A Homogeneous Actor-Based Monitor Language for Adaptive Behaviour

Tony Clark[1], Vinay Kulkarni[2], Souvik Barat[2(✉)], and Balbir Barn[3]

[1] Sheffield Hallam University, Sheffield, UK
[2] Tata Research Development and Design Centre, Pune, India
souvik.barat@tcs.com
[3] Middlesex University, London, UK

Abstract. This paper describes a structured approach to encoding monitors in an actor language. Within a configuration of actors, each of which publishes a history, a monitor is an independent actor that triggers an action based on patterns occurring in the histories. We define a monitor language based on linear temporal logic and show how it can be homogeneously embedded within an actor language. The approach is demonstrated through a number of examples and evaluated in terms of a real-world actor-based simulation.

Keywords: Monitors · Actors · Linear temporal logic
Homogeneous language embedding · Simulation

1 Introduction

1.1 Background

Software system monitors are instruments that are added to applications in order to collect data, diagnose problems and influence behaviour, for example to collect training data [21], recognise health issues [19], control of energy networks [7], dynamically verify system requirements [14] and to adapt to environmental changes [12]. Monitors are typically orthogonal to the system that is monitored and, as such, manage their own thread of control. Our particular interest in monitors arises within the field of simulation to support decision-making [4,20] where levers that control a simulation need to be adapted by monitoring the behaviour of system components as shown in Fig. 1.

The actor model of computation [2] would seem to be a good candidate for representing both systems to be monitored and their associated monitors because an actor has autonomous behaviour and new actors can be added or deleted without significant architectural disruption. Figure 2a shows a simple model of a monitor actor that is attached to a monitored actor. Adaptation occurs by regularly checking the history exported by the monitored actor and taking action when monitor conditions are satisfied by the data in the history.

A. Ricci and P. Haller (Eds.): Programming with Actors, LNCS 10789, pp. 216–244, 2018.
https://doi.org/10.1007/978-3-030-00302-9_8

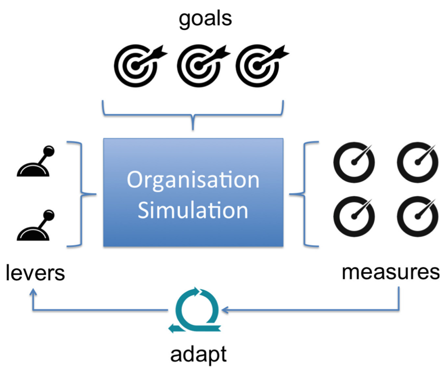
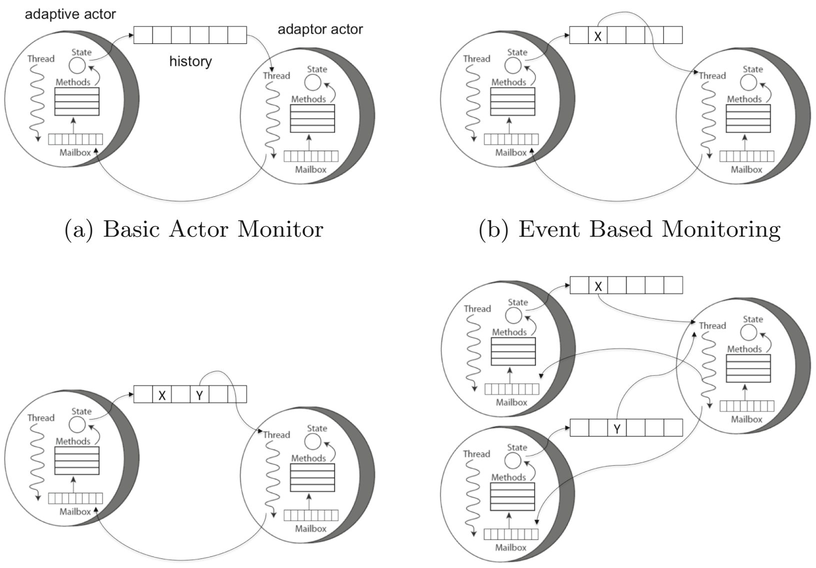


Fig. 1. System adaptation



(a) Basic Actor Monitor (b) Event Based Monitoring

(c) Temporal Relationship (d) Multi-Actor Monitoring

Fig. 2. Actor monitors

Several patterns emerge from the actor arrangement shown in Fig. 2a. An individual monitored actor that publishes events on its history can be monitored so that when the monitoring actor detects the event-data it can take action as shown in Fig. 2b. If the monitoring actor also takes into account the timing of events, it may compare the relative times of two or more events as shown in Fig. 2c. A single monitor may be attached to two or more actors as shown in Fig. 2d in which case we can define *intra*- and *inter*-actor relationships between events that occur over multiple actors.

1.2 Requirements and Contribution

The patterns described in Fig. 2 lead to technical requirements for actor-based monitors: (1) It must be possible to define an actor interface so that monitored actors provide a history in the correct format for monitor actors and are otherwise unaware of monitoring behaviour. (2) Events that are published via histories should have temporal information that monitor actors can use for intra- and inter-actor reasoning. (3) Monitors should be actors to allow them to be added and removed dynamically during system execution (and to support meta-monitoring where that is appropriate). (4) There should be a monitor language, whose denotation is defined in terms of actors, that is open to static analysis.

Our contribution is to show how monitors can be encoded within an actor language in a way that achieves the requirements listed above. In particular, we achieve a homogeneous representation by defining a sub-language of a broader actor language that supports monitors and define a general purpose interface for monitored actors. We perform static analysis, in terms of actor interfaces, and outline in the conclusion how we intend to check that monitors are consistent with the histories published by monitored actors. As described in [9], monitors can themselves be actors and we demonstrate this by defining the semantics of monitors as an homogeneous language embedding [23].

Section 2 describes existing work on actors, agent languages and software monitoring in general. Languages employing temporal operators are often used to specify the behaviour of actor systems, but we employ such operators to define a language of dynamic monitors in Sect. 3. Our work on simulation has led to the implementation of a new actor-based language called ESL that is used to present our contribution; Sect. 4 introduces that part of ESL necessary to understand how monitors are encoded. Section 5 describes how monitors are defined in ESL and Sect. 6 shows how ESL and monitors are used to implement a simulation for a real-world case study.

2 Related Work

The concept of a *monitor* that observes the behaviour of a system and detects the violations of safety properties is well studied area [16]. The early work on monitors focused on off-line monitoring where the historical data or program trace is statically analysed post-execution [13] to detect anomalies. In contrast, the recent research trend primarily addresses on online monitoring that dynamically validates the observed system execution against a specification to achieve preventive and corrective measures.

Monitoring-related research challenges include: (1) the use of an appropriate foundational model for system specification (this can be mapped to Monitoring-Oriented Programming[1]); (2) an efficient implementation of monitors; (3) monitor control patterns. Existing monitor technologies are largely based on temporal logic which has led us to use the same underlying formalism. For example, the

[1] http://fsl.cs.illinois.edu/index.php/Monitoring-Oriented_Programming.

past-time linear temporal logic (PTLTL) is recommended as specification language [18,22] where monitor properties are expressed using the notion of time, as introduced by LTL, with both past and future modalities including *next*, *previous*, *eventually* in the future or past, *always* in the future or past, *etc.* The PTLTL is further extended with call and returns in [24], the Eagle Logic (where the temporal logic is extended with *chop* operator) is proposed in [5], and the Meta Event Definition Language (a propositional temporal logic of events and conditions) is used in the Monitoring and Checking (MaC) toolset [26]. In contrast, an extended regular expression and context free grammar based language is proposed and implemented in [9].

From an implementation dimension, online monitoring technologies are implemented either by adopting an inline approach [25] or a centralised approach [15]. A centralised approach is applied to the system under observation using either a synchronous [10] mode or an asynchronous mode [11]. In general, the inline monitors are computationally efficient as they have access to all system variables, whereas the centralised monitors are better in other dimensions as they enable clear separation of concerns (and thus less error-prone), facilitate distributed computation, and exhibit compositional characteristics.

As part of our work on system simulation, we propose a variant of monitor technology that can be used to evaluate agent goals (similar to monitoring safety properties) and make decisions about appropriate system adaptation. A restricted form of centralised asynchronous monitor is implemented in Akka [3] to realise the monitoring behaviour of supervisor actors of a hierarchical actor system. Recently, a prototype of asynchronous centralised monitor implementation is presented in [9]. In the context of multi-agent systems, the concept of monitor is used for one of the two purposes - (a) to support heterogeneous agent implementations, and (b) to introduce a clear separation of concerns between core agent behaviour and other aspects such as adaptation. With respect to (b) the core agent behaviour (or behavioural specification) is considered to be a black-box, and adaptation is defined as monitor that observes the execution of core behaviour and reacts to specific observations [9].

ESL, simulation and adaptive behaviour are active areas of our research. Work in this area by [28] is likely to prove useful in efforts to specify the structure of histories, perhaps using pattern based rules, and then to verify that monitors are consistent with the monitored actors to which they are applied.

Our proposed language also provides centralised asynchronous monitors. The Akka implementation monitors the occurrence of events within its sub-actors and uses a fixed set of operations such as *stop actor*, *suspend actor* as an adaptation strategy. In contrast our proposed implementation evaluates the historical data to produce adaptation without using a restricted set of operations, effectively achieving a combination of [9] with the adaptation strategy presented in [8] augmented with temporal features and the extensions presented in [22] (*i.e.*, past and future modalities).

3 An Actor-Based Monitor Language

We have motivated a requirement for an actor-based monitor language that can be used in a variety of applications including diagnosis and adaptive behaviour. Since monitors need to reason about behaviour over time, it is reasonable to follow existing approaches to both actor behavioural specification and system monitoring that are based on linear temporal logic (LTL). We show how the proposed LTL-based monitor language can be encoded using actors. This section describes the monitor language and provides a simple example. The rest of the paper provides the semantics of the language by embedding it in an actor language called ESL, and then evaluates the language using a real-world case study.

