Reactive Actors in Isolation for Efficient Analysis Reactive Actors: Isolation for Efficient Analysis

Marjan Sirjani
IDT Department
MDH
Vasteras, Sweden
marjan.sirjani@mdh.se

Ehsan Khamespanah
School of Computer Sceince
Reykjavik University)
City, Country
email address

Fatemeh Ghassemi
School of ECE
University of Tehran)
City, Country
email address

Abstract—**In this paper, we will introduce timed reactive actors for modeling distributed systems and will explain our theories, techniques and tools for model checking and performance evaluation of such models. Rebeca can be used to model asynchronous event-based components in systems, and in the extended version, Timed Rebeca, real time constraints can be captured in the language. We will explain how the isolation of actors, meaning no shared variable, non-preemptive execution of event handlers, and no blocking send or receive, help in developing more efficient analysis techniques. We show how floating-time transition system can be used for model checking of such models when we are interested in event-based properties, and how it helps in state space reduction. We explain how isolation of actors helps in distributed model checking? ... and how it helps in verification of MANETs. The same approach can be used in verification of swarms ...**

Index Terms—**Actors, Real-time systems**

I. Introcution

Message of the paper:

The actor-based language, Rebeca, provides a usable and analyzable model for distributed, concurrent, event-based asynchronous systems (Cyber-Physical systems).

Floating Time Transition System is a natural event-based semantics for timed actors, giving us a significant amount of reduction in the state space, using a non-trivial novel idea.

How models shape the thought and ease the analysis

a) Reactive Systems:

b) Reactive Actors:

c) Faithful Models for Reactive Systems: From Rocco paper: a major challenge in designing languages is to devise appropriate abstractions and linguistic primitives to deal with the specificities of the domain under investigation.

From Gul and Rajesh: programming language should facilitate the process of writing programs by being close to the conceptual level at which a programmer thinks about a problem, rather than at the level at which it may be implemented.

II. Floating Time Transition System

Active objects and actors are encapsulated modules with no shared variables. We also choose to have atomic execution of message servers (i.e. methods, event handlers) which gives us a macro-step semantics and models a non-preemptive execution of the handlers. Our actors are reactive, when sending a message they are not blocked and there is no explicit receive. So, there is no coupling via shared variables, no coupling because of waiting for another actor to return a value for a remote procedure call, and no coupling because of a context dependency caused by having a future construct in the language. This isolation helps in more efficient analysis, and reduces the state space. Moreover, if we are only interested in event-based properties we may be able to abstract even more and just keep the states that are followed by an event which we are interested in. This type of reduction is not straight forward as we need to prove that we are preserving the order of the events.

Floating Time Transition System is proposed based on the above discussion. What we mean by floating time is that in each state of the state space, different actors do not necessary have the same local clock. Actors are not synchronised on their local time in the state space. We consider this as letting the time *float* across the actors in the state space. To avoid confusion, it is important to note the different models in different levels of abstraction, and also layering of models. We have (1) distributed systems, we use (2) Timed Rebeca to model distributed systems, and we model (3) the state space as Floating Time Transition System to do the analysis. Note that at the level of Timed Rebeca, actors have synchronised local clocks which gives us a notion of global time across the model. We use time stamps, and time stamps are comparable accross all actors in the model. This makes our model simpler and more understandable, and our analysis more efficient. But in distributed systems we cannot assume syncronised clocks and time stamps. For that assumption to be a valid and faithful enough to the system, we rely on layering and different responsibilities for different layers. For distributed actors to be able to have synchronised time stamps we rely on the lower-level network protocols to provide that for us.

If we use the standard Timed Transition System to generate the state space for Timed Rebeca model we distinguish three types of transitions: time transitions, events, and τ transitions. In FTTS we reduce that to only event transitions.

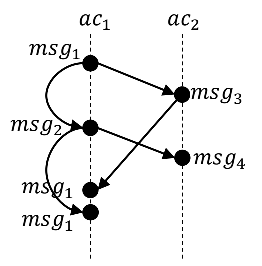
The authors would like to acknowledge DPAC and SEADA.

