DOI 10.1007/s00165-017-0429-z
BCS © 2017
Formal Aspects of Computing (2017) 29: 1051–1086

Formal Aspects of Computing

Modeling and efficient verification of wireless ad hoc networks

Behnaz Yousefi[1], Fatemeh Ghassemi[1] and Ramtin Khosravi[1]
[1]School of Electrical and Computer Engineering, University of Tehran, N. Kargar Ave., Tehran, Iran

Abstract. Wireless ad hoc networks, in particular mobile ad hoc networks (MANETs), are growing very fast as they make communication easier and more available. However, their protocols tend to be difficult to design due to topology dependent behavior of wireless communication, and their distributed and adaptive operations to topology dynamism. Therefore, it is desirable to have them modeled and verified using formal methods. In this paper, we present an actor-based modeling language with the aim to model MANETs. We address main challenges of modeling wireless ad hoc networks such as local broadcast, underlying topology, and its changes, and discuss how they can be efficiently modeled at the semantic level to make their verification amenable. The new framework abstracts the data link layer services by providing asynchronous (local) broadcast and unicast communication, while message delivery is in order and is guaranteed for connected receivers. We illustrate the applicability of our framework through two routing protocols, namely flooding and AODVv2-11, and show how efficiently their state spaces can be reduced by the proposed techniques. Furthermore, we demonstrate a loop formation scenario in AODV, found by our analysis tool.

Keywords: State-space reduction; Mobile ad hoc network; Ad hoc routing protocol; Rebeca; Actor-based language; Model checking

1. Introduction

Applicability of wireless communications is rapidly growing from home networks to satellite transmissions due to their high accessibility and low cost. Wireless communication has a broadcasting nature, as messages sent by each node can be received by all nodes in its transmission range, called *local broadcast*. Therefore, by paying the cost of one transmission, several nodes may receive the message, which leads to lower energy consumption for the sender and throughput improvement [CCH07].

Mobile ad hoc networks (MANETs) consist of several portable hosts with no pre-existing infrastructure, such as routers in wired networks or access points in managed (infrastructure) wireless networks. In such networks, nodes can freely change their locations so the network topology is constantly changing. For unicasting a message to a specific node beyond the transmission range of a node, it is needed to relay the message by some intermediate nodes to reach the desired destination. Due to lack of any pre-designed infrastructure and global network topology information, network functions such as routing protocols are devised in a completely distributed manner and adaptive to topology changes. Topology dependent behavior of wireless communication, distribution and adaptation requirements make the design of MANET protocols complicated and more in need of modeling and

Correspondence and offprint requests to: F. Ghassemi, E-mail: fghassemi@ut.ac.ir
Electronic supplementary material The online version of this article (doi:10.1007/s00165-017-0429-z) contains supplementary material, which is available to authorized users.

verification so that it can be trusted. For instance, MANET protocols like the Ad hoc On Demand Distance Vector (AODV) routing protocol [PB99] has been evolved as new failure scenarios were experienced or errors were found in the protocol design [BOG02, NT15b, FVGH$^+$13].

The actor model [Agh90, Hew77] has been introduced for the purpose of modeling concurrent and distributed applications. It is an agent-based language introduced by Hewitt [Hew77], extended by Agha to an object-based concurrent computation model [Agh90]. An actor model consists of a set of actors communicating with each other through unicasting asynchronous messages. Each computation unit, modeled by an actor, has a unique address and mailbox. Messages sent to an actor are stored in its mailbox. Each actor is defined through a set of message handlers, called *message servers*, to specify the actor behavior upon processing of each message. In this model, message delivery is guaranteed but is not in-order. This policy implicitly abstracts away from effects of the network, i.e., delays over different routing paths, message conflicts, etc., and consequently makes it a suitable modeling framework for concurrent and distributed applications. Rebeca [SMSdB04] is an actor-based modeling language which aims to bridge the gap between formal verification techniques and the real-world software engineering of concurrent and distributed applications. It provides an operational interpretation of the actor model through a Java-like syntax, which makes it easy to learn and use. Rebeca is supported by a robust model checking tool, named Afra [afra], which takes advantage of various reduction techniques [JSM$^+$10, SS10] to make efficient verification possible. With the aim of reducing the state space, computations, i.e., executions of message servers in actors, are assumed to be instantaneous while message delivery is in-order. Consequently, instructions of message servers are not interleaved and hence, execution of message servers becomes atomic in semantic model and each actor mailbox is modeled through a FIFO queue.

In [YGK15] we introduced bRebeca as an extension to Rebeca, to support broadcast communication which abstracts the *global broadcast communications* [BG92]. To abstract the effect of network, the order of receipts for two consequent broadcast communications is not necessarily the same as their corresponding sends in an actor model. Hence, each actor mailbox was modeled by a bag. The resulting framework is suitable for modeling and efficient verification of broadcasting protocols above the network layer, but not appropriate for modeling MANETs in two ways: firstly the topology is not defined, and every actor (node) can receive all messages, in other words all nodes are connected to each other. Secondly, as there is no topology defined, mobility is not considered.

In this paper, we extend the actor-based modeling language bRebeca [YGK15] to address local broadcast, topology, and its changes. The aim of the current paper is to provide a framework to detect malfunctions of a MANET protocol caused by conceptual mistakes in the protocol design, rather than by an unreliable communication. Therefore, the new framework abstracts away from the data link layer services by providing asynchronous reliable local broadcast, multicast, and unicast communications [Pen08, SL04]. Since only one-hop communications are considered, the message delivery is in-order and is guaranteed for connected receivers. Consequently, each actor mailbox is modeled through a queue. The reliable communication services of the data link layer provide feedback (to its upper layer applications) in case of (un)successful delivery. Therefore, our framework provides *conditional unicast* to model protocol behaviors in each scenario (in the semantic model, the status of the underlying topology defines the behavior of actors).

The resulting framework provides a suitable means to model the behavior of ad hoc networks in a compositional way without the need to consider asynchronous communications handled by message storages in the computation model. However, to minimize the effect of message storages on the growth of the state space, we exploit techniques to reduce it. Since nodes can communicate through broadcast and a limited form of multicast/unicast, it is possible to consider actors that have the same neighbors and local states as identical according to the counter abstraction technique [BMWK09, PXZ02, ET99]. Therefore, the states whose number of actors (irrespective of their identifiers) with the same neighbors and local state are the same for each local state value, will be aggregated, thus the state space is reduced considerably. The reduced semantics is strongly bisimilar to the original one.

To examine resistance and adaptation of MANET protocols to changes of the underlying topology, we address mobility via arbitrary changes of the topology at the semantic level. Since network protocols have no control over movement of MANET nodes and mobility is an intrinsic characteristic of such nodes, the topology should be implicitly manipulated at the semantics. In other words, with the aim of verifying behaviors of MANET protocols for any mobility scenario, the underlying topology is arbitrarily changed at each semantic state. We provide mechanisms to restrict this random changes in the topology through specifying constraints over the topology. However, these random changes make the state space grow exponentially while the proposed counter abstraction technique becomes invalid. To this end, each state is instead explored for each possible topology and meanwhile topology information is removed from the state. Therefore, two next states only different in their topologies are consolidated together and hence, the state space is reduced considerably. Due to arbitrary changes of the underlying topology, states with different topologies are reachable from each other (through τ-transitions denoting topology changes). We establish that such states are branching bisimilar, and consequently a set of properties such as ACTL-X [DV90] are preserved. The proposed reduction techniques makes our framework scalable to verify some important properties of MANET protocol, e.g., loop freedom, in the presence of mobility in a unified model (cf. generating a model for each mobility scenario).

The contributions of this paper can be summarized as follows:

- We extend the computation model of the actor model, in particular Rebeca, with the concepts of MANETs, i.e., asynchronous reliable local broadcast/multicast/unicast, topology, and topology changes;
- We apply the counter abstraction in presence of topology as a part of semantic states to reduce state space substantially: actors with the same neighbors, i.e., topological situations, and local states are counted together in the counter abstraction technique;
- We show that the soundness of the counter abstraction technique is not preserved in presence of mobility, and propose another technique to reduce the state space.
- We provide a tool that supports both reduction techniques and examines invariant properties automatically. We illustrate the scalability of our approach through the specification and verification of two MANET protocols, namely flooding and AODV.
- We present a complete and accurate model of the core functionalities of a recent version of AODVv2 protocol (version 11), abstracting from its timing issues, and investigate its loop freedom property. We detect scenarios over which the property is violated due to maintaining multiple unconfirmed next hops for a route without checking them to be loop free. We have communicated this scenario to the AODV group and they have confirmed that it can occur in practice. In response, their route information evaluation was modified, published in version 13 of the draft.[1] Furthermore, we verify the monotonic increase of sequence numbers and packet delivery properties using existing model checkers.

Our framework can also be applied to Wireless Mesh Networks (WMNs). Unlike MANETs, WSNs have a backbone of dedicated mesh routers along with mesh clients. Hence, they provide flexibility in terms of mobility: in contrast to MANETs, the clients mobility has limited effect on the overall network configuration, as the mesh routers are fixed [MKKAR06].

The paper is structured as follows. Section 2 briefly introduces bRebeca, explain the idea behind the counter abstraction technique and its relation to symmetry reduction technique, and explains equivalence relations that validate our reduction techniques. Section 3 addresses the main modeling challenges of wireless networks. Section 4 presents our extension to bRebeca for modeling MANETs. In Sect. 5, we generate the state space compactly with the aim of efficient model checking. To illustrate the applicability of our approach, we specify the core functionalities of AODVv2-11 in Sect. 6. Then, in Sect. 7, we discuss the efficiency of our state-space generation over two case studies: the AODV and the flooding-based routing protocol. We illustrate our tool and possible analysis over the models through a verification of AODV. Finally, we review some related work in Sect. 8 before concluding the paper.

2. Preliminaries

2.1. bRebeca

Rebeca [SMSdB04] is an actor-based modeling language proposed for modeling and verification of concurrent and distributed systems. It has a Java-like syntax familiar to software developers and it is also supported by a tool via an integrated modeling and verification environment [afra]. Due to its design principle it is possible to extend the core language based on the desired domain [SJ11].

[1] https://tools.ietf.org/html/draft-ietf-manet-aodvv2-13.

```
reactiveclass MNode
{
    statevars
    {
        int my_i;
        boolean done;
    }

    msgsrv initial(int j, boolean starter)
    {
        my_i = j;
        if(starter) {
            done = true;
            send(my_i);
        } else
            done = false;
    }

    msgsrv send(int i)
    {
        if (i < my_i) {
            if (!done) {
                done = true;
                send(my_i);
            }
        } else {
            my_i = i;
            done = true;
        }
    }
}
main
{
        MNode n1(1,false);
        MNode n2(2,false);
        MNode n3(3,true);
        MNode n4(4,false);
}
```

Fig. 1. An example in bRebeca: max-algorithm with 4 nodes

For example, different extensions have been introduced in various domains such as probabilistic systems [VK12], real-time systems [RSA$^+$14], software product lines [SK13], and broadcasting environment [YGK15]. As in this paper we intend to extend bRebeca, we briefly review its syntax and semantics.

In bRebeca as well as in Rebeca, actors are the computation units of the system, called rebecs (short for reactive objects), which are instances of the defined *reactive classes* in the model.

