Reactive Actors: Isolation for Efficient Analysis of Distributed Systems

Marjan Sirjani
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
School of Computer Science
Reykjavik University
Reykjavik, Iceland
marjan.sirjani@mdh.se

Ehsan Khamespanah
School of Computer Science
Reykjavik University
Reykjavik, Iceland
School of ECE
University of Tehran
Tehran, Iran
ehsank@ru.is

Fatemeh Ghassemi
School of ECE
University of Tehran
Tehran, Iran
fghassemi@ut.ac.ir

***Abstract*—In this paper we explain how the isolation or decoupling of actors can help in developing efficient analysis techniques. The Reactive Object Language, Rebeca, and its timed extension are introduced as actor-based languages for modeling and analyzing distributed systems. We show how floating-time transition system can be used for model checking of timed actor models when we are interested in event-based properties, and how it helps in state space reduction. We explain how the model of computation of actors helps in devising an efficient state distribution policy in distributed model checking. We show how we use Rebeca to verify the routing algorithms of mobile ad-hoc networks. The paper is written in a way to make the ideas behind each technique clear such that it can be reused in similar domains.**

***Index Terms*—Actors, real-time systems, distributed systems, model checking**

I. Introduction

Distributed systems consist of software components executed on different computers that are linked together by a network. The software components communicate with each other in order to achieve the goal of the distributed system. Nowadays distributed systems are everywhere providing scalability and redundancy. Design and analysis of such systems is overly challenging.

Actor model is a model of concurrent computation for developing parallel, distributed and mobile systems [1], [2]. Each actor is an autonomous object that operates concurrently, and send and receive messages asynchronously. Rebeca [3], [4] is an actor-based language proposed to bridge the gap between software engineers and formal methods community. Rebeca comes with a formal semantics and is the first actor-based language with model checking support [5].

Models can be used for both synthesis and analysis [6]. We build abstract models that serve as a specification of a system to be built, and then we refine the models, adding details until we build the system itself. The process is usually iterative, with the specifications evolving along with their refinements. We may have different analysis purposes like verification, validation, and performance evaluation. Model checking, simulation, and building physical prototypes can all be used as methods for analysis. Simulation, which is the execution of an executable model, reveals one possible behavior of a model with one set of inputs. Model checking reveals all possible behaviors of a model over a family of inputs [6].

According to De Nicola et.al. in [7] a major challenge in designing languages is to devise appropriate abstractions and linguistic primitives to deal with the specificities of the domain under investigation. Karmani and Agha believe that a 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 [8]. In [9], Sirjani defines faithfulness as the similarity of the model and the system; and it is argued that faithfulness can bring in analizability and tracability. According to [9], a modeling language is faithful to a system if the model of computation supported by the language matches the model of computation of [the features of interest of] the system. In [10] a model of computation (MoC) is defined as a collection of rules that govern the execution of the [concurrent] components and the communication between components. Faithfulness can be seen as the key motivation behind domain-specific languages.

Rebeca stands for *Reactive Objects Language*, and is designed as a *faithful* language for distributed reactive software systems. It is an imperative actor-based language while original actor languages are mainly functional languages. It is an event-driven and asynchronous language, with implicit receive and non-blocking send statement. There is no shared variables, and methods are atomic and run to completion. A tool suite for model checking is available at [11]. Theories, techniques and tools for compositional verification [12], symmetry and partial order reduction [13], and slicing [14] are proposed. All the above methods are based on techniques using isolation of actors. Rebeca is used for schedulability analysis of wireless sensor network applications [15], protocol verification [16], design exploration and comparing routing algorithms [17]. It is also used for lightweight preprocessing for agent-based simulation [18].

In the following section we explain what we mean by isolation of actors and how it can help in analysis. We then focus on a model that is presented as the semantics of Timed Rebeca and can help by a significant amount of reduction in the state space while doing the analysis. We believe that similar techniques can be used in different analysis methods for event-based reactive isolated modules. In Section IV, we present a technique used for analysis by distributed model checking which is specific for actors. The efficiency of the technique is checked for model checking Rebeca models. Section V explains how we deployed special techniques to make analysis of mobile ad-hoc networks possible.

II. Analysis of Distributed Systems and Isolated Actors

The syntax of Timed Rebeca is showed in Figure 1. Timed Rebeca is an extension of Rebeca where we can model computational time, network delays, and periodic events [19], [20]. A Rebeca model consists of a set of reactive classes and a main part. In the *main*, we declare the instances of the reactive classes as *rebecs*. Rebec stands for reactive object, and is how actors are called in Rebeca. In a reactive class we define the known rebecs to whom we can send messages, the state variables, and the message servers. Message servers are the methods or event handlers of the class. The statements in Rebeca are very similar to Java. The three keywords of `delay`, `after` and `deadline` are specific to Timed Rebeca. Using `delay` we can model the computational time, using `after` we model network delays, and periodic events, and `deadline` is used to specify the timeout for handling the request we are sending out.

Actors are encapsulated modules with no shared variables. Our actors are reactive, there is no explicit receive and the messages (i.e. events) trigger the execution of the message servers (i.e. event handlers) when they are taken from the message queue. Our actors are also asynchronous, when sending a message they are not blocked. 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 (*futures* are mostly used in active object languages [5]). We call this decoupling of actors isolation, and use isolation of actors in generating more efficient analysis techniques. In Rebeca, we choose to have atomic execution of message servers (i.e. methods, or event handlers) which gives us a macro-step semantics and models a non-preemptive execution of the handlers. Macro-step semantics, by itself, helps in reducing the state space size significantly. The reason we can have atomic execution of message servers without loss of generality (in most cases[1]) is the isolation of actors, and the fact that we assume non-preemptive execution of methods. Moreover, if we are only interested in the event-based properties, we may be able to abstract even more and just keep the states that are followed by a transition that 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 while abstracting away some of the states and transitions (see the following section for more explanation).