III. Distributed Model Checking

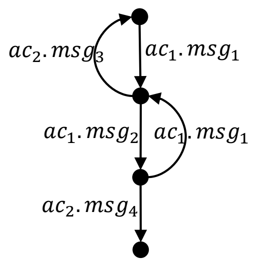
In addition to benefiting from the isolation of actors for reducing the size of state spaces, this property can be used for more efficient analysis of huge state spaces. A major limiting factor in applying model checking for the analysis of real-world systems is the huge amount of space and time required to store and explore state spaces. Distributed model checking is a technique for analyzing these types of state spaces; in which, state spaces are partitioned into some slices and each slice is assigned to a computational node to be analyzed. The efficiency of this technique depends on the communication costs among computational nodes which is related to the distribution policy of states among nodes [**?**]. Another, more fine-grained, representative of communication cost is the number of split transitions; a split transition is a transition between two states, where the hosts of source and destination states are different nodes. In [**?**] we showed how the actor model can be used to reduce teh number of split transitions. We introduced a new state distribution policy based on the so-called Call Dependency Graph (CDG) of actor models. A CDG represents the abstract causality relation among messages of actors. Our abstraction is akin to the dynamic representation of actor's event activation causality proposed by Clinger [**?**].

The most primitive and widely used distribution policy is random state distribution [**?**]. Random state distribution policy distributes states among nodes based on their hash values. Random distribution policy guarantees load balancing. However, it is not an effective technique as cycles are scattered over many different nodes. In [**?**], another state space distribution policy is suggested to improve the locality of cycles. This policy is based on the static analysis of an abstracted model and detects *may* or *must* transition relations among states [**?**]. Based on this analysis, if two states have a *must* relation, they should be stored in a same node. We use a similar idea in our state distribution policy and show that using the CDG improves the locality of cycles by reducing the split transitions in the state space. In other words, we find the *must* relations among the states of actor models using the CDG. Our technique is applicable to other service-oriented models where the unit of concurrency can be modeled as isolated autonomous active objects and message passing is the only way of communication.

Clinger's event diagram comprise vertices (called *dots*) for each event, and edges (called *arrows*) that represent the activation relation of two events. Clinger's event diagram is typically drawn using parallel vertical swim-lanes for actors, where the dots are placed respecting their sequential execution order. Figure 1(a) presents the Clingers' event diagram of an example actor model, shown in Listing 1.



(a) Clinger event diagram of an example actor model



(b) CDG of an example actor model

Fig. 1: Clinger event diagram versus CDG of an example actor model.

Listing 1: An example of a simple actor model

```
reactiveclass AC1 {
  knownrebecs {AC2 ac2;}
  AC1() {
    self.msg1();
  }
  msgsrv msg1() {
    self.msg2();
    ac2.msg3();
  }
  msgsrv msg2() {
    self.msg1();
    ac2.msg4();
  }
}
reactiveclass AC2 {
  knownrebecs{AC1 ac1;}
  statevars {int sv;}
  AC2() {
    sv = 1;
  }
  msgsrv msg3() {
    ac1.msg1();
  }
  msgsrv msg4() {
    if (sv == 1)
      sv = 4;
    else
      sv = 3;
  }
}
main {
    AC1 ac1(ac2):();
    AC2 ac2(ac1):();
}
```

Clinger's event diagrams can be seen as the detailed representations of CDG. Intuitively, a CDG represents the possible activation relations of events derived from a static analysis of the model. Note that as actors are isolated, the only mechanism which may results in activating an event (causality among events) in an actor is sending a message to it. This way, the activation relation of events in a CDG can be extracted from the source codes of actor models by figuring out the message passing among actors. Using static analysis to find message passing results in over-approximatation of events activations in CDGs. Figure 1(b) illustrates the CDG which corresponds to the Clinder's event diagram of Figure 1(a).

In [**?**] we designated and proved a relation between the cycles in the CDG and the cycles in state spaces. We devised a distribution policy for the distributed model checker of *Rebeca* based on the CDG. The new distribution policy increases the efficiency of distributed model checking by increasing the locality of the accepting cycles. Experimental evidence supports that this new policy improves cycle locality, and decreases model checking time and memory in practice.