3.1 Language Definition

Figure 3 defines a monitor construction language. A monitor `p` is applied to a monitored actor that exposes a history as a sequence of data elements. The monitor `p` can *hold* for the supplied history *at a given time*, where time is an integer that indexes the data in the history. Typically time will start with the first element of the history and the modal operator **N** is used to advance time so that `p;` **N**`(q)` holds for a history starting at time `0` when `p` holds at time `0` and `q` holds at time `1`. The formula $\Box$`(p)` is equivalent to `p;`**N**`(p;`**N**`(p;...))`.

The language allows arbitrary conditions `?(c)` to be applied to data in a history at a given time. In order to support actions and system instrumentation, a monitor may be an action `!(a)` where `a` is a function of 0-arguments. The intention is that we use guard monitors of the form: `p` $\Rightarrow$ `!(a)`.

In order to support multi-actor monitoring as shown in Fig. 2d we allow monitored actors to be aggregated using a binary operator. Monitors can manipulate such actors using the disaggregation operator _ $\uparrow$ _ (split) and its inverse _ $\downarrow$ _ (join). For example two monitored actors `a` and `b` can be aggregated to create `Fork(Leaf(a),Leaf(b))`. When such an actor is monitored by `(p` $\uparrow$ `q);r`, `p` monitors `a`, `q` monitors `b` and `r` monitors both `a` and `b`. Note that the _ $\downarrow$ _ operator

```
p,q ::=                  monitors.
      n                  a named monitor.
    | ε                  holds for any history.
    | □(p)               holds when p holds at all times.
    | μ(λ(n)p)           provides recursive definitions: [μ(λ(n)p)/n]p
    | p;q                holds when p and q both hold now.
    | p|q                holds when p or q (or both) holds now.
    | p ⊕ q              holds when p or q (but not both) holds now.
    | p ⇒ q              if p holds then q must hold now.
    | N(p)               holds when p holds for time now + 1.
    | P(p)               holds when p holds for time now - 1.
    | ?(c)               holds when the condition c is true now.
    | !(a)               always holds and performs action a.
    | p ↑ q              splits two merged histories between p and q.
    | p ↓ q              (re)merges two split histories.
```

Fig. 3. Actor monitor language

is implicitly inserted between `(p ↑ q)` and `r` thereby ensuring that the monitor types are consistent.

3.2 A Simple Application of Monitors

The monitor language is influenced by LTL since it uses modal operators to range over time within histories. LTL is used by Bulling *et al.* to specify agent behaviour [6] with respect to an example involving traffic queues. We will use this example to show that an LTL-based monitor language can be used to help adapt traffic-light behaviour.

Traffic flow at an east-west bottle-neck along a single-carriage road is controlled by traffic lights. The bottle-neck can only accommodate one car and therefore the job of the traffic-lights is to ensure that the queues on either side do not become too large. Furthermore, traffic approaches from the east more frequently than that from the west.

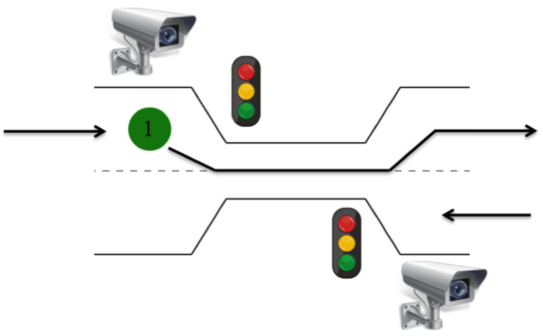
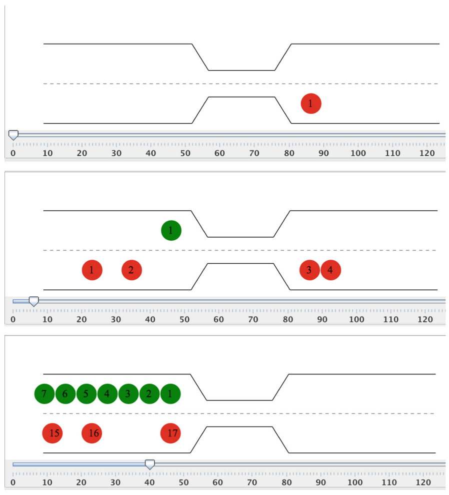


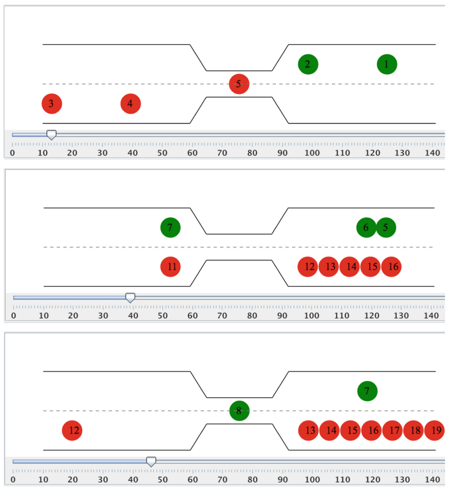
Fig. 4. Traffic flow at a bottle-neck (Color figure online)

Figure 4 shows the situation. The approaches from east and west are represented as actors whose autonomous behaviour supplies cars to their respective queues. The traffic lights are assumed to be in-sync and are passive actors. A single monitor combines data from the cameras that detect the current queue levels at the two approaches.

We will consider the simulation under three levels of monitor control: (1) To test the simulation we leave one of the traffic lights stuck on red and the other on green; (2) The monitor is even-handed and allows traffic to pass where there is a queue; (3) The monitor gives preference to the traffic from the east unless a queue has formed on the western approach and has not moved for a given time period.

The simulation has been encoded in ESL which includes a facility to produce a *filmstrip* consisting of a sequence of diagrams generated from simulation data. Figure 5 shows filmstrips generated using the three different monitor strategies. Figure 5a shows three snapshots at times 0, 6 and 40: the lights are stuck on green


(a) Stuck


(b) Even

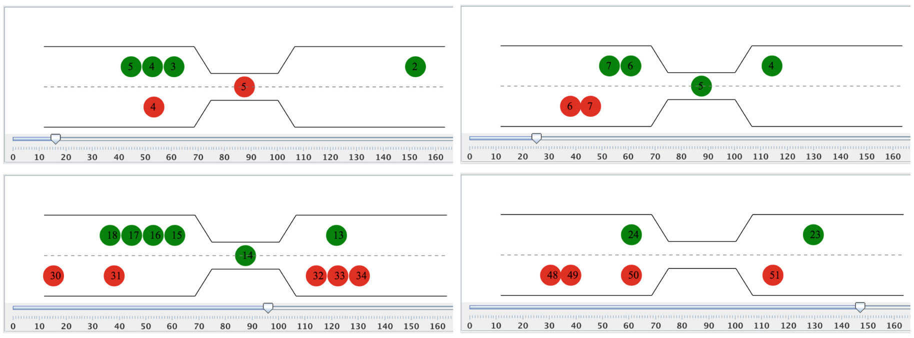

(c) Priority

Fig. 5. Actor monitors (Color figure online)

for east-west and red for west-east and therefore a queue soon builds up on the western approach. Figure 5b shows the situation with an even-handed monitor where, since the east-west traffic is more frequent, a queue quickly builds up. Figure 5c shows the situation at four different times where the monitor prefers east-west traffic, but attempts to prevent queues lingering for too long at the west-east approach. At time 17 a queue has developed and the monitor adapts the traffic light behaviour to clear it by time 25. At time 96 queues have built up on both approaches, but the strategy has cleared them by time 148. Based on the assumptions made by the simulation, the third monitor strategy would seem to be a good one and can be encoded in the monitor language as follows:

```
P0(p) = p                                                       1
Pn(p) = p;P(Pn-1(p))                                            2
□(PmaxQueueDuration(?(gre(maxQueueSize))) ↑ ϵ ⇒ !(westEast) ⊕   3
                                ϵ ↑ ?(gre(0)) ⇒ !(westEast) ⊕   4
                                ?(gre(0)) ↑ ϵ ⇒ !(eastWest) ⊕   5
                                ϵ)                              6
```

The definition **Pn** is used to define a monitor that must hold over `n` previous time units. The monitor consists of three rules that hold at all times and is applied to a pair of monitored actors (the west and east traffic cameras) whose histories can be indexed using the split operator $\_ \uparrow \_$. The first rule on line 3, states that if the west approach has a queue length `maxQueueSize` that has been waiting for `maxQueueDuration` time units then the lights should be changed `westEast`. The rule on line 4 gives queuing traffic on the eastern approach precedence over that queuing on the western approach. A monitor must be satisfied, so ϵ on line 6 ensures that we can move on to the next time unit.

4 ESL

The previous section has introduced a language for actor based monitors. The language has been implemented as a library using ESL[2] which is an actor language designed to support simulations. ESL combines functional and actor-based programming and provides a number of novel features that we believe are important when representing systems with emergent behaviour. The rest of this paper introduces ESL and uses it to define the monitor library. We will conclude with a real-world simulation written in ESL that uses monitors. Section 4.1 provides a brief overview of ESL syntax, Sect. 4.2 describes the operational semantics of ESL in terms of a system executive, and Sect. 4.3 implements a standard actor-based example and shows how it executes concurrently in ESL.

