Noname manuscript No.
(will be inserted by the editor)

VeriVANca Framework: Verification of VANETs by Property Based Message Passing of Actors in Rebeca with Inheritance

Farnaz Yousefi · Ehsan Khamespanah · Mohammed Gharib · Marjan Sirjani · Ali Movaghar

Received: date / Accepted: date

Abstract Vehicular ad-hoc networks have attracted the attention of many researchers during the last years due to the emergence of autonomous vehicles and safety concerns. Most of the frameworks which are proposed for the modeling and analysis VANET applications make use of simulation techniques. Due to the high level of concurrency in these applications, simulation results do not guarantee the correct behavior of the system and more accurate analysis techniques are required. In this paper, we have developed a framework to provide model checking facilities for the analysis of VANET applications. To this end, an actor-based modeling language, Rebeca, is used which is equipped with a variety of model checking engines. We have extended Rebeca with the inheritance mechanism to support model-specific message passing among vehicles, which is crucial for the modeling of VANET applications. To illustrate the applicability of this framework, we modeled and analyzed two warning message dissemination schemes. Reviewing the results of using the model checking technique supports the claim that concurrent behaviors of the system components in VANETs may cause uncertainty which may not be detected by simulation-based techniques. We also observed that considering the interleaving of concurrent executions of the system components affects the performance metrics of it.

Keywords Model Checking · Warning Message Dissemination · Vehicular Ad-Hoc Networks (VANETs) · Rebeca · Actor Model

F. Yousefi · A. Movaghar
Department of Computer Engineering, Sharif University of Technology, Iran

E. Khamespanah
School of Electrical and Computer Engineering, University of Tehran, Iran

E. Khamespanah · M. Sirjani
School of Computer Science, Reykjavik University, Iceland

M. Gharib
School of Computer Science, Institute for Research in Fundamental Science (IPM), Iran

M. Sirjani
School of IDT, Mälardalen University, Sweden

1 Introduction

Safety of the autonomous vehicles is turned into one of the main concerns of future transportation systems. This concern has been attracting the attention of researchers in both academia and industry, during the last years. Using autonomous vehicles in mission-critical applications increases the significance of the problem. Vehicular ad hoc networks (VANETs) are considered as the main communication network in such systems where the main responsibility is the Warning Message Dissemination (WMD). To prevent further potential damage, WMD is used for vehicle to vehicle communication in dangerous situations. In this case, vehicles with the knowledge of hazard broadcast warning messages to inform the other vehicles. To increase the number of vehicles receiving the warning message, the receiving nodes have to forward the message. Such a forwarding operation causes bursty traffic. Different strategies are proposed to hold a fair trade-off between the amount of traffic in the network and the maximum number of vehicles receiving the message [19]. Each strategy proposes how to select the next group of forwarding nodes to enhance the performance, considering the mentioned trade-off. In Section 2 more details about WMD strategies in VANETs are presented.

To validate the correctness of WMD strategies and to evaluate their performance, a number of simulation-

based tools and techniques have been used. Simulation-based approaches cannot provide a high level of confidence for the correct behavior of the system in the presence of the concurrent execution of system components. This parameter reduces the effectiveness of simulation-based techniques in such mission-critical applications. At the failure of simulation-based techniques, formal verification approaches seem to be the perfect technique for achieving reliable results. Formal verification is widely used in applications of VANETs such as cooperative collision avoidance [9], intersection management using mutual exclusion algorithms [3], and collaborative driving [15]. However, to the best of our knowledge, this is the first formal verification work in the WMD application of VANETs. Note that there are some works on formal verification of message dissemination in VANTEs (e.g. Ferreira et al. in [5]), but none of them address analysis of schemes which is an application layer analysis.

In this paper, we introduce Rebeca with Inheritance for modeling and analyzing of the WMD schemes in VANETs. Rebeca [21] is an operational interpretation of the actor model with formal semantics, supported by a variety of analysis tools [20,13]. In the actor model, all the elements that are running concurrently in a distributed system are modeled as actors. Communication among actors takes place by asynchronous message passing. These structures and features match the needs of VANETs as they consist of autonomous nodes that communicate by message passing. This level of faithfulness helps in having a more natural mapping between the actor model and VANETs, making models easier to develop and understand. The way that inheritance is added to Rebeca, enables support for inheritance in a variety of Rebeca extensions including Timed Rebeca [1], Probabilistic Timed Rebeca [10], and Hybrid Rebeca [11]. We developed VeriVANca as a framework for the analysis of WMD schemes in VANETs using Timed Rebeca with inheritance. In Section 3 Timed Rebeca is briefly introduced using the simplified version of the counting-based scheme, a WMD scheme. Then, the inheritance mechanism and its semantics are presented in Section 4. More details about how VeriVANca is developed and its features are described in detail in Section 5. Note that Rebeca family members do not support dynamic actor creation; so, they cannot be used for modeling WMD schemes which require creating actors dynamically.

To illustrate the applicability of VeriVANca, we have modeled a distance-based scheme [22] and a counting-based scheme [23] using VeriVANca. In the case of the distance-based scheme, the model checking results show that concurrent execution of the system components enables multiple execution traces, some of them cause starvation which might not be detected by simulation-based techniques. We also observed that concurrent execution of the components in this model, when considering the interleaving of the components, results in multiple values for the algorithm performance. Further investigations yield that this phenomenon is not limited to one scenario but it is common in this model. More details on similar cases are presented in Section 6.

The next interesting result of this work is about the scalability of the model. To examine the scalability of VeriVANca, a middle-sized model of a four-lane street with about 40 vehicles is analyzed. The model checking results show that scaling the number of vehicles up into a very congested area leads to a dramatical increment in the size of the state space as well as the analysis time of model checking. Scaling up the model without forming new congested areas, however, results in a smooth increment in the size of the state space and analysis time, as presented in Section 6.

This paper is an extended version of our previously published conference paper [27]. This paper extends the conference paper as follows:

- We propose the formal semantics of Rebeca with inheritance in the form of SOS (Structural Operational Semantics) rules.
- We illustrate how the proposed semantics can be used in other extensions of Rebeca family members.
- The experimental results are improved for better illustration of the case studies and the effectiveness of this work.

2 Warning Message Dissemination in VANETs

WMD is an application developed for VANETs that tends to increase the safety and riding experience of passengers. In this application, a warning message is disseminated between vehicles in the case of any abnormal situations such as car accidents or undesirable road conditions. Received warning messages are used either to activate an automatic operation such as reducing speed to avoid chained accidents (increasing safety) or are shown as alerts to inform the driver of the upcoming hazard so that the driver can do operations such as changing their route (improving the riding experience).

Using WMD in safety-critical applications, requires providing high reliability for them in developed solutions. Besides, some characteristics of VANETs such as high mobility of the nodes and fast topology changes, makes routing algorithms commonly used in MANETs (Mobile Ad-hoc NETworks) inapplicable to VANETs [28]. Therefore, the only approach for implementation

of message dissemination in VANETs is multi-hop broadcast of the message. In this approach, the receiving nodes are responsible for re-broadcasting the message to the others. However, this can result in broadcast storm problem in the network. In order to tackle this problem, a number of schemes have been proposed for WMD as described in the following subsection.

2.1 Message Dissemination Schemes

Message dissemination schemes are algorithms that specify how a forwarding node is selected in a VANET. The selection of a forwarding node is performed based on some criteria such as distance between senders and receivers, number of received messages by a node, probabilities associated with nodes, topology of the network, etc. [19]. In this paper, two schemes —a distance-based and a counting-based scheme— are modeled using the proposed framework.

The distance-based scheme, called TLO (The Last One) [22], makes use of location information of the vehicles to select the forwarding node. In this scheme, upon a message broadcast, the farthest receiver in the range of the sender is selected as the forwarding TLO node. Other vehicles in the range know that they are not the farthest node and do not forward the received message. However, they wait for a while to make sure of successful broadcast of the TLO node. Receiving the warning message from the TLO node, means that the sending of the message has been successful and they do not forward the warning message. Otherwise, the algorithm is run once again to select the next TLO forwarding node.

In the counting-based scheme [23], an integer number is defined as counter threshold. Each receiving node counts the number of received messages in a time interval. At the end of that time interval, the receiver decides on being a forwarding node based on the comparison of the value of its counter and the value of counter threshold. If the value of the counter is greater than the value of counter threshold, the receiver assumes that enough warning messages are disseminated in its vicinity; therefore, it avoids forwarding the message. Otherwise, the receiver forwards the warning message.

2.2 Analysis Techniques

Different analysis techniques have been developed for the correctness and performance evaluation of message dissemination schemes in VANETs. Simulation-based approaches are widely used for the analysis of applications of in this domain. Gama et. al. developed a model and analyzed three different message dissemination schemes using Veins simulator [6]. Sanguesa et. al. have used ns-2 simulator in two independent works regarding the selection of optimal message dissemination scheme. In [17], they aim to select the optimal broadcasting scheme for the model in each scenario and in [18], the selection of the optimal scheme is performed for each vehicle based on vehicular density and the topological characteristics of the environment where the vehicle is located in. In a more comprehensive work [19] authors have developed a framework in ns-3 simulator for comparing different schemes. Note that although this approach is used in many applications, it does not guarantee correctness of results as it does not consider concurrent execution of system components.

