
Final accepted version (with author's formatting)

This version is available at: http://eprints.mdx.ac.uk/27818/

Copyright:

Middlesex University Research Repository makes the University's research available electronically. Copyright and moral rights to this work are retained by the author and/or other copyright owners unless otherwise stated. The work is supplied on the understanding that any use for commercial gain is strictly forbidden. A copy may be downloaded for personal, non-commercial, research or study without prior permission and without charge.

Works, including theses and research projects, may not be reproduced in any format or medium, or extensive quotations taken from them, or their content changed in any way, without first obtaining permission in writing from the copyright holder(s). They may not be sold or exploited commercially in any format or medium without the prior written permission of the copyright holder(s).

Full bibliographic details must be given when referring to, or quoting from full items including the author’s name, the title of the work, publication details where relevant (place, publisher, date), pagination, and for theses or dissertations the awarding institution, the degree type awarded, and the date of the award.

If you believe that any material held in the repository infringes copyright law, please contact the Repository Team at Middlesex University via the following email address:

eprints@mdx.ac.uk

The item will be removed from the repository while any claim is being investigated.

See also repository copyright: re-use policy: http://eprints.mdx.ac.uk/policies.html#copy
Making Sense of Actor Behaviour: 
An Algebraic Filmstrip Pattern and its Implementation

Tony Clark
Aston University
Birmingham, UK
tony.clark@aston.ac.uk

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

Vinay Kulkarni, Souvik Barat
TCS Research
Pune, India
vinay.vkulkarni@tcs.com
souvik.barat@tcs.com

ABSTRACT
Sense-making with respect to actor-based systems is challenging because of the non-determinism arising from concurrent behaviour. One strategy is to produce a trace of event histories that can be processed post-execution. Given a semantic domain, the histories can be translated into visual representations of the semantics in the form of filmstrips. This paper proposes a general pattern for the production of filmstrips from actor histories that can be implemented in a way that is independent of the particular data types used to represent the events, semantics and graphical displays. We demonstrate the pattern with respect to a simulation involving predators and prey which is a typical agent-based application.

CCS CONCEPTS
• Software and its engineering → Concurrent programming structures; Software testing and debugging.

KEYWORDS
Actors, Filmstrips

ACM Reference Format:

1 INTRODUCTION
Systems such as smart energy-grids, supply-chain networks and smart factories can be represented using Multi-Agent Systems (MAS) [15, 27, 35] where systems are constructed in terms of independent goal-directed agents that concurrently engage in tasks both independently and collaboratively. The benefits of MAS include resilience [14] and adaptation [3] which are desirable properties for modern complex distributed heterogeneous systems. MAS can also be used to develop simulations of systems [16]. An important reason for using agents for simulation is that the systems of interest are complex and involve, for example, socio-technical features [26].

MAS are inherently non-deterministic and exhibit emergent behaviour which makes debugging and sense-making challenging [33, 40]. Recent work on MAS verification has focussed on static analysis of the communication between agents [34] using interaction protocols [1].

Sense-making incorporates a range of tasks. Such tasks consist of information gathering, re-representation of the information in a schema that aids analysis, the development of insight through the manipulation of this representation, and the creation of some knowledge product or direct action based on the insight [32]. A specialised form of sense-making is debugging. Sense-making can be supported through the use of domain-specific representations of system execution [35]. Augmenting the temporal aspect by a visual representation of the execution data gives improved understanding of systems [2].

Our hypothesis to address these challenges is the use of histories and their subsequent manipulation to perform sense-making. Each agent can produce a history that consists of a description of its local state changes. However, the resulting collection of histories requires combination in the context of a semantic model in order to meaningfully represent the history of a complete system. Therefore, our proposal imposes a semantics on the histories in order to support sense-making. Furthermore, since it is a history, we would like to be able to ‘play’ the history forwards and backwards to understand what happened during the execution traces over time.

The contribution of this paper is a semantics-based filmstrip pattern that can be used to support MAS sense-making that is independent of any particular implementation technology. Filmstrips are generally attributed to D’Souza and Wills in their modelling method, Catalyst [12]. A filmstrip is a sequence of snapshots (objects and relationships) describing system state transitions arising from operation calls in the system. The proposal is evaluated in terms of an implementation using the actor-based language ESL.

This paper motivates the use of filmstrips as a basis for analyzing agent-based systems in section 2 using a standard MAS application involving predators and prey. This is a typical agent-based application [13] that is applied to understanding community dynamics [11], ecology [38], and infectious diseases [37]. The application is written in an actor language ESL [9] whose semantics is presented in section 2.2.

The main contribution of the paper is given as an algebraic pattern in section 3 that can be used to construct filmstrips independently of the data types that are used to represent the event histories and the semantics of the application. The pattern is then
An ESL program consists of a collection of value, function and behavior definitions. Each behavior has a corresponding type sary to precisely capture the debugging challenge and subsequent of predator-prey. This section gives an informal description of the behavior. Example behavior types are listed in lines 1–3; predator-prey behavior. A typical example of this is the predator-prey simulation [30] where a group of predators (in this case wolves) try to catch a prey (in this case a sheep). The purpose of the simulation is to investigate different strategies employed by each category of actor: predators try to catch the prey who in turn tries to evade them. Section 2.1 describes an implementation of predator-prey in the language ESL which is defined in section 2.2. The issues arising from the use of filmstrips for debugging is described in section 2.3 leading to the definition of sense-making requirements to be addressed by the pattern defined in the following section.