4.1 Syntax

The syntax of ESL is shown in Fig. 6. It is statically typed and includes parametric polymorphism, algebraic types and recursive types. Types start with an upper-case letter. An ESL program is a collection of mutually recursive bindings. Behaviour types **Act** `{ ... }` are the equivalent of component interfaces and behaviours **act** `{ ... }` are equivalent to component definitions. A behaviour `b` is *instantiated* to produce an actor using **new** `b` in the same way that class definitions are instantiated in Java. Once created, an actor starts executing a new thread of control that handles messages that are sent asynchronously between actors. Pattern matching is used in arms that occur in **case**-expressions and message handling rules. Uncertainty is supported by **probably**`(p) x y` that evaluates `x` in `p%` of cases, otherwise it evaluates `q`. Functions differ from actors because they are invoked synchronously.

[2] http://tonyclark.github.io/ESL/.

```
exp  ::= name                              variable
      |  num | bool | str                  constant
      |  self                              receiver
      |  null                              undefined
      |  new name[type*] (exp*)            creation
      |  become name[type*] (exp*)         change
      |  exp op exp                        binexp
      |  not exp                           negation
      |  λ(dec*)::type exp                 operation
      |  let bind* in exp                  locals
      |  letrec bind* in exp               recursion
      |  case exp* arm*                    matching
      |  for pat in exp { exp }            looping
      |  { exp* }                          block
      |  if exp then exp else exp          tests
      |  [ exp* ]                          list
      |  []                                empty
      |  exp(exp*)                         apply
      |  Name(exp*)                        term
      |  exp ← exp                         message
      |  name := exp                       update
      |  exp . name                        reference
      |  probably(exp)::type exp exp       uncertain
      |  exp[type*]                        app type

pat  ::= dec                               variable
      |  dec = pattern                     naming
      |  num | bool | str                  const
      |  pat : pat                         cons pair
      |  [ pat* ]                          list
      |  [][type]                          empty
      |  Name[type*](pat*)                 term
```

```
type ::= Name                              type var
      |  Act { export dec* Mes* }          behaviour
      |  (type*) → type                    λ-type
      |  Term                              term type
      |  Int | Bool | Str                  constant
      |  Void                              undefined
      |  [ type ]                          lists
      |  Fun(Name*) type                   type op
      |  ∀(Name*) type                     poly
      |  rec Name . type                   recursion

bind ::= dec = exp                         bind
      |  name(pat*)::type=exp when exp     λ-bind
      |  type Name[Name*] = type           type dec
      |  data Name[Name*] = Term*          algebraic
      |  act name(dec*)::type {            behaviour
           export name*                    interface
           bind*                           locals
           → exp                           initial
           arm*                            behaviour
         }

Term ::= Name(type*)                       term type

dec  ::= name[Name*] :: type               declare

arm  ::= pat* → exp when exp               guarded
```

Fig. 6. ESL syntax

A minimal ESL application defines a single behaviour called `main`, for example:

```
1 type Main = Act{ Time(Int) };
2 act main::Main {
3   Time(100) → stopAll();
4   Time(n::Int) → {}
5 }
```

An ESL application is driven by time messages. The listing defines a behaviour type (line 1) for any actor that can process a message of the form `Time(n)` where `n` is an integer. In this case, the `main` behaviour defines two message handling rules. When an actor processes a message it tries each of the rules in turn and fires the first rule that matches. The rule on line 3 matches at time `100` and calls the system function `stopAll()` which will halt the application. Otherwise, nothing happens (line 4).

4.2 Operational Semantics

ESL compiles to a virtual machine code that runs in Java. Each actor is its own machine and thereby runs its own thread of control. Figure 7 shows the ESL executive that controls the pool of actors. When the executive is called, the global pool `ACTORS` contains at least one actor that starts the simulation. Global time and the current instruction count are initialised (lines 3 and 4)

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

Fig. 7. The ESL executive

before entering the main loop at line 4; the loop continues until one of the actors executes a system call to change the variable `stop`.

Lines 6–7 copy the global pool `ACTORS` so that freshly created actors do not start until the next iteration. If an actor's thread of control has terminated (line 9) then a new thread is created on the actor's VM by scheduling the next message if it is available. The operation `run` continues with the actor's thread of control on a machine that runs a call-by-value functional language [1] extended with actor-based features such as asynchronous message passing.

The executive schedules each actor for `MAX_INSTRS` VM instructions. This ensures that all actors are treated fairly. Once each actor has been scheduled, the existing actors are merged with any freshly created actors (line 13).

The executive measures time in terms of VM instructions. Each clock-time in the simulation consists of `INSTRS_PER_ TIME_UNIT` instructions performed on each actor. When actors need to be informed of a clock-tick (line 14), global time is incremented (line 16), the instruction counter is reset (line 17) and all actors are sent a clock-tick message.

4.3 Factorial

ESL Messages are sent asynchronously between actors. An actor that is at rest selects a new message and processes it in a thread that is independent of other actor threads. When the thread terminates, the actor is ready to process the next message. Consider the concurrent processing of factorials:

```
type Customer = Act { Value(Int) };
type Fact = Act{ Get(Int,Customer) };

act fact::Fact {
  Get(0,c::Customer) → c ← Value(1);
  Get(n::Int,c::Customer) →
```

```
    let cc::Customer = new cust(n,c)
    in self ← Get(n-1,cc)
}

act cust(n::Int,c::Customer)::Customer {
  Value(m::Int) c ← Value(n*m)
}

act main::Customer {
  f::Fact = new fact;
  computeFact(n::Int)::Void = f ← Get(n,self);
  → { computeFact(6); computeFact(6); computeFact(6) }
  Value(n::Int) → print[Int](n)
}
```

An actor of type `Customer` receives an integer value `Value(n)`. An actor of type `Fact` receives a request `Get(n,c)` for the value `!n` to be sent to the customer `c`. The behaviour `main` implements `Customer` and is the end-point for factorial-calculations: when it receives `Value(n)` it prints out the result.

The behaviour `fact` implements `Fact` using two message-handling rules. The first handles `0` and just passes the value `1` to the supplied customer. The second rule receives a request for a factorial of a non-0 number `n`. An actor with behaviour `fact` is able to handle multiple factorial requests at the same time. To do this it creates an auxiliary customer `cc` that is used to handle the return value from `!(n-1)`: this is equivalent to distributing the linked stack-frames of conventional singly-threaded computation to an equivalent number of concurrent customer actors. The behaviour `cust` implements `Customer` and forwards `n*m` to the pending customer `c`.

An actor with behaviour `main` calls `computeFact(6)` three times. Figure 8 shows the resulting message traces as a sequence diagram. Each of the actors is shown as a box with a life-line. Messages sent between actors are shown as arrows between life-lines and are labelled using the following convention `TIME:[THREAD]MESSAGE` where `TIME` is the time at which the message is sent and `THREAD` is a label used to identify each separate calculation of `computeFact(6)`. Messages are encoded to show the different steps: `Start` is the initial message, `Get(n,c,cc)` is a message to calculate `!n` with `cc` as the intermediate customer and `c` as the requesting customer, `One` is the recursion termination step, and `Return(n)` shows the return values between customers.

The sequence diagram shows that the three factorial calculations occur concurrently. The customer actors correspond to conventional stack frames in a singly-threaded language where the calls to factorials occur in sequence and are appropriately nested. Mapping actors to conventional languages has been the subject of several research projects [17,27] where monitors such as those described in this article may be an interesting implementation consideration.

5 Monitor Implementation in ESL

Section 3 introduced an actor-based monitor language and Sect. 4 describes the actor language ESL. This section shows how monitors can be implemented as actors by defining a pair of actor types: `Mtd[T]` for a class of behaviours that

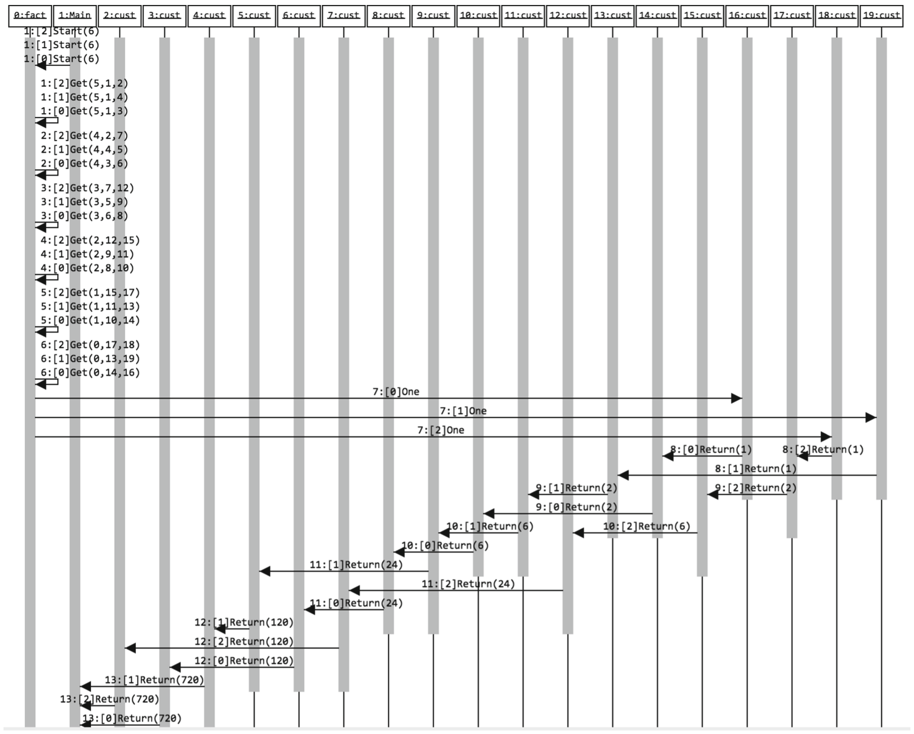


Fig. 8. Concurrent factorial

export histories over type `T`, and `Mtr[T]` for a class of behaviours that monitor histories of type `T`.

