var):cs,b,q,ta),ms,db,t) =
  State(set{Machine(id,cs,b,q,ta),Machine(id',[],b',[],100) | ms},
        Fact(t,New(var))):db,
        t) where (b',id') = newActor(var);
newState(Machine(id,Block(commands):cs,b,q,ta),ms,db,t) =
  State(set{Machine(id,commands + cs,b,q,ta-1) | ms},db,t);
newState(set{Machine(id,Choice(set{c|_}):cs,b,q,ta) | ms},db,t) =
newState(set{Machine(id,c:cs,b,q,ta) | ms},db,t)
newState(m=Machine(id,Consume:cs,b,q,ta),ms,db,t) = State(set{m | ms},db,t)
```

Figure 4: Semantics of ESL

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.

An actor is represented as `Machine(i,cs,b,q,t)` where `i` is a unique identifier, `cs` are machine instructions that are currently executing on the actor's thread of control, `b` is the actor's behaviour, `q` is a message queue, and `t` is an integer that represents the number of instructions left to this actor within this global time unit. The function `step` is defined:

```
is0::(Actor) -> Bool
is0(Machine(i,cs,b,ms,t))        = t=0;

all0::(Set{Actor}) -> Bool
all0(set{})                      = true;
all0(set{a | as})                = is0(a) and all0(as);

tickActor::(Actor) -> Actor
```

```
9    tickActor(Machine(i,cs,b,q,t))    = Machine(i,cs,b,Message('Time'):q,100);
10
11   tick::(Set{Actor}) -> Set{Actor}
12   tick(set{})                       = set{};
13   tick(set{a | s})                  = set{tickActor(a) | tick(s)};
14
15   step::(ESL) -> ESL
16   step(State(a,db,t)) when all0(a)  = State(tick(a),db,t+1);
17   step(State(set{m | ms},db,t))     = newState(m,ms,db,t)
```

where line 6 detects the situation where all actors have exhausted their execution resources for the current time unit and line 17 non-deterministically selects an actor `m` such that `newState` performs an execution step for `m` if that is possible.

Fig. 4 defines function `newState` that performs a single step of execution for an actor with respect to the system. It is defined by case analysis on the actor's control instructions. In the case that the actor has exhausted its control and has no further messages (line 10) it can do nothing. If the control is exhausted and there is a pending message (line 11) then the control is updated to become the new message handler. Otherwise, lines 13-34 show how each command causes the history database to be extended with a fact that is labelled with the current global time. The operation `newActor::(Str)->(Behaviour,Id)` is used to create a new actor when supplied with the name of a behaviour. It is not defined in **Fig. 4** but assumes a global collection of behaviour definitions and allocates a new actor identifier each time it is called.

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=getDB(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;

getDB::(ESL) -> DB;
getDB(State(_,db,_))               = db;

initState::(Set{Actor}) -> ESL;
initState(a)                       = State(a,[],0)

run::(Set{Actor}) -> ESL;
run(a)                           = repeat(step,initState(a),isTerminal);
```

```
act workingMachine(customer::Customer,counter::Int)::Machine {
  customer::Customer = null;
  Do(c::Customer) when counter > 0 -> c <- Busy;
  Do(c::Customer) when counter = 0 -> { counter := 10; customer := c }
  Time(n::Int) when customer <> null and counter > 0 ->
    probably(break) become brokenMachine(customer,10)
    else counter := counter - 1
  Time(n::Int) when customer <> null and counter = 0 -> c <- Done
}

act brokenMachine(customer::Customer,counter::Int)::Machine {
  Do(c::Customer)            ->  c <- Busy;
  Time(n::Int) when counter = 0 -> become workingMachine(customer,10);
  Time(n::Int)               -> counter := counter - 1
}
```

Figure 5: ESL Implementation of Machine

Fig. 5 shows a concrete ESL implementation of the machine from **Fig. 2.** The abstract implementation, using the language defined in **Fig. 3** is as follows:

```
behaviour workingMachine {
        -> block(update(customer),update(counter))
  Do(c) -> choice(send(c,Busy),block(update(counter),update(customer)))
  Time  -> choice(become(brokenMachine),update(counter),send(c,Done))
}
behaviour brokenMachine {
        -> block(update(customer),update(counter))
  Do(c) -> send(c,Busy)
  Time  -> choice(become(workingMachine),update(counter))
}
```

This section has described an abstract operational model for the construction of actor histories. 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 semantics is defined as an interpreter for an abstract actor language that can be used as the basis of designing a similar modification to a wide range of concrete languages and the relationship of the abstract language to ESL has been demonstrated. The next section shows how the histories produced by the interpreter can be interrogated using queries that are expressed using logic programming.