Another technique used for the analysis of WMD in VANETs is the analytical approach. In this approach, a system is modeled by mathematical equations and the analysis is performed by finding solutions to the equation system. For example, in [16], Saeed et. al. have derived difference equations that their solutions yield the probability of all vehicles receiving the emergency warning message. This value is computed as a function of the number of neighbors of each vehicle, the rebroadcast probability, and the dissemination distance. In another work, a probabilistic multi-hop broadcast scheme is mathematically formulated and the packet reception probability is reported for different configurations, taking into account the topology of the network and as a result, major network characteristics such as vehicle density and the number of one-hop neighbors [8]. This approach guarantees achieving correct results but it is not modular and developing mathematical formula needs a high degree of user interaction and a high degree of expertise.

As the third technique, model checking is a general verification approach which provides ease of modeling similarly to simulation-based approaches in addition to guaranteeing the correctness of results due to its mathematical foundation. To the best of our knowledge, there is no framework which provides model checking facilities for the analysis of WMD schemes in VANETs.

3 Rebeca Language

Rebeca is a modeling language based on Hewitt and Agha's actors [2]. Actors in Rebeca are independent units of concurrently running programs that communicate with each other through message passing. The message passing is an asynchronous non-blocking call to the actor's corresponding message server. Message servers are methods of the actor that specify the reaction of the actor to its corresponding received message. In the

Java-like syntax of Rebeca, actors are instantiated from reactive class definitions that are similar to the concept of classes in Java. Actors in this sense can be assumed as objects in Java. Each reactive class declares the size of its message buffer, a set of state variables, and the messages to which it can respond. Reactive classes have constructors with the same name as their reactive class, that are responsible for initializing the actor's state.

Timed Rebeca [14] is an extension on Rebeca with time features which supports modeling and verification of time-critical systems. To this end, three primitives are added to Rebeca to address *computation time*, *message delivery time*, *message expiration*, and *period of occurrence of events*. In a Timed Rebeca model, each actor has its own local clock and the local clocks evolve uniformly. Methods are still executed atomically, however passing time while executing a method can be modeled. In addition, instead of a queue for messages, there is a bag of messages for each actor. We introduce Timed Rebeca using the example of the counting-based scheme presented in the previous section. A Timed Rebeca model consists of a number of reactive class definitions which provide type and behavior specification for the actors instantiated from them. There are two reactive classes `BroadcastingActor` and `Vehicle` in the implementation of counting-based WMD in VeriVANca as shown in Listing 1.

Each reactive class consists of a set of state variables and a message bag with the size specified in parentheses after the name of the reactive class in the declaration. For example, reactive class `Vehicle` has state variables `isAv`, `direction`, `latency`, `counter`, etc. The size of the message bag for this reactive class is set to five. The local state of each actor consists of the values of its state variables and the contents of its message bag. Being an actor-based language, Timed Rebeca benefits from asynchronous message passing among actors. Upon receiving a message, the message is added to the actor's message bag. Whenever the actor takes a message from the message bag, the routine which is associated with that message is executed. These routines are called message servers and are implemented in the body of reactive classes.

As depicted in Listing 1, the message servers of `Vehicle` are `move`, `receive`, `alertAccident`, `stop`, and `finishWait`. In order for an actor to be able to send a message to another actor, the sender has to have a direct reference to the receiver actor. For example, in Line 19, the message `alertAccident` is sent to `self` which represents a reference to the actor itself. However, in order to model a WMD scheme in VANETs, the warning message should reach actors which are in the range of the sender actor.

Listing 1 Counting-based scheme in Timed Rebeca

```
1 env int RANGE = 10;
2 env int THRESHOLD_WAITING = 4;
3 env int MESSAGE_SEND_TIME = 1;
4 env int C_THRESHOLD = 3;
5 abstract reactiveclass BroadcastingActor (5) {
6  statevars { int id, x, y; }
7  abstract msgsrv receive(int data);
8  void broadcast(int data) { ... }
9  double distance(BroadcastingActor bActor,
       BroadcastingActor cActor){...}
10 }
11 reactiveclass Vehicle extends BroadcastingActor(5){
12  statevars{
13   boolean isAV;
14   int direction, latency, destX, destY, counter;
15  }
16  Vehicle (/*List of Parameters*/){
17   /*Variables Initializations*/
18   if (isAV) {
19     self.alertAccident();
20   } else
21    self.move() after(latency);
22  }
23  msgsrv alertAccident(){ ... }
24  msgsrv move() { ... }
25  msgsrv stop () { ... }
26  msgsrv finishWait(int hop) { ... }
27  msgsrv receive(int hopNum) { ... }
28 }
29 main {
30  Vehicle v1():(0,0,10,RIGHT,1,10,10,true);
31  Vehicle v2():(1,10,0,UP,2,10,10,false);
32  Vehicle v3():(2,-1,0,RIGHT,1,10,0,false);
33  Vehicle v4():(3,0,1,DOWN,2,0,-10,false);
34  Vehicle v5():(4,3,0,LEFT,1,-10,0,false);
35 }
```

Basically, in Rebeca, the concept of known rebecs was introduced for an actor to specify the actors to which it can send messages. However, to implement applications in ad-hoc networks, a more flexible sending mechanism is needed. Two Rebeca extensions b-Rebeca [24] and w-Rebeca [25] have been proposed to provide more complex sending mechanism. In b-Rebeca the concept of known rebecs is eliminated and it is assumed that the only communication mechanism among actors is broadcasting; hence, only a fully connected network can be modeled. Note that the type of broadcasting introduced in b-Rebeca is not the same as the location-based broadcasting in VANETs. In location-based broadcasting, only the actors in the range of each other are connected in the Rebeca model. Regarding this assumption, a counter-based reduction technique is used in b-Rebeca to reduce the state space size of the model making it impossible to send messages to a subset of actors.

The other extension w-Rebeca, which is developed for model checking of wireless ad-hoc networks, uses an adjacency matrix in the model checking engine, to con-

sider connectivity of actors. In this approach, by random changes in the value of adjacency matrix, all the possible topologies of the network are considered in the model checking. Note that users are allowed to define a set of topological constraints and the topologies that do not fulfill the constraints are not considered in the model checking. w-Rebeca does not support timing in the model which is essential for developing models in the domain of VANET, since there are some real-time properties that need to be considered. Besides, considering all possible topologies —some of which may not be possible in the reality of the model— results in a bigger state space for the model. In addition, considering these infeasible topologies, may cause false-negative results when checking correctness properties.

In this work, we extended Rebeca to support Inheritance and used the inheritance mechanism of Timed Rebeca to implement this customized sending strategy. More details of this extension will be discussed in the next section.

4 Inheritance in Rebeca

As mentioned before, developing different message dissemination algorithms in Rebeca requires a variety of communication mechanisms that are not supported by the current extensions of Rebeca. So, we extended Rebeca to support the inheritance mechanism and user-defined communication mechanism to enable it for the modeling of this type of applications. In object-oriented design, inheritance mechanism enables classes to be derived from another class and form a hierarchy of classes that share a set of attributes and methods. Using this approach, we encapsulated new communication mechanisms in a base reactive class and all other actors which need that type of communication are inherited from the base reactive class. Note that w-Rebeca and b-Rebeca [26] proposed a broadcasting-based communication mechanism which cannot be used for message dissemination purposes as they mainly work based on distance-based wireless communication.

4.1 Abstract Syntax of Rebeca with Inheritance

To enable formal description of the semantics of Rebeca with inheritance, we have to provide an abstract specification for the Syntax of it. The proposed modifications in the syntax of Rebeca family extensions are developed in a way that the minimum changes are applied to their former syntax.

In the first step, we present the notations used in the rest of the article. These notation are based on the work of [12]. Given a set A, the set A^* is the set of all finite sequences over elements of A, the set $\mathcal{P}(A)$ is the power set of A, and the set $\mathcal{P}_{\mathbb{N}}(A)$ is the power multiset of A. For a sequence $a \in A^*$ of length n, the symbol a_i denotes the i^{th} element of the sequence, where $1 \leq i \leq n$. Using this notation, we may also write the sequence a as $\langle a_1, a_2, \cdots, a_n \rangle$. The empty sequence is represented by ϵ, and $\langle h|T \rangle$ denotes a sequence whose first element is $h \in A$ and $T \in A^*$ is the sequence comprising the elements in the rest of the sequence. For two sequences σ and σ' over A, the operator $\oplus$ is defined as $\oplus : A^* \times A^* \rightarrow A^*$ for the concatenation of two sequences such that $\sigma \oplus \sigma'$ is a sequence obtained by appending σ' to the end of σ. Consequently, getting the prefix of σ with length l takes place using $\ominus : A^* \times \mathbb{N} \rightarrow A^*$ operator.

A Rebeca with inheritance model consists of a set of reactive class declarations and a main block which specifies actors of the model. A reactive class is defined as an instance of type $RClass = CID \times \{\epsilon, CID\} \times \mathcal{P}(Mtds) \times \mathcal{P}(Knowns) \times \mathcal{P}(Vars) \times \mathcal{P}(Mtds)$ such that:

- CID is the set of all reactive class identifiers in the model.
- $Mtds$ is the set of all method declarations.
- $Knowns$ is the set of all the identifiers of known actors.
- $Vars$ is the set of all variable names.

The tuple $(cid, pcid, consts, knowns, vars, mtds)$ defines a reactive class which has the identifier cid, is inherited from the reactive class $pcid$, the constructor method $const$, the set of known actors $knowns$, the set of state variables $vars$, and the set of methods $mtds$. Each method (and the constructor method) is defined as the triple $(m, p, b) \in MName \times Var^* \times Stat^*$, where m is the name of the message the method is used to serve, p is the sequence of the names of the formal parameters, and b contains the sequence of statements comprising the body of the method.