For explaining our approach, we distinguish three levels of abstraction: the distributed software system itself with all the implementation details, the modeling language which is Rebeca here, and the generated state space which is built for the sake of analysis and we model it as a state transition system. There are alternative ways for analysis, like using logical theorems and applying reasoning based on that, but here we use different variations of state transition systems. We sometimes call the transition system the semantics of our model as it shows the behavior in a formal way. Floating Time Transition System, explained in the following section, 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.

III. Timed Rebeca and Floating Time Transition System

Floating Time Transition System (FTTS) is proposed based on the isolation of timed rebecs [21], [22]. The idea behind FTTS is similar to partial order reduction (POR) but the technique does not fit exactly in the definition of POR. In POR we exploit the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. So, we can expand only a representative subset of all enabled transitions and abstract the rest away while preserving the properties of interest. If we consider the standard Timed Transitions System semantics (TTS) of Timed Rebeca, we cannot say that FTTS is derived from TTS by POR, because we are not only abstracting away some of the transitions; we are also changing the states. You cannot necessarily find one state in FTTS which is the same as a state in TTS. Both states and transitions are changed while the order of events are preserved (you may see the proof of property preservation in [22]).

What we mean by floating time is that in each state of the state space, different actors do not necessarily have the same local clock, i.e., 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 across 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 synchronised clocks

[1]In some cases the behavior of a model may be different if we apply macro- or micro- semantics, or what we also call fine-grain and coarse-grain semantics. This discussion is out of scope of this paper.

$$
\begin{aligned}
Model &::= Class^{*}\ Main \\
Main &::= \textbf{main}\ \{\ InstanceDcl^{*}\ \} \\
InstanceDcl &::= className\ rebecName\ (\langle rebecName\rangle^{*}) : (\langle literal\rangle^{*}); \\
Class &::= \textbf{reactiveclass}\ className\ (queueLength)\ \{\ KnownRebecs\ Vars\ Constructor\ MsgSrv^{*}\ \} \\
KnownRebecs &::= \textbf{knownrebecs}\ \{\ VarDcl^{*}\ \} \\
Vars &::= \textbf{statevars}\ \{\ VarDcl^{*}\ \} \\
VarDcl &::= type\ \langle v\rangle^{+}; \\
Constructor &::= className\ (\langle type\ v\rangle^{*})\ \{\ Stmt^{*}\ \} \\
MsgSrv &::= \textbf{msgsrv}\ methodName(\langle type\ v\rangle^{*})\ \{\ Stmt^{*}\ \} \\
Stmt &::= v = e;\ |\ v = ?(e\langle, e\rangle^{+});\ |\ Call;|\ if\ (e)\ \{\ Stmt^{*}\ \}\ [else\ \{\ Stmt^{*}\ \}];|\ \textbf{delay}(t); \\
Call &::= rebecName.methodName(\langle e\rangle^{*})\ [\textbf{after}(t)][\textbf{deadline}(t)]
\end{aligned}
$$

Fig. 1: Abstract syntax of Timed Rebeca (adapted from [21]). Angled brackets $\langle ... \rangle$ are used as meta parenthesis, superscript $+$ for repetition at least once, superscript $*$ for repetition zero or more times, whereas using $\langle ... \rangle$ with repetition denotes a comma separated list. Brackets [...] indicates that the text within the brackets is optional. Identifiers $className$, $rebecName$, $methodName$, $queueLength$, v, $literal$, and $type$ denote class name, rebec name, method name, queue length, variable, literal, and type, respectively; and e denotes an (arithmetic, boolean or nondetermistic choice) expression. In the instance declaration (rule $InstanceDcl$), the list of rebec names ($\langle rebecName\rangle^{*}$) passed as parameters denotes the known rebecs of that instance, and the list of literals ($\langle literal\rangle^{*}$) denotes the parameters of its constructor.

and time stamps for distributed software components, at least not for free[2]. For that assumption to be valid and faithful enough to the system, we rely on the layering and different responsibilities for different layers. For distributed actors (as faithful representatives of distributed software components) to be able to have synchronised clocks and comparable time stamps we rely on the lower-level network protocols to provide that for us.

In Timed Rebeca we have a concept of time and we can consider that each statement is executed at a certain point in time. Note that we are now talking at the level of the Rebeca model, the notion of time is the model time, and we do not need to worry about synchronising the clocks among different components in the distributed system (which are modeled as actors in our Rebeca model). We assume that local clocks of actors are synchronised and we have time stamps on each statement in the actors which are comparable across the actors.

In Timed Rebeca models, we use a `delay(t)` statement to show the computation delay. Other statements are assumed to be executed in zero time. We use `after(t)` in combination with a `send` message statement; it means that the time stamp of the message when it is put in the queue of the receiver is the value of the local clock of the sender (`now` in the sender) plus the value of `t`. The progress of time is forced by the `delay` statement and also by `after`. We can assume that the time stamp of all the statements are zero when a model starts to execute, then in each actor the local time is increased by value of `t` if there is a `delay(t)` statement. A `send` statement with an `after` does not cause any increase in the local time per se. The statement following the `send` statement has the same time stamp as the `send` statement itself. The `after` construct may cause an increase in the time when the actor picks the message annotated by `after` to be executed. The local time of the receiver actor is set to the time stamp of the message, unless it is already greater than that. The latter situation means that the message sits in the queue while the actor is busy executing another message. Remember that messages are executed atomically and are not preempted. The progress of time happens in the case that the time stamp of the message is greater than the local time of the receiver actor, the local time will be pushed forward. The `after` construct can be used to model the network delay, and also to model periodic events.

If we use the standard Timed Transition System (TTS) to generate the state space, for Timed Rebeca model we distinguish three types of transitions: τ transitions, *events*, and *timed* transitions. In FTTS we reduce that to only *events* transitions. We explain TTS and FTTS for Timed Rebeca using an example in Listing 1. Listing 1 shows a simple Rebeca model with two rebecs $r1$ and $r2$ instantiated from two reactive classes $RC1$ and $RC2$. $RC1$ has only one message server ($m1$) in which it triggers the two message servers of $RC2$ ($m2$ and $m3$). The two message servers of $RC2$ are event handlers that do nothing, i.e., there are no internal actions caused by statements like assignments, or `send`, and no `delay` statements. Note that `send` statements are considered as internal or silent actions but they cause a change in the message queue of the receiver by adding the sent message to that queue.

