Generating Filmstrip Models from Actor-Based Systems

Tony Clark
Sheffield Hallam University
Sheffield, United Kingdom
T.Clark@shu.ac.uk

Vinay Kulkarni, Souvik Barat
Tata Consultancy Services Research
Pune, India
{Vinay.Vkulkarni,Souvik.Barat}@tcs.com

Balbir Barn
Middlesex University
London, United Kingdom
B.Barn@mdx.ac.uk

***Abstract*—Actor-based systems are hard to analyse because of their inherent complexity arising from large=scale concurrency and stochastic behaviour. History traces can be produced from such a system that describes what happened during execution leading to a challenge as to how the traces are processed. This tool demonstration shows a novel actor-based language ESL and its development environment EDB that has been extended to produce histories stored in temporal databases, and to use a logic programming language to construct graphical filmstrips from the databases.**

***Index Terms*—Agent-based modeling**

I. Introduction

EDB (the ESL DeBugger) is a tool that supports the analysis of simulation behaviours. Simulations are represented as executable agent models represented using the actor-language ESL (Executable Simulation Language). This demonstration shows how agent models are represented in EDB and how executions can be captured as time-stamped facts in a knowledge-base. ESL supports logic programming in the form of rule-sets that are parameterised with respect to knowledge-bases. The logic programming language provides temporal operators that are used to map temporal knowledge-bases to filmstrips. EDB provides graphical features for displaying and playing filmstrips forwards and backwards. We demonstrate these features using a simple case study involving customers in a shop.

Figure 1 describes the overall EDB-based process for creating and interacting with filmstrips arising from actor-based simulation models represented in ESL. A simulation model is defined in terms of the actor structure and their individual behaviours. The structure is equivalent to a class diagram and the behaviour of each actor is defined as a state-machine driven by the messages received by the associated actor. The models are translated into code that is executed to produce a knowledge-base that contains time-stamped actor-states. Mapping rules are used to process the knowledge-base, producing a sequence of diagram models that are displayed as a filmstrip. The user can then play the filmstrip backwards and forwards.

The rest of this paper is structured as follows: section II describes a simple case study that is modelled in section III; section IV describes the ESL implementation of the simulation model and the production of a knowledge-base; section V shows how the knowledge-base is transformed into filmstrips and how they are displayed by EDB. Section VI places ESL and EDB in relation to other work.

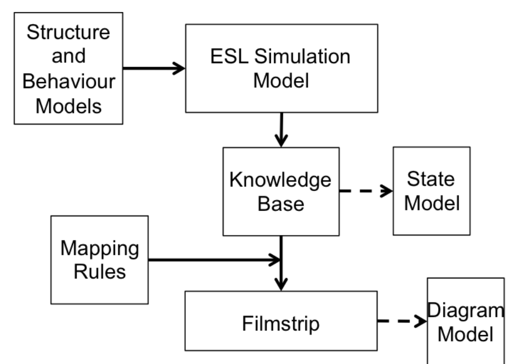


Fig. 1. Overall Process

II. Case Study

Figure 2 shows a case study that is taken from [10]. 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 actor-based implementation of the case study will necessarily have stochastic behaviour and therefore be difficult to verify. A formal approach to verification is often challenging to achieve due to the state explosion when simulations become large. Furthermore, interacting with a running simulation or producing printed output can either change the behaviour or overwhelm the developer. Our approach compliments such approaches by providing a graphical representation of the simulation that can be post-processed.