In Rebeca with inheritance, the set of statements is defined as $Stat = Assign \cup Cond \cup Send \cup \{skip\}$, where different types of statements are defined as below. The meaning of the below statements is the same as the general purpose programming languages. In the following, $Expr$ is the set of integer expressions defined over usual arithmetic operators (with no side effects) and $BExpr$ is the set of Boolean expression defined over usual relational and logic operators. We do not provide more details of the expressions in this article.

- $Assign = Var \times Expr$ is the set of assignment statements. We use the notation $var := expr$ as an alternative to $(var, expr)$.

- $Cond = BExpr \times Stat^* \times Stat^*$ is the set of conditional statements. We use the notation *if* $expr$ *then* σ *else* σ' as an alternative to $(expr, \sigma, \sigma')$.
- $Send = (ID \cup \{self\}) \times MName \times Expr^*$ is the set of send statements. We use the notation $x.m(e)$ as alternative to (x, m, e) to show that message m is sent from actor x with the set of parameters e.
- *skip* is a predefined statement that has no effect.

In the main part of a model, actors are defined as instances of reactive classes. The set of actors is defined as $Actor = CID \times AID \times AID^* \times Expr^*$ such that $(c, a, k, p) \in Actor$ defines an actor instantiated from reactive class c, with identifier a, the set of known actors k, and the set of parameters of its constructor p. Note that supporting inheritance does not result in any modification in the syntax of statements and instantiation part of Rebeca family models.

Having the above definitions, the set of Rebeca models is specified by $\mathcal{P}(RClass) \cup \mathcal{P}(Actor)$, where the first component contains the specification of reactive classes and the second component corresponds to the main block consisting of a sequence of actor instantiations. The BNF presentation of the syntax of Rebeca with inheritance is presented in Fig 1. In comparison with the former grammar, `extends` and `abstract` are two new keywords which are added to the syntax of Rebeca.

4.2 Semantics of Rebeca with Inheritance

In this section, we present the semantics of Rebeca with Inheritance. Prior to presenting the semantics, we present the notations used in the rest of the article.

For a function $f : X \to Y$, we use the notation $f[x \mapsto y]$ to denote the function $\{(a, b) \in f | a \neq x\} \cup \{(x, y)\}$ and $D(f)$ to denote the domain of f (which is X here). Following this, we use the notation $f[x_1 \mapsto y_1 \wedge \cdots \wedge x_n \mapsto y_n]$ to denote the function $\{(a, b) \in f | a \notin \{x_1, \cdots, x_n\}\} \cup \{(x_1, y_1), \cdots, (x_n, y_n)\}$. We also use the notation $x \mapsto y$ as an alternative to (x, y). For $X' \subseteq X$, we write $f|^{X'}$ as the restriction of f to X', i.e., $\{(x, y) \in f | x \in X'\}$. Having two sequences a and b of the same size n, the function $map(a, b)$ returns the mapping of the elements of a into b such that $map(a, b) = \{a_i \mapsto b_i | 1 \leq i \leq n\}$, assuming that the elements of a are distinct.

We also define the following auxiliary functions to be used in defining the formal semantics:

- $body : AID \times MName \to Stat^*$, in which $body(x, m)$ returns the body of the method m of the reactive class which actor identified by x is instantiated from, appended by the special element $endm$, which denotes the end of the method.
- $params : AID \times MName \to Var^*$, in which the function $params(x, m)$ returns the list of formal parameters of the method m of the reactive class which the actor identified by x is instantiated from.
- $svars : AID \to \mathcal{P}(Var)$ which returns the names of the state variables of the reactive class which actor identified by x is instantiated from.
- $eval_v : Expr \to Val$ abstracts away the semantics of expressions by evaluating an expression within the specific context $v : Var \to Val$. Note that Val contains all possible values that can be assigned to the state variables or to be used within the expressions. Here, we have $Val = \mathbb{Z} \cup \{True, False\}$. We assume $eval_v$ is overloaded to evaluate a sequence of expressions: $eval_v(\langle e_1, e_2, \cdots, e_n \rangle) = \langle eval_v(e_1), eval_v(e_2), \cdots, eval_v(e_n) \rangle$. Note that $eval_v(e_1), eval_v(e_2), \cdots, eval_v(e_n)$ are evaluated sequentially not in parallel.
- $unify : \langle (Var \to Val) \rangle \to (Var \to Val)$ is a function that returns the union of a sequence of contexts which is given as its input. This function helps in finding all the variables which are defined or inherited in a reactive class. Note that using inheritance, state variables of a reactive class consists of the inherited state variables. The formal definition of this function is given below.

$$unify(A) = \begin{cases} A_n \cup unify(A')|^{D(A_n)} & A = \langle A' | A_n \rangle \\ \emptyset & A = \varepsilon \end{cases}$$

  Using this definition resolves name clash among state variables as it considers the closest variable's declaration in the inheritance hierarchy.
- $level : AID \times MName \to \mathbb{N}$ returns the level that the closest definition of the message server / method $MName$ for the actor AID that is found in its hierarchy. As we will show later, *level* is used to restrict the access of a message server / method to the state variables which are defined or inherited from its ancestors. For example, assume that there are reactive classes `A`, `B`, `C`, and `D` such that `B` is derived from `A`, `C` from `B`, and `D` from `C`. Also, the method `m()` is defined in `B`. In the case of calling `m` from the actor `ac` which is instantiated from `D`, although `ac` contains all the state variables of `A`, `B`, `C`, and `D`, but `m` only has access to the state variables of `A` and `B`. In this case, the value of *level* is set to 2.
- $upVar : \{(Var \to Val)\}^* \times VName \times newVal \to \{(Var \to Val)\}^*$ updates the value of the closest variable $name$ to the value val for the given function $upVar(v, name, val)$. The formal definition of

$$
\begin{aligned}
\mathit{Model} &::= \mathit{Class}^*\ \mathit{Main}\\
\mathit{Main} &::= \textbf{main}\ \{\ \mathit{InstanceDcl}^*\ \}\\
\mathit{InstanceDcl} &::= \mathit{className}\ \mathit{rebecName}(\langle \mathit{rebecName}\rangle^*) : (\langle \mathit{literal}\rangle^*);\\
\mathit{Class} &::= [\textbf{abstract}]\ \textbf{reactiveclass}\ \mathit{className}\ [\ \textbf{extends}\ \mathit{className}\ ]\\
&\quad \{\ \mathit{KnownRebecs}\ \mathit{Vars}\ \mathit{MsgSrvDef}^*\ \mathit{LocalMethodsDef}^*\ \}\\
\mathit{KnownRebecs} &::= \textbf{knownrebecs}\ \{\ \mathit{RebecDcl}^*\ \}\\
\mathit{Vars} &::= \textbf{statevars}\ \{\ \mathit{VarDcl}^*\ \}\\
\mathit{RebecDcl} &::= \mathit{className}\ \langle v\rangle^+;\\
\mathit{VarDcl} &::= \mathit{Type}\ \langle v\rangle^+;\ |\ \mathit{Type}\ [\ \mathit{number}\ ]^+\ v\\
\mathit{MsgSrvDef} &::= \mathit{MsgSrv} \mid \mathit{AbsMsgSrv}\\
\mathit{MsgSrv} &::= \textbf{msgsrv}\ \mathit{msgName}(\langle \mathit{ExtType}\ v\rangle^*)\ \{\ \mathit{Stmt}^*\ \}\\
\mathit{AbsMsgSrv} &::= \textbf{abstract msgsrv}\ \mathit{msgName}(\langle \mathit{ExtType}\ v\rangle^*);\\
\mathit{LocalMethodsDef} &::= \mathit{LocalMethods} \mid \mathit{AbsLocalMethods}\\
\mathit{LocalMethods} &::= \mathit{methodName}(\langle \mathit{ExtType}\ v\rangle^*)\ \{\ \mathit{Stmt}^*\ \}\\
\mathit{AbsLocalMethods} &::= \textbf{abstract}\ \mathit{methodName}(\langle \mathit{ExtType}\ v\rangle^*);\\
\mathit{Stmt} &::= \mathit{Assignment} \mid \mathit{SendMessage} \mid \mathit{MethodCall} \mid \mathit{ConditionalStmt} \mid \mathit{LoopStmt} \mid \mathit{LocalVars}\\
\mathit{Assignment} &::= v = \mathit{Exp};\ |\ v = ?(\mathit{Exp}\langle, \mathit{Exp}\rangle^+);\\
\mathit{SendMessage} &::= \mathit{rebecExp.msgName}(\langle \mathit{Exp}\rangle^*);\\
\mathit{MethodCall} &::= \mathit{methodName}(\langle \mathit{Exp}\rangle^*);\\
\mathit{ConditionalStmt} &::= \mathit{if}\ (\mathit{Exp})\ \{\ \mathit{Stmt}^*\ \}\ [\mathit{else}\ \{\ \mathit{Stmt}^*\ \}]\\
\mathit{LoopStmt} &::= \mathit{for}\ (\ \mathit{Exp}\ ;\ \mathit{Exp}\ ;\ \mathit{Exp}\ )\ \{\ \mathit{Stmt}^*\ \} \mid \mathit{while}\ (\mathit{Exp})\ \{\ \mathit{Stmt}^*\ \}\\
\mathit{LocalVars} &::= \mathit{ExtType}\ \langle v\rangle^+;\\
\mathit{Exp} &::= e \mid \mathit{rebecExpr}\\
\mathit{rebecExp} &::= \mathit{self} \mid \mathit{rebecTerm} \mid (\mathit{className})\mathit{rebecTerm}\\
\mathit{rebecTerm} &::= \mathit{rebecName} \mid \mathit{sender}\\
\mathit{ExtType} &::= \mathit{Type} \mid \mathit{float} \mid \mathit{double}\\
\mathit{Type} &::= \mathit{boolean} \mid \mathit{int} \mid \mathit{short} \mid \mathit{byte} \mid \mathit{className}
\end{aligned}
$$

Fig. 1 Abstract syntax of Rebeca(a slightly revised version of the syntax presented in [1]). Angle 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. The symbol ? shows non-deterministic choice. Identifiers *className*, *rebecName*, *methodName*, v, *literal*, and *type* denote class name, rebec name, method name, variable, integer number, and type, respectively; and e denotes an (arithmetic, boolean or nondeterministic choice) expression. The parameter t is an expression with natural number result.