[2]Ptides [23] and Spanner [24] are two examples that assume synchronized clocks (up to an error bound) and use logical time stamps. They proposed certain mechanisms to be able to have such assumption.

Listing 1: A simple Timed Rebeca model with two rebecs

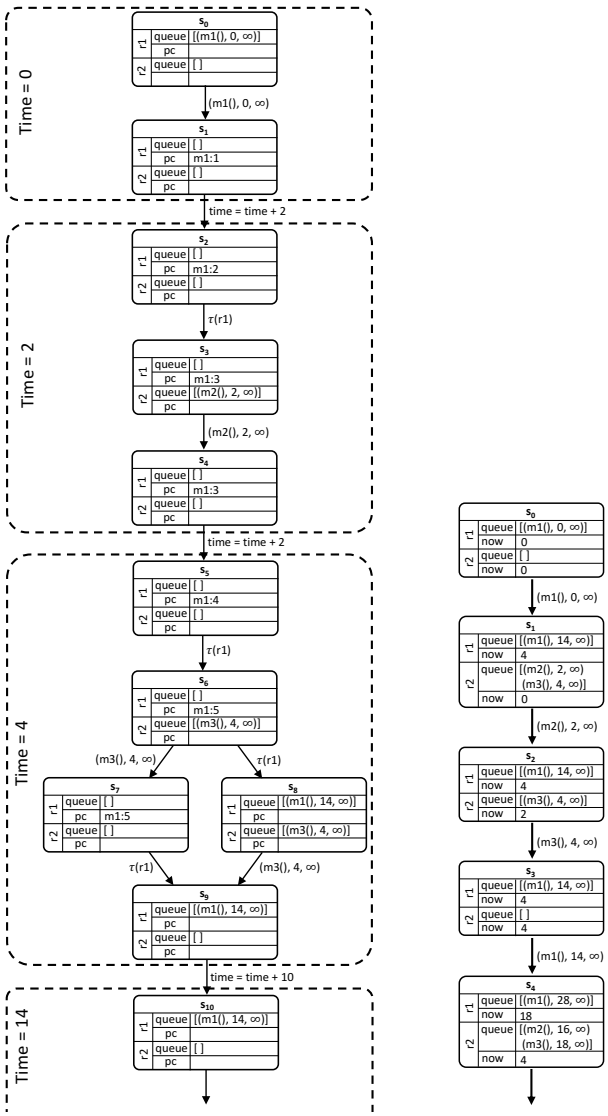
```
1  reactiveclass RC1 (3) {
2    knownrebecs {
3      RC2 r2;
4    }
5    RC1() {
6      self.m1();
7    }
8    msgsrv m1() {
9      delay(2); %PC = 1
10     r2.m2();  %PC = 2
11     delay(2); %PC = 3
12     r2.m3();  %PC = 4
13     self.m1() after (10); %PC = 5
14   }
15 }
16 reactiveclass RC2 (4) {
17   knownrebecs {
18     RC1 r1;
19   }
20   RC2() { }
21   msgsrv m2() { }

23   msgsrv m3() { }
24 }

26 main {
27   RC1 r1(r2):();
28   RC2 r2(r1):();
29 }
```

Figure 2a shows the TTS generated for the model in Listing 1. The constructor of $RC1$ puts the message $m1$ in the queue of $RC1$. So, in $time = 0$ we have the message $m1$ in the queue of $r1$ (see state s_0 in Figure 2a). Also, you see that the message queue of $r2$ is empty. On the transition from s_0 to $s1$ the message is taken from the queue of $r1$, and in the state s_1 the method $m1$ is ready to be executed, i.e., the Program Counter (PC) is at $m1:1$. The first statement in $m1$ is a `delay` statement which is executed and pushes the time forward to $time = 2$ in state s_2. In the state s_2, the PC points at a *send* statement: $r2.m2()$. After this statement is executed as a silent or τ statement, we move to the state s_3 and have the message $m2$ in the queue of $r2$. The message $m2$ is shown as $(m2(), 2, \infty)$ in which 2 is the timestamp of the message and ∞ is the value of `deadline`. The message can only be taken if and when the current time of receiver is greater than or equal its timestamp. If the current time of receiver is less than the timestamp and there is no transition of types τ or *event* enabled then the time of the receiver is advanced to the value of the timestamp (transition of type *time*) and only then the message can be taken.

On the transition from s_3 to s_4 the message $m2$ is fetched to be executed, but the body of the message server $m2$ is empty so nothing happens. We move from s_4 to s_5 by executing another `delay` statement and the time is progressed to four. At state s_5 the `send` statement is executed, the message is put in the queue of $r2$ and we will get to the state s_6. At state s_6, two transitions are enabled and each can be executed first non-deterministically. We go to the state s_7 if the message



(a) TTS (b) FTTS

Fig. 2: TTS and FTTS for the Timed Rebeca model in Listing 1.

in the queue of $r2$ is taken and executed first. We go to the state s_8 if the `send` statement at $m1:5$ is executed first and then the message $m1$ is put in the queue of $r1$. No matter which trace in the diamond shown in the Figure 2a is taken, we will get to the state s_9. In the state s_8, although there is one message in the queue of each rebec, only one transition is enabled. The reason is that the message in the queue of $r1$ has the timestamp of 14 (because of the `after(10)` attached to the `send` statement). Only in the state s_9, when there are no transitions of types τ or *event* enabled the transition of type *time* is taken, and time is advanced for 10 units of time. Now, at the state s_{10} finally the message $(m1(), 14, \infty)$ is enabled.

The FTTS for Listing 1 is shown in Figure 2b. From this figure you may observe that the only transitions in FTTS are the transitions of type *event*, and you may also notice the

reduction in the state space. Another observation is that rebecs may have different current times (*now*) in the same state. That is the reason we use the term *floating time*.