### 2.1 Predator Prey Filmstrips

Figure 2 shows an ESL program that implements a simple version of predator-prey. This section gives an informal description of the program and section 2.2 provides a formal definition that is necessary to precisely capture the debugging challenge and subsequent definition of the filmstrip pattern.

ESL combines functional and actor-based programming [8–10] making it an ideal candidate for the proposed filmstrip pattern. An ESL program consists of a collection of value, function and behaviour definitions. Each behaviour has a corresponding type definition listing the messages that can be received by any actor with the behaviour. Example behaviour types are listed in lines 1–3; Predator and Prey both define a message Move, and Main defines a Time message. The latter is key to the ESL semantics which provides all actors with a message telling them the current time at regular intervals and which can be used to drive the application and eventually terminate it (line 54).

Line 4 defines a union data type Message that has two alternatives: PredAt(i,x,y) meaning that a predator with identity i is at position (x,y), and PreyAt(x,y) meaning that the prey is at position (x,y). The data value messages on line 7 is a list of messages and is initialized to the empty list. This will be used as the application history with all actors posting messages to the end of the list.

The behaviours predator (lines 9–20) and prey (lines 22–33) define what happens when an actor with the respective behaviours handles a Move message. Both behaviours have initialisation arguments that

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>type Predator = Act { Move }</td>
</tr>
<tr>
<td>2</td>
<td>type Prey = Act { Move }</td>
</tr>
<tr>
<td>3</td>
<td>type Main = Act { Time(Int) }</td>
</tr>
<tr>
<td>4</td>
<td>data Message = PredAt(Int,Int,Int)</td>
</tr>
<tr>
<td>5</td>
<td>data Pos = Point(Int,Int);</td>
</tr>
<tr>
<td>6</td>
<td>messages := [Message] = [];</td>
</tr>
<tr>
<td>7</td>
<td>act predator(id:Int,x:Int,y:Int):Predator {</td>
</tr>
<tr>
<td>8</td>
<td>Move → grab(messages) {</td>
</tr>
<tr>
<td>9</td>
<td>let dx:Int = randomMove();</td>
</tr>
<tr>
<td>10</td>
<td>dy:Int = randomMove();</td>
</tr>
<tr>
<td>11</td>
<td>in if isNearerPredy(id,dx,dy) and canMove(x=dx,y=dy)</td>
</tr>
<tr>
<td>12</td>
<td>then {</td>
</tr>
<tr>
<td>13</td>
<td>x := x + dx;</td>
</tr>
<tr>
<td>14</td>
<td>y := y + dy;</td>
</tr>
<tr>
<td>15</td>
<td>messages := messages + [PredAt(x,y)];</td>
</tr>
<tr>
<td>16</td>
<td>}</td>
</tr>
<tr>
<td>17</td>
<td>}</td>
</tr>
<tr>
<td>18</td>
<td>act prey(x: Int, y: Int): Prey {</td>
</tr>
<tr>
<td>19</td>
<td>Move → grab(messages) {</td>
</tr>
<tr>
<td>20</td>
<td>let dx:Int = randomMove();</td>
</tr>
<tr>
<td>21</td>
<td>dy:Int = randomMove();</td>
</tr>
<tr>
<td>22</td>
<td>in if isAwayFromPred(dx,dy) and canMove(x=dx,y=dy)</td>
</tr>
<tr>
<td>23</td>
<td>then {</td>
</tr>
<tr>
<td>24</td>
<td>x := x + dx;</td>
</tr>
<tr>
<td>25</td>
<td>y := y + dy;</td>
</tr>
<tr>
<td>26</td>
<td>messages := messages + [PreyAt(x,y)];</td>
</tr>
<tr>
<td>27</td>
<td>}</td>
</tr>
<tr>
<td>28</td>
<td>}</td>
</tr>
<tr>
<td>29</td>
<td>predators:{Predator} =</td>
</tr>
<tr>
<td>30</td>
<td>[ new predator(p,random(width),random(height))</td>
</tr>
<tr>
<td>31</td>
<td>p:Int ← 0..numOfPredators ];</td>
</tr>
<tr>
<td>32</td>
<td>thePrey:Prey = new prey(random(width),random(height));</td>
</tr>
<tr>
<td>33</td>
<td>rocks:Pos =</td>
</tr>
<tr>
<td>34</td>
<td>[ Point(random(width),random(height))</td>
</tr>
<tr>
<td>35</td>
<td>r:Int ← 0..numOfRocks ];</td>
</tr>
<tr>
<td>36</td>
<td>act main:Main {</td>
</tr>
<tr>
<td>37</td>
<td>Time(n:Int) when n &lt; limit → {</td>
</tr>
<tr>
<td>38</td>
<td>for p:Predator in predators do</td>
</tr>
<tr>
<td>39</td>
<td>p← Move;</td>
</tr>
<tr>
<td>40</td>
<td>thePrey ← Move;</td>
</tr>
<tr>
<td>41</td>
<td>wait();</td>
</tr>
<tr>
<td>42</td>
<td>}</td>
</tr>
<tr>
<td>43</td>
<td>Time(n:Int) → {</td>
</tr>
<tr>
<td>44</td>
<td>showFilmstrip(messages);</td>
</tr>
<tr>
<td>45</td>
<td>stopAll();</td>
</tr>
<tr>
<td>46</td>
<td>};</td>
</tr>
</tbody>
</table>