Interrogation of Histories

Simulations consist of many autonomous agents with independent behaviour and motivation. Consequently, the 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 such histories. We define a typed logic programming language and define an extended Prolog meta-interpreter with history interrogation features.

Typed Logic Programming

A basis for the history query language 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.

Standard Prolog, as shown above, does not provide support for history interrogation. Histories are temporally ordered facts, so history interrogation will involve queries that need to express ordering relationships between, what are otherwise, standard Prolog facts. This suggests that adding temporal operators to Prolog [19] and integrating the history facts with a Prolog rule database will provide a suitable basis for interrogation. During execution, a query is at a particular time unit in the history and can match against any of the facts at that time in addition to matching against rules in the rule-base. Operators can be used to move forwards and backwards in time to adjust the portion of the history that is used to establish facts.

```
data Value =          // Values occurring in rules.
  Term(Str,[Value])   // A term is a named sequence of values.
| Var(Str)            // A named logic variable.
| I(Int)              // An integer.
| S(Str);             // A string.

data Body =           // An element in the body of a rule.
  Call(Str,[Value])   // Call a rule, supplying values.
| Is(Str,Value)       // Unify a variable with a value.
| Start               // The start of the history.
| End                 // The end of the history.
| Next([Body])        // Move forward one unit of time.
| Prev([Body])        // Move back one unit of time.
| Always([Body])      // Body is true from now on.
| Eventually([Body])  // Body is true at some time from now.
| Past([Body])        // Body is true at some time previously.
| Forall([Body],Value,Value);// All ways in which body is true.
```

Figure 6: Data Type Definition

Quantification over the time variable to allow queries such as: *fact f exists at some point in the history from this point*, and *fact f exists at all points in the history before this point* to provide a suitably expressive basis for defining history interrogations with a logic programming framework. The rest of this section defines such a mechanism.

Meta Representation

This section defines a data representation for logic-programming rules where the rule-body elements support temporal operators over histories. The data type `Body` describes the elements that can occur in a rule body as depicted in Fig. 6. 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* that is used to index 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 at now `+1`, similarly `Prev(es)` at now `-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.

```
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('+',[l,r]),e,I(i)) <  eval(l,e,lv), eval(r,e,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(t,eot,db,[e|es],i,o,p) <- try(t,eot,db,e,i,j,p), trys(t,eot,db,es,j,o,p);
try::(Time,Time,DB,Body,Env,Env,Prog);
try(t,eot,db,Call(n,vs),i,o,p) <- derefs(vs,vs',i,o),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(t,eot,db,Next(es),i,o,p) <-  t' := t + 1; trys(t',eot,db,es,i,o,p);
try(0,eot,db,Prev(es),env,env,prog) <- !, false;
try(t,eot,db,Prev(es),i,o,p) <-  t' := t - 1, trys(t',eot,db,es,i,o,p);
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(t,eot,db,Eventually(es),i,o,p) <- trys(t,eot,db,es,i,o,p);
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(t,eot,db,Past(es),i,o,p) <- t' := t - 1, try(t',eot,db,Past(es),i,o,p);
try(t,eot,db,Is(n,exp),e,e,p) <- eval(exp,e,v), lookup[Value](n,v,e);
```

Figure 7: Query Language Meta Interpreter

Meta Interpreter

Fig. 7 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 the program.

The meta-interpreter is based on a standard operational semantics for Prolog that is extended with features to process the supplied database (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 8-13 where a rule named `n` with an appropriate arity is found in the program and is supplied with the argument values using `matchs`.

Fig. 7 extends conventional Prolog rule calling by allowing the fact to be present in the history at *the current time* (line 8). 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 41-63) by modifying the value of the current time appropriately, For example, the rule for `Next` (lines 48-49) 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[new(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.