The relation between TTS and FTTS of a Timed Rebeca model is not trivial and cannot be explained as partial order reduction, i.e., you may find states in FTTS which are not in TTS. For example in Figure 2 for 3 out of 5 states in FTTS there are no similar states in TTS. Nevertheless, in [22] we proved a bisimulation relation between FTTS and TTS to prove that the order of events is preserved in FTTS.

It is not easy to understand the FTTS semantics of Timed Rebeca if we start from the standard TTS semantics. A better way is to start from the Timed Rebeca model, and think of an event-based semantics. By focusing on the event-based properties as the properties of interest, we narrow down the interesting transitions to the *event* transitions which are taking the messages from the queue and executing the corresponding message server. This way, we have macro-step semantics where on each transition we take the enabled event (i.e. message) and execute the corresponding message server all in one transition. This is similar to the original semantics of Rebeca presented in [3], but here we have messages in the queue tagged by timestamps (in Timed Rebeca we usually call the message queues the message bag with time-tagged messages). The main technical point here is how to choose the message to take next. The algorithm for making the state space looks into the queues of all the rebecs and picks the message that is enabled *earlier* than the others. The tricky point is to get the definition of *enabled earlier* correctly. The first idea that comes to mind is to pick the message with the least timestamp, but we also need to check the current time or *now* of the receiver rebec. If the value of *now* is larger than the timestamp then that would be the time that the message can really be taken. So, for each rebec we need to find the maximum between the timestamps and the value of *now* of the receiver rebec, and we need to do that for all the rebecs and all the messages in their queues, and find the least among these values. That would be the message that will be taken first and will be the event on the next transition. There may be more than one message with that characteristic, and in that case the choice is nondeterministic.

The Bounded Floating Time Transition System (BFTTS) for Listing 1 is shown in Figure 3. In many reactive systems the behavior becomes recurrent after a while, and it gives us the chance to have a bounded number of states and transitions. When we add the notion of time to our model and consequently to the transition system, when the time is increasing in the model we are at the risk of having an unbounded state space even when the behavior is recurrent. In BFTTS, if the only difference between two states is an equal shift in *time-dependent* values (we call it *shift-equivalency*) we merge those two states to one and will make the shift clear on the transition. Note that this can work because there is no statement in Timed Rebeca that allows assigning an absolute value to *now*, or accessing the absolute value of *now* (if necessary, we may allow that but the modeler has to be careful not to break the shift-equivalency).

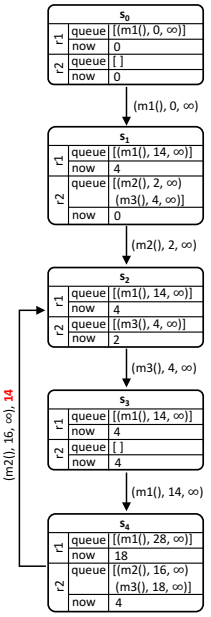


Fig. 3: Bounded FTTS for the Timed Rebeca model in Listing 1.

We have an example of a recurrent behavior in Figure 3. After state s_4 we move to a state which is equivalent to state s_2 with a shift in both now variables (to be 18 for $r1$ and 16 for $r2$), and also a shift 14 for the time tags of the messages (to be 28 for the message in the queue of $r1$ and 18 for the message in the queue of $r2$). So, we do not create a new state, instead we put a transition back to s_2, and make the shift of 14 for the *time-dependent* values clear on this transition. The model checking tool of Timed Rebeca is developed using the idea of BFTTS to be able to generate bounded state spaces and it is integrated in Afra. To illustrate the efficiency of FTTS, the result of comparing the state space size and model checking time consumption of a set of Timed Rebeca examples are presented in Table I.

IV. Distributed Model Checking

In addition to benefiting from the isolation of actors for reducing the size of state space, this property can be used for more efficient analysis of a large state space. A major limiting factor in applying model checking for the analysis of real-world systems is the amount of memory space and the time required to store and explore state space. Distributed model checking is a technique for analyzing the state space where the state space is partitioned into slices, and each slice is assigned to a computational node to be analyzed. Efficiency of this technique depends on the amount of communications among the computational nodes which is affected by the distribution policy of states among the nodes [26].

To reduce the amount of required communication among the nodes, split transitions have to be avoided; a split transition is a transition between two states where the hosts of the

	Config	FTTS			TTS			Reduction	
		States	Trans.	Time	States	Trans.	Time	States	Trans.
Hadoop YARN Scheduler	**1 AM**	25	41	< 1 sec	132	148	< 1 sec	19%	28%
	2 AMs	499	1.02K	1 sec	3.01K	4.28K	1 sec	17%	24%
	3 AMs	5.10K	13.3K	1 sec	34.3K	66.7K	1 sec	15%	20%
	4 AMs	28.19K	92.87K	2 secs	219K	553K	7 secs	13%	17%
	5 AMs	220K	840K	11 secs	1.94M	5.69M	70 secs	11%	15%
WSAN Application	**33-6-4-2**	386	548	< 1 sec	1.58K	2.69K	< 1 sec	24%	20 %
	25-6-4-2	1.21K	1.63K	< 1 sec	5.04K	8.59K	< 1 sec	24%	19%
	30-6-4-2	2.90K	3.25K	< 1 sec	13.00K	20.54K	2 secs	22%	16%
	25-5-4-10	3.54K	5.01K	< 1 sec	19.59K	38.61K	2 secs	18%	13%
	25-7-5-10	4.87K	6.66K	< 1 sec	25.42K	48.29K	2 secs	19%	14%
	50-9-3-2	9.20K	11.79K	1 sec	49.33K	91.93K	3 secs	18%	13%
Ticket Service System	**1 Customer**	5	6	< 1 sec	13	16	< 1 secs	38%	37 %
	2 Customers	51	77	< 1 sec	155	285	< 1 sec	33%	27%
	3 Customers	252	418	< 1 sec	842	1894	< 1 sec	30%	22%
	4 Customers	1.29K	2.21K	< 1 sec	4.75K	12.6K	1 sec	27%	18%
	5 Customers	7.53K	12.8K	< 1 sec	29.1K	85.9K	2 sec	26%	15%
	6 Customers	51.6K	84.7K	3 secs	195.3K	599.3K	9 secs	26%	14%
	7 Customers	408K	650K	11 secs	1.46M	4.34M	67 secs	28%	15%