**Figure 2: ESL Definition Predator-Prey Behaviours**
Making Sense of Actor Behaviour:
An Algebraic Filmstrip Pattern and its Implementation

ISEC’19, February 14–16, 2019, Pune, India

Actor-based systems are highly concurrent which makes debugging the application shows a filmstrip constructed from the history and stops the application. The rest of the paper describes the details of keeping predator, prey and rock positions separate is created in line 39. A list of rock positions is created (lines 41–43); messages drive the movement strategy in terms of some functional programming and actor-languages. The filmstrip pattern that is defined in ESL in section 4 is independent of the data type used to represent the semantic domain used to structure the snapshots.

A list of predators is created in lines 35–37 and a single prey is created in line 39. A list of rock positions is created (lines 41–43); the details of keeping predator, prey and rock positions separate is omitted.

An ESL program starts by creating a single actor with the behaviour main defined on lines 45–56. The time messages drive the main actor to send move messages to each of the predator and prey actors. The messages are sent asynchronously and Actor model of computation guarantees that the messages will be received and computation will be fairly distributed. Once the time limit is reached (lines 52–55) the application shows a filmstrip constructed from the history and stops the application. The rest of the paper describes how the filmstrip is constructed and displayed.

2.2 ESL

Actor-based systems are highly concurrent which makes debugging them a challenge. This section defines the semantics of ESL based on a standard actor semantics [29, 31] which is extended with monitors as used by grab in the previous section. The filmstrip pattern defined in ESL makes use of polymorphism in order to be independent of the semantic domain used as a basis for sense-making. This section also defines a type relation for polymorphic ESL that is suitable for the filmstrip pattern definitions.

Figure 3 defines the operational semantics of ESL. An ESL configuration is \( \langle \alpha; \mu; y \rangle \) where \( \alpha \) is a set of actors, \( \mu \) is a multi-set of pending messages and \( y \) is a set of monitors that are currently locked. An actor \( a \) \( \in \alpha \) can either be busy or inactive. A busy actor is represented as \( \langle E \vdash R(e) \rangle_a \) where \( E \) is the local state of the actor, and \( R \) is a reduction context filled with expression \( e \) that is currently being executed. An inactive actor is waiting for a message and is represented as \( \langle E \vdash b \rangle_a \) where \( b \) is its behaviour. A message to \( a \) that is pending is represented as \( a \equiv v \).

The language of ESL actor behaviours is standard (as noted in [31] and represented by \( \text{fun}(a) \)), the reduction relation \( \rightarrow \lambda \) in figure 3 is therefore not fully defined except for the novel feature of monitors given by rules grab, release where \( \text{rules} \) grab, release provide exclusive access and causes other actors that concurrently attempt to grab the same monitor to wait until the monitor is released.

Rule new defines an ESL term \( \langle E \vdash (E', b') \rangle_a \) where \( b' \) differs from that given in [31] to note that a new actor captures both the current context \( E \), but also creates its own local context \( E' \). Since ESL supports side effects, this allows actors to share state that can be managed via monitors.

Rule term applies when an actor exhausts its current message handler and becomes inactive. Rule recv shows how an inactive actor starts to process a message and rule snd describes message passing.

Rule time injects time messages into the actor community. ESL does not define when these messages occur - they are used to ensure that otherwise idle actors can regularly perform computation and are provided with time \( t \) in milliseconds since the start of the application.

The semantic relation \( \rightarrow \) defined in figure 3 places no further constraints on the order in which actor execution proceeds. Behaviour is highly concurrent and message passing is asynchronous making it difficult to trace threads of execution.

ESL is a statically typed language that merges features from functional programming and actor-languages. The filmstrip pattern that is defined in ESL in section 4 is independent of the data type used to represent the semantic domain used to structure the snapshots. Figure 4 defines that part of ESL type relation used by the examples in this paper. The relation is defined as

\[
\begin{align*}
\Gamma & \vdash t : T \\
\Gamma & \vdash t : T
\end{align*}
\]

that defines a function of type \( \forall T. \ (T) \rightarrow [T] \). When the function is used, the type must be supplied pair[Int[10]:Int].

A data definition (as shown in figure 2 on line 4 introduces a union type. Such a type defines a number of constructors; in this case PredAt and PreyAt which are used to inject values into the union data type. Therefore, PredAt(1, 20, 38) is a value of type Message. Values
of a union type can be projected onto their constituent elements using a case-expression. Values of a union type are also used as messages in ESL where the message handlers are used to project the values.