Given that actors are autonomous, we require a mechanism to synchronise monitors and actor histories. The basic mechanism is described in Sect. 5.1 and then encoded in ESL as described in Sect. 5.2.

5.1 Processing Histories

A history is a list of public state information and as such each monitor processes a list of data. For example, suppose that we want to express a monitor `fff` that causes action `a` to be performed every time a sequence of 3 fails, `000`, is detected:

```
anF(0)::Bool = true
anF(1)::Bool = false
aT(n::Int)::Bool = not(anF(n))
any(n::Int)::Bool = anF(n) or aT(n)

pxn[T](n::Int,p::(T) → Bool)::Mtr[T] =
  case n {
    0 → idle
    else ?(p) ; N(pxn(n-1,p)))
  }

fff::Mtr[Int] = □((pxn(3,anF) ⇒ !(a)) ⊕ ?(any))
```

The predicates `anF` and `any` are defined to detect the appropriate state elements. The history formula `fff` uses the operator `_;_` to compose three `F` detectors one after another in the history. The operator **N** is used to advance through the history. Finally, the history predicate `fff` combines the three `F` detector with an alternative detector ?`(any)` that skips a state value. The monitor `p` ⇒ `q` checks `p` first, if `p` fails then `q` is checked, so line 12 will use three `F`'s as a guard on the action `a`, if the guard fails then the head of the history is skipped.

The history of an actor is produced incrementally over time. Therefore an expression written in the language defined in Fig. 3 must continually monitor the actor's history. The expression can be thought of as a state machine whose nodes correspond to monitor states and whose transitions consume parts of actor histories. Each transition is triggered by a clock-tick and can proceed when there is some history to consume, otherwise the machine must stay in its current state and try again when the next tick occurs.

Figure 9 shows the machine corresponding to `fff`. Each transition is triggered by a clock-tick, the labels on the transitions are: ϵ when no history is available; `F` occurs when the next state element in the history is an `F`; ○ occurs when there is at least one element at the head of the history and causes the element to be consumed; * denotes the situation when the next state element in the history is anything but `F`.

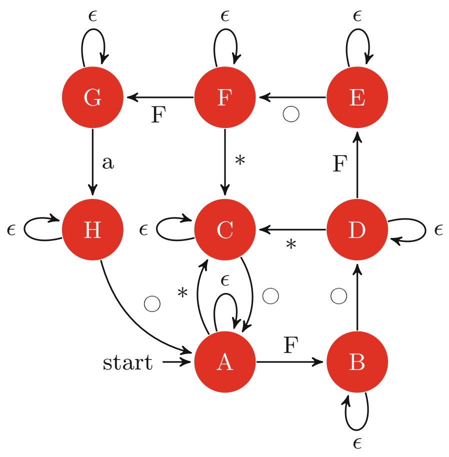


Fig. 9. A monitor state machine

5.2 Monitor Implementation

The implementation of monitors in ESL has four parts: (1) An actor type for a class of monitored behaviours, (2) A compositional data type that allows monitored actors to be combined, (3) An actor type for a class of monitors,

(4) The definition of the operators in Fig. 3. This section addresses each of these parts in turn.

Monitored Behaviours: An actor that can be monitored must export a list of data values called `history`. The order of the values in the history is important since it corresponds to the temporal operators in the monitor language. The actual type of the data elements in the list is not important so the definition of the type `Mtd` is parametric with respect to the type `T`:

```
type Mtd[T] = Act {
  export history::[T];
  Time(Int)
}
```

Composition: As shown in Fig. 2, monitors may be attached to single or multiple actors. Therefore, we require a mechanism that will combine the histories of two actors into a single history. Such a binary operator, can then be used successively to compose an arbitrarily large history that can be processed by a single monitor.

The type `MTree[T]` is the type of potentially aggregated monitored actors. Given a single monitored actor `m::Mtd[T]`, we create an aggregate singleton `Leaf(m)`. Given two aggregate monitored actors: `m1::MTree[T]` and `m2::MTree[T]` the composition `Fork(m1,m2)::MTree[T]` is also an aggregate monitored actor. The data type is defined below:

```
data MTree[T] = Leaf(Mtd[T])  | Fork(MTree[T],MTree[T])
```

Monitor Behaviours: A monitor `m::Mtr[T]` can be sent a message `Check(a,c,s,f)` that causes it to check whether it holds for the monitored actor `a`. The integer `c` is an index into the history that is exported by `a` and represents the *current time*. Checking whether the monitor is satisfied or not by `a` at time `c` may be delayed while `m` waits for `a` to produce the required amount of history. Therefore, `m` must keep polling `a` for its history until it can determine whether it is satisfied. At this point `m` is either satisfied with the history or not. These two outcomes are handled by the arguments `s` and `f` in terms of *success* and *fail* continuations. The argument `s` is a monitor to which `m` will send `a` when it is satisfied. The argument `f` is an operation that is invoked in the case that `m` is not satisfied with `a` at time `c`. The data type is as follows:

```
type Fail = () → Void
type Mtr[T] = rec M. Act {
  Check(Mtd[T],Int,M,Fail);
  Time(Int)
}
```

Operator Definition: The monitor behaviours are defined in Fig. 10, note that where the clock-tick handler is `Time(n::Int) → {}` it is omitted. A monitor of type `T` is created by instantiating a behaviour with type `Mtr[T]`; for example, **new** `ϵ[T]()`. A monitor is activated by sending it a `Check(a,c,s,f)` message where `a` is a tree of monitored actors, `c` is the current history-index (initially `0`), s is a success-monitor, and f is a failure-monitor. The behaviour of each type of monitor, as defined in Fig. 10, is outlined below:

```
act ϵ[T]::Mtr[T] {
  Check(a::MTree[T],c::Int,s::Mtr[T],f::Fail) →
    s ← Check(a,c,self,f)
}

idle[T]::Mtr[T] = new ϵ[T]()

act ![T](command::() → Void)::Mtr[T] {
  Check(a::MTree[T],c::Int,s::Mtr[T],f::Fail) → {
    command();
    s ← Check(a,c,idle[T],f)
  }
}

act (_;_)[T](p::Mtr[T],q::Mtr[T])::Mtr[T] {
  Check(a::MTree[T],c::Int,s::Mtr[T],f::Fail) →
    p ← Check(a,c,new (q;s)[T],f)
}

act (_|_)[T](p::Mtr[T],q::Mtr[T])::Mtr[T] {
  Check(a::MTree[T],c::Int,s::Mtr[T],f::Fail) →
    p ← Check(a,c,s,λ()::Void q ← Check(a,c,s,
      f))
}

act (_⊕_)[T](p::Mtr[T],q::Mtr[T])::Mtr[T] {
  Check(a::MTree[T],c::Int,s::Mtr[T],f::Fail) → {
    p ← Check(a,c,s,f);
    q ← Check(a,c,s,f)
  }
}

□[T](p::Mtr[T])::Mtr[T] =
  new μ[T](λ(q::Mtr[T])::Mtr[T]
    new seq[T](p,new next[T](q)))

act N[T](p::Mtr[T])::Mtr[T] {
  Check(a::MTree[T],c::Int,s::Mtr[T],f::Fail) →
    p ← Check(a,c+1,s,f)
}

act (_ ⇒ _)[T](p::Mtr[T],q::Mtr[T])::Mtr[T] =
  Check(a::MTree[T],c::Int,s::Mtr[T],f::Fail) →
    p ← Check(a,c,new (q;s),λ()::Void s ←
      Check(a,c,idle[T],f)
}
```

```
act P[T](p::Mtr[T])::Mtr[T] {
  Check(a::MTree[T],c::Int,s::Mtr[T],f::Fail) →
    p ← Check(a,c-1,s,f)
}

act ?[T](pred::(T) → Bool)::Mtr[T] {
  Check(t::MTree[T],c::Int,s::Mtr[T],f::Fail) →
    case t {
      Leaf(a::Mtd[T]) →
        if length[T](a.history) > c
        then {
          if pred(nth[T](a.history,c))
          then s ← Check(t,c,idle[T],f)
          else f()
        } else self ← Check(t,c,s,f)
    }
}

act μ[T](g::(Mtr[T]) → Mtr[T])::Mtr[T] {
  Check(a::MTree[T],c::Int,s::Mtr[T],f::Fail) →
    g(new rec[T](g)) ← Check(a,c,s,f)
}

act (_↑_)[T](p::Mtr[T],q::Mtr[T])::Mtr[T] {
  Check(t::MTree[T],c::Int,s::Mtr[T],f::Fail) →
    case t {
      Fork[T](t1::MTree[T],t2::MTree[T]) →
        let j::Mtr[T] = new (t1 ↓ t2)
        in {
          p ← Check(t1,c,j,f);
          q ← Check(t2,c,j,f)
        }
    }
}

act (_↓_)[T](t1::MTree[T],t2::MTree[T])::Mtr[T]{
  done::Bool = false
  Check(a::MTree[T],c::Int,s::Mtr[T],f::Fail) →
    if not(done)
    then done := true
    else s ← Check(Fork(t1,t2),c,s,f)
}
```

Fig. 10. ESL monitor behaviours

- ϵ() immediately activates the success-monitor by sending it a message. The global actor idle can be used as the identity monitor.
- !(c) receives a Check message, performs the command c, and then activates the success-monitor.
- p;q forwards the Check message to p and creates a new success-monitor q;s.
- p|q forwards the Check message to p using q as the fail-monitor. Therefore, q will be tried in the case that p fails. Both p and q will use f as their fail-monitor.
- p ⊕ q tries both p and q in parallel and assumes that only one will succeed.