TABLE I: Comparing the number of states and transitions in TTS and FTTS of three different example (from [25]).

source and destination states are located at different nodes. In [27], we show how for an actor model we can reduce the number of split transitions. We introduce 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 the message passing of actors. Our abstraction is inspired from the Clinger's event diagram [28] that shows the trace of system events and causality relation among events. It also can be count as an special kind of System Dependence Graph in which its intra-procedure dependency relations are omitted. [29]

In a Clinger's diagram, events are arrival of messages to receivers' message queues. A Clinger's event diagram comprises 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 each actor, where the dots are placed for each event respecting their sequential execution order. Figure **??** represents the Clingers' event diagram of a simple actor model, shown in Listing 2. In contrast, sent messages of actors are events in CDG and events are associated with edges instead of vertices. The other major difference between Clinger's event diagram and CDG is that Clinger's event diagram is an infinite graph which shows trace of events in actor programs. However, a CDG is generated for an actor model using static analysis of its source code and can accurately represent the flow of message passing among actors. A CDG shows that by handling a message, which messages *may* or *must* be sent to other actors. The idea of dividing edges to *may* and *must* is inspired from [30] and as we will show later, it helps in having a more effective state distribution policy. Figure **??** illustrates the CDG which corresponds to the actor model of Listing 2.

Listing 2: A simple actor model (from [27])

```
1  reactiveclass AC1 {
2      knownrebecs {
3          AC2 ac2;
4      }
5      AC1() {
6          self.msg1();
7      }
8      msgsrv msg1() {
9          self.msg2();
10         ac2.msg3();
11     }
12     msgsrv msg2() {
13         self.msg1();
14         ac2.msg4();
15     }
16 }
17 reactiveclass AC2 {
18     knownrebecs{
19         AC1 ac1;
20     }
21     statevars {
22         int sv;
23     }
24     AC2() {
25         sv = 1;
26     }
27     msgsrv msg3() {
28         ac1.msg1();
29     }
30     msgsrv msg4() {
31         if (sv == 1)
32             sv = 4;
33         else
34             sv = 3;
35     }
36 }
37 main {
38     AC1 ac1(ac2):();
39     AC2 ac2(ac1):();
40 }
```

The most primitive and widely used distribution policy is

Problem	Size	#Transitions	#Split Transitions		
			Random	CDG	improvement
Asynch. Resource Manager	**4 clients**	7,76K	5,83K	4516	23%
	5 clients	83,19K	66,52K	50,46K	25%
	6 clients	1,02M	850,74K	635,14K	26%
	7 clients	14,34M	12,30M	9,01M	27%
Dining Philosophers	**3 phils**	10,30K	6,97K	4,81K	31%
	4 phils	206,00K	154,76K	75,86K	51%
	5 phils	3,78M	3,02M	1,60M	47%
Train Controller	**5 trains**	33,30K	26,66K	18,17K	32%
	6 trains	265,89K	221,61K	148,32K	34%
	7 trains	2,30M	1,97M	1,30M	35%
	8 trains	21,83M	19,11M	12,40M	36%

TABLE II: The number of split edges in the random and the CDG-based distribution policies (from [27]).

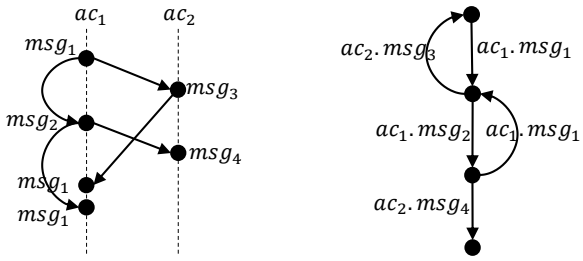


Fig. 4: (a) Clinger event diagram and (b) CDG of the actor model in Listing 2 (from [27]).

the random state distribution [31]. 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 it scatters cycles in state spaces over different nodes. Note that detecting cycles of state spaces is crucial for model checking against LTL-like properties; so, splitting them among nodes (i.e., reducing locality of cycles) dramatically increases the time consumption of model checking. In [27] we proved that there is a relation between cycles in state spaces of actor models and cycles in their corresponding CDGs. So, we devise a distribution policy that is based on associating each cycle of a CDG with a computational node of our distributed model checker. When assigning the states of the state space to different nodes, we assign the state to the node which is associated to the corresponding CDG. This way, we reduce the number of cycles which are scattered among different nodes. In the case that there is a state in CDG which is shared among more than one cycle, the cycle that contains an edge which is associated with a *must* label determines the host of the state. When none of the cycles include a *must* label then one of the cycles is randomly selected and the new state is distributed to its host. We have implemented the CDG-based distribution policy in the distributed model checking tool of Rebeca, and the result of using CDG-based distribution policy is shown in Table II for three different examples. Experimental evidence supports that this new policy improves cycles locality, and decreases the model checking time and memory consumption.

For using similar techniques in distribution of states to nodes, we can think of System Dependence Graphs (SDG) that is presented as an standard formalism to model structures and dependencies in programs [29]. SDGs are the basis for multiple applications in program analysis, such as slicing and testing. Similar to CDGs for actor-based models, SDGs can be used in showing causality relations in models, but for a wider range of textual imperative modeling languages including the model checking languages of Promela [32] and NuSMV [33]. However, as all the dependencies between the statements of models are shown in SDGs, there are many cycles in SDGs which are not related to cycles in the state space. So, using SDGs for state distribution policy to localize cycles most likely will not be practically effective. The reason that there is a relation among the cycles in the CDG and the state space of an actor model is inherent in the model of computation of actors and their isolation. Our technique is not limited to Rebeca and applicable to other distributed systems where the unit of concurrency can be modeled as isolated autonomous reactive objects and message passing is the only means of communication.