2.3 Sense-Making Requirements

Our aim is to determine whether or not the ESL program defined in figure 2 exhibits the behaviour we expect. In general, it is difficult to achieve this through instrumentation due to the highly concurrent and non-deterministic nature of actor computation. In principle we could apply static verification techniques to the program to investigate a required behaviour, however for applications of any size the state-space explosion makes this approach unusable. Therefore, we propose to use a post-execution, human-based, machine-assisted technique where the history of execution is analysed. Consider a system history for the predator-prey example:

```
PreyAt(10,20) // The prey starts at (10,20)
PredAt(1,20,10) // Predator 1 starts at (20,10)
PreyAt(9,28) // The prey moves to (9,28)
PredAt(1,19,11) // Predator 1 moves to (19,11)
```

Such a sequence of actions is difficult to interpret because the semantics of any global state is the aggregation of previous actions. Furthermore, some actions overwrite previous actions: the movement of predator 1 above. In order to make sense of any given history we propose a filmstrip that can be run forwards and backwards.

A filmstrip is a visual semantic description of the system in terms that allow us to spot issues of interest and to perform some sense-making analysis. A typical example of a filmstrip is shown in figure 5 where the sequence of predator-prey messages has been transformed into a sequence of snapshots displayed via a slider that can be dragged forwards and backwards to display different points in time. Figure 5a shows the predators moving towards the prey and figure 5b clearly shows the prey strategy to be unintelligent in time. Figure 5a shows the predators moving towards the prey and figure 5b clearly shows the prey strategy to be unintelligent in time. Figure 5a shows the predators moving towards the prey and figure 5b clearly shows the prey strategy to be unintelligent in time.

The use of system visualisation and filmstrips in particular is a known technique for parallel systems [4, 24] and for MAS [39]. Whilst these approaches acknowledge the need to integrate events, the context of MAS. The following features are required to create such an integrated structure:

![Figure 5: Filmstrip](image_url)

(a) Predators Move In  (b) Simple Prey Strategy

Figure 5: Filmstrip
Making Sense of Actor Behaviour: An Algebraic Filmstrip Pattern and its Implementation

event history The application must produce a history of events in a form that can be aggregated to produce snapshots as described below.

semantic domain The history consists of individual messages that must be aggregated based on a semantics for the application. It is useful if the semantic domain is compositional since the individual components of the history are produced by different actors.

state transitions The history of the application must be mapped to a sequence of state transitions each of which is defined as a semantic snapshot.

display domain Each snapshot must be mapped to a visual representation such as that described in figure 5. The displays should be designed so as to exhibit behaviour of interest.

3 THE FILMSTRIP PATTERN
This paper proposes a filmstrip pattern that is independent of the semantic and display domains that are used. The pattern places conditions on these domains but leaves the details to the particular applications. This section defines the pattern; section 3.1 provides some basic definitions, section 3.2 presents the pattern definition, section 3.3 describes how messages are translated to state transitions, section 3.4 describes how the transitions are translated to semantic values, section 3.5 describes how semantic values are translated to displays. The pattern is independent of particular data types, however section 3.6 states the properties for any data types that are used.

3.1 Basic Definitions
A monoid $(M, m_0, +)$ is a set of values $M$ together with a value $m_0 : M$ and an associative binary operation $+ : (M, M) \rightarrow M$ such that $m_0$ is the left and right identity of +. Given a list of values $xs : [X]$ the lists $\uparrow xs$ and $\downarrow xs$ are defined to be the prefix of $xs$ and the last element of $xs$ respectively such that $\uparrow xs + \downarrow xs = xs$. The function $[\downarrow]$ is defined as $[\downarrow] = [\downarrow]xs$. Given a function $f : X \rightarrow Y$, the function $f^+ : [X] \rightarrow [Y]$ maps $f$ over a list of type $[X]$ to produce a list of type $[Y]$. Given an associative binary operator $\ast : (Y, Y) \rightarrow Y$, and a value $y : Y$ the function $\gamma_{f, \ast, y}$ maps a list $[x_1, x_2, \ldots]$ to produce $f(x_1) \ast f(x_2) \ast \ldots \ast y$. $|l|$ is the length of the list $l$.

3.2 Pattern Definition
An actor-based system executes in terms of messages. When a message is received by an actor it may change state. The state changes can be recorded as events which, over the duration of an application, build up a history of execution. Each actor has a unique identity which can be used to tag the events it produces leading to a structure $(\langle E \rangle, i : E \rightarrow Int)$ of event histories.

A filmstrip $f : [D]$ is a sequence of display elements. The displays represent elements that can be drawn on a screen and have no knowledge of system executions. We would like to define a mapping $\omega : \langle E \rangle \rightarrow [D]$ from sequences of events to filmstrips that preserves a semantic structure that we define for the system. The data type for displays should be defined so that it forms a monoid $(D, \otimes, \delta)$ where $\delta : D$ is the empty display and $\omega : [E] \rightarrow [D]$ composes displays.

The mapping $\omega : \langle E \rangle \rightarrow [D]$ from histories to filmstrips is to be defined in terms of three mappings: $\sigma : \langle E \rangle \rightarrow \langle [E] \rangle$ that maps event histories to state transitions; $\mu^* : \langle [E] \rangle \rightarrow [S]$ that maps sequences of state transitions to sequences of semantic values; $\gamma^* : [S] \rightarrow [D]$ that maps sequences of semantic values to sequences of displays.

The pattern is defined in figure 6; each of the components are defined in the rest of this section.