Rebecs communicate with each other only through broadcasting message which is asynchronous. Every sent message eventually will be received and processed by its *potential* receivers. In Rebeca, the rebecs defined as the *known rebecs* of a sender, the sender itself using the "self" keyword, or the sender of the message currently processed using the keyword "sender" are considered as the potential receivers. However, in bRebeca, it is assumed the network is fully connected and therefore, all rebecs of a model constitute the potential receivers. In other words, a broadcast message is received by all the nodes to which a sender has a (one-hop/multi-hop) path. So, unlike Rebeca, there is no need for declaring the known rebecs in the reactive class definition. Due to unpredictability of multi-hop communications, the arrival order of messages must be considered arbitrary. Therefore, as the second difference with Rebeca, received messages are stored in an unordered *bag* in each node.

Every reactive class has two major parts, first the *state variables* to maintain the state of the rebec, and second the *message servers* to indicate the reactions of the rebec on received messages. The local state of a rebec is defined in terms of its state variables together with its message bag. Whenever a rebec receives a message which has no corresponding message server to respond to, it simply discards the message. Each rebec has at least one message server called "initial", which acts like a constructor in object-oriented languages and performs the initialization tasks.

A rebec is said to be *enabled* if and only if it has at least one message in its bag. The computation takes place by removing a message from the bag and executing its corresponding message server atomically, after which the rebec proceeds to process the other messages in its bag (if any). Processing a message may have the following consequences:

- it may modify the value of the state variables of the executing rebec, or
- some messages may be broadcast to other rebecs.

Each bRebeca model consists of two parts, the *reactive classes* part and the *main* part. In the *main* part the instances of the reactive classes are created initially while their local variables are initialized.

As an example, Fig. 1 illustrates a simple max finding algorithm modeled in bRebeca, referred to as "Max-Algorithm" [DK86]. Every node in a network contains an integer value and they intend to find the maximum value of all nodes in a distributed manner. The `initial` message server has a parameter, named `starter`. The rebec with the `starter` value *true* initiates the algorithm by broadcasting the first message. Whenever a node receives a value from others, it compares this value with its current value and one of the following scenarios happens:

- if it has not broadcast its value yet and its value is greater than the received one, it broadcasts its value to others;
- if its current value is less than the received one, it gives up broadcasting its value and updates its current value to the received one;
- if it has already sent its value, it only checks whether it must updates its value.

This protocol does not work on MANETs as nodes give up to rebroadcast their value after their first broadcast. The Max-Algorithm should find the maximum value among the connected nodes in MANETs. To this aim, if a node moves and connects to new nodes, it has to re-send its value as its value may be the maximum value in the currently connected nodes.

2.2. Counter abstraction

Since model checking is the main approach of verification in Rebeca, we need to overcome state-space explosion, where the state space of a system grows exponentially as the number of components in the system increases. One way to tackle this well-known problem is through applying reduction techniques such as symmetry reduction [CEJS98] and counter abstraction [BMWK09, PXZ02, ET99]. Counter abstraction is indeed a form of symmetry reduction and, in case of full symmetry, it can be used to avoid the *constructive orbit problem*, according to which finding a unique representative of each state is NP-hard [CEJS98]. The idea of using counters and counter abstraction in model checking was first introduced in [ET99]. However, the term of *counter abstraction* was first presented in [PXZ02] for the verification of parameterized systems and further used in different studies such as [BMWK09, Kat11].

The idea of counter abstraction is to record the global state of a system as a vector of counters, one per local state. Each counter denotes the number of components currently residing in the corresponding local state. In our work, by"components" we mean the actors of the system. This technique turns a model with an exponential size in n, i.e. m^n, into one of a size polynomial in n, i.e. $\binom{n+m-1}{m}$, where n and m denote the number of components and local states, respectively. Two global states S and S' are considered identical up to permutation if for every local state s, the number of components residing in s is the same in the two states S and S', as permutation only changes the order of elements. For example, consider a system which consists of three components that each have only one variable v_i of boolean type. Three global states $(true, true, false)$, $(false, true, true)$, and $(true, false, true)$ are equivalent and can be abstracted into one global state represented as $(true : 2, false : 1)$.

2.3. Semantic equivalence

Strong bisimilarity [Plo81] is used as a verification tool to validate the counting abstraction reduction technique on labeled transition systems. A labeled transition system (LTS), is defined by the quadruple $\langle S, \rightarrow, L, s_0\rangle$ where S is a set of states, $\rightarrow \subseteq S \times L \times S$ a set of transitions, L a set of labels, and s_0 the initial state. Let $s \xrightarrow{\alpha} t$ denote $(s, \alpha, t) \in \rightarrow$.

Definition 2.1 (*Strong bisimilarity*) A binary relation $\mathcal{R} \subseteq S \times S$ is called a strong bisimilation if and only if, for any s_1, s_1', s_2, and s_2' and $\alpha \in L$, the following transfer conditions hold:

- $s_1 \mathcal{R}\, s_2 \wedge s_1 \xrightarrow{\alpha} s_1' \Rightarrow (\exists\, s_2' \in S : s_2 \xrightarrow{\alpha} s_2' \wedge s_1' \mathcal{R}\, s_2')$,
- $s_1 \mathcal{R}\, s_2 \wedge s_2 \xrightarrow{\alpha} s_2' \Rightarrow (\exists\, s_1' \in S : s_1 \xrightarrow{\alpha} s_1' \wedge s_1' \mathcal{R}\, s_2')$.

Two states s and t are called strong bisimilar, denoted by $s \sim t$, if and only if there exists a strong bisimulation relating s and t.

As explained in Sect. 1, mobility is addressed through random changes of underlying topology at each semantic state, modeled by τ-transitions. We propose to remove such transitions while the behavior of each semantic state is explored for all possible topologies. We exploit branching bisimilarity [vGW96] to establish the reduced semantic is branching bisimilar to the original one. Let $\xrightarrow{\tau}^{*}$ be reflexive and transitive closure of τ-transitions:

- $t \xrightarrow{\tau}^{*} t$;
- $t \xrightarrow{\tau}^{*} s$, and $s \xrightarrow{\tau} r$, then $t \xrightarrow{\tau}^{*} r$.

Definition 2.2 (*Branching bisimilarity*) A binary relation $\mathcal{R} \subseteq S \times S$ is called a branching bisimilation if and only if, for any $s_1,\ s_1',\ s_2$, and s_2' and $\alpha \in L$, the following transfer conditions hold:

- $s_1 \mathcal{R} s_2 \wedge s_1 \xrightarrow{\alpha} s_1' \Rightarrow ((\alpha = \tau \wedge s_1' \mathcal{R} s_2) \vee (\exists s_2', s_2'' \in S : s_2 \xrightarrow{\tau}^{*} s_2'' \xrightarrow{\alpha} s_2' \wedge s_1 \mathcal{R} s_2'' \wedge s_1' \mathcal{R} s_2'))$,
- $s_1 \mathcal{R} s_2 \wedge s_2 \xrightarrow{\alpha} s_2' \Rightarrow ((\alpha = \tau \wedge s_1 \mathcal{R} s_2') \vee (\exists s_1', s_1'' \in S : s_1 \xrightarrow{\tau}^{*} s_1'' \xrightarrow{\alpha} s_1' \wedge s_1'' \mathcal{R} s_2 \wedge s_1' \mathcal{R} s_2'))$.

Two states s and t are called branching bisimilar, denoted by $s \simeq_{br} t$, if and only if there exists a branching bisimulation relating s and t.

3. Modeling topology and mobility

In this section, we discuss issues brought up by extending bRebeca to model and verify MANETs, and our solutions to overcome these challenges. We assume that the number of nodes is fixed (to make the state space finite as explained in [DSZ11]).

3.1. Network topology and mobility

Every rebec represents a node in the MANET model. A node can communicate only with those located in its communication range, so-called *connected*. bRebeca does not define a "topology" concept as the network graph is considered to be connected, all nodes are globally connected.

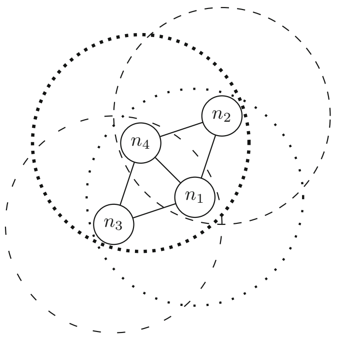
Mobility is the intrinsic characteristic of MANET nodes. Furthermore, network protocols have no control over the movement of MANET nodes, and hence, topology changes cannot be specified as a part of the specification. Additionally, to verify a protocol with respect to any mobility scenario, we need to consider all possible topology changes while constructing the state space. To this end, we consider the topology as a part of the states and randomly change the underlying topology at the semantic level. To this aim, a topology is modeled as an $n \times n$ matrix in each (global) state of the semantic model, where n is the number of nodes in the network. Each element of this matrix, denoted by $e_{i,j}$, indicates whether $node_i$ is *connected* to $node_j$ ($e_{i,j} = 1$) or not ($e_{i,j} = 0$). As the communication ranges of all nodes are assumed to be equal, connectivity is a bidirectional concept, and hence, the resulting matrix will be symmetric. The main diagonal elements are always 1 to make it possible for nodes to unicast messages to themselves. (However, in the case of broadcast, our semantic rules prevent a node from receiving its own message, see Sect. 4.2). Changing the topology is considered an unobservable action, modeled by a τ transition, which alters the topology matrix. Hence, each τ-transition represents a set of (bidirectional) link setups/breakdowns in the underlying topology.

To set up the initial topology of the network, the *known-rebecs* definitions, provided by the Rebeca language, is extended to address the connectivity of rebecs. Fig. 2a shows the communication range of the nodes in a simple network. To configure the initial topology of this network, *known-rebecs* of each rebec should be defined as shown in Fig. 2b during its instantiation (cf. Fig. 1). The corresponding semantic representation (as a part of the initial state) is shown in Fig. 2c.

The connectivity matrix has $n \times n$ elements which can be either 0 or 1, and since on the main diagonal we will exclusively have 1s, we have $2^{((n \times n)-n)/2}$ possible topologies. For example, in a network of 4 nodes, we have $2^{(16-4)/2} = 2^6$ possible topologies. Considering all these topologies may lead to a state-space explosion. Hence, we provide a mechanism to limit the possible topologies by applying some *network constraints* to characterize the set of topologies in terms of (dis)connectivity relations to (un)pin a set of the links among the nodes. We use the notations $con(i,j)$ or $!con(i,j)$ to show that two nodes i and j are connected or disconnected, respectively, and $and(\mathcal{C}_1, \mathcal{C}_1)$ to denote both $\mathcal{C}_1$ and $\mathcal{C}_2$ hold. For example, $!con(n_1, n_2)$ specifies that n_1 (n_2) never gets connected to n_2 (n_1), in other words, n_1 never enters into n_2's communication range, and vice versa. Therefore a topology γ is called *valid* for the network constraint $\mathcal{C}$, denoted as $\gamma \vDash \mathcal{C}$, if:

$$\gamma \vDash con(i,j) \Leftrightarrow \gamma_{i,j} = 1 \qquad \gamma \vDash and(\mathcal{C}_1, \mathcal{C}_2) \Leftrightarrow \gamma \vDash \mathcal{C}_1 \wedge \gamma \vDash \mathcal{C}_2$$
$$\gamma \vDash !con(i,j) \Leftrightarrow \gamma_{i,j} = 0 \qquad \gamma \vDash true$$

where $\gamma_{i,j}$ represents the element $e_{i,j}$ of the corresponding semantic model of γ, and $true$ characterizes all possible topologies.