$upVar$ is given below.

$$upVar(vs, n, v) = \begin{cases} T \oplus vs'[n \to v] & vs = \langle T | vs' \rangle,\\ & n \in D(vs')\\ upVar(T, n, v) \oplus vs' & o.w. \end{cases}$$

Now, the semantics of Rebeca with inheritance can be defined. For a given Rebeca model $\mathcal{M}$, the semantics of the model is defined in terms of transition system $TS = (S, s_0, Act, \to, AP, L)$, where S is the set of states, s_0 is the initial state, Act is the set of actions, $\to \subseteq S \times Act \times S$ is the transition relation, AP is the set of atomic propositions, and $L : S \to 2^{AP}$ is the labeling function, described as the following.

- The global state of a Rebeca model is represented by a function $s : AID \to (Var \to Val) \times \mathcal{P}_{\mathbb{N}}(Msg) \times (Stat^*, \mathbb{N})$, which maps an actor's identifier to the local state of the actor. The local state of an actor is defined by a tuple like $(v, q, (\sigma, l))$, where v :

$Var \rightarrow Val$ gives the values of the state variables of the actor, $q : \mathcal{P}_{\mathbb{N}}(Msg)$ is the message queue of the actor, $\sigma : Stat^*$ contains the sequence of statements the actor is going to execute to finish the service to the message currently being processed, and l shows the level of the currently executing message server. Here, $Msg = AID \times MName \times (Var \rightarrow Val)$ is used as the type for the messages which are passed among actors. In a message $(i, m, r) \in Msg$, i is the identifier of the sender of this message, m is the name of its corresponding method, r is a function mapping argument names to their values. Note that the sequence of statements is put as a part of the states to make the operation semantics easier to understand and more readable not for supporting dynamic statement definition and configuration. Also, as mentioned before, we assume that actors communicate via message passing and put their incoming messages into message bags.

- In the initial state, the values of state variables and content of the actors' message queues are set based on the statements of their constructor methods.
- The set of actions is defined as $Act = MName \cup \{\tau\}$.
- The transition relation $\rightarrow \subseteq S \times Act \times S$ defines the transitions between states which are taking a message from the message queue and continuing the execution of statements. The SOS rules of Table 1 define these transitions. Note that we associated a rule name with τ transitions to relate τ transitions to their corresponding rules.
- AP contains the name of all of atomic propositions.
- The function $L : S \rightarrow 2^{AP}$ associates a set of atomic propositions with each state, shown by $L(s)$ for a given state s.

Finally, we assumed that Rebeca models are well-formed. The following rules define the well-formedness of a Rebeca model which is hard to (or cannot be) described in the grammar, but may be statically checked.

- **Unique Identifiers.** The actor identifiers are unique within a Rebeca model.
- **Unique Variables.** The names of the state variables of an actor are unique.
- **Unique Methods.** The names of the methods of an actor are unique.
- **Unique Parameters.** The names of the formal parameters of a method are unique and different from the state variables of the enclosing actor.
- **Type Safety.** The model is well typed, i.e.,
  - expressions are well-typed,
  - the type of the right side of an assignment is upcastable to the type of the left side,
  - the conditions of the conditional statements are of type Boolean, and
  - the receiver of a message has a method with the same name as the message.
- **Well-Formed Arguments.** The list of actual arguments passed to a message send statement conforms to the list of formal parameters of the corresponding method, in both length and type.

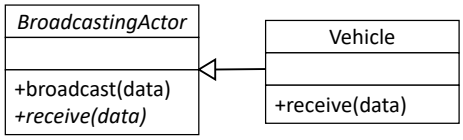


Fig. 2 The UML class diagram of VeriVANca

4.3 Inheritance for Extensions of Rebeca

Reviewing the abovementioned semantic rules illustrates that only minor modifications are needed to enable the other extensions of Rebeca to support inheritance. These modifications are in how to specify state variables of reactive classes, the scope part of the *eval* function, resolving name of method / message server, and storing the level of the method / message server which currently being executed.

Considering Timed Rebeca [13], applying these modifications to the semantic rules results in modifying the set of SOS rules which are presented in Table 2. Note that the state variable part of all of the other semantic rules has to be modified which is straightforward.

These modifications are sufficient to enable Probabilistic Timed Rebeca [10] to support inheritance too.

5 The VeriVANca Framework

In object-oriented design, inheritance mechanism enables classes to be derived from another class and form a hierarchy of classes that share a set of attributes and methods. Using this approach, VeriVANca is developed as a framework that encapsulates broadcasting mechanism in a reactive class called `BroadcastingActor` and all other actors are inherited from it to use the broadcasting mechanism. Figure 2 illustrates this fact in a UML class digram.

In `BroadcastingActor`, the `broadcast` method that is shown in Listing 2 mimics the distance-based sending mechanism of vehicles in VANETs. In VeriVANca behaviors of vehicles and warning message dissemination scheme are implemented in the `Vehicle` reactive class. Broadcasting data by a vehicle results in receiving a message containing that data by the vehicles in the range of the sender actor. In the body of this method, all

(**take – message**)	$\frac{s(x) = (v, \langle (ac, mg, pr) \vert T\rangle, (\epsilon, \epsilon))}{s \xrightarrow{mg} s[x \mapsto (v \oplus \{pr \cup \{(self, x)\} \cup \{(sender, ac)\}, T, (body(x, mg)) \oplus \textbf{endmsgsrv}, level(x, mg))]}$
(**method – call**)	$\frac{s(x) = (v, q, (\langle method(e) \vert \sigma\rangle, l))}{s \xrightarrow{\tau} s[x \mapsto (v, q, (body(x, method) \oplus \textbf{endm} \oplus \textbf{level} : \mathbf{l} \oplus \sigma, level(x, method)))]}$
(**assignment**)	$\frac{s(x) = (v, q, (\langle var := expr \vert \sigma\rangle, l))}{s \xrightarrow{\tau} s[x \mapsto (upVar(v, var, eval_{unify(v \ominus l)}(expr)), q, (\sigma, l))]}$
(**Conditional$_\mathbf{T}$**)	$\frac{s(x) = (v, q, (\langle \textbf{if } expr \textbf{ then } \sigma \textbf{ else } \sigma' \vert \sigma''\rangle, l)) \wedge eval_{unify(v \ominus l)}(\text{expr}) = \textbf{True}}{s \xrightarrow{\tau} s[x \mapsto (v, q, (\sigma \oplus \sigma'', l))]}$
(**Conditional$_\mathbf{F}$**)	$\frac{s(x) = (v, q, (\langle \textbf{if } expr \textbf{ then } \sigma \textbf{ else } \sigma' \vert \sigma''\rangle, l)) \wedge eval_{unify(v \ominus l)}(\text{expr}) = \textbf{False}}{s \xrightarrow{\tau} s[x \mapsto (v, q, (\sigma' \oplus \sigma'', l))]}$
(**nondet – assign**)	$\frac{s(x) = (v, q, (\langle var :=?(expr_1, expr_2, \cdots, expr_n) \vert \sigma\rangle, l))}{\bigvee_{1 \le i \le n} s \xrightarrow{\tau} s[x \mapsto (v, var, eval_{unify(v \ominus l)}(expr_i))], q, (\sigma, l))]}$
(**send**)	$\frac{s(x) = (v, q, (\langle y.m(e_1) \vert \sigma, l)\rangle) \wedge s(y) = (v', q', (\sigma', l')) \wedge p = params(y, m)}{s \xrightarrow{\tau} s[x \mapsto (v, q, (\sigma, l)) \wedge y \mapsto (v', q' \oplus \{(m, (map(p, eval_{unify(v \ominus l)}(e_1))))\}, (\sigma', l')]}$
(**skip**)	$\frac{s(x) = (v, q, (\langle \textbf{skip} \vert \sigma\rangle, l))}{s \xrightarrow{\tau} s[x \mapsto (v, q, (\sigma, l))]}$
(**end – msgSrv**)	$\frac{s(x) = (v, q, (\langle \textbf{endmsgsrv}\rangle, l))}{s \xrightarrow{\tau} s[x \mapsto (v \ominus (\vert v\vert - 1), q, (\epsilon, \epsilon))]}$
(**end – method**)	$\frac{s(x) = (v, q, (\langle \textbf{endm} \vert \sigma\rangle, l))}{s \xrightarrow{\tau} s[x \mapsto (v \ominus (\vert v\vert - 1), q, (\sigma, l))]}$
(**change – level**)	$\frac{s(x) = (v, q, (\langle \textbf{level} : \mathbf{l}' \vert \sigma\rangle, l))}{s \xrightarrow{\tau} s[x \mapsto (v, q, (\sigma, l'))]}$