3.3 Producing State Transitions
A system state can be expressed as a collection of facts that describe the current state of each actor. If the event history contains a record of the complete state of an actor each time it changes then a sequence of events can be transformed into a sequence of states by taking all the prefixes of the history. However, states produced in this way may contain contradictory facts about a given actor since the state may change over time. Therefore, we must filter the prefixes so that the latest state of each actor is retained. The mapping $\sigma$ is defined by specifying its inverse $\sigma^{-1} : \langle [E] \rangle \rightarrow \langle E \rangle$ such that the following two conditions hold:

\[
\langle \downarrow \rangle + \langle \downarrow \rangle (\text{ess}) = \text{es}
\]

\[
\forall j \in \{1, \ldots, \text{ess}\} \uparrow \text{ess}_j = [m \mid m \in \text{ess}_{j-1}, i(m) \neq \text{id} \langle \downarrow \rangle \text{ess}_j] \uparrow
\]

\[
\forall j \in \{1, \ldots, \text{ess}\} \langle \downarrow \rangle \text{ess}_j = \text{ess}_{j-1} + 1
\]

Condition 1 states that the concatenation of the last element of each state must produce the original history. Condition 2 states that the prefix of each state must not contain a message whose id is that of the suffix. Together, these conditions ensure that $\sigma$ generates a step-by-step state transition that does not contain contradictory information about any element. Condition 3 requires the state transitions to be incremental.

3.4 Producing Semantic Values
A key feature of the pattern is the requirement to define a semantic domain $S$ that is used as the anchor-point of filmstrip production. The semantics is defined in order to reflect the features of the domain that we would like to examine. For example in the case of the predator-prey scenario, the semantic domain is a world containing positions of the predators and prey. Other semantic domains may be more complex, however there is a requirement that the domain can be expressed as a monoid in order that it has an empty element and a composition operator. This allow the mapping between sequences of system states $\langle [E] \rangle$ and sequences of semantic values $[S]$ to be defined in terms of a simple mapping $e : E \rightarrow S$ between events and semantic values such that $\mu = \gamma_{e, \otimes, e}$ and therefore $\mu^* : \langle [E] \rangle \rightarrow [S]$ as required.
3.5 Producing Displays

Given that we have defined filmstrips as a monoid over displays it is possible to define the mapping \(\gamma^*\) in terms of a simple display mapping \(d : S \rightarrow D\) since this can be generalised in the same way as \(e\) above:

\[
y([1]) = [1] \\
y(e) = \delta \\
y(s) = d(s) \\
y(s_1 \oplus s_2) = y(s_1) \oplus g(s_2)
\]

The individual mappings \(e\) and \(d\) can be composed in order to translate directly from state transitions to displays:

\[
y \circ \mu = (\lambda d, \ominus, \delta) \circ (\lambda e, \ominus, \delta) = \lambda d e, \ominus, \delta
\]

Giving the following mapping:

\[
(y \circ \mu)^*: \{[E]\} \rightarrow \{D\}
\]

3.6 Key Types, Mappings and Filmstrip Laws

The filmstrip production pattern identifies several key definitions and some laws that the definitions must satisfy. The following key components must be provided: data types for events \(E\), semantics \(S\), and displays \(D\). The semantics and displays should be monoids, and the semantics may include further domain-specific constraints. The events should provide an identity mapping \(i\) that is the basis for a standard state-transition mapping \(\sigma\) that must satisfy the specification in section 3.3. The mapping from states to displays can be constructed from two mappings \(e\) and \(d\) from events to semantic values and from semantic values to displays respectively. The semantic value mapping and the display monoid must satisfy the equations defined in section 3.5. The next section uses the language ESL to implement the pattern.

4 PATTERN REPRESENTATION IN ESL

The previous section has defined a filmstrip pattern that is independent of any implementation language and the data types used for events, semantics, and the displays. This section uses polymorphic functions in ESL to define the pattern in terms of its constituent mappings: section 4.1 defines the state transition mapping and then section 4.2 defines \(\omega\) that expects the pattern component mappings as arguments.

4.1 State Transitions

State transitions are implemented using a structure \(([E], i: E \rightarrow \text{Int})\) and a mapping \(\sigma\) that maps a history of events to a sequence of state transitions where each state is a sub-sequence of the events. The ESL definition of \(\sigma\) is shown in figure 7. The function \(\text{combine}\) is used to ensure that the specification of \(\sigma\), as defined in section 3.3, is satisfied.