Evaluation

The approach has been implemented as part of the ESL Workbench. ESL is an actor language that has been designed to support simulations. It has static types and

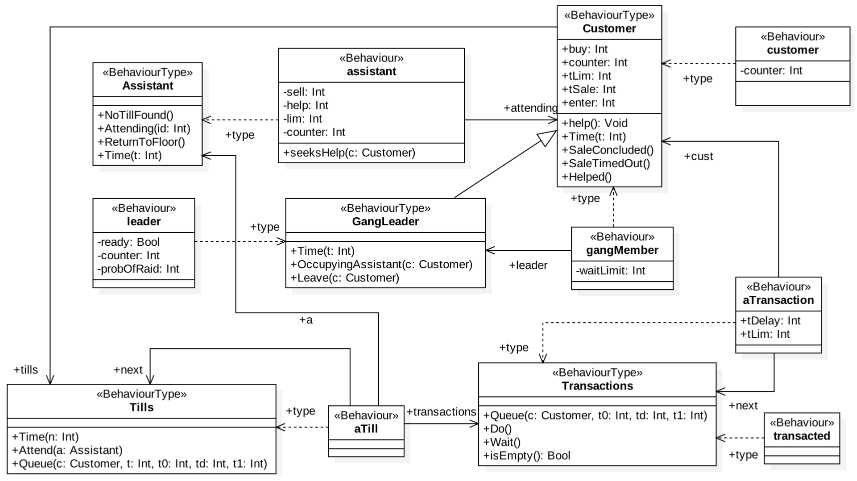


Figure 8: Structure of Shop Actors

compiles to run on a virtual machine (VM) implemented in Java. We have extended the ESL VM with features to produce histories and then integrated a query language, implemented as a Prolog VM, extended with features to process histories. This section evaluates the validity of the approach by applying it to a case study. ESL has been used to construct a number of simulations including an IT service provider, a research institute, and the effect of the 2016 Indian Demonetisation initiative. In this chapter we use a case study that is based on existing work on agent-based organisation simulations [38] involving a shop where customers browse for items, seek help from assistants, and queue to buy chosen items. Customers become unhappy if they wait too long for help or in a queue, and unhappy customers leave the shop.

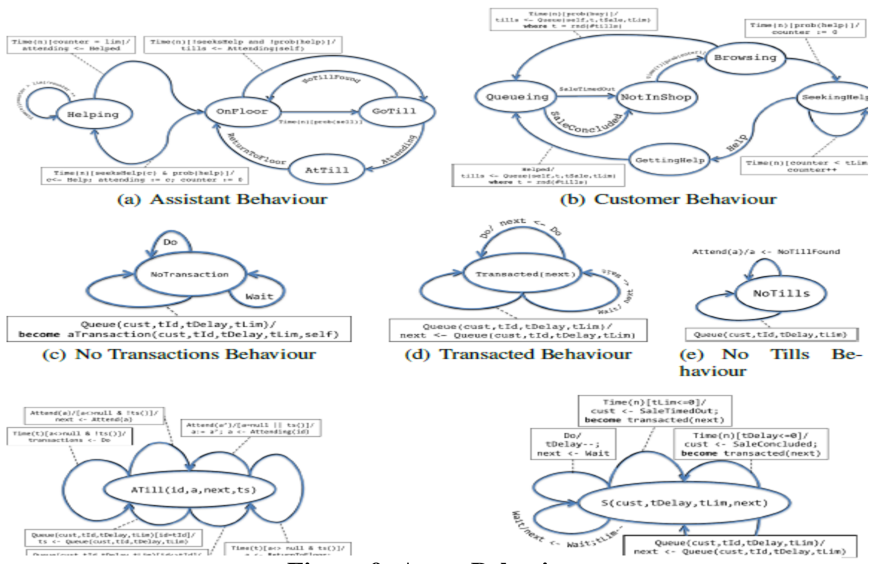
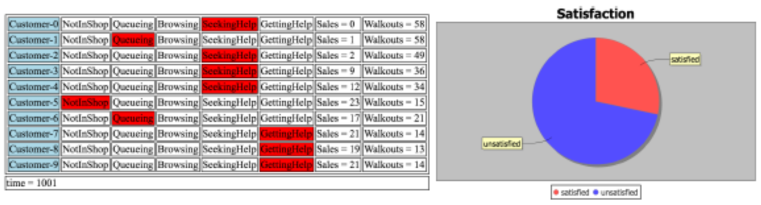


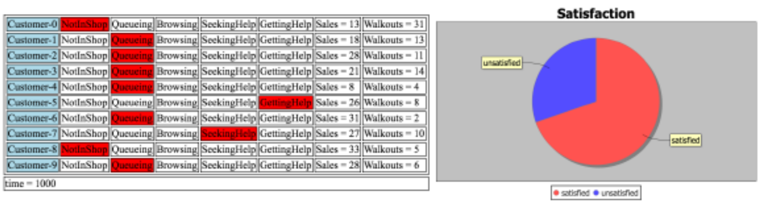
Figure 9: Actor Behaviours



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 10: Shop Simulation Output

The shop would like to simulate customer and assistant behavior in order to minimize unhappy customers. 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 analyzed in terms of its efficiency based on the implementation described in the previous section.