- $\square$(p) checks the monitored actor using p with respect to history-indices `c`, `c+1`, `c+2` *etc.* Note that checking occurs in parallel with all other monitors and any check with respect to a particular history-index will wait, due to the definition of `?(_)`, until the indexed history element has been generated by the monitored actor.
- **N**(p) checks the monitored actor using p with respect to history-index `c+1`.
- `p` $\Rightarrow$ `q` if `p` holds then `q` should also hold, otherwise the fail-monitor is used.
- **P**`(p)` checks the monitored actor using p with respect to history-index `c-1`.
- `?(g)` when this monitor is activated by a `Check` message, the history element at index `c` is checked using guard `g`. If the result is `true` then the success-monitor is activated, otherwise the fail-monitor is activated. Note that if an element at index `c` is not yet available, the monitor sends itself a `Check` message that will be processed at some time in the future, thereby delaying the guard. Note also that the monitored actor is actually a tree: it is the responsibility of the monitor to use `_↑_` and `_↓_` to access a leaf of the tree when applying a guard.
- μ`(g)` recursive monitors are created by supplying a function `g` whose argument is a cyclic monitor. For example:

  ```
  μ[Int](λ(fStar::Mtr[Int])::Mtr[Int]  isF; N(fStar))
  ```

  is a monitor that will expect a history to contain an infinite sequence of `0`s.

5.3 Traffic Monitoring

Section 3.2 describes a simple use of monitors to achieve adaptive behaviour at a traffic light. This section provides the ESL implementation of the simulation and shows how the adaptor language works by providing a fragment of the resulting sequence diagram.

The ESL program in Fig. 11 is a slightly simplified version of that which generates the outputs shown in Fig. 5 where the details of generating sequence diagram output have been omitted. The behaviour type `Approach` is a sub-type of `Mtd[Int]` and exports a history of integers being the time-sequenced number of cars queuing at an approach. An approach actor is created by supplying the behaviour `approach` with an identifier, a traffic light and the probability of new car arrival. Two approaches called `left` and `right` are created and the operations `westEast` and `eastWest` are used to control the traffic lights.

Each approach is autonomous and receives a `Time(n)` message at regular intervals. When this occurs, either a new car will arrive or the next available car will move from the approach if the lights are green.

The monitor defined in Sect. 3.2 is attached to the monitored actors `left` and `right` by creating an aggregate `Fork(Leaf(left),Leaf(right))::MTree[Int]`. Figure 12 shows the initial steps performed by the simulation and its associated monitor. The actors are labelled with their unique identifier and behaviour where `E` stands for the behaviour `nothing`, `L` is the left approach and `R` is the right approach. Messages of the form `Check(a,c,s,f)` are represented on the sequence diagram as `C(a,c,s)` since the failure continuation is not particularly informative. Message passing starts at actor `14:rec` which is created by the monitor definition

```
type TrafficLight = Act { export colour::Str; change::() → Void }
type Approach = Act < Mtd[Int] { export history::[Int]; Time(Int) }

act light(colour::Str)::TrafficLight {
  export colour, change;
  change()::Void =
    case colour {
      'RED' → colour := 'GREEN';
      'GREEN' → colour := 'RED'
    }
}

act approach(id::Str,light::TrafficLight,probOfNewCar::Int)::Approach {
  export history;
  history::[Int] = [0];
  Move        → if light.colour = 'GREEN' and head[Int](history) > 0 then self ← DeQueue;
  DeQueue     → queue := (head[Int](history) - 1) : history;
  Queue       → queue := (head[Int](history) + 1) : history
  Time(n::Int) → probably(probOfNewCar) self ← Queue else self ← Move
};

l1::TrafficLight = new light('RED');
l2::TrafficLight = new light('GREEN');
left::Approach = new approach('left',l1,10);
right::Approach = new approach('right',l2,20);

westEast()::Void =  if l2.colour = 'RED' then { l2.change(); l1.change() }
eastWest()::Void =  if l1.colour = 'RED' then { l1.change(); l2.change() }
```

Fig. 11. The traffic simulation

for □. The **rec** behaviour creates a loop that is used to increment the history counter `c` that starts at `0` and is incremented twice by successive messages from `18:`**`next`** to `14:`**`rec`**. Notice that the left and right approach actors concurrently queue and de-queue cars at the same time that the monitor is processing their histories.

6 Evaluation

Our claim is that actor-based systems can benefit from monitors that meet the requirements outlined in Sect. 1.2 and in particular actor-based simulations can use monitors to encode adaptive behaviour. We have proposed a monitor-language and shown how it can be encoded in an actor language called ESL. This section evaluates the claims by describing a real-world case study that we have implemented in ESL and that uses monitor-based adaptation to influence actor behaviour.

6.1 Case Study Overview

The cash in circulation in the Indian economy has steadily been increasing over the years. In 2001, the total cash in circulation was 2.1 trillion rupees and by early November it had reached 15.4 trillion. On November 8, 2016, the dominance of cash-based transactions and the relentless growth of a shadow economy triggered a sudden fiscal intervention by the Indian government with the withdrawal from circulation of 500 Rupees and 1000 Rupees notes. This action resulted in 87% of the total cash in the system being pulled out. The primary objective of this

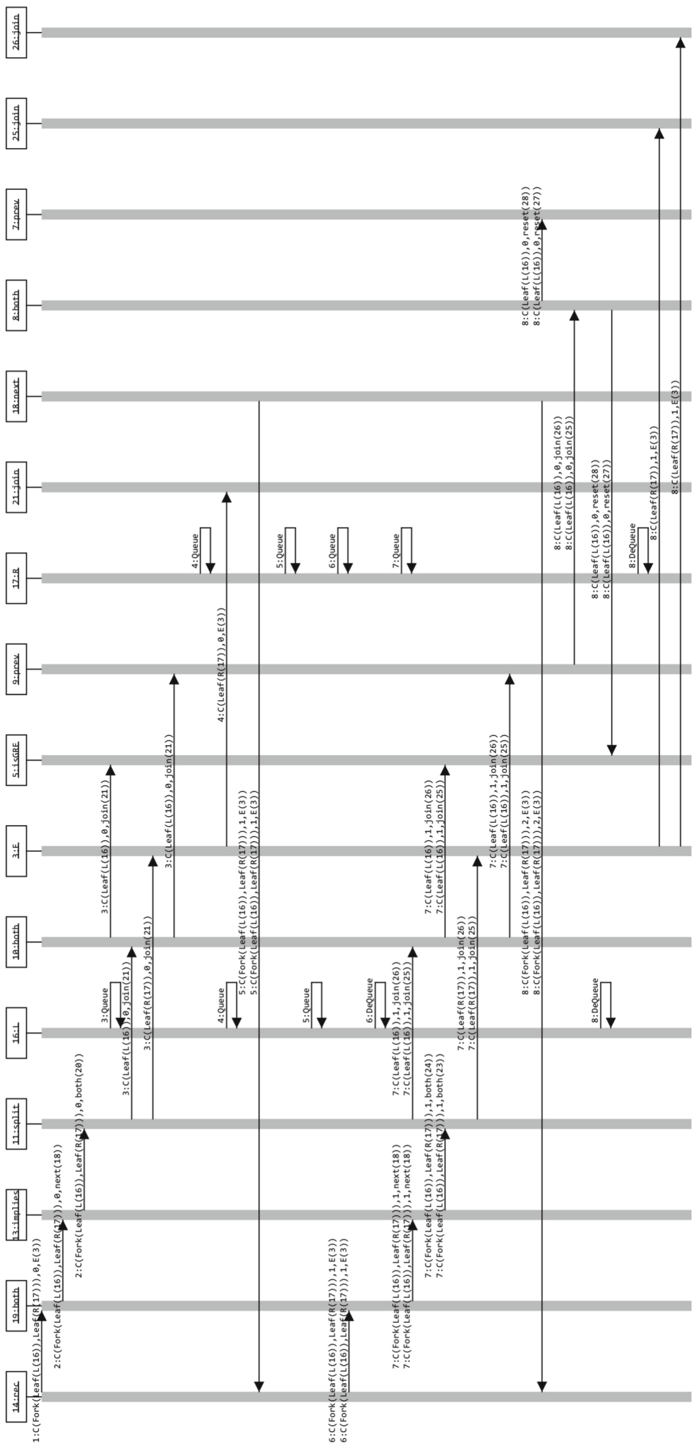

14:rec
19:both
13:implies
11:split
16:L
10:both
3:E
5:isGRE
9:prev
17:R
21:join
18:next
8:both
7:prev
25:join
26:join
1:C(Fork(Leaf(L(16)),Leaf(R(17))),0,E(3))
2:C(Fork(Leaf(L(16)),Leaf(R(17))),0,next(18))
2:C(Fork(Leaf(L(16)),Leaf(R(17))),0,both(20))
3:Queue
3:C(Leaf(L(16)),0,join(21))
3:C(Leaf(R(17)),0,join(21))
3:C(Leaf(L(16)),0,join(21))
3:C(Leaf(L(16)),0,join(21))
4:Queue
4:C(Leaf(R(17)),0,E(3))
4:Queue
5:C(Fork(Leaf(L(16)),Leaf(R(17))),1,E(3))
5:C(Fork(Leaf(L(16)),Leaf(R(17))),1,E(3))
5:Queue
5:Queue
6:C(Fork(Leaf(L(16)),Leaf(R(17))),1,E(3))
6:C(Fork(Leaf(L(16)),Leaf(R(17))),1,E(3))
6:DeQueue
6:Queue
7:C(Fork(Leaf(L(16)),Leaf(R(17))),1,next(18))
7:C(Fork(Leaf(L(16)),Leaf(R(17))),1,next(18))
7:C(Fork(Leaf(L(16)),Leaf(R(17))),1,both(24))
7:C(Fork(Leaf(L(16)),Leaf(R(17))),1,both(23))
7:C(Leaf(L(16)),1,join(26))
7:C(Leaf(L(16)),1,join(25))
7:C(Leaf(R(17)),1,join(26))
7:C(Leaf(R(17)),1,join(25))
7:C(Leaf(L(16)),1,join(26))
7:C(Leaf(L(16)),1,join(25))
7:C(Leaf(L(16)),1,join(26))
7:C(Leaf(L(16)),1,join(25))
7:Queue
8:C(Fork(Leaf(L(16)),Leaf(R(17))),2,E(3))
8:C(Fork(Leaf(L(16)),Leaf(R(17))),2,E(3))
8:DeQueue
8:C(Leaf(L(16)),0,join(26))
8:C(Leaf(L(16)),0,join(25))
8:C(Leaf(L(16)),0,reset(28))
8:C(Leaf(L(16)),0,reset(27))
8:DeQueue
8:C(Leaf(R(17)),1,E(3))
8:C(Leaf(R(17)),1,E(3))
8:C(Leaf(L(16)),0,reset(28))
8:C(Leaf(L(16)),0,reset(27))

Fig. 12. Initial messages sent by traffic monitor

demonetisation was seen as a way of purging black money from the system with the key assumption that such a large amount of money would not come back to the system as holders of illicit wealth would be wary of prosecution by tax authorities. Further, cash would be slowly replenished with newly minted currency notes.