V. 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 applicable where no pre-existing infrastructure, such as routers in wired networks or access points in wireless networks are available. Node A can receive data from node B if A is located in the communication range of B, i.e., A is directly connected to B. The union of connectivity/disconnectivity relations among nodes forms the underlying topology. In such networks, nodes can freely move, so the topology dynamically changes. 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 ad-hocly to find a valid route from a source to a destination. A route is valid if the corresponding path exists in the topology. Routes are partially maintained in the routing tables of nodes by indicating the next-hop(s) via which an intended destination is accessible. As topology changes, the routes maintained by nodes should be updated to prevent using invalid routes. 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. *Loop-*

```
1  reactiveclass Node{
2    statevars {
3      int sn,ip;
4      int [] dsn, rst ,hops,nhop;
5    }
6    Node(int i, boolean starter )
7    {
8      ... /* Initialization code*/
9    }
10   msgsrv rec_newpkt(int data, int dip_)
11   {
12     if( rst[dip_]==1)
13     {... /*forward packet*/}
14     else {
15       sn++;
16       rec_rreq (0,dip_,
17         dsn[dip_], self ,sn, self ,5);}
18     }
19   msgsrv rec_rreq (int hops_, int dip_ , int dsn_ ,
         int oip_ , int osn_ , int sip_, int maxHop)
20   {
21   boolean gen_msg = false;
22    ... /*processing code*/
23   if (gen_msg == true) {
24     if (ip == dip_) {
25       sn = sn+1;
26       unicast(nhop[oip_],
27         rec_rrep(0 , dip_ , sn , oip_ , self))
28       succ:{
29         rst[oip_] = 1;
30       }
31       unsucc:{
32         if( rst[oip_] == 1)
33           {... /*error*/}
34         rst[oip_] = 2;}
35       } else {
36         hops_ = hops_ + 1;
37         if(hops_<maxHop) {
38           rec_rreq(hops_,dip_,dsn_,oip_,
39             osn_, self ,maxHop);}
40   }}}
41   msgsrv rec_rrep( int hops_ , int dip_ , int dsn_ ,
42     int oip_ , int sip_){
43     boolean gen_msg = false;
44      ... /*processing code*/
45     if(gen_msg == true){
46       if(ip == oip_ ){
47         ... /*forward packet*/ }
48       else {
49         hops_= hops_+1;
50         unicast(nhop[oip_], rec_rrep
51           (hops_,dip_,dsn_,oip_, self ));
52         ...
53   }}}
54   msgsrv rec_rerr( int source_ ,
55     int sip_, int [] rip_rsn )
56    {... /*error recovery code*/}
57   }
58   main{
59     Node n1(n2,n4):(0, true );
60     Node n2(n1,n4):(1, false );
61      ...
62    constraints {
63     and(con(n1,n2), con(n3,n4))
64   }
65 }
```

Fig. 5: The AODV protocol specified by wRebeca (adapted from [16])

freedom is one of the main required properties of MANET routing protocols. Because the topology is changing all the time, finding a scenario that leads to protocol malfunction (for example formation of a loop in the routing protocol), is very unlikely if we use simulation-based approaches. So, these approaches are not very helpful in designing such protocols in practice.

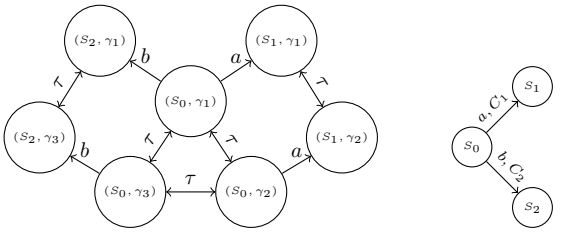
An extension of Rebeca, called wRebeca, is introduced in [16], and is used to model and verify MANET protocols addressing dynamically changing topology. To support modeling such protocols, wRebeca provides *unicast*, *multicast*, and *broadcast* for communication. The wRebeca model of an abstract version of Ad-hoc On Demand Vector (AODV) routing protocol [34] is given in Figure 5. Each node in the network is represented as an actor while the routing protocol is modeled through the message servers of the actor. The network topology and its mobility are not explicitly modeled in the wRebeca code, instead we address them at the level of the state transition system. 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 in 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 processing a request message, forward the request (line 38) if they do not have any route to the destination until the request reaches to the destination (line 24). The destination replies by unicasting a reply message, rec_rrep, in response (line 26) to the sender of the request ($oip\_$), called origin. As links may not be bidirectional, the origin may not be directly connected to the destination, so the destination unicasts to the next-hop towards the origin ($nhop[oip\_]$). The modeler can specify what should be done in case the unicast message is delivered successfully ($succ$ in line 28) or the delivery fails ($unsucc$ in line 31). The reply message is resent by the middle nodes (line 50) until it arrives to the source node (line 47).

To reason about MANET protocols by model checking technique, the state transition systems of their models have to be generated. In wRebeca tool, the global states (denoted as (S, γ)) are defined by the local states of rebecs and the underlying topology. The state-space generator produces two types of transitions; transitions for atomic handling of messages in the actor queues, and transitions for random modification of the underlying topology to address mobility. A resulting state space is shown partially in Figure 6a. The message handling transitions are represented by a/b-transitions while random modification of the topology is shown by τ-transitions. For generating the first type of transition, the effect of topology on

TABLE III: Comparing the size of state space with/without applying topology-elimination reduction (from [16]).

No. of nodes	No. of 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 execution of the communication statement, like broadcast, is considered. When an actor broadcasts, only those that are directly connected to the sending actor will receive the message. We remark that the underlying topology is not changed by the first type of transitions, for example note the following transitions in Figure 6a: $(S_0, \gamma_1) \xrightarrow{a} (S_1, \gamma_1)$ and $(S_0, \gamma_1) \xrightarrow{b} (S_2, \gamma_1)$. For a network of n nodes, there are $2 \times \binom{n}{2}$ possible uni-directional links among the nodes, and as each link can be up/down, there are $2^{(n^2-n)}$ possible topologies. So, for each global state, $2^{(n^2-n)}$ transitions are generated to randomly change the topology, and the state space grows exponentially, resulting state-space explosion. It is notable that the local states of rebecs are not changed by τ-transitions like $(S_1, \gamma_1) \xrightarrow{\tau} (S_1, \gamma_2)$ and $(S_2, \gamma_1) \xrightarrow{\tau} (S_2, \gamma_3)$.