(a)

$\mathsf{MNode}\ n_1\ (n_2, n_3, n_4) : (1, \mathit{false})$
$\mathsf{MNode}\ n_2\ (n_1, n_4) : (2, \mathit{false})$
$\mathsf{MNode}\ n_3\ (n_1, n_4) : (3, \mathit{true})$
$\mathsf{MNode}\ n_4\ (n_2, n_3, n_1) : (4, \mathit{false})$

(b)

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

(c)

Fig. 2. A sample of an initial topology and its corresponding syntactic and semantic representations. **a** The network, **b** syntactic definition during instantiation, **c** semantic representation

If the only valid topology of a network constraint is equal to the initial topology, then the underlying topology will be static. This case can be useful for modeling WMNs with stable mesh routers with no mesh clients.

3.2. Restricted delivery guarantee

The nature of communications in the wireless networks is based on broadcast. The aim of the current paper is to provide a framework to detect malfunctions of a MANET protocol caused by conceptual mistakes in the protocol design, rather than by an unreliable communication. Therefore, we consider the wireless communications in our framework, namely local broadcast, multicast, and unicast, to be asynchronous and reliable in order to abstract the data link layer services. In this way, we abstract the issues related to contention management and collision detection following the approach of [KLN11]. This work abstracts the services of data link layer[2] with the aim to design/analyze MANET protocols irrespective to the network radio model that implements them (its effect is captured by three delays functions). It provides reliable local broadcast communication, with timing guarantees on the worst-case amount of time for a message to be delivered to all its recipients, total amount of time the sender receives its acknowledgment, and the amount of time for a receiver to receive some message among those currently being transmitted by its neighbors, expressed by *delay functions*. Therefore, our approach to specify protocols relying on the abstract data link layer simplifies the study of such protocols, and is valid as its real implementation with such reliable services exists [Pen08, SL04]. In these implementations, a node can broadcast/multicast/unicast a message successfully only to the nodes within its communication range. Therefore, message delivery is *guaranteed* for the connected nodes to the sender. In the case of unicast, if the sender is located in the receiver communication range, it will be notified, otherwise it assumes that the transmission was unsuccessful so it can react appropriately. Therefore, we extended bRebeca with *conditional unicast* so that it enables the model to react accordingly based on the status of underlying topology (which defines the delivery status in reliable communications).

Since we only consider one-hop communications (in contrast to the broadcast in bRebeca), the assumption about the unpredictability of multi-hop communications (with different delays) is not valid anymore, and message storages in wRebeca are modeled by queues instead of bags.

[2] Data link layer [the second layer of Open Systems Interconnection (OSI) model] is responsible for transferring data across the physical link. It consists of two sublayers: Logical Link Control sublayer (LLC) and Media Access Control sublayer (MAC). LLC is mainly responsible for multiplexing packets to their protocol stacks identified by their IP addresses, while MAC manages accesses to the shared media.

$$
\begin{aligned}
\text{Model} &::= \text{ReactiveClass}^{+}\ \text{Main}\\
\text{Main} &::= \mathsf{main}\ \{\text{RebecDecl}^{+}\ \text{ConstraintPart}\ \}\\
\text{List(X)} &::= \langle X, \rangle^{*} X \mid \epsilon\\
\text{RebecDecl} &::= C\ R\ (\text{List}(R))\ :\ (\text{List}(V));\\
\text{ConstraintPart} &::= \mathsf{constraint}\ \{\text{Constraint}\}\\
\text{Constraint} &::= \text{ConstrainDec} \mid !\ \text{ConstrainDec} \mid \mathsf{and}(\text{Constraint}\ ,\ \text{Constraint})\\
\text{ConstrainDec} &::= \mathsf{con}(R\ ,\ R) \mid\ \mathsf{true}\\
\text{ReactiveClass} &::= \mathsf{reactiveclass}\ C\ \{\ \text{StateVars MsgServer}^{*}\ \}\\
\text{StateVars} &::= \mathsf{statevars}\ \{\ \text{VarDecl}^{*}\ \}\\
\text{MsgServer} &::= \mathsf{msgsrv}\ M(\text{List}(T\ V))\ \{\ \text{Statement}^{*}\ \}\\
\text{VarDecl} &::= T\ V;\\
\text{Statement} &::= \text{VarDecl} \mid \text{Assign} \mid \text{Conditional} \mid \text{Loop} \mid \text{Broadcast} \mid \text{Multicast} \mid \text{Unicast} \mid \mathsf{break};\\
\text{Assign} &::= V = Expr;\\
\text{Conditional} &::= \mathsf{if}\ (Expr)\ \text{Block}\ \mathsf{else}\ \text{Block}\\
\text{Block} &::= \text{Statement} \mid \{\ \text{Statement}^{*}\ \}\\
\text{Loop} &::= \mathsf{while}(Expr)\ \text{Block}\\
\text{Broadcast} &::= M(\text{List}(Expr));\\
\text{Multicast} &::= \mathsf{multicast}\ (\ V\ , M(\text{List}(Expr)));\\
\text{Unicast} &::= \mathsf{unicast}\ (\ \text{Rec}\ , M(\text{List}(Expr)))\ \mathsf{succ} : \text{Block}\ \mathsf{unsucc} : \text{Block}\\
\text{Rec} &::= \mathsf{self} \mid V
\end{aligned}
$$

Fig. 3. wRebeca language syntax: *angle brackets* ($\langle\ \rangle$) are used as metaparentheses. *Superscript* * indicates zero or more times repetition. The *symbols* C, R, T, M, and V denote the set of classes, rebec names, types, method and variable names, respectively. The *symbol* $Expr$ denotes an expression, which can be an arithmetic or a boolean expression

4. wRebeca: syntax and semantics

In this section, we extend the syntax of bRebeca, introduced in Sect. 2.1, with conditional unicast and multicast, topology constraint, and known rebecs to set up the initial topology. Next, we provide the semantics of wRebeca models in terms of LTSs.

4.1. Syntax

The grammar of wRebeca is presented in Fig. 3. It consists of two major parts: reactive classes and main part. The definition of reactive classes is almost similar to the one in bRebeca. However, the main part is augmented with the ConstraintPart, where constraints are introduced to reduce all possible topologies in the network. The instances of the declared reactive classes are defined in the main part, before the ConstraintPart, by indicating the name of a reactive class and an arbitrary rebec name along with two sets of parentheses divided by the character :. The first couple of parentheses is used to define the neighbors of the rebec in the initial topology. The second couple of parentheses is used to pass values to the initial message server. Rebecs here communicate through broadcast, multicast, and unicast. In the broadcast statement, we simply use the message server name along with its parameters without specifying the receivers of a message. In contrast, when unicasting/multicasting a message, we also need to specify the receiver/receivers of the message. However, there is no delivery guarantee, depending on the location of the receiver. In case of unicasting, the sender can react based on the delivery status. Let unicast(Rec, $M(\text{List}(Expr))$) indicate unicast(Rec, $M(\text{List}(Expr))$) succ :{} unsucc :{} when the delivery status has no effect on the rebec behavior.

```
reactiveclass Node
{
    statevars
    {
        int IP;
    }

    msgsrv initial (boolean source,int ip_)
    {
        IP=ip_;
        if(source==true)
            relay_packet (55,0,3) ;
    }

    msgsrv relay_packet(int data,int hopNum,int
      destination)
    {
        if(IP==destination)
            unicast( self , deliver_packet (data));
        else if(hopNum<3)
        {
            hopNum++;
            relay_packet(data,hopNum,destination);
        }
    }
    msgsrv deliver_packet(int data)
    {

    }
}

main
{
    Node node0 (node1):(true,0);
    Node node1 (node0,node2,node3):(false,1);
    Node node2 (node1,node3):(false,2);
    Node node3 (node1,node2):(false,3);

    constraint
    {
        and(con(node0,node1),!con(node0,node2))
    }
}
```

Fig. 4. Flooding protocol in a network consisting of four nodes

In addition to communication statements, there are assignment, conditional, and loop statements. The first one is used to assign a value to a variable. The second is used to branch based on the evaluation of an expression: if the expression evaluates to *true*, then the if part, and otherwise the else part will be executed. Let if ($Expr$) Block denote if ($Expr$) Block else { }. Finally, the third is used to execute a set of statements iteratively as long as the loop condition, i.e., the boolean expression $Expr$, holds. Furthermore, break can be used to terminate its nearest enclosing loop statement and transfer the control to the next statement. For the sake of readability, we use for ($T\ x = Expr_1$; $Expr_2$; $Expr_3$){ Statement* } to denote $T\ x = Expr_1$; while ($Expr_2$){ Statement* $Expr_3$ }. A variable can be defined in the scope of message servers as a statement similar to programming languages.

A given wRebeca model is called *well-formed* if no state variable is redefined in the scope of a message server, no two state variables, message servers or rebec classes have identical names, identifiers of variables, message servers and classes do not clash, and all rebec instance accesses, message communications and variable accesses occur over declared/specified ones and the number and type of actual parameters correctly match the formal ones in their corresponding message server specifications. Each break should occur within a loop statement. Furthermore, the initial topology should satisfy the network constraint and be symmetric, i.e., if n_1 is the known rebec of n_2, then n_2 should be the known rebec of n_1. By default, the network constraint is true if no network constraint is defined, and all the nodes are disconnected if no initial topology is defined.

Example The flooding protocol is one of the earliest methods used for routing in wireless networks. The flooding protocol modeled in wRebeca is presented in Fig. 4. Every node upon receiving a packet checks whether it is the packet's destination. If so it processes the message, otherwise it broadcasts the message to its neighbors. To reduce the number of transferred messages, each message contains a counter, called `hopNum`, which shows how many times it has been re-broadcast. If the `hopNum` is more than the specified bound, it quits re-broadcasting.

4.2. Semantics

The formal semantics of a well-formed wRebeca is expressed as an LTS. In the following, we formally define the states, transitions, and initial states of the semantic model generated for a given wRebeca specification. To this aim, the given specification is decomposed into its constituent components, i.e., rebec instances, reactive classes, initial topology, and network constraint represented by the wRebeca model $\mathcal{M}$. The topology is implicitly changed as long as the given network constraint is satisfied. As explained in Sect. 1, message server executions are atomic and their statements are not interleaved. Intuitively, the global state of a wRebeca model is defined by the local states of its rebecs and the underlying topology. Consequently, a state transition occurs either upon atomic execution of a message server (i.e., when a rebec processes its corresponding message in its queue), or at a random change in the topology (modeled through unobservable τ-transitions).

Let V denote the set of variables ranged over by x, and Val denote the set of all possible values for the variables, ranged over by e. Furthermore, we assume that the set of types T consists of the integer and boolean data types, i.e., $T = \{int, bool\}$. We consider the default value $0 \in Val$ for the integer and boolean variables since the boolean values $true$ and $false$ can be modeled by 1 and 0 in the semantics, respectively. The variable assignment in each scope can be modeled by the valuation function $V \rightarrow Val$ ranged over by θ. An assignment can be extended by writing $\theta \cup \{y \mapsto e\}$. To monitor value assignments regarding scope management, we specify the set of all environments as $Env = Stack(V \rightarrow Val)$, ranged over by υ. Let $upd(\upsilon, \{y \mapsto e\})$ extend the variable assignments of the current scope, i.e, the top of the stack, by $\{y \mapsto e\}$ if the stack is not empty. Assume $Stack()$ denotes an empty environment. By entering into a scope, the environment υ is updated by $push(\theta, \upsilon)$ where θ is empty if the scope belongs to a block (which will be extended by the declarations in the block). Upon exiting from the scope, it is updated by $pop(\upsilon)$ which removes the top of the stack. Let $eval(expr, \upsilon)$ denote the value of the expression $expr$ in the context of environment υ, and $\upsilon[x := e]$ the environment identical to υ except that x is assigned to e.