The initiative involved several financial restrictions. For example, a limitations were imposed on the exchange of old notes wherein the citizens were allowed to exchange up to 4000 rupees with the remaining deposited to their bank account; ATM withdrawal limits were reduced to 2000 rupees in a day for Indian citizens, and there was a cap of 10,000 rupees on bank withdrawal in a day along with a weekly withdrawal restriction of 20000 rupees per citizen. In addition to these restrictions, cash-less payment modes, such as mobile wallet and card payments, were incentivised. For example, on Dec 8, e-transactions for fuel included a 0.75% discount. Despite all preventative measures, the demonetisation initiative resulted into prolonged cash shortages. Citizens were inconvenienced due to non-availability of new currency notes in the banks and ATM machines. Even as recently as Feb 16, estimates indicate that at least 30% of ATMs still run dry. Overall, the initiative has faced a lot of criticism as being poorly thought through, inadequately planned, inefficiently executed and unfair to a significant segment of cash dependent citizen[3].

6.2 Adaptation

When creating a simulation of the demonetisation case study, adaptation occurs in a number of ways. The banks, commercial suppliers, and citizens were continuously monitoring and adopting their behaviour to cope with the emerging situation. For example, banking transaction limits were changed multiple times to control cash flow, commercial suppliers adopted alternative payment options to stay viable, and citizens changed their behaviour to avoid undesired consequences.

The behaviour of citizens changed along multiple dimensions: (1) individuals started using alternate payment modes such as mobile wallet and credit/debit cards; (2) individuals changed their needs and suppliers catering to those needs so as to support cash-less transactions; (3) some individuals felt a greater sense of security in having cash-in-hand in excess of their requirements, *i.e.* hoarding behaviour emerged.

These adaptations to individual behaviour collectively impacted the overall system in a non-linear manner. In particular, the frequent changes to banking transaction limits, uncertainty in availability of cash with banks and ATM machines, circular dependencies between availability of cash and behaviour of individuals, and non-linearity in cash-in-hand of an individual and cash hoarding tendency led to an emergent system behaviour.

[3] http://bit.ly/2mpgGRb.

The use of actors and actor-monitors within a simulation can help to understand the effect of demonetisation. Actors are used to encode the individual behaviours and monitors are used for post-demonetisation adaptations.

6.3 Case Study Model

We formulate a society comprised of three key identities: Citizens, Banks, Shops, and a basic element termed Item as shown using a class diagram in Fig. 13. The term Item represents the needs of citizens that include merchandise and services; shops are the locations where items can be purchased and services can be acquired. The activities that we consider: citizens consume items to cater to daily needs; citizens purchase items from shops when the item quantity dips below a threshold value; citizens withdraw cash when cash-in-hand dips below a threshold value. A citizen may hold Cards, and a citizen who holds a card may choose to pay by cash or by card for a purchase, and may withdraw cash from ATM machine or bank counter. In contrast, a citizen without a card always pays by cash and withdraws cash from bank counters. The purchase behaviour and cash withdrawal behaviour are illustrated using state-machines in Fig. 14 (the firm lines describe the pre-demonetisation behaviour).

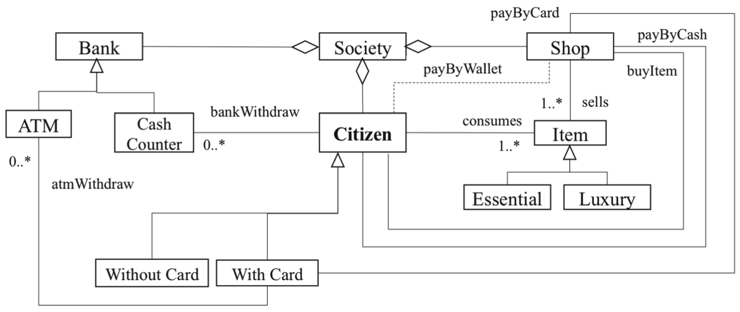
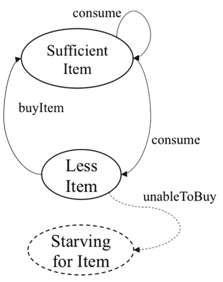
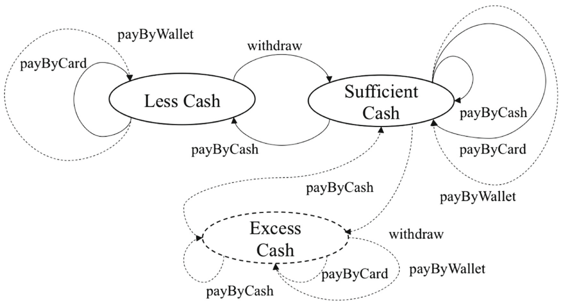


Fig. 13. Structural representation of society

We assume citizens are able to satisfy their daily needs *i.e.*, poverty and other societal aspects are not considered in the case study. We further consider: there is sufficient cash with the banks to service citizens through ATMs and Bank counters *i.e.*, no denial of service from bank; there is sufficient stock in shops; and citizens are able to withdraw cash when in need during pre-demonetisation phase. We replicate demonetisation by eliminating cash abruptly from banks and citizens, and replenishing cash at slow rate (around 0.7% of cash in circulation at pre-demonetisation) up to a certain percentage (*e.g.*, 70% of cash in circulation at pre-demonetisation). The key identities of society start behaving differently during post-demonetisation phase. They adopt different strategies which are very



(a) Purchasing



(b) Cash Withdrawal

Fig. 14. Citizen behaviour

specific to individuals. The adaptation strategies considered in this case study are:

(1) Bank: banks impose restrictions on cash withdrawals *e.g.*, ATM withdrawal limit is 2000 rupees in a day for a citizen, bank withdrawal limit is 10,000 rupees in a day for a citizen, and the weekly withdrawal limit is 20000 rupees per citizen. These changes are deterministic and associated with the demonetisation event.

(2) Shop: shops may (a stochastic behaviour) adapt themselves to accept alternate payment options such as mobile wallet and card payment whenever they observed a drop in sales record.

(3) Citizen: a citizen, as an individual, may adopt (as a stochastic behaviour) an appropriate strategy (with multiple options selected based on personal intuition and experience) to avoid entering an undesired state. The strategies can be visualised along two independent dimensions: **Payment Pattern:** Citizens start using mobile wallet and/or card as a payment option to save cash for the future. However, not everyone will start using alternate option, an individual's decision will be based on several factors such as availability and familiarity with payment technology, and whether the citizen is an early or late adopter. **Cash Withdrawal Pattern:** A group of citizens may start making attempt to withdraw cash (from ATM machine and/or Bank counter) even when the cash is not required (temporary hoarding behaviour) to safe guard from future consequences.

6.4 Case Study Monitors and Adaptation

The case study exhibits a variety of adaptive behaviour each of which is realised using monitors of the types shown in Fig. 2. For example, a shop tries to understand the situation in terms of its sales target, and adapts if the sales target is not met. A citizen observes the financial status (of itself and others) and adapts depending on circumstances; for example, one citizen may choose an alternate payment option as a mobile wallet, whereas another citizen may adapt to cash hoarding behaviour to avoid a cash shortage.

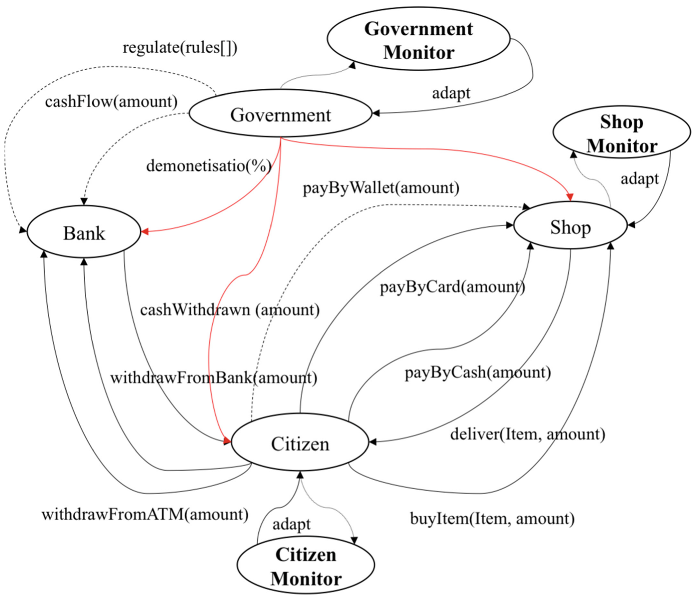


Fig. 15. Case study adaptors

In this case study, the shops and citizens exhibit significant individualistic behaviours and often adapt new strategies to deal with changing situations. A set of shops may change their behaviour proactively immediately after the demonetisation event, whereas another set of shops may wait until they observe a significant decline in sales before changing their behaviour. We attach a single event based monitor to represent the former scenario, and a simple history based monitor to represent latter.