IV. Verification of Mobile Ad-hoc Protocols

Mobile Ad-hoc wireless networks (MANETs) consist of mobile nodes equipped by wireless transceivers by which they communicate. These networks are useful where no pre-existing infrastructure, such as routers in wired networks or access points in managed (infrastructure) wireless networks are needed, so the network operates in a completely distributed manner. A node A can receive data from a node B if A is located in the communication range of B, i.e., A is directly connected to B. The union of (dis)connectivity relations among the node forms the underlying topology. Due to the mobility of node, (dis)connectivity relations among nodes change, and the topology is dynamic. As all nodes are not directly connected, they rely on each other to communicate indirectly with those not within their range. To this aim, they collaborate on ad-hoc to find a valid route to a destination. A route is valid if the corresponding path exists in the topology, and it is partially maintained in the routing tables of nodes by indicating the next-hop(s) to the destination. As topology changes, the routes maintained by nodes should be updated to prevent from useless communications leading to the energy consumption and network degradation. During the process of route maintenance, it is possible that by following next-hops, a route visits the same node more than once, and a loop is formed. To avoid cycling data in a MANET, *loop-freedom* is one of the main properties of routing protocols MANET routing. Due to enormous number of topology changes, finding the mobility scenario leading to protocol malfunction, e.g., formation of a loop for routing protocols, is not possible by simulation-based approaches, and so their design is error-prone in practice.

Rebeca was extended in [**?**], called wRebeca, to model and verify MANET protocols addressing dynamically changing topology. To support modeling such protocols, wRebeca provides unicast, multicast, and broadcast for communication. The wRebeca model of an abstract version of Ad-hoc On Demand Vector (AODV) routing protocol [**?**] is given in Figure 2. Each node in the network is modeled as an actor, and the routing protocol is represented through the message servers of the actor. The network topology and its mobility are captured while analyzing the model, and are not explicitly modeled in the wRebeca code. In message server rec_newpkt (line 10), whenever a source node intends to send a data packet to a destination ($dip\_$), first it looks up at its routing table, $rst[dip\_]$, to see if it has a valid route to the destination to forward the data packet. Otherwise it starts a route discovery by broadcasting a route *request* message, rec_rreq, at line 16. Nodes upon receiving a request message, forward the request if they do not have any route to the destination until the request reaches to the destination (line 24). Now the destination replies by unicasting a *reply* message, rec_rep, in response (line 26) to the sender of the request ($oip\_$) via its next-hop. The unicast message will be delivered successfully ($succ$ in line 28) if the receiver node is in the wireless range, or the delivery can fail ($unsucc$ in line 31) if the receiver node is not in the wireless range. The reply message is resent by the middle nodes until it arrives to the source node (line 48).

The global states are defined by the local state of rebecs and the underlying topology. To capture topology changes, for each global state, there are a set of state transitions by which the underlying topology is randomly changed. For a network of n nodes, there are $(n^2 - n)$ possible links among them, so the state pace grows with a factor of $2^{(n^2-n)}$.

To tackle the state space explosion, we eliminate the topology from the global states and combine those states that are only different in their topologies. To capture the effect of topology, the (dis)connectivity of those links that result the same behaviors are added to the labels on the transitions. A set of (dis-)connectivity relations among two actors, expressing the status of their link, is called *network constraints* [**?**], [**?**]. For instance, $and(con(n1, n2), !con(n3, n4)$ denotes that the actor $n1$ is connected to $n2$ while n_3 is disconnected from $n4$. The network constraint of each state transition isolates those connected actors and computes the effect of processing of events. We elaborate the idea with an example. Recall that a destination node, upon receiving a request message unicasts rec_rep to the next-hop of the request origin ($oip\_$) (line 26 in Figure 2). In the reduced state space, for the global states in which the destination has a rec_req message to handle, at least two next states are considered; one for the case that the destination is connected to the next-hop, and one for the case that they are disconnected. In the first case, the destination and next-hop are considered in isolation and the effect of processing a request message is computed by inserting a rec_rep message in the queue of the next-hop. The second case partitions actors in further as the destination should informs its upstream nodes that the next-hop is not reachable through it by broadcasting an error message (this part of code has been abstracted). In other words, the effect of processing the request event is computed for those subsets of actors excluding the next-hop in isolation. This scenario has been illustrated in Figure 3 in a MANET with four nodes. Assume that n_1 and n_3 are the source and destination, respectively, and n_3 has received its request message from n_2. The next-hop to n_2 for n_3 is n_4.