Assume $Seq(D)$ denotes the set of all sequences of elements in D; we use notations $\langle d_1 \ldots d_n \rangle$ and ϵ for a non-empty and empty sequence, respectively. Note that the elements in a sequence may be repeated. A FIFO queue of elements of D can be viewed as a $Seq(D)$. For instance, $\langle 2\ 3\ 2\ 4 \rangle \in Seq(\mathbb{N})$ denotes a FIFO queue of natural numbers where its head is 2. For a given FIFO queue $f : Seq(D)$, assume $f \rhd d$ denotes the sequence obtained by appending d to the end of f, while $d \rhd f$ denotes the sequence with head d and tail f.

A wRebeca model is defined through a set of reactive classes, rebec instances, an initial topology, and a network constraint. Let C denote the set of all reactive classes in the model ranged over by c, R the set of rebec instances ranged over by r, and $\mathbb{C}$ the set of network constraints ranged over by $\mathcal{C}$. Assume Γ is the set of all possible topologies ranged over by γ. Each reactive class c is described by a tuple $c = \langle V_c, M_c \rangle$, where V_c is the set of class state variables and M_c the set of message types ranged over by m that its instances can respond to. We assume that for each class c, we have the state variable $self \in V_c$, and $c \in M_c$ which can be seen as its constructor in object-oriented languages. For the sake of simplicity, we assume that messages are parameterized with one argument, so Msg_c, where $M_c = Val \rightarrow Msg_c$ defines the set of all messages that rebec instances of the reactive class c can respond to. The formal parameter of a message can be accessed by $fm : M_c \rightarrow V$. Let $Statement$ denote the set of statements ranged over by σ, δ (we use σ^*, δ^* to denote a sequence of statements), and $body : M_c \rightarrow Seq(Statement)$ specify the sequence of statements executed by a message server. A block, denoted by β, is either defined by a statement or a sequence of statements surrounded by braces.

A rebec instance r is specified by the tuple $\langle c, e_0 \rangle$ where $c \in C$ is its reactive class, and e_0 defines the value passed to the message c which is initially put in the rebec's queue. We assume a unique identifier is assigned to each rebec instance. Let $I = \{1 \ldots n\}$ denote a finite set of all rebec identifiers ranged over by i and j. Furthermore, we use r_i to denote the rebec instance r with the assigned identifier i. As explained in Sect. 2.1, a rebec in wRebeca, like Rebeca, holds its received messages in a FIFO queue (unlike bRebeca, in which messages are maintained in a bag).

All rebecs of the model form a closed model, denoted by $\mathcal{M} = \langle \|_{i \in I} r_i, C, \gamma_0, \mathcal{C} \rangle$, where $r_i = \langle c, e_0^i \rangle$ for some $c \in C$ and $\mathcal{C} \in \mathbb{C}$. By default, $\mathcal{C} = true$ and $\forall\, i, j \leq n(\gamma_{0_{i,i}} = 1 \wedge (i \neq j \Rightarrow \gamma_{0_{i,j}} = 0))$ if no network constraint and initial topology were defined. The (global) state of the $\mathcal{M}$ is defined in terms of rebec's local states and the underlying topology.

Definition 4.1 The semantics of a wRebeca model $\mathcal{M} = \langle \|_{i \in I} r_i, C, \gamma_0, \mathcal{C} \rangle$ is expressed by the LTS $\langle S, L, \rightarrow, s_0 \rangle$ where

- $S \subseteq S_1 \times \ldots \times S_n \times \Gamma$ is the set of global states such that $(s_1, \ldots, s_n, \gamma) \in S$ iff $\gamma \vDash \mathcal{C}$, and $S_i = Env \times FIFO_i$ is the set of local states of rebec $r_i = \langle c, e_0^i \rangle$ where $FIFO_i = Seq(Msg_c)$ models a FIFO queue of messages sent to the rebec r_i. Therefore, each s_i can be denoted by the pair (ν_i, f_i). We use the dot notations $s_i.\nu$ and $s_i.f$ to access the environment and FIFO queue of the rebec i, respectively.
- $L = Act \cup \{\tau\}$ is the set of labels, where $Act = \bigcup_{c \in C} Msg_c$;
- The transition relation $\rightarrow \subseteq S \times L \times S$ is the least relation satisfying the semantic rules in Table 1;

Table 1. wRebeca natural semantic rules

$Term$:	$\nu_i, f_1, \ldots, f_n, \epsilon \leadsto_\gamma \nu_i, f_1, \ldots, f_n, \top$
$Assign$:	$\nu_i, f_1, \ldots, f_n, x := expr; \leadsto_\gamma \nu_i[x := eval(expr, \nu_i)], f_1, \ldots, f_n, \top$
$VDecl$:	$\nu_i, f_1, \ldots, f_n, T\ x; \leadsto_\gamma upd(\nu_i, \{x \mapsto 0\}), f_1, \ldots, f_n, \top$
$Block$:	$\dfrac{push(\emptyset, \nu_i), f_1, \ldots, f_n, \sigma^* \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \zeta}{\nu_i, f_1, \ldots, f_n, \{\sigma^*\} \leadsto_\gamma pop(\nu'_i), f'_1, \ldots, f'_n, \zeta}$
$Cond_1$:	$\dfrac{eval(expr, \nu_i) = true \quad \nu_i, f_1, \ldots, f_n, \beta_1 \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \zeta}{\nu_i, f_1, \ldots, f_n, if\ expr\ \beta_1\ else\ \beta_2 \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \zeta}$
$Cond_2$:	$\dfrac{eval(expr, \nu_i) = false \quad \nu_i, f_1, \ldots, f_n, \beta_2 \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \zeta}{\nu_i, f_1, \ldots, f_n, if\ expr\ \beta_1\ else\ \beta_2 \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \zeta}$
$Loop_1$:	$\dfrac{\begin{array}{c} eval(expr, \nu_i) = true \\ \nu_i, f_1, \ldots, f_n, \beta \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \top \\ \nu'_i, f'_1, \ldots, f'_n, while(expr)\ \beta \leadsto_\gamma \nu''_i, f''_1, \ldots, f''_n, \top \end{array}}{\nu_i, f_1, \ldots, f_n, while(expr)\ \beta \leadsto_\gamma \nu''_i, f''_1, \ldots, f''_n, \top}$
$Loop_2$:	$\dfrac{\begin{array}{c} eval(expr, \nu_i) = true \\ \nu_i, f_1, \ldots, f_n, \beta \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \bot \end{array}}{\nu_i, f_1, \ldots, f_n, while(expr)\ \beta \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \top}$
$Loop_3$:	$\dfrac{eval(expr, \nu_i) = false}{\nu_i, f_1, \ldots, f_n, while(expr)\ \beta \leadsto_\gamma \nu_i, f_1, \ldots, f_n, \top}$
$BCast$:	$\nu_i, f_1, \ldots, f_n, m(expr); \leadsto_\gamma \nu_i, f'_1, \ldots, f'_n, \top$, where $\forall\, k \leq n(k \neq i \wedge (\gamma_{i,k} == 1) \Rightarrow [f'_k = f_k \rhd m(eval(expr, v_i))][f'_k = f_k])$
$MCast$:	$\nu_i, f_1, \ldots, f_n, multicast(rcvs, expr); \leadsto_\gamma \nu_i, f'_1, \ldots, f'_n, \top$, where $\forall\, k \leq n(k \in rcvs \wedge (\gamma_{i,k} == 1) \Rightarrow [f'_k = m(eval(expr, v_i)) \rhd f_k][f'_k = f_k])$
$UCast_1$:	$\dfrac{\begin{array}{c} (\gamma_{i,j} == 1) \\ f'_j = f_j \rhd m(eval(expr, v_i)) \wedge \forall\, k \neq j(f'_k = f_k) \\ \nu_i, f'_1, \ldots, f'_n, \beta_1 \leadsto_\gamma \nu'_i, f''_1, \ldots, f''_n, \zeta \end{array}}{\nu_i, f_1, \ldots, f_n, unicast(j, m(expr))\ succ : \beta_1\ unsucc : \beta_2 \leadsto_\gamma \nu'_i, f''_1, \ldots, f''_n, \zeta}$
$UCast_2$:	$\dfrac{(\gamma_{i,j} == 0) \quad \nu_i, f_1, \ldots, f_n, \beta_2 \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \zeta}{\nu_i, f_1, \ldots, f_n, unicast(j, m(expr))\ succ : \beta_1\ unsucc : \beta_2 \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \zeta}$
Seq_1:	$\dfrac{\nu_i, f_1, \ldots, f_n, \sigma_1 \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \top \quad \nu'_i, f'_1, \ldots, f'_n, \sigma_2^* \leadsto_\gamma \nu''_i, f''_1, \ldots, f''_n, \zeta}{\nu_i, f_1, \ldots, f_n, \sigma_1\sigma_2^* \leadsto_\gamma \nu''_i, f''_1, \ldots, f''_n, \zeta}$
Seq_2:	$\nu_i, f_1, \ldots, f_n, break;\ \sigma^* \leadsto_\gamma \nu_i, f_1, \ldots, f_n, \bot$
$Handle$:	$\dfrac{\begin{array}{c} s_i.f = m(e) \rhd f_i \wedge \forall\, k \neq i(f_k = s_k.f) \\ \nu_i = push(\{fm(m) \mapsto e\}, s_i.\nu) \\ \nu_i, f_1, \ldots, f_n, body(m) \leadsto_\gamma \nu'_i, f'_1, \ldots, f'_n, \top \end{array}}{(s_1, \ldots, s_n, \gamma) \xrightarrow{m(e)} (s'_1, \ldots, s'_n, \gamma)}$, where $\forall\, k \neq i(s'_k = (s_k.\nu, f'_k)) \wedge s'_i = (pop(\nu'_i), f'_i)$
Mov	$(s_1, \ldots, s_n, \gamma) \xrightarrow{\tau} (s_1, \ldots, s_n, \gamma')$, where $\gamma' \models \mathcal{C}$

- s_0 is the initial state which is defined by the combination of initial states of rebecs and the initial topology, i.e., $s_0 = \{(s_0^1, \ldots, s_0^n, \gamma_0)\}$, where for the rebec $r_i = \langle c, e_0^i \rangle$, $s_0^i = (push(\theta_0, stack()), \langle c(e_0^i)\rangle)$ which denotes that the class variables (i.e., V_c) are initialized to the default value, denoted by θ_0, and its queue includes only the message $c(e_0^i)$, and $\gamma_0 \vDash \mathcal{C}$.