(a) Before reduction (b) After reduction

Fig. 6: A simple state space before and after applying the reduction. The network constraint C_1 expresses the common links of the topologies γ_1 and γ_2, and C_2 represents the common links of γ_1 and γ_3.

To tackle the state-space explosion, we propose a reduction technique that eliminates the topology from the global states and combines those states that are only different in their topologies. A similar technique is used in a previous work in an algebraic framework using Process Algebra [35]. By using wRebeca for solving the same problem, we not only gain a more modular model but also much more reduction in the state space. The isolation of rebecs which allows us to use a macro-step semantics with no loss of generality helps in reducing the state space significantly.

As we want to eliminate the topology from the global states and combine those states, the states (S_0, γ_1), (S_0, γ_2), and (S_0, γ_3) of Figure 6a are aggregated to S_0 as demonstrated in Figure 6b. As a consequence, transitions modifying the topology are removed. So, a state transition like $S_0 \xrightarrow{a} S_1$ is possible in Figure 6b after the reduction. As message handlers are executed with regard to the topology, the connectivity/disconnectivity of those links that are common in both topologies make the effect of handling the message to be the same. The connectivity/disconnectivity relations of these links, called *network constraint* [36], [37], are added to the labels on the transitions. For example, the links common in both γ_1 and γ_2 that result in the same next local states, i.e., S_1 in Figure 6a, are added by the network constraint C_1 to the label of the transition $S_0 \xrightarrow{a,C_1} S_1$ in Figure 6b. By the topology-elimination reduction technique, the state-space generator for each local states of actors S generates the next local actor states for handling messages considering all the possible connectivity/disconnectivity of handling actors to other actors. For a network of n nodes, each node has 2^{n-1} possible connectivity/disconnectivity relations with others. So, the topology-elimination reduction technique reduces the number of states from $\mathbb{S} \times 2^{(n^2-n)}$ to $\mathbb{S} \times 2^{n-1}$, where $\mathbb{S}$ is the total number of possible local states of actors. If $\mathbb{S}$ be a large number, the reduction achieved by topology-elimination is not sufficient enough for tackling state-space explosion, as we experienced in the algebraic framework of [35]. The macro-step semantics of Rebeca helps in reducing $\mathbb{S}$, and hence, makes this reduction at the state-space level more effective for the analysis of real-world protocols like AODV.

Table III illustrates the amount of reduction achieved by applying topology-elimination reduction on isolated actors. To restrict the state-spaces generator in considering the possible topologies, we have enforced a set of stable connectivity/disconnectivity relations among the actors. This set are specified by network constraint at the *constraints* block of the model. For instance, the network constraint given in line 68 of Figure 5 enforces the state-space generator to consider the topologies that n_3 is connected to n_4, expressed by $con(n_3, n_4)$. The second column of Table III indicates the number of possible topologies considered for generating the state space. We proved that the reduced state space is branching bisimilar to the original one, and consequently a set of properties such as ACTL-X are preserved [16]. The network constraints on transition are used during model checking [35], [38] to verify the topology-sensitive properties.

VI. Hybrid Reactive Systems

In Hybrid Rebeca [39], Timed Rebeca is extended with physical behavior to support hybrid systems. Like in Timed Rebeca, Hybrid Rebeca allows modeling of 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). 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 which define the continuous behavior of the actor. A physical actor must always have one active mode. The active mode can be changed upon handling a message that is received 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.

In [40], we show that how using Hybrid Rebeca, can reduce the cost of modifying models compared to the cost of changing a hybrid automata. The reason is that in Hybrid Rebeca, modeling concepts like message passing and message buffering can be handled in the model at a higher level of abstraction compared to hybrid automata. Furthermore, modeling these features directly in hybrid automata can decrease the analyzability of the models. We are currently working on more efficient analysis techniques for Hybrid Rebeca. We believe such techniques can be developed based on the ease in modeling which we gained by using Hybrid Rebeca.

ACKNOWLEDGMENT

The work of the first author is supported in part by DPAC Project (Dependable Platforms for Autonomous Systems and Control) at Malardalen University, and MACMa Project (Modeling and Analyzing Event-based Autonomous Systems) at Software Center, Sweden. The work of the first and second authors is supported by the project Self-Adaptive Actors: SEADA (nr 163205-051) of the Icelandic Research Fund. We would like to thank Edward Lee and Libero Nigro for their detailed comments on the paper.

REFERENCES

[1] C. Hewitt, "Viewing control structures as patterns of passing messages," *Journal of Artificial Intelligence*, vol. 8, no. 3, pp. 323–363, 1977.

[2] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, no. 1, pp. 1–72, 1997.

[3] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Modeling and verification of reactive systems using Rebeca," *Fundam. Inform.*, vol. 63, no. 4, pp. 385–410, 2004.

[4] M. Sirjani, "Rebeca: Theory, applications, and tools," in *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006*, 2006, pp. 102–126.

[5] F. S. de Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang, "A survey of active object languages," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 76:1–76:39, 2017.

[6] E. A. Lee and M. Sirjani, "What good are models?" in *Formal Aspects of Component Software - 15th International Conference, FACS 2018*, 2018, pp. 3–31.

[7] R. De Nicola, G. L. Ferrari, R. Pugliese, and F. Tiezzi, "A formal approach to the engineering of domain-specific distributed systems," in *Coordination Models and Languages - COORDINATION 2018, Proceedings*, 2018, pp. 110–141.

[8] R. K. Karmani and G. Agha, "Actors," in *Encyclopedia of Parallel Computing*, 2011, pp. 1–11.