The structure of the shop simulation is shown in **Fig. 8.** and its behaviour is shown in **Fig. 9.** A snapshot of the output from ESL is shown in **Fig. 10** where 10 customers are mostly unsatisfied (a) and mostly satisfied (b).

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 **Fig. 9(g).** to show how customers waiting at a till can time out and ultimately leave the shop. The value supplied to a 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 organize its as-

```
raid::(Int,[Int]);
raid(n,raiders) <-
  customers(customers), !,
  subset[Int](raiders,customers),
  length[Int](raiders,n),
  findAllHelped(raiders);

findAllHelped::([Int]);
findAllHelped(raiders) <- allHelped(raiders,t), !;
findAllHelped(raiders) <- next[findAllHelped(raiders)];

allHelped::([Int],Int);
allHelped([],t)      <- !;
allHelped([c|cs],t) <-
  update(c,'state',GettingHelp,t),
  allHelped(cs,t);
```

Figure 11: Raids: Finding a Pattern in a History

sistants, sales and floor-walking strategies in order to minimize unhappy customers. **Fig. 10** shows the ESL Workbench output from two different simulation configurations. Figure **Fig. 10(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 **Fig. 10(b)** where the situation is reversed. Note that the simulation has many random elements and therefore each run is different, but the two outputs characterize the relative differences.

Fig. 11 shows a simple ESL query that interrogates the history produced by the shop simulation where `raid(n,cs)` is satisfied when `cs` is a list of `n` customer identifiers where the customers are all receiving help from sales assistants at the same time. Line 3 queries the history for all the customers and line 4 constructs a subset of all the customers. The rule *subset* is defined to allow backtracking through all the possible subsets, so line 5 will filter the subsets to select just those of the required length. The rules `findAllHelped` and `allHelped` query the history to ensure that the selected customers are in a `GettingHelp` state at the same time. The ESL implementation of the shop simulation can be downloaded as part of the open-source ESL system[1].

Conclusion

In this chapter we have sought to address the challenge of creating and analyzing actor-histories. We have shown how to extend a general actor-based language to produce histories of facts and how to extend a Prolog engine with temporal operators that can query the histories to establish whether patterns of facts exist. The proposal has been evaluated by showing that it can support a typical simulation, and that it can be implemented. Whilst other approaches to agent-based systems have used temporal operators to specify behaviour, and such operators have been used to interrogate system traces, use of this approach to analyze agent-based simulations is novel.

Whilst we have evaluated the approach in several different ways, the following issues and threats to validity remain: (Threat-1) The case study that has been used to evaluate the approach is taken from the literature and we can claim it to be representative of a class of simulations. Further work is needed to establish whether this case study is representative of a sufficiently broad class. (Threat-2) The histories that are produced by ESL-based simulations are typically intended to reflect some aspects of a real-world situation. Although the query approach described in this chapter does not rely on a valid history, in practice there must be some way to validate the simulation output. One possibility is to model an accepted theory, for example from social science or organisational management, and to show that the

[1] https://github.com/TonyClark/ESL

simulation model and its results is consistent with the theory. We intend to investigate this approach in the context of ESL simulations. (Threat-3) The efficiency of the approach has been established in the context of the example. This relies on knowledge of query semantics in order to ensure they are executed efficiently. It remains to be seen whether this is reasonable and whether efficiency can be improved. (Threat-4) Histories can be very large for long simulation runs. We have defined a compact implementation format, but further work is required to ensure that histories are no larger than is required. One option is to pre-process histories based on partial knowledge of query structures.

The approach described in this chapter is similar to existing approaches that analyze system traces, however we go further by defining precisely how the execution histories are produced and give a complete specification of the query language that can be used to analyze them. As such the work is novel and solves a problem that arises with actor-based systems where the behavior is both complex and non-deterministic. It remains to be seen whether the results meet users needs in terms of their ability to construct appropriate queries. One option is to be able to display and compare the results graphically, and this is an area for further work.