To describe the semantics of transitions in wRebeca in Table 1, we exploit an auxiliary transition relation $\rightsquigarrow_\gamma \subseteq (Env \times FIFO_1 \times \ldots \times FIFO_n \times Seq(Statement)) \to (Env \times FIFO_1 \times \ldots \times FIFO_n \times \{\top, \bot\})$ to address the effect of statement executions on the given environment of the rebec (which executes the statements) and the queue of all rebecs. Upon execution, the statements are either successfully terminated, denoted by $\top$, or abnormally terminated, denoted by $\bot$. Let ζ range over $\{\top, \bot\}$. Rule $Term$ explains that an empty statement terminates successfully. The effect of an assignment statement, i.e., $x := expr;$, is that the value of variable x is updated by $eval(expr, \nu_i)$ in ν_i as explained by the rule $Assign$. The variable declaration $T\ x;$ extends the variable valuation corresponding to the current scope by the value assignment $x \mapsto 0$, where 0 is the default value for the types of T, as explained in the rule $VDecl$. The behavior of a block is expressed by the rule $Block$, based on the behavior of the statements (in its scope) on the environment $push(\emptyset, \nu_i)$, where the empty valuation function may be extended by the declarations in the scope (by rule $VDecl$). Thereafter, to find the effect of the block, the last scope is popped from the environment. Rules $Cond_{1,2}$ specify the effect of the *if* statement: If $eval(expr, \nu_i)$ evaluates to $true$, its effect is defined by the effect of executing the *if* part, otherwise the *else* part. Rules $Loop_{1-3}$ explain the effect of the $while$ statement; If the loop condition evaluates to $true$, the effect of the $while$ statement is defined in terms of the effect of its body by the rules $Loop_{1,2}$, otherwise it terminates immediately as specified by the rule $Loop_3$. If the body of the $while$ statement terminates successfully, the effect of the $while$ statement is defined in terms of the effect of the $while$ statement on the resulting environment and queues of its body execution as explained by $Loop_1$. Rule $Loop_2$ expresses that if the body of the $while$ statement terminates abnormally (due to a $break$ statement) while its condition evaluates to $true$, then it terminates successfully while taking the effect of its body execution into account. The effect of a sequence of statements is specified by the rules $Seq_{1,2}$. Upon successful execution of a statement, the effect of its next statements is considered (rule Seq_1). A $break$ statement makes all its next statements be abandoned (rule Seq_2).

The expression $b \Rightarrow [C_1][C_2]$ in the post-conditions of rules $BCast$ and $MCast$ abbreviates $(b \Rightarrow C_1) \wedge (\neg b \Rightarrow C_2)$. The effects of broadcast and multi-cast communications are specified by the rules $BCast$ and $MCast$, respectively: the message $m(eval(expr, \nu_i))$ is appended to the queue of all connected nodes to the sender in case of broadcast, and all connected nodes among the specified receivers (i.e., $rcvs$) in case of multi-cast. Rules $UCast_{1,2}$ express the effect of unicast communication upon its delivery status. If the communication was successful (i.e., the sender was connected to the receiver), the message is appended to the queue of the receiver while the effect of the *succ* part is also considered (rule $UCast_1$), otherwise only the effect of the *unsucc* part is considered (rule $UCast_2$).

The rule $Handle$ expresses that the execution of a wRebeca model progresses when a rebec processes the first message of its queue. In this rule, the message $m(e)$ is processed by the rebec r_i as $s_i.f = m(e) \rhd f_i$. To process this message, its corresponding message server, i.e. $body(m)$ is executed. The effect of its execution is captured by the transition relation $\rightsquigarrow_\gamma$ on the environment of r_i, updated by the variable assignment $\{fm \mapsto e\}$ for the scope of the message server of m, and the queue of all rebecs while message $m(e)$ is removed from the queue of r_i. Finally, the rule Mov specifies that the underlying topology is implicitly changed at the semantic level, and the new topology satisfies $\mathcal{C}$.

Example Consider the global state $(s_0, s_1, s_2, s_3, \gamma)$ such that $s_0 = ((\{\!\{IP \mapsto 0\}\!\}, \langle relay\_packet(55, 0, 3)\rangle)$, $s_1 = (\{\!\{IP \mapsto 1\}\!\}, \epsilon)$, $s_2 = (\{\!\{IP \mapsto 2\}\!\}, \epsilon)$, $s_3 = (\{\!\{IP \mapsto 3\}\!\}, \epsilon)$, and $\gamma : \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix})$ for the wRebeca model in Fig. 4 where $\{\!\{IP \mapsto i\}\!\}$ denotes $push(\{IP \mapsto i\}, Stack())$. Regarding our rules, the following transition is derived:

$$\cfrac{\cfrac{\nu_2, \epsilon, \epsilon, \epsilon, \epsilon, hopNum{+}{+} \rightsquigarrow_\gamma \nu_3, \epsilon, \epsilon, \epsilon, \epsilon, \top \qquad \nu_3, \epsilon, \epsilon, \epsilon, \epsilon, rel(data, \ldots) \rightsquigarrow_\gamma \nu_3, \epsilon, \langle rel(55, 1, 3)\rangle, \epsilon, \epsilon, \top}{\nu_2, \epsilon, \epsilon, \epsilon, \epsilon, hopNum{+}{+};\ rel(data, \ldots) \rightsquigarrow_\gamma \nu_3, \epsilon, \langle rel(55, 1, 3)\rangle, \epsilon, \epsilon, \top}\ Seq_1}{\nu_1, \epsilon, \epsilon, \epsilon, \epsilon, \{hopNum{+}{+};\ rel(data, \ldots)\} \rightsquigarrow_\gamma \nu_4, \epsilon, \langle rel(55, 1, 3)\rangle, \epsilon, \epsilon, \top \ \ : (*)}\ Block$$

The following inference tree uses the result of the first tree, denoted by $(*)$, as a part of its premise to derive the transition.

$$\dfrac{\dfrac{eval(IP == des, \nu_1) = false \qquad \dfrac{eval(hopNum < 3, \nu_1) = true \qquad (*)}{\nu_1, \epsilon, \epsilon, \epsilon, \epsilon, if(hopNum < 3)\ldots \rightsquigarrow_\gamma \nu_4, \epsilon, \langle rel(55, 1, 3)\rangle, \epsilon, \epsilon, \top}\,Cond_1}{\nu_1, \epsilon, \epsilon, \epsilon, \epsilon, if\,(IP == \ldots \rightsquigarrow_\gamma \nu_1', \epsilon, \langle rel(55, 1, 3)\rangle, \epsilon, \epsilon, \top}\,Cond_2}{(s_0, s_1, s_2, s_3, \gamma) \xrightarrow{rel(55,0,3)} (s_0', s_1', s_2, s_3, \gamma)}\,Handle$$

where $\nu_1 = push(\{data \mapsto 55, hopNum \mapsto 0, des \mapsto 3\}, \{\{IP \mapsto 0\}\})$, $\nu_2 = push(\emptyset, \nu_1)$, $\nu_3 = \nu_2[hopNum := 1]$, $\nu_4 = pop(\nu_3)$, $\nu_1' = pop(\nu_4)$, $s_0' = (\{\{IP \mapsto 0\}\}, \epsilon)$, and $s_1' = (\{\{IP \mapsto 1\}\}, \langle rel(55, 1, 3)\rangle)$. Note that des denotes `destination`, and rel refers to `relay_packet` message. By the rule $Handle$, the message $rel(55, 0, 3)$ in the queue of $node_0$ is processed. To this aim, the body of its message server, i.e., *if* $(IP == \ldots$ is executed. Since $eval(IP == des, \nu_1) = false$, by the rule $Cond_2$, the *else* part (i.e., *if* $(hopNum < 3) \ldots)$ is executed. Due to $eval(hopNum < 3, \nu_1) = true$, by the rule $Cond_1$, the *if* part is executed.

5. State-space reduction

We extend application of the counter abstraction technique to wRebeca models when the topology is static. To this end, the local states of rebecs and their neighborhoods are considered. Later, we inspect the soundness of the counter abstraction technique in the presence of mobility. As a consequence, we propose a reduction technique based on removal of τ-transitions. Recall that the topology is static when the only valid topology of the network constraint is equal to the initial topology.

5.1. Applying counter abstraction

Assume S_c is the set of local states that the instances of the reactive class c can take (i.e., $S_c = Env_c \times FIFO_c$) and I is the set of rebec identifiers. To apply counter abstraction, rebecs with an identical local state and neighbors that are *topologically equivalent* are counted together. Two nodes $i, j \in I$ are said to be topologically equivalent, denoted by $i \approx_\gamma j$, iff $\forall\, k \in I \setminus \{i, j\}(\gamma_{ik} = \gamma_{jk})$. Intuitively, two topologically equivalent nodes have the same neighbors (except themselves). So if either one broadcasts, the same set of nodes (except themselves) will receive, and if they are also connected to each other, their counterpart (that is symmetric to the sender) will receive. Nodes in $\mathcal{N} \subseteq I$ are called topologically equivalent iff $\forall\, i, j \in \mathcal{N}\,(i \approx_\gamma j)$. This definition implies that all topologically equivalent nodes should be either all connected to each other, or disconnected, while they should have the same neighbors (except themselves). Therefore, topologically equivalent nodes will affect the same nodes when either one broadcasts. Hence, topologically equivalent nodes with an identical local state can be aggregated. To this aim, nodes of the underlying topology are partitioned into the maximal sets of topologically equivalent nodes, denoted by $\mathcal{N}_1, \ldots, \mathcal{N}_\ell$. We define the set of *distinct local states* as $S^d = \bigcup_{c \in C} S_c$, and the set of topology equivalence classes as $\mathbb{T} = \{\mathcal{N}_1, \ldots, \mathcal{N}_\ell\}$. Consequently, each global state $(s_1, \ldots, s_n, \gamma)$ is abstracted into a vector of elements $(s_i^d, \mathcal{N}_i) : c_i$ where $s_i^d \in S^d$, $\mathcal{N}_i \in \mathbb{T}$, and c_i is the number of nodes in the topology equivalence class $\mathcal{N}_i$ that reside in the very local state s_i^d. The reduced global state, called *abstract global state*, is presented as follows, where n and m donate the number of all rebecs and distinct local states (i.e., $m = |S^d|$), respectively:

$$S = ((s_1^d, \mathcal{N}_1) : c_1, \ldots, (s_k^d, \mathcal{N}_k) : c_k), \quad \forall\, i \leq k(c_i > 0 \wedge \mathcal{N}_i \in \mathbb{T}), \quad \sum_{i=1}^{k} c_i = n, \quad k \leq n$$

For instance, nodes n_1, n_4, and n_2, n_3 in Fig. 2 have the same neighbors, so if their state variables and queue contents are the same, then they can be counted together.

Recall that when the underlying topology is static, a global state may only change upon processing a message by a rebec, since in wRebeca the bodies of message servers execute atomically. Thus, its corresponding abstract global state may also only change upon processing a message by a rebec.