Table 1 The SOS rules of Rebeca with Inheritance

(**take – message**)	$\frac{s(x) = (v, \langle (ac, mg, pr, ar, dl) \vert T\rangle, (\epsilon, \epsilon), t, \epsilon) \wedge ar \le t \wedge dl \ge t}{s \xrightarrow{mg} s[x \mapsto (v \oplus pr \cup \{(self, x)\} \cup \{(sender, ac)\}, T, (body(x, mg)) \oplus \textbf{endmsgsrv}, level(x, mg)), t, t)]}$
(**internal**)	$\frac{s(x) = (v, q, (\langle st, \sigma\rangle\, l), t, r) \wedge t = r}{s \xrightarrow{\tau} s[effect(st, x)]}$
(**time – progress**)	$\frac{s \overset{mg}{\not\to} \wedge\, s \overset{\tau}{\not\to} \wedge\, n_1 = \min_{x \in AID}\{ar \vert s(x) = (v, \langle (ac, mg, pr, ar, dl) \vert T\rangle, \epsilon, t, \epsilon)\} \wedge n_2 = \min_{x \in AID}\{r \vert s(x) = (v, q, (\sigma, l), t, r)\}}{s \to s[\forall x \in AID \cdot x = (v, q, (\sigma, l), t, r) \mapsto (v, q, (\sigma, l), \min\{n_1, n_2\}, r)]}$
(**delay**)	$\frac{s(x) = (v, q, (\langle \textbf{delay}(e) \vert \sigma\rangle, l), t, r) \wedge r = t}{s \xrightarrow{\tau} s[x \mapsto (v, q, (\sigma, l), t, r + eval_{unify(v \ominus l)})]}$

Table 2 The SOS rules of Timed Rebeca with Inheritance

actors —that are derived from `BroadcastingActor`— are examined in terms of their distance to the sender (Line 5). If the distance between an actor and the sender is less than the threshold `RANGE` (Line 6), the data is sent to the actor by an asynchronous message server call of `receive` (Line 7). As `BroadcastingActor` has no idea about the behavior of vehicles, upon receiving the `receive` message, the template method design pattern [7] is used in the implementation of `receive`. So, the `receive` message server is defined as an abstract message server in `BroadcastingActor` and its body is implemented in `Vehicle`.

Listing 2 Body of broadcast Method in Broadcasting Actor

```
1 void broadcast(int data) {
2  ArrayList<ReactiveClass> allActors=getAllActors();
3  for(int i = 0; i < allActors.size(); i++) {
4   BroadcastingActor ba =
        (BroadcastingActor)allActors.get(i);
5   double distance = distance (ba , self);
6   if(distance < RANGE) {
7    ba.receive(data) after (MESSAGE_SEND_TIME);
8   }
9  }
10 }
11 double distance(BroadcastingActor bActor ,
       BroadcastingActor aActor){
12    int xPart = pow(aActor.x - bActor.x, 2);
13    int yPart = pow(aActor.y - bActor.y, 2);
14    return sqrt(xPart + yPart);
15 }
```

Using this separation significantly improves the usability and flexibility of VeriVANca and other not known rebec based communication mechanisms can be implemented in the same way. For example, in the case of distance-based delay in communication, there is no limitation on the range of message sending but the receiving time is set based on the location of moving items. This behavior is implemented in Listing 3. As shown in lines 5 and 6, the communication delay for `receive` message is set based on the distance of vehicles. Note that the definition of `distance` in this example is the same as that of Listing 2.

Listing 3 Body of distance-based communication delay in BroadcastingActor

```
1 void broadcast(int data) {
2  ArrayList<ReactiveClass> allActors=getAllActors();
3  for(int i = 0; i < allActors.size(); i++) {
4   BroadcastingActor ba =
        (BroadcastingActor)allActors.get(i);
5   int distDelay = (int)distance (ba , self);
6   ba.receive(data) after (distDelay);
7   }
8  }
9 }
```

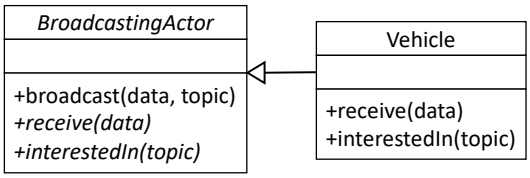


Fig. 3 The UML class diagram of Publisher/Subscriber communication mechanism

In addition to the broadcasting-based communication mechanisms, more complex communication mechanisms can be implemented in VeriVANca. For example, *publish/subscribe* communication mechanism can be implemented using another helper function. Figure 3 shows the UML class diagram representation of publish/subscribe communication mechanism in Rebeca and its implementation is presented in Listing 4.

In this implementation, each actor implements its own `interestedIn` method in a way that it returns `true` if this actor interested in the given topic as the parameter of `interestedIn`.

Listing 4 Body of broadcast method in publish/subscribe communication mechanism

```
1 void broadcast(int data, int topic) {
2  ArrayList<ReactiveClass> allActors=getAllActors();
3  for(int i = 0; i < allActors.size(); i++) {
4   BroadcastingActor ba =
        (BroadcastingActor)allActors.get(i);
5   boolean interested = ba.interestedIn(topic);
6   if(interested) {
7    ba.receive(data) after (MESSAGE_SEND_TIME);
8   }
9  }
10 }
```

6 Experimental Results

In this section we present two different case studies and illustrate how functional analysis and performance evaluation can be made using VeriVANca. To demonstrate the applicability of VeriVANca, both of them are analyzed in different configurations. As mentioned before, concurrent behaviors of the system components may cause uncertainty which is clearly observable in the presented scenarios, but may not be detected using simulation-based techniques. For the case of the TLO scheme, we show that nondeterminism causes starvation and for the case of the counting-based scheme, it causes different results in the performance of the algorithm. Furthermore, we illustrate that the approach is scalable regarding the number of cars with traffic patterns that do not contain congested areas. Note that the

following experiments have been executed on a Macbook Air with Intel Core i5 1.3 GHz CPU and 8GB of RAM, running macOS Mojave 10.14.2 as the operating system. Development of these experiments are performed in Afra, modeling and verification IDE of Rebeca family languages [4].

6.1 Counting-Based Scheme in VeriVANca

The model of Counting-Based scheme in Rebeca is presented in Listing 5. Note that definition of `broadcast` and `distance` methods is omitted in Listing 5, as it is the same as that of Listing 2. In the following implementation, three message servers `alertAccident`, `finishWait`, and `receive` provide the behavior of the scheme. When `Vehicle` actors are instantiated, their constructor methods are executed resulting in sending one of the following messages to themselves:

- `alertAccindent`: sent by the accident vehicle (i.e. `v1` as shown in Line 65) to start the WMD algorithm (Line 34),
- `move`: sent by the other actors to begin moving with their pre-defined `latency` (Line 36); an actor performs this through sending `move` message periodically to itself (Lines 39 to 48).

The algorithm of Counting-Based scheme, as implemented in Listing 5, begins by serving `alertAccident` message in the accident vehicle. Upon the execution of `receive` (Lines 56 to 62), if the `counter`, which is initially set to zero for all actors (Line 32), is zero — meaning that it is the first time the actor is receiving the warning message— a watchdog timer is started. This is implemented by sending the `finishWait` message to the actor itself with `THRESHOLD_WAITING` as its arrival time(Line 58). In addition, the value of `counter` is set to one to indicate that this is the first call of `receive` (Lines 60). The next calls of `receive` result in increasing the value of `counter`, which represents the number of received warning messages. When message server `finishWait` is executed by an actor, showing that the watchdog timer is expired, as shown in Line 51, the value of `counter` is compared with `C_THRESHOLD`, i.e. the threshold considered for the counter. By not exceeding the threshold, i.e., the area around the actor is not covered by enough number of warning messages, the actor broadcasts the warning message (Line 52).

Listing 5 Counting-based scheme in Timed Rebeca

```
1 env int RANGE = 10;
2 env int THRESHOLD_WAITING = 4;
3 env int C_THRESHOLD = 3;
4 env int RIGHT = 0;
5 env int LEFT = 1;
6 env int UP = 2;
7 env int DOWN = 3;
8 abstract reactiveclass BroadcastingActor(5) {
9  statevars {
10  int id, x, y;
11 }
12 abstract msgsrv receive(int data);
13 void broadcast(int data) { ... }
14 double distance(BroadcastingActor bActor ,
       BroadcastingActor cActor) { ... }
15 }
16 reactiveclass Vehicle extends BroadcastingActor(5) {
17 statevars{
18  boolean isAV;
19  int direction, latency;
20  int destX, destY;
21  int counter;
22 }
23 Vehicle(int vid, int X, int Y, int dir, int vLatency,
       int dX, int dY, boolean isAccidentVehicle) {
24  id = vid;
25  x = X;
26  y = Y;
27  direction = dir;
28  latency = vLatency;
29  destX = dX;
30  destY = dY;
31  isAV = isAccidentVehicle;
32  counter = 0;
33  if (isAV) {
34   self.alertAccident();
35  } else
36   self.move() after(latency);
37 }
38 msgsrv alertAccident() { broadcast(0); }
39 msgsrv move() {
40  switch (direction) {
41   case 0: x++; break;
42   case 1: x--; break;
43   case 2: y++; break;
44   case 3: y--; break;
45  }
46  if (x != destX || y != destY)
47   self.move() after(latency);
48 }
49 msgsrv stop() { stop(); }
50 msgsrv finishWait(int hopNum) {
51  if (counter < C_THRESHOLD)
52   broadcast(hopNum++);
53  else
54   stop();
55 }
56 msgsrv receive(int hopNum) {
57  if (counter == 0) {
58   finishWait(hopNum) after(THRESHOLD_WAITING);
59  }
60  counter++;
61 }
62 }
63
64 main {
65  Vehicle v1():(0,0,10,RIGHT,1,10,10,true);
66  Vehicle v2():(1,10,0,UP,2,10,10,false);
67  Vehicle v3():(2,-1,0,RIGHT,1,10,0,false);
68  Vehicle v4():(3,0,1,DOWN,2,0,-10,false);
```

```
69  Vehicle v5():(4,3,0,LEFT,1,-10,0,false);
70 }
```

The configuration depicted in Figure 4(a) is used for the analysis of the Counting-Based scheme. In this scenario, the value of `C_THRESHOLD` is set to 2 and the `RANGE` is set to 4. The scenario begins with the vehicle A broadcasting the warning message (Figure 4(b)). This broadcast results in increasing the counters of the vehicles A, B, C, and E by one. In the next round two following cases may happen.