References

1. Agha, G.A.: Actors: A model of concurrent computation in distributed systems. Tech. rep., DTIC Document (1985)
2. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Jou nal of Functional Programming 7(01), 1–72 (1997)
3. Al-Kateb, M., Ghazal, A., Crolotte, A., Bhashyam, R., Chimanchode, J., Pakala, S.P.: Temporal query processing in teradata. In: Proceedings of the 16th International Conference on Extending Database Technology, pp. 573–578. ACM (2013)
4. Barat, S., Kulkarni, V., Clark, T., Barn, B.: Enterprise modeling as a decision making aid: A systematic mapping study. The Practice of Enterprise Modeling - 9th IFIP WG 8.1. Working Conference, PoEM 2016, Sk¨ovde, Sweden, pp. 289–298.
5. Barat, S., Kulkarni, V., Clark, T., Barn, B.: A simulation-based aid for organisational decisionmaking. 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016., pp. 109–116.
6. Barat, S., Kulkarni, V., Clark, T., Barn, B.: A model based realisation of actor model to conceptualise an aid for complex dynamic decision-making. Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21,2017., pp. 605–616.
7. Borgwardt, S., Lippmann, M., Thost, V.: Temporal query answering in the description logic dllite. In: International Symposium on Frontiers of Combining Systems, pp. 165–180. Springer (2013)
8. Borgwardt, S., Lippmann, M., Thost, V.: Temporalizing rewritable query languages over knowledge bases. Web Semantics: Science, Services and Agents on the World Wide Web 50–70 (2015)
9. Bosse, T., Duell, R., Memon, Z.A., Treur, J., Van DerWal, C.N.: Multi-agent model for mutual absorption of emotions. ECMS 2009, 212–218 (2009)

10. Bosse, T., Jonker, C.M., Van der Meij, L., Sharpanskykh, A., Treur, J.: Specification and verification of dynamics in cognitive agent models. In: IAT, pp. 247–254. Citeseer (2006)
11. Bosse, T., Jonker, C.M., Van Der Meij, L., Treur, J.: LEADSTO: a language and environment for analysis of dynamics by simulation. In: German Conference on Multiagent System Technologies, pp. 165–178. Springer (2005)
12. Bulling, N., Van der Hoek, W.: Preface: Special issue on logical aspects of multi-agent systems. Studia Logica,(Special Issue), 2016 (2016)
13. Caillou, P., Gaudou, B., Grignard, A., Truong, C.Q., Taillandier, P.: A simple-to-use BDI architecture for agent-based modeling and simulation. In: The Eleventh Conference of the European Social Simulation Association (ESSA 2015) (2015)
14. Clark, T., Kulkarni, V., Barat, S., Barn, B.: Actor monitors for adaptive behaviour. Proceedings of the 10th Innovations in Software Engineering Conference, ISEC 2017, Jaipur, India, February 5-7, 2017, pp. 85–95.
15. Clark, T., Kulkarni, V., Barat, S., Barn, B.: ESL: an actor-based platform for developing emergent behaviour organisation simulations. Advances in Practical Applications of Cyber-Physical Multi-Agent Systems: The PAAMS Collection - 15th International Conference, PAAMS 2017, Porto, Portugal, June 21-23, 2017.
16. De Koster, J., Van Cutsem, T., De Meuter, W.: 43 years of actors: a taxonomy of actor models and their key properties. In: Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, pp. 31–40. ACM (2016)
17. 18. Ferber, J., Gutknecht, O.: A meta-model for the analysis and design of organizations in multiagent systems. In: Multi Agent Systems, 1998. Proceedings. International Conference on, pp. 128–135. IEEE (1998)
18. Fishwick, P.A.: Computer simulation: growth through extension. Transactions of the Society for Computer Simulation 14(1), 13–24 (1997)
19. Gaintzarain, J., Lucio, P.: Logical foundations for more expressive declarative temporal logic programming languages. ACM Transactions on Computational Logic (TOCL) 14(4), 28 (2013)
20. Galland, S., Knapen, L., Gaud, N., Janssens, D., Lamotte, O., Koukam, A., Wets, G., et al.: Multi-agent simulation of individual mobility behavior in carpooling. Transportation Research Part C: Emerging Technologies 45, 83–98 (2014)
21. Hewitt, C.: Actor model of computation: scalable robust information systems. arXiv preprint arXiv:1008.1459 (2010)
22. Hindriks, K.V.: Debugging is explaining. In: International Conference on Principles and Practice of Multi-Agent Systems, pp. 31–45. Springer (2012)
23. Imam, S., Sarkar, V.: Savina-an actor benchmark suite. In: 4th International Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE (2014)
24. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: a comparative analysis. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, pp. 11–20. ACM (2009)
25. Kaufmann, M., Manjili, A.A., Vagenas, P., Fischer, P.M., Kossmann, D., F¨arber, F., May, N.: Timeline index: A unified data structure for processing queries on temporal data in SAP HANA. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 1173–1184. ACM (2013)
26. Kaufmann, M., Vagenas, P., Fischer, P.M., Kossmann, D., F¨arber, F.: Comprehensive and interactive temporal query processing with sap hana. Proceedings of the VLDB Endowment 6(12), 1210–1213 (2013)
27. Koeman, V.J., Hindriks, K.V.: Designing a source-level debugger for cognitive agent programs. In: International Conference on Principles and Practice of Multi-Agent Systems, pp. 335–350. Springer (2015)
28. Kruse, R., Steinbrecher, M., Moewes, C.: Temporal pattern mining. In: Signals and Electronic Systems (ICSES), 2010 International Conference on, pp. 3–8. IEEE (2010)