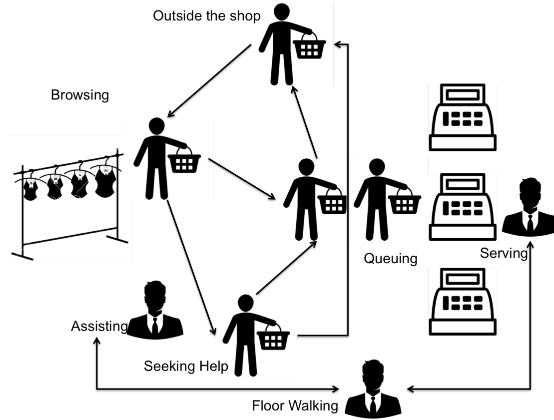


Fig. 2. Shopping Simulation (Based On [10])

III. Input Models

ESL is a statically typed actor language. Actors have behaviour types that are equivalent to component interface types, and have behaviour definitions that are equivalent to module definitions. Actor behaviour in ESL is implemented as message handling rules that can be generated from state machines. This section outlines the behaviour type (structure) and behaviour (state-machine) models that are derived from the case study. Figure 3 shows the behaviour types for the shop simulation. In addition to domain actors such as `customer` and `assistant`, the transactions at a till are represented as actors. Figure 4 shows the behaviour of the actors represented as state machines. An actor receives messages that are either sent by other actors or are `Time` messages produced automatically by ESL. Optionally, an actor can replace its behaviour when it handles a message, for example this occurs in figure 4(c) where a customer joins a till-queue.

IV. ESL Implementation

ESL is a statically typed text-based language. Actor models, such as those described in the previous section, are translated into ESL. EDB provides support for ESL development in the form of static type checking. The following code fragment shows how the shop models are translated into ESL.

```
data State =            NotInShop(Int)
| Browsing(Int)       | Queuing(Int,Int)
| SeekingHelp(Int)    | GettingHelp(Int,Int)
| OnFloor(Int)        | GoTill(Int)
| AtTill(Int,Int)     | Helping(Int,Int)
| Raid                | NoRaid;

type Customer = Act {
  export state::State; getId::() → Int;
  help::(Assistant) → Void;
  Time(Int);
  SaleConcluded();
  SaleTimedOut();
  Helped()
}

type Facts = KB[State];
facts::Facts = kb[Facts]{ NoRaid };

act customer(id::Int,tills::Tills)::Customer {
  state::State = NotInShop(id);
  getId()::Int = id;
  counter::Int = 0;
  helpedBy::Assistant = null;

  Time(n::Int) when state = NotInShop(id) →
    probably(probOfEnteringShop) {
```

Fig. 3. Structure of Shop Actors

```
      state := Browsing(id);
      facts.add(state,time++);
    }
}
```

Actor states are defined as an algebraic data type `State` as shown above. The states are used to control the behaviour of the actors and are also recorded in a knowledge-base for subsequent processing by the filmstrip mapping rules.

Each actor has a behaviour type definition such as `Customer` containing the operation and message signature. All message names are implicitly exported, property names and operations are local in any behaviour implementation unless they are exported by the type.

The knowledge-base `facts` conforms to the type `State` and therefore contains time-stamped state information.

The behaviour definition `customer` shows a fragment of the implementation of the customer behaviour defined in figure 4. All actors in ESL are supplied with `Time` messages at regular intervals in order to drive the simulation. The `customer` behaviour shows a message handler that responds to a `Time` message that occurs when the customer is not in the shop and has a given probability of entering. Note how the change of state is recorded in the knowledge-base.

V. FILMSTRIP GENERATION

Filmstrips are produced by translating temporal knowledge-bases into sequences of pictures. This is achieved in a two-stage process: (1) the knowledge-base is translated into a sequence of system states; (2) each state is translated into a picture. Both stages are performed using the features of ESL: (1) temporal logic programming; (2) functional pattern matching. Mapping from a knowledge-base to a sequence of system states is performed using the rules shown in figure 5 where a history is created as a sequence of shop-states as defined by the rule `hist` which is supplied with customer, assistant and till identifiers. The `hist` rule uses the temporal operator **next** to step through the knowledge-base. At each time-step, the rule `state` is used to construct a shop-state. The rules in figure 5 show how the `cOut` and `raid` components of the shop-state are extracted from the knowledge-base. A raid is detected by rule `isRaid` when there is a current `Raid` state recorded in the knowledge base; notice that is may be necessary to step backwards in time through the facts in the knowledge-base before the appropriate fact is found. The identifiers of customers who