1. **The watchdog timer of vehicle E expires after receiving the message from B:** In this case, as the counter has reached the threshold, E does not forward the warning message as shown in Figure 4(c). Following this case, the algorithm continues with vehicles D, H, and F being selected as forwarding nodes and rebroadcasting the message (Figures 4(d) to 4(f). As a result, it takes 5 hops for all the vehicles to get informed of the warning message. Note that the same scenario happens when C forwards the message before the expiration of the watchdog timer of E.
2. **The watchdog timer of vehicle E expires before receiving warning message from B and C:** In this case, since the counter of E is less than the threshold, E must forward the warning message (Figure 5(a)). In the next step, vehicle F broadcasts the message and all non-informed vehicles receive the warning message and algorithm finishes in 3 hops.

Achieving two different numbers for performance of this algorithm shows that beside correctness properties, providing guaranteed values for performance results requires applying formal verification techniques as well. We analyzed this scenario with different values for range and counter threshold, the result of three of them are shown in Figure 6.1. The results show that this phenomenon is not rare and can be observed in many cases.

For the purpose of scalability analysis, we have modeled a four-lane street which contains about 30 vehicles. These vehicles are distributed in a way that there is no congested area in the street as shown in Figure 7(a). Note that we assumed cars are fixed and do not move. Analyzing this model using Afra results in the execution time of 11 seconds, reaching 19,588 states and 110,627 transitions. To examine the scalability of the model, we added new cars in two different areas. First, we increased the length of the street and added new vehicles to the tail of the street of Figure 7(a). The newly added cars follow the same distribution to avoid creating congested areas. This way of scaling resulted in the execution time of 15 seconds, 23,734 states, and 133,255 transitions for 35 vehicles and 18 seconds, 25,872 states, and 143,727 transitions for 40 vehicles (i.e. about 1.3 times more than the first case).

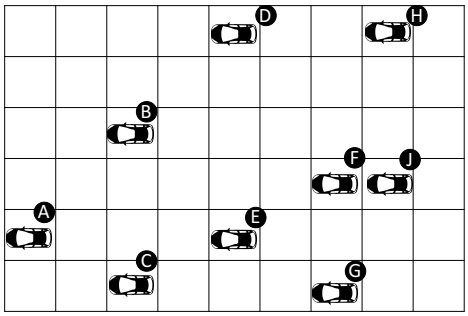
In the second way, the newly added vehicles increased congestion in some areas (Figure 7(b)). Scaling in this way increases the execution time of the model to 120 seconds and the number of reached states and transitions to 157,086 and 1,265,839, respectively (i.e. about 10 times more than the previous case). This is because of the fact that in a congested area, the number of delivered warning messages to each vehicle grows rapidly and all of the possible orders of execution for messages with the same execution time are considered in the model checking. This results in a sharp growth in the size of the state space and model checking time consumption.

6.2 TLO Scheme in VeriVANca

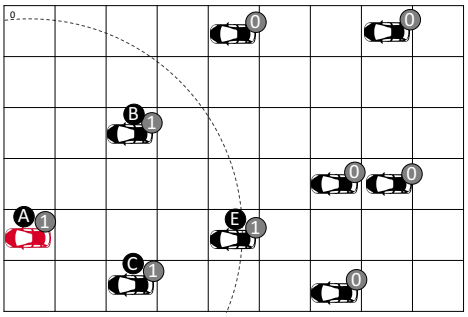
In the TLO scheme, explained in Section 2.1, upon receiving the warning message for the first time, the `runTLO` method is called. In the body of this method, if the actor has not received the duplicate warning message from a selected TLO node as a sign of its successful broadcast, the `isTLO` method is called. This property is checked by examining the value of state variable `received` in Line 86. The `isTLO` method is implemented in the `BroadcastingActor` and checks if the actor is the furthest node in the range of the sender and returns the result as a boolean value. If the return value is true, the actor is the last one in the range and is selected as the TLO node to forward the warning message.

Then the value of `received` is set to true to show that broadcasting has been successful (Line 89). In the case that the actor is not the last one in the range, the actor should wait for a while to make sure that the selected TLO node has successfully broadcasted the warning message (Line 91 and 92). To this end, the actor sets the value of `isWaiting` to true to show that it is in the waiting mode. In this case a watchdog timer is set for the actor by sending message `finishWait` to itself by execution time of `THRESHOLD_WAITING` (Line 92).

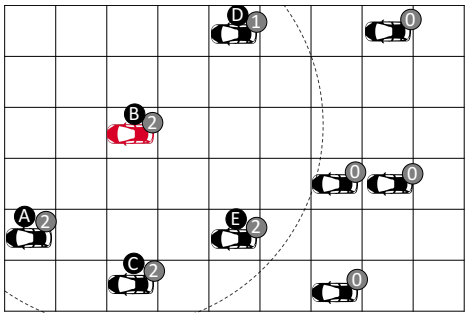
The message server `receive`, mimics receiving the warning message. In the body of this message server, if the actor is not in the waiting mode, `isTLO` is executed to select the TLO forwarding node (Lines 80 and 81). Otherwise, `isWaiting` is set to false since this message is interpreted as a successful broadcast of the TLO node (line 83). The `finishWait` message server is executed upon expiration of the watchdog timer and it checks the value of `isWaiting` (Line 76). In the case of false



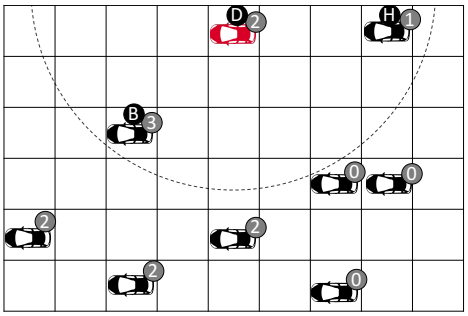
(a) Configuration of the scenario



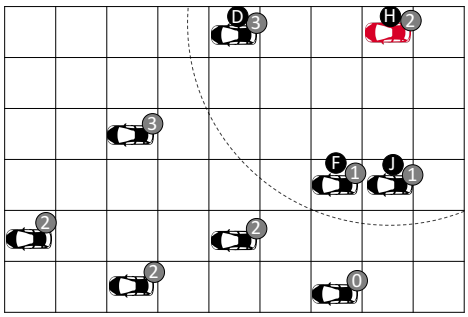
(b) Vehicle A starts broadcasting



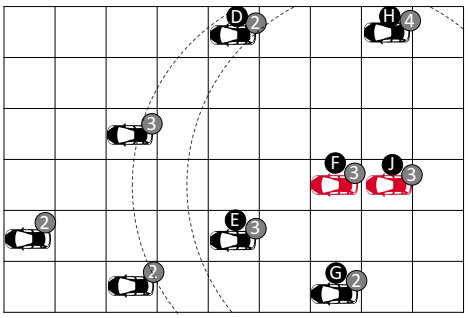
(c) B broadcasts before expiration of the watchdog timer of E



(d) D forwards the warning message

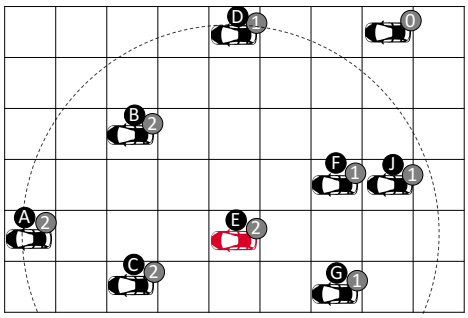


(e) H rebroadcasts the message

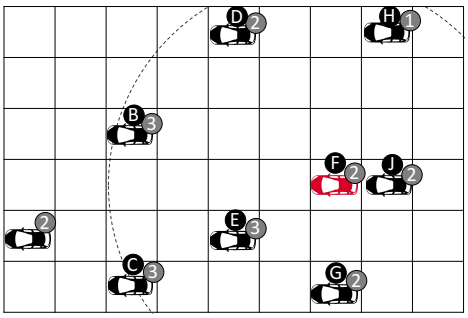


(f) F and (or) J forward(s) the message and algorithm finishes

Fig. 4 A case of the scenario for the counting-based scheme

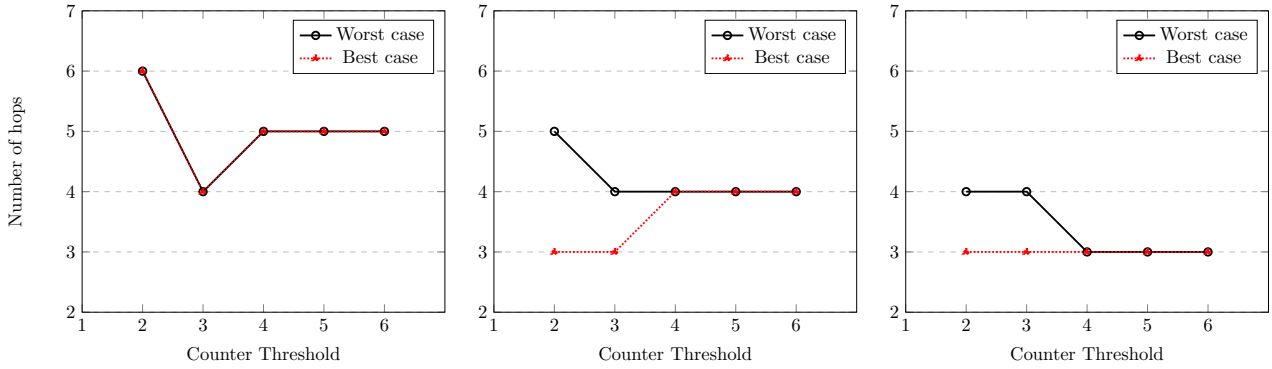


(a) E is selected as forwarder (instead of B as depicted in Figure 4(c))