A citizen may adapt their behaviour (a) right after the demonetisation event, (b) on demonetisation followed by several consecutive failures to authorise transactions, (c) when they observe low cash-in-hand for a given number of days and are unable to withdraw cash, or (d) when they observe a given number of citizens with multiple authorisation failures. Thus we see the need for all types of monitors to represent citizen adaptation. In contrast to citizen and shop actors, the bank actor primarily follows standard regulations that are defined by a government actor. Hence we associate a monitor actor with the government actor and allow government to control banks through their event interactions.

6.5 Simulation Organisation

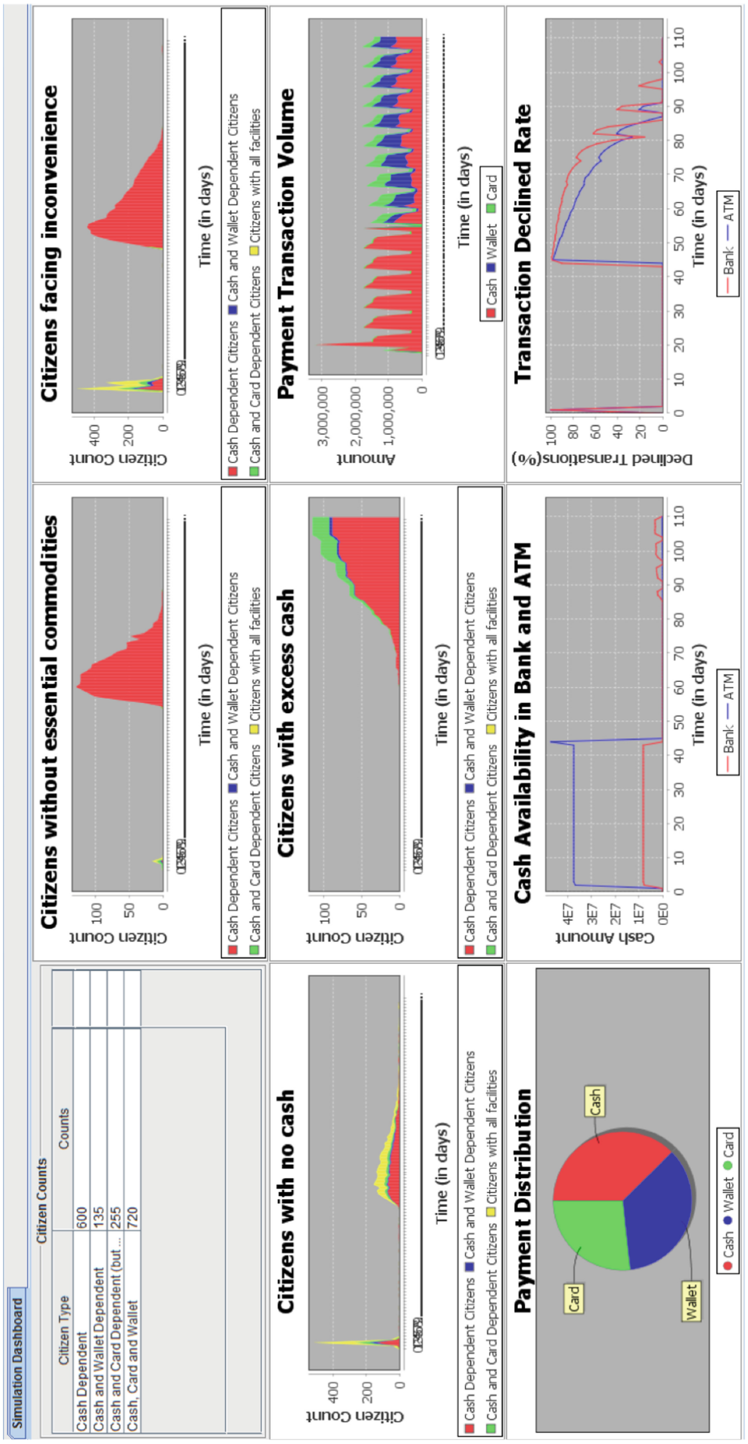
We set up a simulation by forming a society with actors representing the government, citizens, shops, and banks. Monitors are attached to shop and citizen actors for specifying the individualistic adaptation. The simulation progresses with a time event that represents a ‘day’. Each day, citizen actors consume items, buy items from shops if any item is below a certain threshold, pay for the purchases, and make an attempt to withdraw cash if needed. Similarly, bank actors try to stock up cash to fulfil ATM and Bank withdrawal requests, and shop actors stock up the items for their customers. The government actor triggers 'demonetisation' at specific day of a simulation run.

We divide a simulation run into three phases: setup, pre-demonetisation, and post-demonetisation. Figure 16 shows part of the simulation: the levers are displayed in the top left hand corner and the measures are displayed in real-time as the simulation progresses. Pre-demonetisation is an observation phase to validate conditions that include (a) cash at banks are adequate to serve all citizens *i.e.*, no denial of service at ATM machines and Bank counters; (b) the stock are sufficient at each shops to serve their citizens; and (c) citizens can buy items and withdraw cash as needed. This phase also monitors the cash-flows to the banks, and cash in circulation.

In this simulation setting, we firstly observe the impact of demonetisation by removing cash and reducing cash-flows. We then explore various what-if scenarios by changing the parameters and/or by attaching different actor monitors to understand the impact of courses of action. For example: the impact of demonetisation if the government replenishes cash at a faster rate, or the impact if the government decides to replenish 60% cash instead of 70%. Similarly, one can explore the impact if none of the citizen exhibits cash hoarding behaviour by detaching the monitors that are responsible for cash hoarding behaviours. In this paper, we limit our analyses to two scenarios (a) understand the effect of demonetisation for standard setting that closely represents the real Indian demonetisation event, and (b) understand the positive/negative effect on overall society when there is no hoarders. We have chosen these two relatively intuitive scenarios to illustrate the efficacy of using actors and monitors.

6.6 Simulation Results

We simulated the demonetisation case study with one government, one bank, 15 shop and 1710 citizen actors for 150 days, where the first 15 days are considered for setup phase, next 30 days are the pre-demonetisation phase, and 105 days are the post-demonetisation phase. A snapshot of ESL simulation dashboard at the day of 115 (*i.e.*, after 70 days of demonetisation) is depicted in Fig. 16. The dashboard shows useful states of the society and its identities: the 'Citizen Type' table describes the citizens and their card/wallet usage capabilities, (b) the 'Payment Distribution' pie chart shows distribution of Card (green), Wallet (blue) and Cash (red) payments, (c) the 'Payment Transaction Volume' chart describes the history of overall payment transactions where card transactions are green, wallet transactions are blue, and cash transactions are red, (d) the 'Cash Availability in Bank and ATM' graph shows the history of cash availability at Banks and ATMs using red and blue respectively, (e) the 'Transaction Declined Rate' graph describes the denial of service at Banks and ATMs using red and blue respectively. In addition, the 'Citizen with no Cash' and 'Citizen with excess Cash' charts describe the financial condition of the citizens: the former chart


Simulation Dashboard

Citizen Counts

Citizen Type	Counts
Cash Dependent	600
Cash and Wallet Dependent	135
Cash and Card Dependent (but ...	255
Cash, Card and Wallet	720

Citizens with no cash
Citizens without essential commodities
Citizens facing inconvenience
Citizens with excess cash
Payment Transaction Volume
Payment Distribution
Cash Availability in Bank and ATM
Transaction Declined Rate
Citizen Count
Amount
Cash Amount
Declined Transactions(%)
Time (in days)
🟥 Cash Dependent Citizens 🟦 Cash and Wallet Dependent Citizens
🟩 Cash and Card Dependent Citizens 🟨 Citizens with all facilities
🟥 Cash 🟦 Wallet 🟩 Card
● Cash ● Wallet ● Card
Cash
Wallet
Card
— Bank — ATM