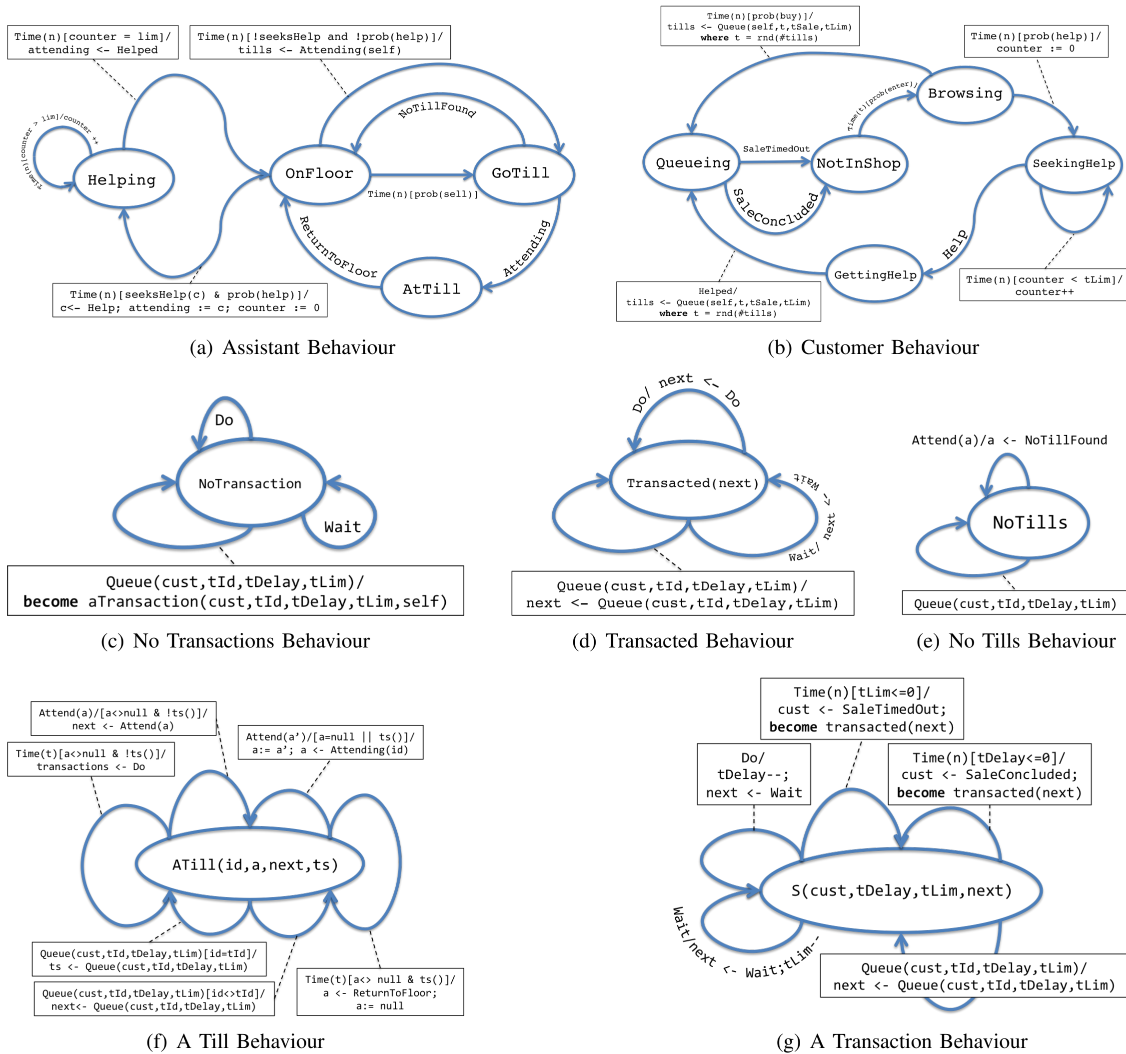


Fig. 4. Shop Actor Behaviour

are currently outside the shop is calculated by the call `getCStates(n,cIds,cOut)` which extracts the nth customer state and filters the list of customer identifiers for those customers currently in that state. The calculation uses the rule `recent` to ensure that the desired state is the most recent: this will be false if the particular customer is most recently in an alternative state. A fact `f` is currently in the knowledge-base if `fact(f)` is true.

```
type PictureElement =
   Rectangle(Int,Int,Int,Int,Str)
|  Circle(Int,Int,Int,Str)
|  Line(Int,Int,Int,Int,Str)
|  Image(Int,Int,Int,Int,Str)
|  Text(Int,Int,Str,Str)
type Pics = [Picture(Int,Int,[PictureElement])];
type EDB = Act {
  Filmstrip(Str,Pics)
}
```

The data type above shows the different picture elements that can be drawn on each frame of a film-strip. The message `Filmstrip(title,pictures)` displays the sequence of pictures with a slider that can be used to move forwards and backwards; moving the slider changes the picture that is displayed.

```
type Solver = Rules {
  hist([Int],[Int],[Int],[ShopState])
};
solver::Solver = rules {
 hist::([Int],[Int],[Int],[ShopState]);
 hist(cIds,aIds,tIds,[s]) ←
   state(cIds,aIds,tIds,s), end, !;
 hist(cIds,aIds,tIds,[s|ss]) ←
   state(cIds,aIds,tIds,s), !,
   next[hist(cIds,aIds,tIds,ss)];

 custStates::(Int,[State]);
 custStates(id,[NotInShop(id),
                Browsing(id),
                SeekingHelp(id),
                GettingHelp(id,aId),