For example if \(E = \text{Message}\) then given the following sequence of messages:

\[
h = \{\text{PreyAt}(10,20), \\
\text{PredAt}(1,20,18), \\
\text{PredAt}(9,28), \\
\text{PredAt}(1,19,11), . . .\}
\]

\(\sigma[\text{Message}](h)\) produces:

\[
\begin{align*}
\text{combine}[E](i: (E) \rightarrow \text{Int}, ids: \{\text{Int}\}, h: [E], m: E): [E] = \\
\text{case } h ( \{ \} \rightarrow [E]; \\
\text{if member}[\text{Int}](i(m), ids) \text{ then } [E] \text{ else } [E]; \\
hh: [\text{Int}] + [\text{m}: E] \rightarrow [\text{Int}] + [\text{m}: E] \\
\text{combine}[E](i, ids, hh, mm); \\
\text{else } \text{combine}[E](i, ids + \{i,m\}, hh, mm) + [E]; \\
\})
\end{align*}
\]

**Figure 7: State Transitions in ESL**

\[
\begin{align*}
\text{map} [M,N](f: M \rightarrow \text{Int}, id: \{\text{Int}\}): [N] = \\
\text{case } id ( \{ \} \rightarrow [N]; \\
ms: \{M\} \rightarrow [\text{Int}]; \\
m: ms \rightarrow (f(n)): \text{map} [M,N](f, ms); \\
})
\end{align*}
\]

\[
\text{foldr} [M,N](\text{map}: M \text{ to } N, op: \text{ N to } \text{ N}, empty: \text{ N to } \text{ N}, list: \{M\}): [N] = \\
\text{case } list ( \{ \} \rightarrow \text{empty}; \\
\text{h: M to } \text{ N}: \text{ N to } \text{ N}; \\
hm: \text{list} \rightarrow op(\text{map}(h), \text{foldr}(\text{M}, \text{N})(\text{map}, \text{op}, \text{empty}, \text{t})); \\
})
\]

\[
\omega[E, S, D](events: \{E\}, \\
i: (E) \rightarrow \text{Int}, \\
e: (E) \rightarrow S, \\
d: (S) \rightarrow D, \\
\ominus: (D, D) \rightarrow D, \\
\delta: D \rightarrow [D] = \\
\text{let } ms: \{E\} \rightarrow \text{foldr}[E, D](\ominus, \delta, ms) \\
in \text{map}[E, D](n, \sigma[E](i, events))
\]

**Figure 8: Filmstrip Pattern Implemented in ESL**

\[
[[], \\
[\text{PreyAt}(10,20)], \\
[\text{PreyAt}(18,20)], \text{PredAt}(1,20,18)], \\
[\text{PredAt}(1,20,18), \text{PredAt}(9,28)], \\
[\text{PredAt}(9,28), \text{PredAt}(1,19,11)], . . .
\]

**Theorem 4.1.** The definition of \(\sigma\) given above satisfies the requirements 1, 2 and 3 given in section 3.3.

**Proof:** By induction on the length of \(h\).

4.2 Filmstrip Mapping in ESL

The filmstrip mapping \(\omega\) maps sequences of events to sequences of displays. It relies on constituent mappings as defined in figure 6 and combines them using \(\text{fold}\) (that implements \(\_\ominus\_\)) and \(\text{map}\) (that implements \(\_\delta\_\)). Figure 8 shows the definition of the mapping in ESL.

5 PREDATOR-PREY FILMSTRIPS IN ESL

Given a collection of messages generated by ESL actors, the filmstrip is created as an ESL sequence of display elements by supplying \(\omega\) with the messages, mappings and display monoid components:
The semantic mapping must translate a state into a semantic value.

The semantic domain structure for the predator-prey application is a board that contains locations. Each location can be empty, a rock, a predator or a prey. The location elements can be represented as a single data type and the board is a two-dimensional list:

```plaintext
data Location = EmptyLoc | PredLoc | PreyLoc | Rock;
type Board = [[Location]];```

The semantic domain should form a monoid. The following is the empty board:

```plaintext
emptyBoard:Board = [[if member[Pos](Point(x,y),rocks) then Rock else EmptyLoc | x:Int ← 0..width] | y:Int ← 0..height];
```

The monoid combination operation must be associative and have empty_board as a left and right identity assuming rocks are always in the same place:

```plaintext
mergeBoards(b1:Board,b2:Board):Board = [[
mergeLocs(b1[x][y],b2[x][y]) | x:Int ← width] | y:Int ← height];
```

mergeLocs(rock:Location,rock:Location):Location = Rock;
mergeLocs(rock:Location,empty:Location):Location = rock;
mergeLocs(empty:Location,rock:Location):Location = rock;
mergeLocs(empty:Location,empty:Location):Location = empty;
mergeLocs(prey:Location,rock:Location):Location = prey;
mergeLocs(rock:Location,pred:Location):Location = pred;
mergeLocs(prey:Location,pred:Location):Location = pred;

The semantic domain structure for the predator-prey application is therefore (Board,mergeLocs,emptyBoard).

### 5.2 Semantic Mapping

The semantic mapping must translate a state into a semantic value. Since the source and target of the semantic mapping both form a monoid, the mapping can be generated using a map for a single event as follows:

```plaintext
mapMessage(m:Message):Board = case m of
    PredAt(_,x0,y0:Int) → 
        [[ if (x=x0) and (y=y0) then PredLoc else if member[Pos](Point(x,y),rocks) then Rock else EmptyLoc | x:Int ← 0..width] | y:Int ← 0..height];
    PreyAt(_,x0,y0:Int) → [[ if (x=x0) and (y=y0) then PreyLoc
```

The semantic mapping can be defined as follows:

```plaintext
µ = foldr[Message,Board](mapMessage,mergeBoards,emptyBoard)
```

and generalised to \(µ^n\) using the definition of \(\mu\) as required. The history is therefore produced by copying forward snapshot fragments until an actor causes a change when it processes a message.

### 5.3 Displays

The filmstrips are represented as sequences of displays. ESL provides a number of display types that can be used to populate the filmstrip pattern. The predator-prey example can be displayed as a two-dimensional board that can be represented as a nested collection of trees containing horizontal and vertical boxes, shapes and images:

```plaintext
data Tree = 
    TreeNode((Shape)) // A picture made up of shapes.
    HBox([[Tree]]) // A box of elements arranged horizontally.
    VBox([[Tree]]) // A box of elements arranged vertically.
    Rectangle(Int,Int) // Rectangle(width,height).
    Circle(Int) // Circle(radius).
    Shape
    VBox((Tree)) // A box of elements arranged vertically.
    VBox((Tree)) // A box of elements arranged horizontally.
```

A space can be represented using:

```plaintext
space:Tree = TreeNode(Rectangle(size,size));
```

A simple two-dimensional board representing a predator-prey display can be represented as follows:

```plaintext
VBox(
    HBox((space,space,rock.png)),
    HBox((space,space,sheep.png)))
```

To comply with the filmstrip pattern, the language of displays must form a monoid. In the case of a two-dimensional predator-prey world the empty display is:

```plaintext
VBox(
    HBox((space,space)),
    HBox((space,space)))
```

The binary display combination operator is implemented as follows where \(I[1]\) indexes an element in a list:

```plaintext
mergeDisplays(d1:Tree,d2:Tree):Tree = 
    case d1,d2 of
        VBox(I1:Tree),VBox(I2:Tree) → 
            VBox((mergeDisplays(I1,I2) | 1:Int ← ⋆(I1)));
        HBox(I1:Tree),HBox(I2:Tree) → 
            HBox((mergeDisplays(I1,I2) | 1:Int ← ⋆(I1)));
        ⋆ when d1 = space → d2;
        ⋆ when d2 = space → d1;
```

Assuming that emptyDisplay is a tree of the appropriate size and shape that contains only spaces then the displays form a monoid (Tree,mergeDisplays,emptyDisplay) as required.
with ESL both in terms of the type system and the run-time in order to support a range of different displays that can be used to a virtual machine written in Java. Graphics libraries are integrated into the filmstrip produced by ESL.

Consider a 3-by-3 predator-prey world with a single rock at position (2,2). Given the following messages:

```
messages = [PreyAt(1,0),PredAt(1,0),PreyAt(1,1),PredAt(1,0),PredAt(1,0)]
```

the filmstrip produced by `filmstrip(messages)` is shown in figure 9.

5.4 Display Mapping

Since the semantic domain and displays both form monoids, the display mapping can be generated from a single map from a board to a tree:

```
rockIcon:Tree = TreeNode(Image(size,size,'rock.png'));
predIcon:Tree = TreeNode(Image(size,size,'wolf.png'));
preyIcon:Tree = TreeNode(Image(size,size,'sheep.png'));
emptyDisplay:Tree = mapBoard(emptyBoard);
```