To enforce a set of stable (dis)connectivity relations among the actors to govern the level of isolation, they can be specified in *constraints* block at line 68 of Figure 2. This approach reduces the state space substantially for real-world protocols and hence, makes the model checking technique possible as shown in Table I. When the number of nodes increases from 4 to 5 for AODV, and the network constraint results 16 possible topologies, the classic state space cannot be generated due to the memory limitation on a computer with 8GB RAM. We proved that the reduced state space is branching bisimilar to the original one, and consequently a set of properties such

```
reactiveclass  Node(){
  statevars {
    int  sn,ip;
    int []  dsn, rst ,hops,nhop;
  }
  msgsrv  initial  ( int  i ,
       boolean  starter ){
    ...  /*  Initialization   code*/
  }
  msgsrv rec_newpkt(int  data , int  dip_)
  {
    if ( rst [dip_]==1)
    {...  /*forward  packet */}
    else  {
      sn++;
      rec_rreq (0, dip_,
        dsn[dip_], self ,sn, self ,5) ;}
    }
  msgsrv rec_rreq  ( int  hops_,  int  dip_  ,  int  dsn_  ,
        int  oip_  ,  int  osn_  ,  int  sip_,  int  maxHop)
  {
  boolean  gen_msg = false;
   ...  /*processing  code*/
   if  (gen_msg == true) {
    if  (ip  == dip_)  {
      sn  =  sn+1;
      unicast (nhop[oip_],
        rec_rrep (0  ,  dip_  ,  sn  ,  oip_  ,   self ))
      succ:{
        rst [oip_]  =  1;
      }
      unsucc:{
        if ( rst [oip_]  == 1)
          {...  /*error */}
        rst [oip_]  = 2;}
      }  else  {
           hops_ = hops_ + 1;
           if (hops_<maxHop) {
             rec_rreq (hops_,dip_,dsn_,oip_,
               osn_, self ,maxHop);}
  }}}
  msgsrv  rec_rrep ( int  hops_  , int  dip_  , int  dsn_  ,
    int  oip_  , int  sip_ ){
   boolean gen_msg = false;
    ...  /*processing  code*/
    if (gen_msg == true){
      if (ip  == oip_  ){
        ...  /*forward  packet */  }
      else  {
        hops_= hops_+1;
        unicast (nhop[oip_], rec_rrep
          (hops_,dip_,dsn_,oip_, self ))
        succ:{
          rst [oip_]=1;
        }
        unsucc:{
          if ( rst [oip_]  == 1)
            {...}  /*error */
         rst [oip_]  = 2;}
  }}}
  msgsrv  rec_rerr ( int  source_  ,
    int  sip_,  int []  rip_rsn )
  {...  /*error  recovery  code*/}
  }
  main{
    Node  n1(n2,n4) :(0, true );
    Node  n2(n1,n4) :(1, false );
     ...
   constraints {
    and(con(n1,n2),  con(n3,n4))
  }
}
```

Fig. 2: The AODV protocol specified by wRebeca [?]