Fig. 16. Case study dashboard

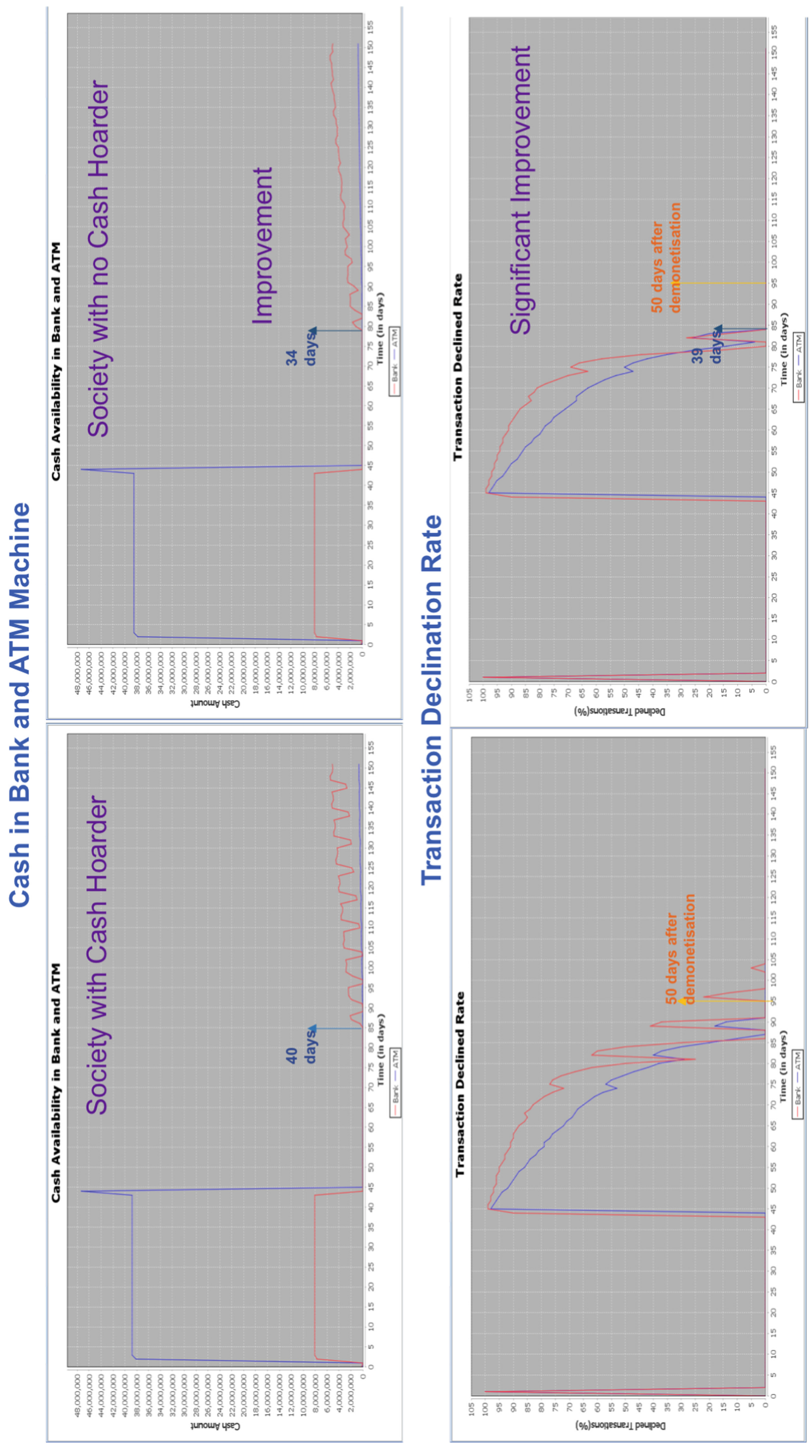


Fig. 17. Case study results

describes the number of citizens having considerably less cash, and the latter represents the number of cash hoarders (the red, blue, green and yellow colours signify the cash dependent citizens, cash and wallet dependent citizens, cash and card dependent citizens, and citizen with all facilities respectively as classified in table). The 'Citizens without essential commodities' and 'Citizen facing inconvenience' charts represent the number of citizens starving for essential items and luxury items respectively.

We observe that the graphs are unstable for first 15 days of the simulation runs as the simulator is trying to set the values based on actor behaviours and their interactions. The simulation outcomes for pre-demonetisation phase is stable: no bank withdrawal request is denied, no citizen is facing any financial crisis, and citizens are not having any deficiency for essential or luxury items. The demonetisation event is triggered at day 45, causing a sudden reduction of cash from the bank and ATM machines. Subsequently, the withdrawals from bank and ATM decline whilst wallet payment and card payment increase significantly: the citizens have started facing a financial crisis and the citizens who are solely dependent on cash have started starving for essential and/or luxury items. The adverse effects continue for almost 50 days and then the situation returns to normal.

As we can observe in graph with title 'Citizen with excess cash' in Fig. 16, 115 citizens started hoarding cash when the situation is on the verge of returning back to normal. One may hypothesise that cash hoarding behaviour is significantly slowing down the stabilisation process. We validate the hypothesis on hoarding behaviour by removing the monitors that are responsible for turning a citizen into a cash hoarder. We simulated the same society with no hoarders, and relevant simulation results are depicted in Fig. 17. We observe improvements in 'cash in bank and ATM' and 'transaction declination rate' for the society with no hoarders. The cash condition is returning back to normal in 40 days instead of 50 days. Thus the result is supporting the hypothesis regarding hoarding behaviour and also providing an indication of possible improvement.

7 Conclusion

This paper has proposed a homogeneous actor-based language for monitors that achieve adaptive behaviour. It is interesting because we have used the prevailing LTL-based approach to expressing behaviours in actor and multi-agent systems in order to define monitors that are also actors and therefore can be freely mixed with other actors at run-time. The language has been given a semantics by defining it using ESL which is a function-oriented actor language and we have demonstrated the utility of the approach using a real-world case study based on the recent demonetisation event in India. The case study demonstrates how monitors are used in the context of a simulation that exhibits emergent behaviour.

A number of limitations are identified in this work. The example described in Sect. 6 involves several thousand actors, and completes a simulation in roughly 30 s. We recognise the need to scale up to more realistic actor configurations which may then require further investigation in how to make actor behaviour, including adaptation, more efficient. A larger simulation model of our case study would also allow us to calibrate more precisely the simulation results with real world events as documented in the Indian national newspapers. As with all actor based simulations, the results need to be carefully interpreted given the underlying assumptions which we make given the complexity of the example. Regardless, the ability of the language to define actors with their own behaviours and adaptations and the flexibility for testing different conditions provide a means evaluating different policies and options. The ability to visualise and quantify the simulations results is also promising but again, we recognise that much more work on visualisation is required. Time in this simulation remains challenging however, we are similar in our approach to other efforts to map simulation time with real world time. Our monitor language semantics is currently defined using ESL which is currently in use in our research groups. It may be appropriate to document other options for defining the semantics of the monitor language to demonstrate other external validity routes.

ESL, simulation and adaptive behaviour are active areas of our research. An interesting extension of this work would be to specify the structure of histories, perhaps using pattern based rules, and then to verify that monitors are consistent with the monitored actors to which they are applied. Other case studies are also being explored to validate the technology and to produce a simulation development method based on ESL.

References

1. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming, pp. 8–19. ACM (2003)
2. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)
3. Allen, J.: Effective Akka. O'Reilly Media Inc. (2013)
4. Bharat, S., Kulkarni, V., Clark, T., Barn, B.: A simulation-based aid for organisational decision-making. In: Maciaszek, L.A., Cardoso, J.S., Ludwig, A., van Sinderen, M., Cabello, E. (eds.) Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016), ICSOFT-PT, Lisbon, Portugal, 24–26 July 2016, vol. 2, pp. 109–116. SciTePress (2016)
5. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_5
6. Bulling, N., Dastani, M., Knobbout, M.: Monitoring norm violations in multi-agent systems. In Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems, pp. 491–498. International Foundation for Autonomous Agents and Multiagent Systems (2013)

7. Cahn, A., Hoyos, J., Hulse, M., Keller, E.: Software-defined energy communication networks: from substation automation to future smart grids. In: 2013 IEEE International Conference on Smart Grid Communications (SmartGridComm), pp. 558–563. IEEE (2013)
8. Cassar, I., Francalanza, A.: Runtime adaptation for actor systems. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 38–54. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_3
9. Cassar, I., Francalanza, A.: On implementing a monitor-oriented programming framework for actor systems. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 176–192. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_12
10. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. In: ACM SIGPLAN Notices, vol. 42, pp. 569–588. ACM (2007)
11. Colombo, C., Francalanza, A., Gatt, R.: Elarva: a monitoring tool for Erlang. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 370–374. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_29
12. D'Ippolito, N., Braberman, V., Kramer, J., Magee, J., Sykes, D., Uchitel, S.: Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: Proceedings of the 36th International Conference on Software Engineering, pp. 688–699. ACM (2014)
13. Dodd, P.S., Ravishankar, C.H.: Monitoring and debugging distributed real-time programs. Softw., Pract. Exper. **22**(10), 863–877 (1992)
14. Fotrousi, F., Fricker, S.A.: QoE probe: a requirement-monitoring tool. In: REFSQ Workshops (2016)
15. Francalanza, A.: A theory of monitors. In: Jacobs, B., Löding, C. (eds.) FoSSaCS 2016. LNCS, vol. 9634, pp. 145–161. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49630-5_9
16. Goodloe, A.E., Pike, L.: Monitoring distributed real-time systems: a survey and future directions (2010)
17. Haller, P., Odersky, M.: Scala Actors: unifying thread-based and event-based programming. Theoret. Comput. Sci. **410**(2), 202–220 (2009)
18. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
19. Juen, J., Cheng, Q., Prieto-Centurion, V., Krishnan, J.A., Schatz, B.: Health monitors for chronic disease by gait analysis with mobile phones. Telemed. e-Health **20**(11), 1035–1041 (2014)
20. Kulkarni, V., Barat, S., Clark, T., Barn, B.: 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, 30 September–2 October 2015, pp. 368–377 (2015)
21. Müller, H., Bosse, S., Wirth, M., Turowski, K.: Collaborative software performance engineering for enterprise applications. In: Proceedings of the 50th Hawaii International Conference on System Sciences (2017)
22. Pradella, M., San Pietro, P., Spoletini, P., Morzenti, A.: Practical model checking of LTL with past. In: ATVA03: 1st Workshop on Automated Technology for Verification and Analysis (2003)
23. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 380–404. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_19

24. Roşu, G., Chen, F., Ball, T.: Synthesizing monitors for safety properties: this time with calls and returns. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 51–68. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89247-2_4
25. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Proceedings of the 26th International Conference on Software Engineering, pp. 418–427. IEEE Computer Society (2004)
26. Sokolsky, O., Sammapun, U., Lee, I., Kim, J.: Run-time checking of dynamic properties. Electron. Notes Theoret. Comput. Sci. **144**(4), 91–108 (2006)
27. Srinivasan, S., Mycroft, A.: Kilim: isolation-typed actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_6
28. Yasutake, S., Watanabe, T.: Actario: a framework for reasoning about actor systems. In: Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE) (2015)