Counting abstraction is beneficial when the reactive classes do not have a variable that will be assigned uniquely to its instances, such as "unique address" as a state variable. (Note that at the semantics, rebecs have identifiers which are not a part of their local states.) For example, counter abstraction is not effective on the specification of the *flooding protocol* given in Fig. 4, since its nodes are identified uniquely by their IP addresses, and hence their state variables can not be collapsed. Therefore, to take benefit of this abstraction, we revise the example in the way that nodes are not distinguished by their IP addresses.

```
reactiveclass Node
{
    statevars
    {
        boolean destination;
    }

    msgsrv initial (boolean source,boolean dest)
    {
        destination=dest;
        if(source==true)
            relay_packet(55,1);
    }


    msgsrv relay_packet(int data,int hopNum)
    {
        if(destination==true)
            unicast( self , deliver_packet(data));
        else if(hopNum<2)
        {
            hopNum++;
            relay_packet(data,hopNum);
        }
    }
    msgsrv deliver_packet(int data)
    {

    }
}

main
{
    Node node0 (node1):(true,false);
    Node node1 (node0,node2,node3):(false,false);
    Node node2 (node1,node3):(false,false);
    Node node3 (node1,node2):(false,true);

}
```

Fig. 5. The revised version of the flooding protocol to make counter abstraction applicable in a network consisting of four nodes

$$\left(\begin{array}{l} (\{\!\{i \mapsto 1\}\!\}, \epsilon), \\ (\{\!\{i \mapsto 2\}\!\}, \langle msg \rangle), \\ (\{\!\{i \mapsto 1\}\!\}, \epsilon), \\ (\{\!\{i \mapsto 0\}\!\}, \epsilon), \end{array} \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \right)$$

(a)

$$\left(\begin{array}{l} ((\{\!\{i \mapsto 1\}\!\}, \epsilon), \{1,3\}) : \{1,3\}, \\ ((\{\!\{i \mapsto 2\}\!\}, \langle msg \rangle), \{2,4\}) : \{2\}, \\ ((\{\!\{i \mapsto 0\}\!\}, \epsilon), \{2,4\}) : \{4\} \end{array} \right)$$

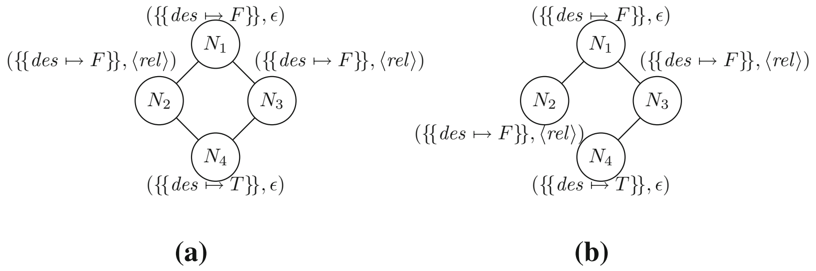
(b)

Fig. 6. An abstract global state and its corresponding transposed global state: assume $\{\!\{i \mapsto e\}\!\}$ denotes $push(\{i \mapsto e\}, Stack())$. **a** Before applying counter abstraction, **b** after applying counter abstraction

To this aim, the IP variable is replaced by the boolean variable `destination` which identifies the sink node, while the last parameter of the `relay_packet` message server is removed. The revised version is shown in Fig. 5.

The reduction takes place on-the-fly while constructing the state space. To this end, each global state $(s_1, \ldots, s_n, \gamma)$ is transformed into the form $((s_1^d, \mathcal{N}_1) : n_1, (s_2^d, \mathcal{N}_2) : n_2, \ldots, (s_k^d, \mathcal{N}_k) : n_k)$ such that $n_i \subseteq \mathcal{N}_i$ is the set of node identifiers that are topologically equivalent with the local state equal to s_i^d, where $\mathcal{N}_i \in \mathbb{T}$. This new presentation of the global state is called *transposed global state*. The sets n_i are leveraged to update the states of the potential receivers (known by the underlying topology) when a communication occurs. To generate the abstract global states, each transposed global state is processed by taking an arbitrary node from the set assigned to a distinct local state and a topology equivalence class if the distinct local state consists of a non-empty queue. The next transposed global state is computed by executing the message handler of the head message in the queue. This is repeated for all the pairs of a distinct local state and a topology equivalence class of the transposed global state. After generating all the next transposed global states of a transposed state, the transposed state is transformed into its corresponding abstract global state by replacing each n_i by $|n_i|$. A transposed global state is processed only if its corresponding abstract global state has not been previously computed. During state-space generation, only the abstract global states are stored. Fig. 6 illustrates a global state and its corresponding transposed global state. It is assumed that the network consists of four nodes of the reactive class with only one state variable i and message server msg. Each row in Fig. 6a represents a local state, i.e., valuation of the local state variable and message queue, while each row in Fig. 6b represents a distinct local state and a set of topologically equivalent identifiers together with those nodes of the set that reside in that distinct local state. As the topology is static, it can be removed from the abstract/transposed global states. Furthermore, each topology equivalence class of nodes can be represented by its unique representative, e.g., the one with the minimum identifier.

The following theorem states that applying counter abstraction preserves semantic properties of the model modulo strong bisimilarity. To this aim, we prove that states that are counted together are strong bisimilar. For instance, the global state similar to the one in Fig. 6a except that the distinct local states of nodes 2 and 4 are swapped, is mapped into the same abstract global state that corresponds to Fig. 6b.



$(\{\{des \mapsto F\}\}, \epsilon)$
N_1
$(\{\{des \mapsto F\}\}, \langle rel \rangle)$
$(\{\{des \mapsto F\}\}, \langle rel \rangle)$
N_2
N_3
N_4
$(\{\{des \mapsto T\}\}, \epsilon)$
(a)
$(\{\{des \mapsto F\}\}, \epsilon)$
N_1
$(\{\{des \mapsto F\}\}, \langle rel \rangle)$
N_2
N_3
$(\{\{des \mapsto F\}\}, \langle rel \rangle)$
N_4
$(\{\{des \mapsto T\}\}, \epsilon)$
(b)

and(con(N_1, N_2), and(con(N_1, N_3),
and(con(N_3, N_4), and(!con(N_1, N_2),
!con(N_2, N_3)))))

(c)

Fig. 7. Two possible topologies for the given constraint on the flooding protocol. **a** Topology 1, **b** topology 2, **c** an example of network topology constraint

Theorem 5.1 (Soundness of counter abstraction) Assume two global states S_1 and S_2 such that for all pair of $s^d \in S^d$ and $\mathcal{N} \in \mathbb{T}$, the number of topologically equivalent nodes of $\mathcal{N}$ that have the distinct local state s^d are the same in S_1 and S_2. Then they are strongly bisimilar.

Proof Since the topology is static, the only transitions these states have are the result of processing messages in their rebec queues. Suppose $S_1 \xrightarrow{m(e)} S_1'$ since there is a node i with the local state (ν_i, f_i) in the topology equivalence class $\mathcal{N}$, where $m(e)$ is the head of f_i using the semantic rule *Handle* in Table 1. Assume that i belongs to the topologically equivalent nodes $\mathcal{N}_1 \subseteq \mathcal{N}$, where $((\nu_i, f_i), \mathcal{N}) : \mathcal{N}_1$ is an element of the transposed global state corresponding to S_1. Due to the assumption, there exist topologically equivalent nodes $\mathcal{N}_2 \subseteq \mathcal{N}$ in S_2 with the distinct local state (ν_i, f_i) where $|\mathcal{N}_1| = |\mathcal{N}_2|$. We choose an arbitrary node j in $\mathcal{N}_2$ and prove that it triggers the same transition as i. We claim that $\forall(s_k^d, \mathcal{N}')$, the number of nodes in the topology equivalence class $\mathcal{N}'$ that are a neighbor of i, denoted by nb_i, and reside in the local state s_k^d is the same to the number of the nodes in the topology equivalence class $\mathcal{N}'$ that are a neighbor of j, denoted by nb_j, with the local state s_k^d. Assume for the arbitrary transposed global state element $(s_l^d, \mathcal{N}'')$ this does not hold, and we consider the case where nb_i has more topologically equivalent nodes than nb_j in $(s_l^d, \mathcal{N}'')$. As the links are bidirectional, due to the definition of abstract/transposed global states, i is the neighbor of nodes in $\mathcal{N}''$. Furthermore, as the topology is the same for S_1 and S_2 and $i, j \in \mathcal{N}$, then j is also the neighbor of nodes in $\mathcal{N}''$. However, due to the assumption, the number of topologically equivalent nodes of $\mathcal{N}''$ in S_1 and S_2 that have the distinct local state s_l^d are the same. So there are some topologically equivalent nodes of $\mathcal{N}''$ with the local state s_l^d that are not in nb_j, which contradicts to fact that j is the neighbor of nodes in $\mathcal{N}''$.

As both i and j handle the same message, they execute the same message server, and consequently the effects on their own local state and their neighbors will be the same. Therefore, $S_2 \xrightarrow{m(e)} S_2'$ while $\forall(s_o^d, \mathcal{N}^*)$ the number of topologically equivalent nodes from the equivalence class $\mathcal{N}^*$ in S_2' that have the distinct local state s_o^d is the same to S_1'. A similar argumentation holds when $S_2 \xrightarrow{m(e)} S_2'$ while the inequality between nb_i and nb_j goes the other way.

As mentioned before, the reduction is only applicable if the network is static. This is due to the fact that if node neighborhoods may change, then nodes which are in the same equivalence class in some state may no longer be equivalent in the next state. Consider the flooding protocol (Fig. 5) for the two topologies shown in Fig. 7a, b (satisfying the network constraint in Fig. 7c). By applying counter abstraction, nodes N_2 and N_3 are considered equivalent under topology 1, but not under topology 2.

To illustrate that counter abstraction is not applicable to systems with a dynamic topology, Fig. 8 shows a part of the state space of the flooding protocol with a change in the underlying topology (from Fig. 7a, b) with/without applying counter abstraction, where only these two topologies are possible. As predicted, the reduced state space is not strong bisimilar (see Sect. 2.3 for the definition) to its original state space. During transposed global state generation, the next state is only generated for node 2 with the distinct local state $(\{\{des \mapsto F\}\}, \langle rel \rangle)$ from the equivalence class {2, 3}. Therefore, it is obvious that the next states in the left LTS of Fig. 8 can be matched to the states with the solid borders in the right LTS. However, the solid bordered states are not strong bisimilar to the dotted ones in the right LTS. As explained in Sect. 1, the reduced LTS should be strong bisimilar to its original one to preserve all properties of its original model.

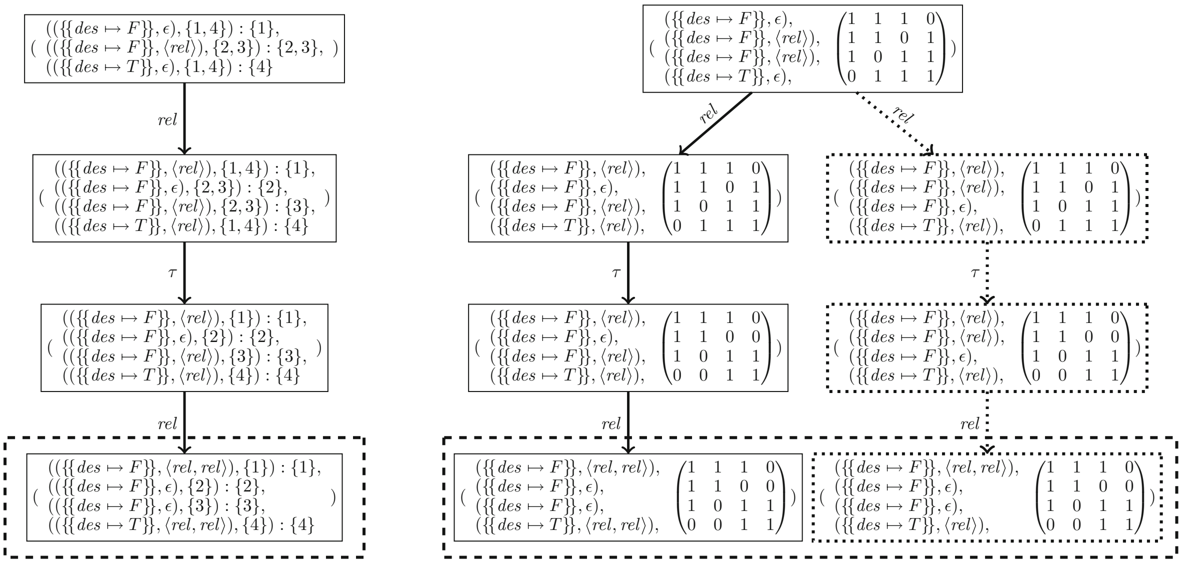


Fig. 8. Comparing a part of the flooding protocol's state space with/without applying counter abstraction in a dynamic network. The *two dashed bordered* states are not strong bisimilar since in the *right figure* there is a global state in which only one node has two *rel* messages in its queue while in the *left figure* there are two nodes with queues containing two *rel* messages. Note that T, F stand for *true*, *false*, *des* denotes `destination`, and *rel* refers to `relay_packet` messages. For simplicity the message parameters are not shown in the figure.