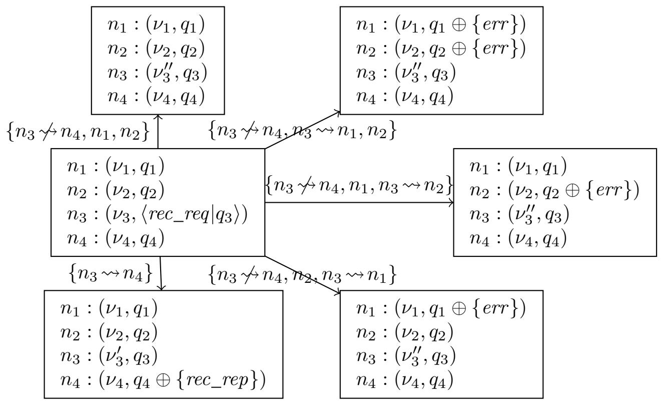

$n_1 : (\nu_1, q_1)$
$n_2 : (\nu_2, q_2)$
$n_3 : (\nu_3'', q_3)$
$n_4 : (\nu_4, q_4)$
$n_1 : (\nu_1, q_1 \oplus \{err\})$
$n_2 : (\nu_2, q_2 \oplus \{err\})$
$n_3 : (\nu_3'', q_3)$
$n_4 : (\nu_4, q_4)$
$\{n_3 \not\rightsquigarrow n_4, n_1, n_2\}$
$\{n_3 \not\rightsquigarrow n_4, n_3 \rightsquigarrow n_1, n_2\}$
$n_1 : (\nu_1, q_1)$
$n_2 : (\nu_2, q_2)$
$n_3 : (\nu_3, \langle rec\_req | q_3 \rangle)$
$n_4 : (\nu_4, q_4)$
$\{n_3 \not\rightsquigarrow n_4, n_1, n_3 \rightsquigarrow n_2\}$
$n_1 : (\nu_1, q_1)$
$n_2 : (\nu_2, q_2 \oplus \{err\})$
$n_3 : (\nu_3'', q_3)$
$n_4 : (\nu_4, q_4)$
$\{n_3 \rightsquigarrow n_4\}$
$\{n_3 \not\rightsquigarrow n_4, n_2, n_3 \rightsquigarrow n_1\}$
$n_1 : (\nu_1, q_1)$
$n_2 : (\nu_2, q_2)$
$n_3 : (\nu_3', q_3)$
$n_4 : (\nu_4, q_4 \oplus \{rec\_rep\})$
$n_1 : (\nu_1, q_1 \oplus \{err\})$
$n_2 : (\nu_2, q_2)$
$n_3 : (\nu_3'', q_3)$
$n_4 : (\nu_4, q_4)$

Fig. 3: Possible next states upon handling of a request message: connectivity and disconnectivity relation are represented by the notation $\rightsquigarrow$ and $\not\rightsquigarrow$, respectively.

as ACTL-X are preserved [?]. The network constraints on transition are used during model checking [?], [?] to verify the topology-sensitive properties.

The reduction technique can be improved further if the topology be stable. As rebecs have no shared variable, the state space can be reduced by the *counter abstraction* technique [?]. By this technique, the global state is defined by a vector of counters, one for each local state of nodes. If there are n nodes in a network where each node can have m possible local states, the global state is shown as $\langle c_1, \ldots, c_m \rangle$, where $c_i \leq n$ denotes the number of nodes residing in the local state $i \leq m$. Therefore, the state space is reduced from m^n to $\binom{n+m-1}{m}$. To apply counter abstraction concerning the topology, rebecs with an identical local state and neighbors, called *topologically equivalent*, are counted together. Intuitively, all topologically equivalent nodes should be either all connected to each other, or disconnected, while they should have the same neighbors (except themselves). So if either one broadcasts, the same set of nodes (except themselves) will receive, and if they are also connected to each other, their counterpart (that is symmetric to the sender) will receive. For example, the nodes $1, 3$ and $2, 4$ are topologically equivalent in Figure 4. To compute the reduced state space, nodes of

TABLE I: Comparing the size of state spaces with/without applying reduction [**?**]