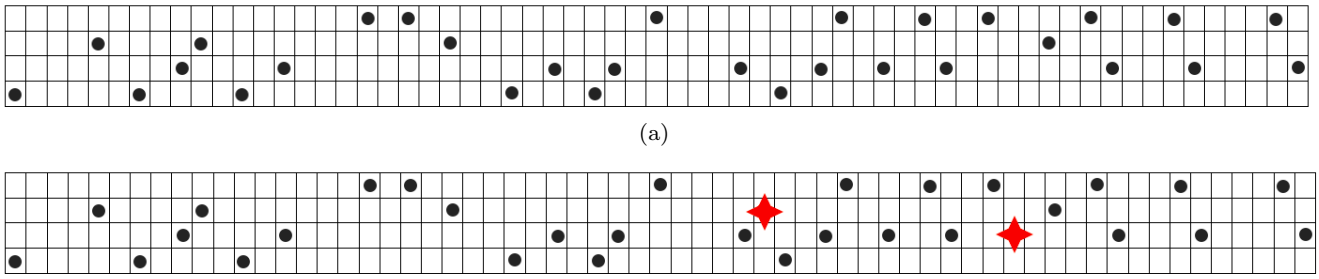
(b) F broadcasts the message and algorithm finishes

Fig. 5 Another case of the scenario for the counting-based scheme



(a) Range is 3 (b) Range is 4 (c) Range is 5

Fig. 6 Analysis results of the counting-based scheme with different values for the range and counter threshold (Note that Y axis shows the number of hops required for termination of the algorithm)



(a)

(b)

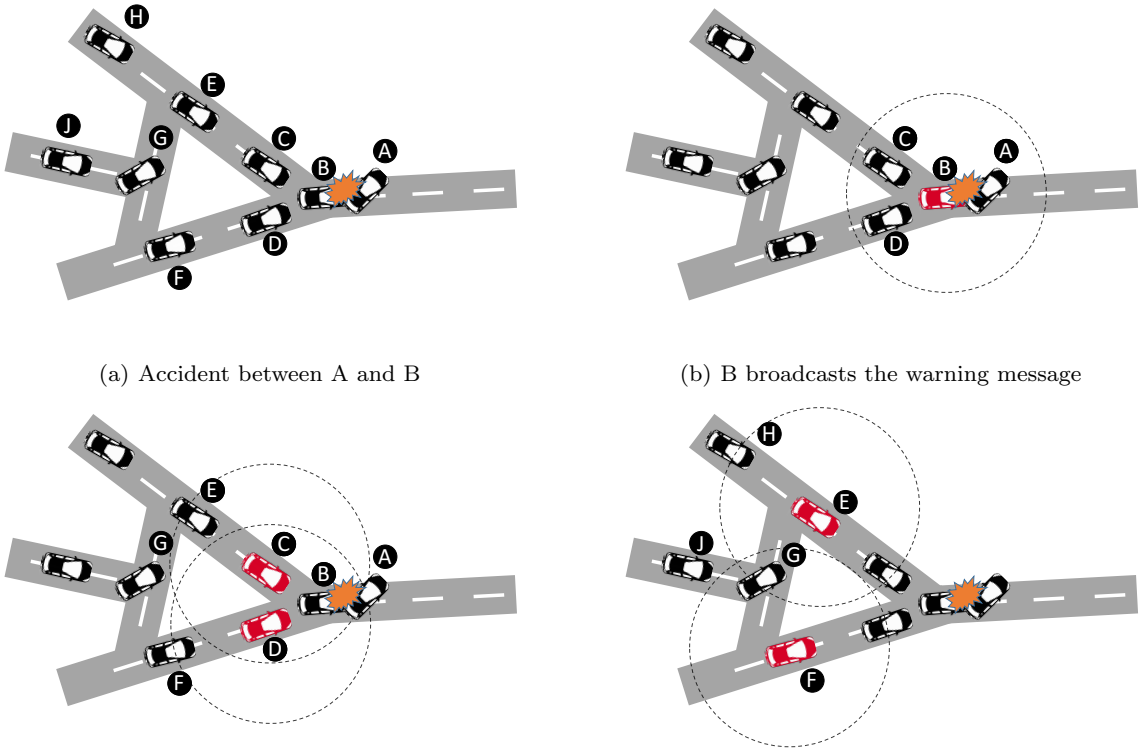
Fig. 7 Configuration of the scenario used for scalability analysis

value for `finishWait`, the actor has not received any warning message from the already selected TLO node; so, `runTLO` is called to select the next TLO forwarding node.

Analyzing the mode of Listing 6, a starvation condition can be detected. Using this implementation of TLO scheme causes starvation and affects the reliability of the scheme in some executions. The steps of the scenario is depicted in Figure 8. In 8(a), the position of the vehicles is shown in the time of the accident between vehicles A and B. In the next step, vehicle B starts broadcasting the warning message and vehicles C and D receive the message as they are in the range of B (Figure 8(b)). Upon receiving the warning message, these vehicles execute the TLO algorithm and since they both have the same distance from B, they forward the received warning message and the vehicles E and F receive the warning message from these two vehicles. When vehicles E and F execute the TLO algorithm, racing between the following two scenarios happen.

1. **E broadcasts before F:** vehicles G and H receive the warning message from E. Upon execution of TLO algorithm by G and H, Vehicle H is selected as the TLO forwarding node and forwards the message. Meanwhile, vehicle G is waiting for receiving the warning message from H to make sure that the broadcasting has been successful. If in the waiting time of G, vehicle H forwards the warning message, the message will be interpreted as acknowledgement of the successful broadcast of H and although G is TLO node in this step, it will not forward the message. In this case, the vehicle J does not receive the warning message.
2. **F broadcasts before E:** vehicle G receive the warning message from F and after the execution of TLO algorithm, it forwards the message as the selected TLO node and vehicle J will receive the warning message in this scenario.

This example shows that concurrent execution of the algorithm in nodes causes nondeterministic behavior which may violate correctness properties of the application. To avoid such cases, all the possible nondeterministic behaviors have to be considered in any analysis framework. However, simulation-based techniques, fail to report a result by considering all the possible execution traces.



(a) Accident between A and B

(b) B broadcasts the warning message

(c) C and D are both selected as TLO nodes to forward the warning message

(d) Order of broadcasting between E and F results in two cases

Fig. 8 A scenario of TLO scheme which results in two execution alternatives that one of them causes starvation for vehicle J

Listing 6 TLO scheme in Timed Rebeca

```
1 env int RANGE = 9;
2 env int THRESHOLD_WAITING = 2;
3 env int MESSAGE_SEND_TIME = 1;
4 env int RIGHT = 0;
5 env int LEFT = 1;
6 env int UP = 2;
7 env int DOWN = 3;
8 abstract reactiveclass BroadcastingActor (5) {
9  statevars {
10   int id, x, y;
11  }
12  abstract msgsrv receive(int data);
13  void broadcast() { ... }
14  double distance(BroadcastingActor bActor ,
        BroadcastingActor cActor){ ... }
15  boolean isTLO () {
16   boolean isTLO = true;
17   ArrayList<ReactiveClass> all = getAllActors();
18   BroadcastingActor senderData;
19   for(int i = 0 ; i < all.size(); i++) {
20    BroadcastingActor rns =
          (BroadcastingActor)all.get(i);
21    BroadcastingActor sn = (BroadcastingActor)sender;
22    if (rns.id == sn.id)
23     senderData = rns;
24   }
25   double myDistance = distance (senderData , self);
26   for(int i = 0; i < allActors.size(); i++) {
27    BroadcastingActor ba =
          (BroadcastingActor)allActors.get(i);
28    double distance = distance (ba ,
          (BroadcastingActor)senderData);
29    if(distance < RANGE && distance > myDistance) {
30     isTLO = false;
31    }
32   }
33   return isTLO;
34  }
35 }
36
37 reactiveclass Vehicle extends BroadcastingActor(5){
38  statevars{
39   boolean isAV;
40   int direction, latency;
41   int destX, destY;
42   boolean isWaiting, received, isAware;
43  }
44
45  Vehicle (int vid, int X , int Y , int dir , int
        vLatency , int dX , int dY , boolean
        isAccidentVehicle) {
46   id = vid;
47   x = X;
48   y = Y;
49   direction = dir;
50   latency = vLatency;
51   destX = dX;
52   destY = dY;
53   isAV = isAccidentVehicle;
54   isWaiting = false;
55   received = false;
```

```
56   if (isAV) {
57    self.alertAccident();
58    isAware = true;
59    received = true;
60   } else
61    self.move() after(latency);
62  }
63  msgsrv alertAccident(){ broadcast(); }
64  msgsrv move() {
65   switch (direction) {
66    case 0: x++; break;
67    case 1: x--; break;
68    case 2: y++; break;
69    case 3: y--; break;
70   }
71   if (x != destX || y != destY)
72    self.move() after(latency);
73  }
74  msgsrv stop (){ stop(); }
75  msgsrv finishWait() {
76   if (isWaiting) runTLO();
77  }
78  msgsrv receive(int data) {
79   isAware = true;
80   if(!isWaiting)
81    runTLO();
82   else
83    isWaiting = false;
84  }
85  void runTLO() {
86   if (!received) {
87    if (isTLO()) {
88     broadcast();
89     received = true;
90    } else {
91     isWaiting = true;
92     self.finishWait() after (THRESHOLD_WAITING);
93    }
94   }
95  }
96 }
97
98 main {
99  Vehicle v1():(0,0,10,RIGHT,1,10,10,true);
100 Vehicle v2():(1,10,0,UP,2,10,10,false);
101 Vehicle v3():(2,-1,0,RIGHT,1,10,0,false);
102 Vehicle v4():(3,0,1,DOWN,2,0,-10,false);
103 Vehicle v5():(4,3,0,LEFT,1,-10,0,false);
104 Vehicle v6():(5,0,-7,UP,2,0,10,false);
105 }
```