To take a better advantage of the reduction technique, the message storages can be modeled as bags. However, such an abstraction results in more interleavings of messages which do not necessarily happen in reality, and hence, an effort to inspect if a given trace (of the semantic model) is a valid scenario in the reality is needed. This effort is only tolerable if the state space reduces substantially.

5.2. Eliminating τ-transitions

Instead of modifying the underlying topology, modeled by τ-transitions, messages can be processed with respect to all possible topologies (not only to the current underlying topology). Therefore, all τ-transitions are eliminated and only those that correspond to processing of messages are kept. The following theorem expresses that removal of τ-transitions and topology information from the global states preserves properties of the original model modulo branching bisimulation, such as ACTL-X [DV90]. In fact, by exploiting a result from [DV90] about the correspondence between the equivalence induced by ACTL-X and branching bisimulation, the ACTL-X fragments of CACTL [GAFM13], introduced to specify MANET properties, and μ-calculus are also preserved. We show in Sect. 7.3 that important properties of MANET protocols can be still verified over reduced state spaces.

Theorem 5.2 (Soundness of τ-transition elimination) For the given LTS $T_0 \equiv \langle S \times \Gamma, \rightarrow, L, (s_0, \gamma_0)\rangle$, assume that $(s, \gamma) \xrightarrow{\alpha} (t, \gamma') \Rightarrow (\gamma = \gamma') \vee (\alpha = \tau \wedge s = t)$, and $\forall \gamma, \gamma' \in \Gamma : (s, \gamma) \xrightarrow{\tau} (s, \gamma')$. If $T_1 \equiv \langle S, \rightarrow', L, s_0\rangle$, where $\rightarrow' = \{(s, \alpha, t) \mid ((s, \gamma), \alpha, (t, \gamma)) \in \rightarrow\}$, then $(s_0, \gamma_0) \simeq_{br} s_0$.

Proof Construct $\mathcal{R} = \{((s, \gamma), s) \mid s \in S, \gamma \in \Gamma\}$ as shown in Fig. 9. We show that $\mathcal{R}$ is a branching bisimulation. To this aim, we show that it satisfies the transfer conditions of Definition 2.2. For an arbitrary relation $(s, \gamma)\, \mathcal{R}\, s$, assume $(s, \gamma) \xrightarrow{\alpha} (t, \gamma')$. If $\alpha = \tau$, then two cases can be distinguished: (1) either $\gamma \neq \gamma'$, and hence by definition of T_0, $s = t$ holds which concludes $(t, \gamma')\, \mathcal{R}\, s$, (2) or $\gamma = \gamma'$ and by definition of T_1, $s \xrightarrow{\alpha}' t$, and $(t, \gamma')\, \mathcal{R}\, t$.

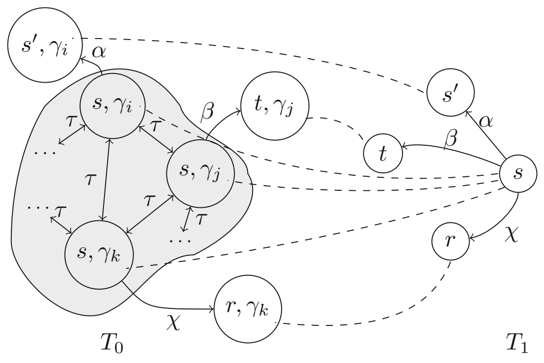


Fig. 9. Relation $\mathcal{R}$ matches states (s, γ) of T_0 to s of T_1.

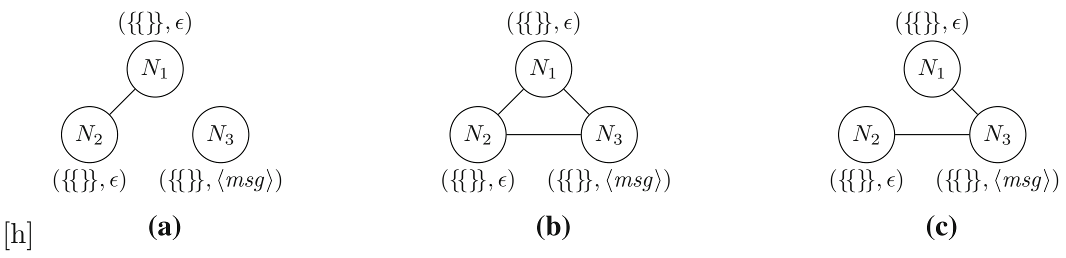


Fig. 10. All possible topologies considered during state-space generation of Fig. 11. **a** Topology γ_1, **b** topology γ_2, **c** topology γ_3

If $\alpha \neq \tau$, then by definition of T_0, $\gamma = \gamma'$ and hence by definition of T_1, $s \overset{\alpha}{\rightarrow}' t$, and $(t, \gamma') \mathcal{R}\, t$. Whenever $s \overset{\alpha}{\rightarrow}' t$, then by definition of T_1 there exists γ' such that $(s, \gamma') \xrightarrow{\alpha} (t, \gamma')$ and hence, $(t, \gamma') \mathcal{R}\, t$. Consequently $\mathcal{R}$ is a branching bisimulation relation.

We remark that the labeled transitions T_0 and T_1 in the Theorem 5.2 specify the state space of wRebeca models before and after elimination of τ-transitions, respectively. As an example, consider a network which consists of three nodes, which are the instances of a reactive class with no state variable and only one message, *msg*. The message server *msg* has only one statement to broadcast the message *msg* to its neighbors. We assume that the set of all possible topologies is restricted by a network constraint to the three topologies depicted in Fig. 10. Consider the global state in which only N_3 has one *msg* in its queue.

The state space of the above imaginary model before reduction is presented in Fig. 11a, where transitions take place by processing messages or changing the topology. Fig. 11b illustrates the state space after eliminating τ-transitions and topology information. Connectivity information is removed from the global states, as in each state its transitions are derived for all possible topologies. In this approach, transition labels are paired with the topology to denote the topology-dependent behavior of transitions. The two transitions labeled with γ_2 and γ_3 can be merged by characterizing the links that make communication from N_3 to N_1 and N_2; i.e., from the sender to the receivers. Such links can be characterized by the network constraints depicted in Fig. 11c. In this model, a state is representative of all possible topologies. The resulting semantic model, called *Constrained Labeled Transition System* (CLTS), was introduced in [GFM11] as the semantic model to compactly model MANET protocols. Another advantage of a CLTS is its model checker to verify topology-dependant behavior of MANETs [GAFM13]. The properties in wireless networks are usually pre-conditioned to existence of a path between two nodes. This model checker takes benefit of network constraints over transitions and assures a property holds if the required paths hold (inferred from the traversed network constraints).

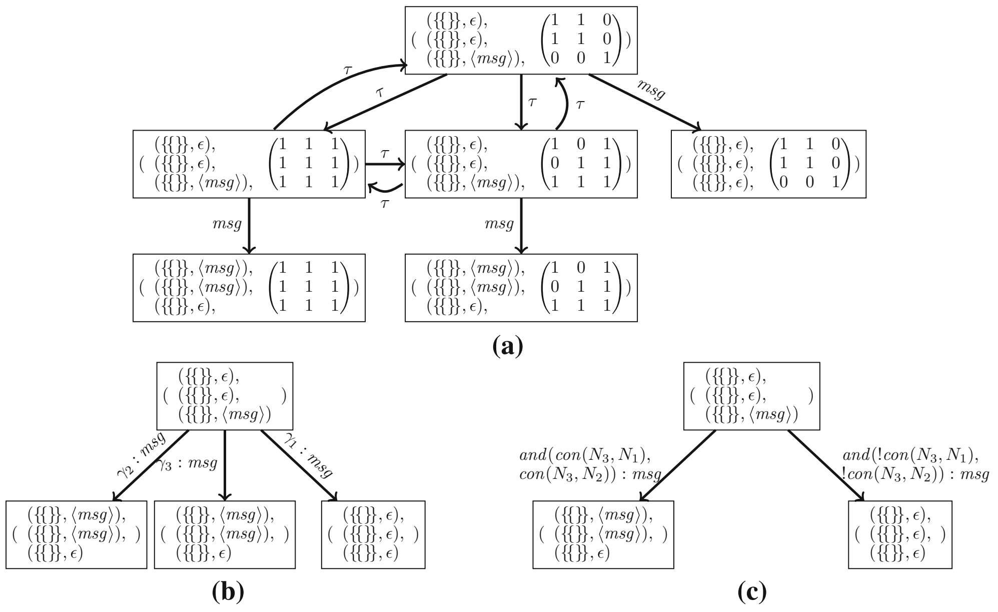

τ
msg
(a)
γ₂ : msg
γ₃ : msg
γ₁ : msg
(b)
and(con(N₃, N₁), con(N₃, N₂)) : msg
and(!con(N₃, N₁), !con(N₃, N₂)) : msg
(c)

Fig. 11. State space before and after applying reduction. **a** State space before reduction, **b** reduced State space after eliminating τ-transitions and topology information, **c** reduced state space with labels characterized by network constraints

6. Modeling the AODVv2 protocol

To illustrate the applicability of the proposed modeling language, the AODVv2[3] (i.e., version 11) protocol is modeled. The AODV is a popular routing protocol for wireless ad hoc networks, first introduced in [PB99], and later revised several times.

In this algorithm, routes are constructed dynamically whenever requested. Every node has its own routing table to maintain information about the routes of the received packets. When a node receives a packet (whether it is a route discovery or data packet), it updates its own routing table to keep the shortest and freshest path to the source or destination of the received packet. Three different tables are used to store information about the neighbors, routes and received messages:

- neighbor table: keeps the adjacency states of the node's neighbors. The neighbor state can be one of the following values:
  - *Confirmed: indicates that a bidirectional link to that neighbor exists. This state is achieved either through receiving a rrep message in response to a previously sent rreq message, or a RREP_Ack message as a response to a previously sent rrep message (requested an RREP_Ack) to that neighbor.*
  - *Unknown: indicates that the link to that neighbor is currently unknown. Initially, the states of the links to the neighbors are unknown.*
  - *Blacklisted: indicates that the link to that neighbor is unidirectional. When a node has failed to receive the RREP_Ack message in response to its rreq message to that neighbour, the neighbor state is changed to blacklisted. Hence, it stops forwarding any message to it for an amount of time, ResetTime. After reaching the ResetTime, the neighbor's state will be set to unknown.*
- route table: contains information about discovered routes and their status: The following information is maintained for each route:
  - *SeqNum: destination sequence number*
  - *route_state: the state of the route to the destination which can have one of the following values:*

[3] https://tools.ietf.org/html/draft-ietf-manet-aodvv2-11.