                Queuing(id,tId)]);

 state::([Int],[Int],[Int],ShopState);
 state(cIds,aIds,tIds,Shop(cOut,f,b,h,t,raid)) ←
   isRaid(raid),
   getCStates(0,cIds,cOut),
   // extraction of f,b,h,t omitted.
// listing continues in right-hand column.
```

```
 isRaid::(Int);
 isRaid(0) ← fact(NoRaid), !;
 isRaid(1) ← fact(Raid), !;
 isRaid(raid) ← prev[isRaid(raid)];

 getCStates::(Int,[Int],[Int]);
 getCStates(_,[],[]) ← !;
 getCStates(n,[id|idsIn],[id|idsOut]) ←
   custStates(id,ss),
   nth[State](n,ss,s),
   delete[State](s,ss,ss'),
   recent(id,s,ss'), !,
   getCStates(n,idsIn,idsOut);
 getCStates(n,[id|idsIn],idsOut) ←
   getCStates(n,idsIn,idsOut);

 recent::(Int,State,[State]);
 recent(id,f,fs) ← fact(f), !;
 recent(id,f,fs) ← facts(fs), !, false;
 recent(id,f,fs) ← prev[recent(id,f,fs)];

 facts::([State]);
 facts([]) ← false, !;
 facts([f|fs]) ← fact(f), !;
 facts([_|fs]) ← facts(fs);
}
```

Fig. 5. Logic Rules for History Production

```
toPicture(s::ShopState)::PictureElement =
 case s {
   Shop(cOut,aFloor,cBrowse,h,ts,raid) →
    letrec customerOutside(ids::[Int],x::Int)::[PictureElement] =
      if ids = []
      then []
      else [Image(x,0,iconWidth,iconHeight,customerIcon),
            Text(x,(iconHeight+textHeight),head[Int](ids)+'','')] +
           customerOutside(tail[Int](ids),x+iconWidth);
     ...
    in Picture(pictureWidth,pictureHeight,customerOutside(cOut,0) +..)
  };
show hist(cIds,aIds,tIds,history) from solver using facts {
  edb ← Filmstrip('ShopFilmstrip',[ toPicture(s) | s::ShopState ← history ])
}
```