7 Conclusion and Future Work

Lack of a framework for formal modeling and efficient verification of warning message dissemination schemes in VANETs is the main obstacle in using these schemes in real-world applications. In this paper, we presented VeriVANca, an actor-based framework, developed using Timed Rebeca for modeling warning message dissemination schemes in VANETs. Model of schemes developed in VeriVANca can be analyzed using Afra, the model checking tool of Timed Rebeca. We showed how warning message dissemination schemes can be modeled using VeriVANca by implementing two of these schemes. Scenarios in these schemes were explored to illustrate the effectiveness of the approach in checking correctness properties and performance evaluation of the schemes. We further explained how easily the model of a scheme can be transformed to present another scheme by making minor modifications. Providing this level of guarantee in correctness and performance of warning message dissemination schemes, enables engineers to benefit from these schemes in the development of smart cars.

Considering different members of Rebeca family modeling language, VeriVANca can be used for addressing other characteristics of schemes such as their probabilistic behavior. Since Afra supports different members of Rebeca family, models with these characteristics can be analyzed using Afra.

VeriVANca can be used for the analysis of scenarios with limited congested areas. However, to be able to use the framework for large-scale models containing congested areas, we are going to develop a partial order reduction technique. This reduction relies on the fact that reaction of a vehicle to received warning messages is independent of their sender; therefore, different orders of execution (interleaving) for messages received at the same time can be ignored without affecting the result of model checking.

Acknowledgements The work on this paper has been supported in part by the project "Self-Adaptive Actors: SEADA" (163205-051) of the Icelandic Research Fund and by DPAC Project (Dependable Platforms for Autonomous Systems and Control) at Mälardalen University, Sweden.

References

1. Luca Aceto, Matteo Cimini, Anna Ingólfsdóttir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. Modelling and simulation of asynchronous real-time systems using timed rebeca. In Mohammad Reza Mousavi and António Ravara, editors, *Proceedings 10th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2011, Aachen, Germany, 10th September, 2011*, volume 58 of *EPTCS*, pages 1–19, 2011.
2. Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings*, volume 206 of *Lecture Notes in Computer Science*, pages 19–41. Springer, 1985.
3. Aoxueluo, Weigang Wu, Jiannong Cao, and Michel Raynal. A generalized mutual exclusion problem and its algorithm. In *ICPP*, pages 300–309. IEEE Computer Society, 2013.

4. Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017.
5. Bruno Ferreira, Fernando A. F. Braz, Antonio A. F. Loureiro, and Sérgio V. A. Campos. A probabilistic model checking analysis of vehicular ad-hoc networks. In *IEEE 81st Vehicular Technology Conference, VTC Spring 2015, Glasgow, United Kingdom, 11-14 May, 2015*, pages 1–7. IEEE, 2015.
6. Óscar Gama, Maria João Nicolau, António Costa, Alexandre Santos, Joaquim Macedo, and Bruno Dias. Evaluation of message dissemination methods in vanets using a cooperative traffic efficiency application. In *IWCMC*, pages 478–483. IEEE, 2017.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
8. Mozhdeh Gholibeigi and Geert Heijenk. Analysis of multi-hop broadcast in vehicular ad hoc networks: A reliability perspective. In *Wireless Days*, pages 1–8. IEEE, 2016.
9. Michael R. Hafner, Drew Cunningham, Lorenzo Caminiti, and Domitilla Del Vecchio. Cooperative collision avoidance at intersections: Algorithms and experiments. *IEEE Trans. Intelligent Transportation Systems*, 14(3):1162–1175, 2013.
10. Ali Jafari, Ehsan Khamespanah, Marjan Sirjani, Holger Hermanns, and Matteo Cimini. Ptrebeca: Modeling and analysis of distributed and asynchronous systems. *Sci. Comput. Program.*, 128:22–50, 2016.
11. Iman Jahandideh, Fatemeh Ghassemi, and Marjan Sirjani. Hybrid rebeca: Modeling and analyzing of cyber-physical systems. In Roger D. Chamberlain, Walid Taha, and Martin Törngren, editors, *Cyber Physical Systems. Model-Based Design - 8th International Workshop, CyPhy 2018, and 14th International Workshop, WESE 2018, Turin, Italy, October 4-5, 2018, Revised Selected Papers*, volume 11615 of *Lecture Notes in Computer Science*, pages 3–27. Springer, 2018.
12. Ehsan Khamespanah. *Modeling, Verification, and Analysis of Timed Actor-Based Models*. PhD thesis, Computer Science, Menntavegi 1, 101 Reykjavík, 6 2018.
13. Ehsan Khamespanah, Ramtin Khosravi, and Marjan Sirjani. An efficient TCTL model checking algorithm and a reduction technique for verification of timed actor models. *Sci. Comput. Program.*, 153:1–29, 2018.
14. Ehsan Khamespanah, Marjan Sirjani, Mahesh Viswanathan, and Ramtin Khosravi. Floating time transition system: More efficient analysis of timed actors. In *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, pages 237–255, 2015.
15. Shou-pon Lin and Nicholas F. Maxemchuk. The fail-safe operation of collaborative driving systems. *J. Intellig. Transport. Systems*, 20(1):88–101, 2016.
16. Taqwa Saeed, Yiannos Mylonas, Andreas Pitsillides, Vicky Papadopoulou, and Marios Lestas. Modeling probabilistic flooding in vanets for optimal rebroadcast probabilities. *IEEE Trans. Intelligent Transportation Systems*, 20(2):556–570, 2019.
17. Julio A. Sanguesa, Manuel Fogue, Piedad Garrido, Francisco J. Martinez, Juan-Carlos Cano, Carlos Miguel Tavares Calafate, and Pietro Manzoni. On the selection of optimal broadcast schemes in vanets. In *MSWiM*, pages 411–418. ACM, 2013.
18. Julio A. Sanguesa, Manuel Fogue, Piedad Garrido, Francisco J. Martinez, Juan-Carlos Cano, Carlos Miguel Tavares Calafate, and Pietro Manzoni. RTAD: A real-time adaptive dissemination system for vanets. *Computer Communications*, 60:53–70, 2015.
19. Julio A. Sanguesa, Manuel Fogue, Piedad Garrido, Francisco J. Martinez, Juan-Carlos Cano, and Carlos T. Calafate. A survey and comparative study of broadcast warning message dissemination schemes for vanets. *Mobile Information Systems*, 2016:8714142:1–8714142:18, 2016.
20. Marjan Sirjani and Mohammad Mahdi Jaghoori. Ten years of analyzing actors: Rebeca experience. In *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 20–56. Springer, 2011.
21. Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S De Boer. Modeling and verification of reactive systems using rebeca. *Fundamenta Informaticae*, 63(4):385–410, 2004.
22. Kanitsom Suriyapaibonwattana and Chotipat Pomavalai. An Effective Safety Alert Broadcast Algorithm for VANET. In *2008 International Symposium on Communications and Information Technologies*, pages 247–250. IEEE, oct 2008.
23. Yu-Chee Tseng, Sze-Yao Ni, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. *Wireless Networks*, 8(2-3):153–167, 2002.
24. Behnaz Yousefi, Fatemeh Ghassemi, and Ramtin Khosravi. Modeling and efficient verification of broadcasting actors. In *Fundamentals of Software Engineering - 6th International Conference, FSEN 2015 Tehran, Iran, April 22-24, 2015, Revised Selected Papers*, pages 69–83, 2015.
25. Behnaz Yousefi, Fatemeh Ghassemi, and Ramtin Khosravi. Modeling and efficient verification of wireless ad hoc networks. *Formal Aspects of Computing*, 29(6):1051–1086, 2017.
26. Behnaz Yousefi, Fatemeh Ghassemi, and Ramtin Khosravi. Modeling and efficient verification of wireless ad hoc networks. *Formal Asp. Comput.*, 29(6):1051–1086, 2017.
27. Farnaz Yousefi, Ehsan Khamespanah, Mohammed Gharib, Marjan Sirjani, and Ali Movaghar. Verivanca: An actor-based framework for formal verification of warning message dissemination schemes in vanets. In Fabrizio Biondi, Thomas Given-Wilson, and Axel Legay, editors, *Model Checking Software - 26th International Symposium, SPIN 2019, Beijing, China, July 15-16, 2019, Proceedings*, volume 11636 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2019.
28. Sherali Zeadally, Ray Hunt, Yuh-Shyan Chen, Angela Irwin, and Aamir Hassan. Vehicular ad hoc networks (VANETS): status, results, and challenges. *Telecommunication Systems*, 50(4):217–241, 2012.