No. of nodes	No. of valid topologies	No. of states before reduction	No. of transitions before reduction	No. of states after reduction	No. of transitions after reduction
4	4	3,007	16,380	763	1,969
4	8	12,327	113,480	1,554	3,804
4	16	35,695	610,816	2,245	5,549
4	32	93,679	3,097,792	2,942	7,596
4	64	258,447	16,797,536	4,053	10,629
5	16	>655,441	>11,276,879	165,959	598,342

the underlying topology are partitioned into the maximal sets of topologically equivalent nodes, and then a counter is considered for each pair of topology equivalence class and a local state that a rebec can reside in. For instance, assume a network with four rebecs of a same class where i and msg are its state variable and message server name, respectively. When the underlying topology is the one shown in Figure 4, as rebecs 1 and 3 have the same local state (expressed by a pair of the state variable valuation and message queue) as shown in Figure 5, and they are topologically equivalent, the global state can be represented as shown in this figure. As a consequence, the resulting state merges two states into one: one as shown in the figure, and the other in which the local states of nodes 2 and 4 swap. We proved that the reduced transition system is strongly bisimilar to the original one, and the state space reduction is considerable for models with homogeneous actors that are no more distinguished by their identifiers. This technique is beneficial for finding an error during the design of a new version of a protocol. If we know that a certain topology leads to malfunctioning of a previous version of the protocol, we can check the new version of the protocol using that certain topology.

V. Verification of Hybrid Reactive Systems

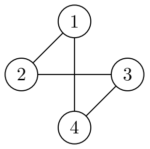
Embedded systems consist of microprocessors which control physical behavior. In such *hybrid* systems, physical and cyber behaviors, characterized as continuous and discrete respectively, affect each other; the physical components may trigger the cyber components which change (by (de)activating) physical components in response. The new generation of embedded systems, cyber-physical systems (CPSs) composed of microprocessors controlling other software/physical systems via networks. For instance, in automotive systems, there are components like sensors, actuators, and controllers that communicate asynchronously with each other through a CAN network. The computational model of Rebeca provides a suitable level of abstraction to faithfully model such distributed asynchronously communicating systems in an intuitive way.

Timed Rebeca was extended by Hybrid Rebeca [] with physical behavior to support hybrid systems. Such an extension allows non-determinism inherent in concurrent and distributed systems, e. g., in the case of simultaneous arrival of messages (and no explicit priority-based policy to choose one over the other) to model check the possible implementations of systems. In Hybrid Rebeca, physical behaviors are encapsulated in so-called physical actors. Each physical actor, in addition to message handlers, is defined by a set of modes. Each mode defines the continuous behavior of the actor. A physical actor (which is instantiated from a physical class) must always have one active mode. By changing the active mode of a physical actor, it's possible to change the continuous behavior of the actor. The active mode can be changed upon receiving a message from either a software actor (controller) or a physical actor. The semantics of Hybrid Rebeca is defined as a hybrid automaton, for which many verification algorithms and tools are available.

We have shown that by using Hybrid Rebeca, the cost of improving and modifying models is vastly reduced compared to modeling in hybrid automata [] as the computational model of Hybrid Rebeca encapsulates many complexities. These complexities subsume high-level concepts like message passing and message buffering. Furthermore, modeling these complexities directly in hybrid automata can hugely decrease the analyzability of the models. Concluding that the abstraction resulted from choosing actors as the basic units of computation, offers more friendliness towards cyber-physical systems compared to the low-level languages like hybrid automata.

Acknowledgment

References

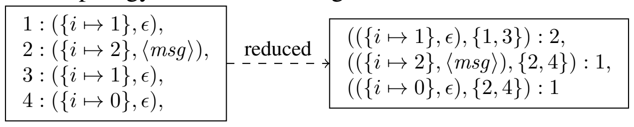


0.3

(a)
b

Fig. 4: A topology: the links among the nodes are bidirectional

0.6


$1 : (\{i \mapsto 1\}, \epsilon),$
$2 : (\{i \mapsto 2\}, \langle msg \rangle),$
$3 : (\{i \mapsto 1\}, \epsilon),$
$4 : (\{i \mapsto 0\}, \epsilon),$
reduced
$((\{i \mapsto 1\}, \epsilon), \{1, 3\}) : 2,$
$((\{i \mapsto 2\}, \langle msg \rangle), \{2, 4\}) : 1,$
$((\{i \mapsto 0\}, \epsilon), \{2, 4\}) : 1$

(a)
b

Fig. 5: Counter abstraction reduction

Fig. 6: Applying counter abstraction reduction to a network with four nodes and the assumed topology, each having a local variable i