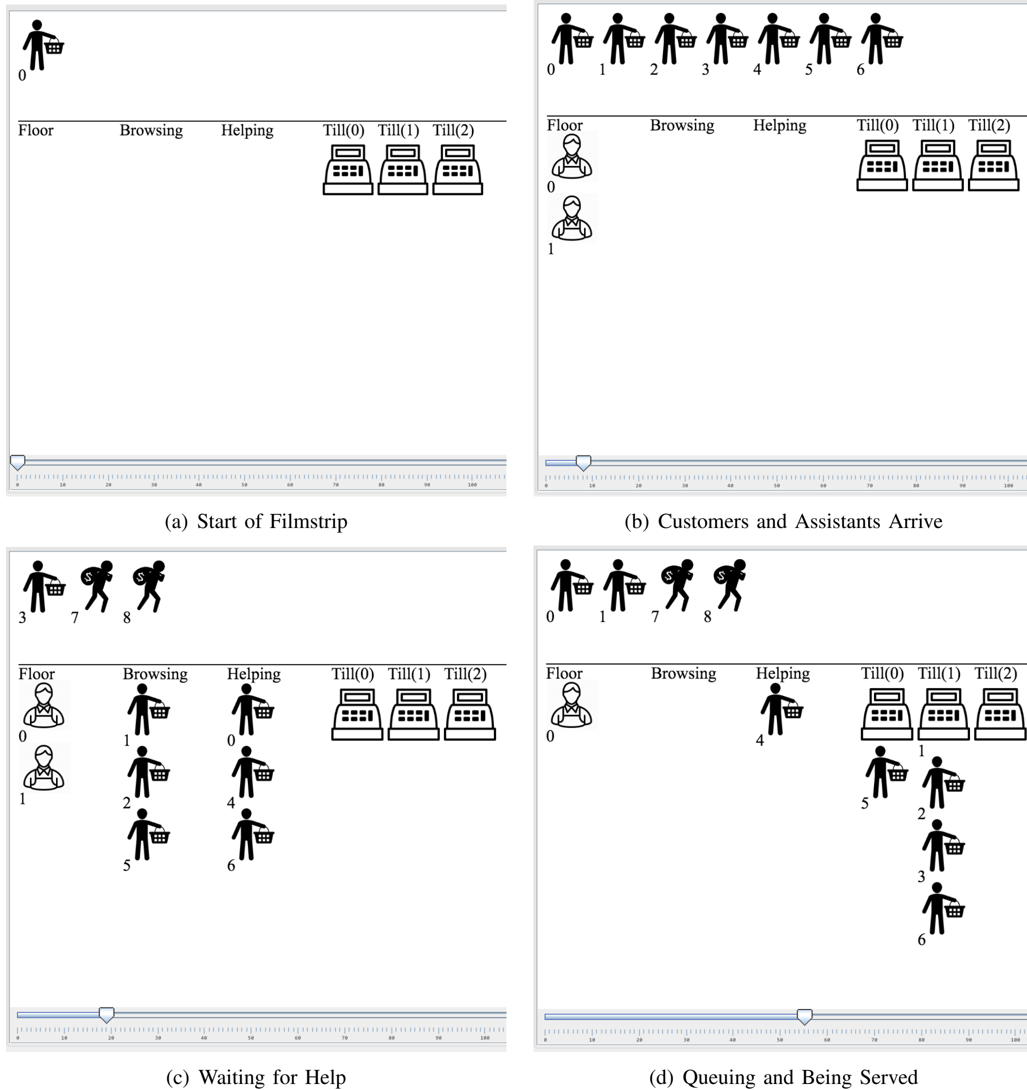
Fig. 6. Producing A Picture

Once the simulation has completed and populated the knowledge-base, the filmstrip is displayed by sending a `Filmstrip` message to EDB. A rule-base is paired with a knowledge-base using the `show` construct in EDB where `show q from r using k c` will try to establish fact `q` in rule base `r` using knowledge-base `k` and then perform command `c`. Figure 6 shows a fragment of the function `toPicture` that maps system states to pictures using pattern-matching functions.