[9] M. Sirjani, "Power is overrated, go for friendliness! expressiveness, faithfulness and usability in modeling - the actor experience," in *Principles of Modeling*, ser. LNCS 10760, 2018, pp. 424–449.

[10] C. Ptolemaeus, *System Design, Modeling, and Simulation using Ptolemy II*. Berkeley, CA: Ptolemy.org, 2014.

[11] Rebeca, "Rebeca Homepage: http://www.rebeca-lang.org/."

[12] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Model checking, automated abstraction, and compositional verification of rebeca models," *J. UCS*, vol. 11, no. 6, pp. 1054–1082, 2005.

[13] M. M. Jaghoori, M. Sirjani, M. R. Mousavi, E. Khamespanah, and A. Movaghar, "Symmetry and partial order reduction techniques in model checking rebeca," *Acta Inf.*, vol. 47, no. 1, pp. 33–66, 2010.

[14] H. Sabouri and M. Sirjani, "Actor-based slicing techniques for efficient reduction of rebeca models," *Sci. Comput. Program.*, vol. 75, no. 10, pp. 811–827, 2010.

[15] E. Khamespanah, M. Sirjani, K. Mechitov, and G. Agha, "Modeling and analyzing real-time wireless sensor and actuator networks using actors and model checking," *STTT*, vol. 20, no. 5, pp. 547–561, 2018.

[16] B. Yousefi, F. Ghassemi, and R. Khosravi, "Modeling and efficient verification of wireless ad hoc networks," *Formal Aspect of Computing*, vol. To appear, 2017.

[17] Z. Sharifi, M. Mosaffa, S. Mohammadi, and M. Sirjani, "Functional and performance analysis of network-on-chips using actor-based modeling and formal verification," *ECEASST*, vol. 66, 2013.

[18] J. de Berardinis, G. Forcina, A. Jafari, and M. Sirjani, "Actor-based macroscopic modeling and simulation for smart urban planning," *Sci. Comput. Program.*, vol. 168, pp. 142–164, 2018.

[19] A. H. Reynisson, M. Sirjani, L. Aceto, M. Cimini, A. Jafari, A. Ingólfsdóttir, and S. H. Sigurdarson, "Modelling and simulation of asynchronous real-time systems using timed rebeca," *Sci. Comput. Program.*, vol. 89, pp. 41–68, 2014.

[20] M. Sirjani and E. Khamespanah, "On time actors," in *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, 2016, pp. 373–392.

[21] E. Khamespanah, M. Sirjani, Z. Sabahi-Kaviani, R. Khosravi, and M. Izadi, "Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system," *Sci. Comput. Program.*, vol. 98, pp. 184–204, 2015.

[22] E. Khamespanah, M. Sirjani, M. Viswanathan, and R. Khosravi, "Floating time transition system: More efficient analysis of timed actors," in *Formal Aspects of Component Software, FACS 2015*, 2015, pp. 237–255.

[23] P. Derler, E. A. Lee, and S. Matic, "Simulation and implementation of the PTIDES programming model," in *International Symposium on Distributed Simulation and Real-Time Applications*, 2008, pp. 330–333.

[24] J. C. Corbett and et.al, "Spanner: Google globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[25] E. Khamespanah, "Modeling, Verification, and Analysis of Timed Actor-Based Models," Ph.D. dissertation, Reykjavik University, School of Computer Science, June 2018.

[26] S. Orzan, J. van de Pol, and M. V. Espada, "A state space distribution policy based on abstract interpretation," *Electr. Notes Theor. Comput. Sci.*, vol. 128, no. 3, pp. 35–45, 2005.

[27] E. Khamespanah, M. Sirjani, M. R. Mousavi, Z. Sabahi-Kaviani, and M. Razzazi, "State distribution policy for distributed model checking of actor models," *ECEASST*, vol. 72, 2015.

[28] W. D. Clinger, "Foundations of actor semantics," Cambridge, MA, USA, Tech. Rep., 1981.

[29] J. Graf, "Speeding up context-, object- and field-sensitive SDG generation," in *SCAM 2010*, 2010, pp. 105–114.

[30] K. G. Larsen and B. Thomsen, "A modal process logic," in *Proceedings of (LICS '88)*. IEEE Computer Society, 1988, pp. 203–210.

[31] H. Garavel, R. Mateescu, and W. Serwe, "Large-scale distributed verification using CADP: beyond clusters to grids," *Electr. Notes Theor. Comput. Sci.*, vol. 296, pp. 145–161, 2013.

[32] G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2011.

[33] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: A new symbolic model verifier," in *CAV 99*, 1999, pp. 495–499.

[34] C. E. Perkins and E. M. Belding-Royer, "Ad-hoc on-demand distance vector routing," in *Proc. 2nd Workshop on Mobile Computing Systems and Applications*. IEEE, 1999, pp. 90–100.

[35] F. Ghassemi and W. J. Fokkink, "Model checking mobile ad hoc networks," *Formal Methods in System Design*, vol. 49, no. 3, pp. 159–189, 2016.

[36] F. Ghassemi, W. J. Fokkink, and A. Movaghar, "Equational reasoning on mobile ad hoc networks," *Fundamenta Informaticae*, vol. 103, pp. 1–41, 2010.

[37] F. Ghassemi and W. J. Fokkink, "Reliable restricted process theory," *Fundamenta Informatica*, vol. 165, no. 1, pp. 1–41, 2019.

[38] F. Ghassemi, "Verification of mobile ad hoc network processes with data," *The CSI Journal on Computer Science and Engineering*, vol. 15, no. 2, pp. 44–52, 2018.

[39] I. Jahandideh, F. Ghassemi, and M. Sirjani, "Hybrid Rebeca: Modeling and analyzing of cyber-physical systems," in *Proc. Model-Based Design of Cyber Physical Systems*, ser. LNCS, vol. 11615. Springer, 2019, pp. 1–25.

[40] ——, "An actor-based framework for asynchronous event-based cyber-physical systems," *CoRR*, vol. abs/1709.01786v2, 2019. [Online]. Available: http://arxiv.org/abs/1709.01786v2