29. Kulkarni, V., Barat, S., Clark, T., Barn, B.: A wide-spectrum approach to modelling and analysis of organisation for machine-assisted decision-making. In: Enterprise and Organizational Modeling and Simulation - 11th InternationalWorkshop, EOMAS 2015, Held at CAiSE 2015, Stockholm, Sweden, June 8-9, 2015.
30. Kulkarni, V., Barat, S., Clark, T., Barn, B.S.: Toward overcoming accidental complexity in organisational decision-making. In: 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015, pp. 368–377 (2015)
31 Tony Clark, Vinay Kulkarni, Souvik Barat, and Balbir Barn. Sense-making in a complex and complicated world. IBM systems journal 42(3), 462–483 (2003)
32. McDermott, T., Rouse, W., Goodman, S., Loper, M.: Multi-level modeling of complex sociotechnical systems. Procedia Computer Science 16, 1132–1141 (2013)
33. Morgan, G.P., Carley, K.M.: An agent-based framework for active multi-level modeling of organizations. In: International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction and Behavior Representation in Modeling and Simulation SBP-BRiMS 2016. Springer (2016)
34. Pynadath, D.V., Tambe, M.: An automated teamwork infrastructure for heterogeneous software agents and humans. Autonomous Agents and Multi-Agent Systems 7(1-2), 71–100 (2003)
35. R¨aim, M., Di Ciccio, C., Maggi, F.M., Mecella, M., Mendling, J.: Log-based understanding of business processes through temporal logic query checking. In: OTM Conferences, pp. 75–92. Springer (2014)
36. Ricci, A., Agha, G., Bordini, R.H., Marron, A.: Special issue on programming based on actors, agents and decentralized control. Science of Computer Programming 98, 117–119 (2015)
37. Santos, G., Pinto, T., Morais, H., Sousa, T.M., Pereira, I.F., Fernandes, R., Prac¸a, I., Vale, Z.: Multi-agent simulation of competitive electricity markets: Autonomous systems cooperation for european market modeling. Energy Conversion and Management 99, 387–399 (2015)
38. 40. Siebers, P., Aickelin, U.: A first approach on modelling staff proactiveness in retail simulation models. J. Artificial Societies and Social Simulation 14(2) (2011). URL http://jasss.soc.surrey.ac.uk/14/2/2.html
39. Singh, D., Padgham, L., Logan, B.: Integrating BDI agents with agent-based simulation platforms. Autonomous Agents and Multi-Agent Systems 30(6), 1050–1071 (2016)
40. Sukthankar, G., Sycara, K.: Simultaneous team assignment and behavior recognition from spatio-temporal agent traces. In: AAAI, vol. 6, pp. 716–721 (2006)
41. 44. Vasconcelos, W.W., Kollingbaum, M.J., Norman, T.J.: Normative conflict resolution in multiagent systems. Autonomous Agents and Multi-Agent Systems 19(2), 124–152 (2009)
42. Winikoff, M., Cranefield, S.: On the testability of BDI agent systems. J. Artif. Intell. Res.(JAIR) 51, 71–131 (2014)
43. Divine T. Ndumu, Hyacinth S. Nwana, Lyndon C. Lee, and Jaron C. Collis. Visualising and debugging distributed multi-agent systems. In Proceedings of the ird Annual Conference on Autonomous Agents, AGENTS '99, pages 326–333, New York, NY, USA, 1999. ACM.