Interactions with the resulting filmstrip in EDB is shown in figure 7 where figure 7(a) shows the start of time, figure 7(b) shows the situation after all the customers and assistants have joined the simulation; figure 7(c) shows customers have entered the shop and started to browse and some are looking for help, notice also that gang members have also arrived; figure 7(d) shows a customer (5) queueing at till 0, customers (2,3,6) are queueing at till 1 where assistant 1 is serving.

VI. Related Work

The use of MAS for system simulation has been explored by a number of researchers, for example in [5], [1], [9], [11], [4]. In [1], 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



(a) Start of Filmstrip

(b) Customers and Assistants Arrive

(c) Waiting for Help

(d) Queuing and Being Served

Fig. 7. Filmstrip Animation

and quantitative aspects, and subsumes specification languages based on differential equations. However, this is not an executable language. It has been specialised for simulation and has produced the LEADSTO language [2] which 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 animating the resultant behaviour. Temporal logics have been used to specify the behaviour of MAS [3] and to analyse 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. The need to analyse agent-based simulations is related to the field of agent-based system testing.

As noted in [12] *attempting to obtain assurance of a system's correctness by testing the system as a whole is not feasible*. Our approach is intended to be a pragmatic partial solution that is used selectively in collaboration with a domain expert. Managing temporal data is becoming increasingly important for many applications [7], [6]. Our work is related to process mining from the event logs that are created by enterprise systems [8] where queries can be formulated in terms of a temporal logic and applied to data produced by monitoring real business systems.

REFERENCES

[1] 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.

[2] 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.

[3] Nils Bulling and Wiebe Van der Hoek. Preface: Special issue on logical aspects of multi-agent systems. *Studia Logica,(Special Issue), 2016*, 2016.

[4] Philippe Caillou, Benoit Gaudou, Arnaud Grignard, Chi Quang Truong, and Patrick Taillandier. 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.

[5] Stephane Galland, Luk Knapen, Nicolas Gaud, Davy Janssens, Olivier Lamotte, Abderrafiaa Koukam, Geert Wets, et al. Multi-agent simulation of individual mobility behavior in carpooling. *Transportation Research Part C: Emerging Technologies*, 45:83–98, 2014.

[6] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. 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*, pages 1173–1184. ACM, 2013.

[7] Rudolf Kruse, Matthias Steinbrecher, and Christian Moewes. Temporal pattern mining. In *Signals and Electronic Systems (ICSES), 2010 International Conference on*, pages 3–8. IEEE, 2010.

[8] Margus Räim, Claudio Di Ciccio, Fabrizio Maria Maggi, Massimo Mecella, and Jan Mendling. Log-based understanding of business processes through temporal logic query checking. In *OTM Conferences*, pages 75–92. Springer, 2014.

[9] Gabriel Santos, Tiago Pinto, Hugo Morais, Tiago M Sousa, Ivo F Pereira, Ricardo Fernandes, Isabel Praça, and Zita Vale. Multi-agent simulation of competitive electricity markets: Autonomous systems cooperation for european market modeling. *Energy Conversion and Management*, 99:387–399, 2015.

[10] 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.

[11] Dhirendra Singh, Lin Padgham, and Brian Logan. Integrating BDI agents with agent-based simulation platforms. *Autonomous Agents and Multi-Agent Systems*, 30(6):1050–1071, 2016.

[12] Michael Winikoff and Stephen Cranefield. On the testability of BDI agent systems. *J. Artif. Intell. Res.(JAIR)*, 51:71–131, 2014.