```
mapBoard(b:Board):Tree =
let mapRow(row:[Location]):Tree =
    HBox([ case i 1 |
        PredLoc → predIcon;
        EmptyLoc → space;
        Rock → rockIcon
    ] | i:Location ← row )
in VBox([ mapRow(D[i]) | i:Int ← 0..height ])
```

5 IMPLEMENTATION

ESL source code is translated to an instruction set that is executed on a virtual machine written in Java. Graphics libraries are integrated with ESL both in terms of the type system and the run-time in order to support a range of different displays that can be used to show the results of actor-based programs. Figure 10 shows example filmstrips that have been produced by ESL applications.

Figure 10a shows a snapshot of an ESL filmstrip that simulates a shop consisting of assistants, customers and criminals. The customers wait for help while browsing and queue at tills to be served. Figure 10b shows the event and semantic domains for the shop: the customer, assistant and till ids are defined to be integers and a shop state is `Shop(o,a,b,h,t)` where o are the customers outside the shop, a are the assistants on the floor, b are the customers who are browsing, h represents customers waiting for help and being helped, and t are the tills that may have an assistant serving and have a (possibly empty) queue of customers.

Figure 10c shows a snapshot of an ESL filmstrip that simulates traffic flow at a junction. The junction state is shown in figure 10d where `Road(left_l,right_l,left_r,j,right_r)` contains the current state of the left and right traffic lights (`left_l` and `right_l`), the traffic flow queuing and leaving the left and right roads (`left_r` and `right_r`) and the car passing the single-track mid-point of the junction `j`.

Figure 10e shows a snapshot of an ESL filmstrip that implements the dining philosophers. The state is shown in figure 10f where `Dining(f,p)` contains the fork identifiers `f` and the philosopher states `p` where a philosopher state is `Phil(i,1,r)` where `i` is the identifier of the philosopher and `l` and `r` are the left and right forks or `nothing` if the philosopher is not holding a fork.

In all cases it is possible to map the events defined in figures 10b, 10d and 10f to their respective semantic data domains. Furthermore, it should be clear that the semantic domains form monoids based on an empty value and a binary composition operator.

7 RELATED WORK

The general notion of sense-making processes originated in the field of intelligence analysis through the seminal work of Pirolli and Card [32]. However, there is a growing amount of work that addresses the problems of the more specialised form of sense-making, debugging of actor and agent-based systems such as [39]. The latter for example, exhibits many of the characterisation of the Pirolli-Card sense-making process such as re-representation through multiple complementary abstractions of the underlying multi-agent system to identify new insights and actions.

Many of the implementation level concerns of sense-making and debugging are described in [25]. The tool in [28] produces static diagrams of agent communication topologies using a society tool. They support off-line video-style replay facilities with forward and backward video modes as a powerful sense-making aid, although the structure of agents seems to be fixed.

The challenges and an approach to source-level agent-based debugging is described in [22]. Several systems support run-time instrumentation of actor-based systems, for example [36] and [23] which monitors a run-time system for semantic properties. Our work differs from all these approaches since we aim to understand a system in terms of its solution-domain instead of the implementation-domain.

The use of system traces to address the challenge of understanding the behaviour of MAS is described by Bürdalo et al. in [6] where a standard model of trace data and an architecture that supports trace-processing is presented. The work in [6] does not describe
Making Sense of Actor Behaviour: 
An Algebraic Filmstrip Pattern and its Implementation

**Figure 10: Filmstrip Examples**

- **(a) Shop Filmstrip**
- **(b) Shop Event and Semantic Domains**
  ```plaintext
type Cid = Int;
type Aid = Int;
type Tid = Int;
data ShopE = NotInShop(Cid) | Browsing(Cid) | Queueing(Cid,Tid) | SeekingHelp(Cid) | GettingHelp(Cid,Aid) | OnFloor(Aid) | ATill(Aid,Tid);
data Helping = Help(Cid,Possibly[Aid]);
data Possibly[T] = Just(T) | Nothing;
data Till = Tilt(Tid,Possibly[Aid],[Cid]);
data ShopS = Shop([Cid],[Aid],[Cid],[Helping],[Till]);
```

- **(c) Traffic Filmstrip**
- **(d) Traffic Event and Semantic Domains**
  ```plaintext
type Vid = Int;
data TrafficLight = Left | Right;
data Colour = Red | Amber | Green;
data RoadE = QueueLeft(Vid) | QueueRight(Vid) | Advance(Vid) | LeaveLeft(Vid) | LeaveRight(Vid) | Change(TrafficLight,Colour);
data Road = Road([Vid],[Vid]);
data RoadS = Road(Colour,Colour,Road,Possibly[Vid],Road);
```

- **(e) Dining Philosophers Filmstrip**
- **(f) Dining Philosopher Event and Semantic Domains**
  ```plaintext
type Pid = Int;
type Fid = Int;
type Forks = [Fid];
data Side = Left | Right;
data Philosopher = Phil(Pid,Possibly[Fid],Possibly[Fid]);
type Philosophers = [Philosopher];
data DinerE = Pickup(Philosopher,Fid,Side) | Eat(Pid) | DropFork(Philosopher,Fid,Side);
data DinerS = Dining(Forks,Philosophers);
```

How to process the trace data and the pattern presented in this paper could be incorporated into that work.

Other approaches to sense-making include the interrogation of system traces and source-level debuggers. The visualization of Java execution traces in terms of object diagrams and sequence diagrams is proposed as a means for sense-making in [20]. Our approach provides a structured framework for defining many types of diagram including object and sequence. Queries are applied to AgentSpeak execution histories [41] in order to determine whether certain behaviours occurred. The ESL language supports similar queries (described in [10]) which are complementary to the animations described in this paper.

Model-checking can be used to formally express system properties of actor-based systems, for example [19] uses a model-checker called McErlang to check safety properties of Timed Rebeca that are translated to Erlang. Whilst this approach can be very successful for particular types of properties, we argue that the approach described in this paper has wider application and is more scalable.

Process event logs can be used as a basis for analysis of complex business applications. The event logs are similar to the histories described in this paper. In some cases visualisation has been used to compare different processes [5, 42], although there is no description of a general pattern for constructing the visual output.

Filmstrips, first attributed to D’Souza and Wills [12] provide important visual support for examining histories. In their approach, a filmstrip is a set of contiguous snapshots that describe how a system state evolves through a specific scenario. Filmstrips have also been applied in areas such as functional testing [7]. Efforts to incorporate filmstrips, include the recent efforts by Gogolla et al. use filmstrip models for automatic validation of model dynamics of applications [18]. Gil and Kent, in 1995, proposed the use of filmstrips as an important component for three dimensional software modelling in an effort to move away from a topological graph metaphor [17].
8 CONCLUSION

Actor-based systems exhibit non-deterministic behaviour that makes sense-making activities such as debugging challenging. Semantically based visualisation is a powerful tool in helping understand such execution. We have described how filmstrips can be used to examine histories of executions and hence function as a sense making tool. We have presented a generalisation of the necessary machinery (event histories, domain-independent filmstrip representation and the operations possible over the event histories) as an algebraic pattern. The pattern has potential for use in environments where agent based simulation histories are key output for analysis.

The pattern has been implemented in the open-source language ESL that supports both actors and polymorphic functions, and has been used to implement a number of actor-based applications including predator-prey, shop and traffic simulations, and dining philosophers.

REFERENCES