- · unconfirmed*: when the neighbor state of the next hop is unknown;*
- · active*: when the link to the next hop has been confirmed, and the route is currently used;*
- · idle*: when the link to the next hop has been confirmed, but it has not been used in the last ACTIVE_INTERVAL;*
- · invalid*: when the link to the next hop is broken, i.e., the neighbor state of the next hop is blacklisted.*

- *Metric: indicates the cost or quality of the route, e.g., hop count, the number of hops to the destination*
- *NextHop: IP address of the next hop to the destination*
- *Precursors (optional feature): the list of the nodes interested in the route to the destination, i.e., upstream neighbors.*

- route message table, also known as *RteMsg Table*: contains information about previously received route messages such as $rreq$ and $rrep$, so that we can determine whether the new received message is worth processing or redundant. Each entry of this table contains the following information:
  - *MessageType: which can be either $rreq$ or $rrep$*
  - *OrigAdd: IP address of the originator*
  - *TargAdd: IP address of the destination*
  - *OrigSeqNum: sequence number of the originator*
  - *TargSeqNum: sequence number of the destination*
  - *Metric*

When one node, i.e., source, intends to send a package to another, i.e., destination, it looks up its routing table for a valid route to that destination, i.e., a route of which the route state is not invalid. If there is no such a route, it initiates a route discovery procedure by broadcasting a $rreq$ message. The freshness of the requested route is indicated through the sequence number of the destination that the source is aware of. Whenever a node initiates a route discovery, it increases its own sequence number, with the aim to define the freshness of its route request. Every node upon receiving this message checks its routing table for finding a route to the requested destination. If there is such a path or the receiver is in fact the destination, it informs the sender through unicasting a $rrep$ message. However, an acknowledgment is requested whenever the neighbor state of the next hop is *unconfirmed*. Otherwise, it re-broadcasts the $rreq$ message to examine if any of its neighbors has a valid path. Meanwhile, a reverse forwarding path is constructed to the source over which *rrep* messages are going to be communicated later. In case a node receives a $rrep$ message, if it is not the source, it forwards the $rrep$ after updating its routing table with the received route information. Whenever a node fails to receive a requested $RREP\_Ack$, it uses a $rerr$ message to inform all its neighbors intended to use the broken link to forward their packets.

In our model, each node is represented through a rebec (actor), identified by an IP address, with a routing table and a sequence number (sn). In addition, every node keeps track of the adjacency status to its neighbors by means of a neighbor table, through the $neigh\_state$ array, where $neigh\_state[i] = true$ indicates that it is adjacent to the node with the IP address i, while $false$ indicates that its adjacency status is either *unknown* or *blacklisted* (since timing issues are not taken into account, these two statuses are considered the same). As the destinations of any two arbitrary rows of a routing table are always different, the routing table has at most n rows, where n is the number of nodes in the model. Therefore, the routing table is modeled by a set of arrays, namely, dsn, $route\_state$, $hops$, $nhops$, and $pres$, to represent the $SeqNum$, $route\_state$, $Metric$, $NextHop$, and $Precursors$ columns of the routing table, respectively. The arrays dsn and $route\_state$ are of size n, while the arrays $hops$, $nhops$, and $pers$ are of size $n \times n$. For instance, $dsn[i]$, keeps the sequence number of the destination with IP address i, while $nhops[i][j]$ contains the next hop of the j-th route to the destination with the IP address i.

- dsn: destination sequence number
- $route\_state$: an integer that refers to the state of the route to the destination and can have one of the following values:
  - $route\_state[i] = 0$: the route is *unconfirmed*, there may be more than one route to the destination i with different next hops and hop counts;

  - $route\_state[i] = 1$: the route is *valid*, the link to the next hop has been confirmed, the route state in the protocol is either **active** or **idle**; since we abstract from the timing issues, these two states are depicted as one;
  - $route\_state[i] = 2$: the route is *invalid*, the link to the next hop is broken;

- $hops$: the number of hops to the destination for different routes
- $nhop$: IP address of the next hop to the destination for different routes
- $pres$: an array that indicates which of the nodes are interested in the routes to the destination, for example $pres[i][j] = true$ indicates that the node with the IP address j is interested in the routes to the node with the IP address i.

Since we have considered a row for each destination in our routing table, to indicate whether the node has any route to each destination until now, we initially set $dsn[i]$ to -1 which implies that the node has never known any route to the node with the IP address i. We refer to the all above mentioned arrays as *routing arrays*. Initially all integer cells of arrays are set to -1 and all boolean cells are set to $false$. To model expunging a route, its corresponding next hop and hop count entries in the arrays *nhops* and *hops* are set to -1. Since we have only considered one node as the destination and one node as the source, the information in $rreq$ and $rrep$ messages has no conflict and consequently the route message table can be abstracted away. In other words, the routing table information can be used to identify whether the new received message has been seen before or not, as the stored routes towards the source represent information about $rreq$s and the routes towards the destination represent $rrep$s.

Note that $rreq$ and $rrep$, i.e., all route messages, carry route information to their source and destination, respectively. Therefore, a bidirectional path is constructed while these messages travel through the network. Whenever a node receives a route message, it processes incoming information to determine whether it offers any improvement to its known existing routes. Then, it updates its routing table accordingly in case of an improvement. The processes of evaluating and updating the routing table are explained in the following subsections.

6.1. Evaluating route messages

Every received route message contains a route and consequently is evaluated to check for any improvement. Note that a $rreq$ message contains a route to its source while a $rrep$ message contains a route to its destination. Therefore, as the routes are identified by their destinations (denoted by des), in the former case, the destination of the route is the originator of the message (i.e., $des = oip\_$), and in the latter, it is the destination of the message (i.e., $des = dip\_$). The routing table must be evaluated if one of the following conditions is realized:

1. no route to the destination has existed, i.e., $dsn[des] = -1$
2. there are some routes to the destination, but all their route states are *unconfirmed*
3. there is a valid or invalid route to the destination in the routing table and one of following conditions holds:
   - the sequence number of the incoming route is greater than the existing one
   - the sequence number of the incoming route is equal to the existing one, however the hop count of the incoming route is less than the existing one (the new route offers a shorter path and also is loop free)

6.2. Updating the routing table

The routing table is updated as follows:

- if no route to the destination has existed, i.e., $dsn[des] = -1$, the incoming route is added to the routing table.
- if the route states of existing routes to the destination are *unconfirmed*, the new route is added to the routing table.

- the incoming route has a different next hop from the existing one in the routing table, while the next hop's neighbor state of the incoming route is *unknown* and the route state of the existing route is *valid*. The new route should be added to the routing table since it may offer an improvement in the future and turn into *confirmed*.
- if the existing route state is *invalid* and the neighbor state of the next hop of the incoming route is *unknown*, the existing route should be updated with information of the received one.
- if the next hop's neighbor state of the incoming route is *confirmed*, the existing route is updated with new information and all other routes with the route state *unconfirmed* are expunged from the routing table.

As described earlier, there are three types of route discovery packets: $rreq$, $rrep$ and $rerr$. There is a message server for handling each of these packet types:

- rec_rreq is responsible for processing a route discovery request message;
- rec_rrep handles a reply request message;
- rec_rerr updates the routing table in case an error occurs over a path and informs the interested nodes about the broken link.

There are also two message servers for receiving and sending a data packet. All these message servers will be discussed thoroughly in the following subsections.

6.3. $rreq$ message server

This message server processes a received route discovery request and reacts based on its routing table, shown in Fig. 14. The $rreq$ message has the following parameters: $hops\_$ and $maxHop$ as the number of hops and the maximum number of hops, $dsn\_$ as the destination sequence number, and $oip\_$, $osn\_$, dip, and $sip\_$ respectively refer to the IP address and sequence number of the originator, and the IP address of the destination, and the IP address of the sender. Whenever a node receives a route request, i.e., $rec\_rreq(hops\_, dip\_, dsn\_, oip\_, osn\_, sip\_, maxHop)$ message, it checks incoming information with the aim to improve the existing route or introduce a new route to the destination, and then updates its routing table accordingly (see also Sections 6.1 and 6.2). During processing an $rreq$ message, a backward route, from the destination to the originator is built by manipulating the routing arrays with the index $oip\_$. Similarly, while processing an $rrep$ message, it constructs a forwarded route to the destination by addressing the routing arrays with the index $dip\_$. Therefore, the procedure of evaluating the new route and updating the routing table is the same for both $rreq$ and $rrep$ messages, except for different indices $oip\_$ and $dip\_$, respectively.

Updating the routing table Figure 12 depicts this procedure which includes both evaluating the incoming route and updating the routing table (the code is the body of *if*-part in the line 7 of Fig. 14). If no route exists to the destination, the received information is used to update the routing table and generate discovery packets, lines (1–10). The route state is set based on the neighbor status of the sender: if its neighbor status is *confirmed*, the route state is set to *valid*, otherwise to *unconfirmed*. The next hop is set to the sender of the message, i.e., $nhop[oip\_][0] = sip\_$. If a route exists to the destination (i.e., $oip\_$), one of the following conditions happens:

- the route state is *unconfirmed*, lines (11–36): it either updates the routing table if there is a route with a next hop equal to the sender, or adds the incoming route to the first empty cells of $nhop$ and $hops$ arrays. If the neighbor status of the sender is *confirmed*, then all other routes with the same destination are expunged while the route state is set to *valid*, lines (21-30).
- the route state is *invalid* or it is valid, but the neighbor status of the sender is *confirmed*, lines (38–48): if the incoming message contains a greater sequence number, or an equal sequence number with a lower hop count, then it updates the current route while a new discovery message is generated.
- the route state is *valid* and the neighbor status of the sender is *unknown*, lines (50–66): the incoming route is added to the routing table and a new discovery message is generated if it provides a fresher or shorter path.

```
if(dsn[oip_]==−1) {
      dsn[oip_]=osn_;
      if(neigh_state[sip_]==true)
           { route_state[oip_]=1; }
      else
           { route_state[oip_]=0; }
      hops[oip_][0]=hops_;
      nhop[oip_][0]=sip_;
      gen_msg = true;
} else {
      if(route_state[oip_]==0) {
             dsn[oip_]=osn_;
             route_num = 0;
             for(int i=0;i<4;i++)
             {
                    if(nhop[oip_][i]==−1 || nhop[oip_][i]==sip_) {
                          route_num = i;
                          break;
                    }
             }
             if(neigh_state[sip_]==true) {
                    route_state[oip_]=1;
                    for(int i=0;i<4;i++)
                    {
                          hops[oip_][i]=−1;
                          nhop[oip_][i]=−1;
                    }
                    hops[oip_][0]=hops_;
                    nhop[oip_][0]=sip_;
             }
             else {
                    route_state[oip_]=0;
                    hops[oip_][route_num]=hops_;
                    nhop[oip_][route_num]=sip_;
             }
      }
      else {
             if(route_state[oip_]==2 || neigh_state[sip_]==true) {
                    /* update the existing route */
                    if((dsn[oip_]==osn_ && hops[oip_][0]>hops_) || dsn[oip_]<osn_ ) {
                          dsn[oip_]=osn_;
                          if(neigh_state[sip_]==true) route_state[oip_]=1;
                               else route_state[oip_]=0;
                          hops[oip_][0]=hops_;
                          nhop[oip_][0]=sip_;
                          gen_msg = true;
                    }
             }
             else {
          route_num = 0;
          for(int i=0;i<4;i++)
          {
             if(nhop[oip_][i]==−1 || nhop[oip_][i]==sip_)
             {
                route_num = i;
                break;
             }
          }
          if((dsn[oip_]==osn_ && hops[oip_][0]>hops_) || dsn[oip_]<osn_ )
          {
              dsn[oip_]=osn_;
              hops[oip_][route_num]=hops_;
              nhop[oip_][route_num]=sip_;
              gen_msg = true;
          }
          }
   }
}
```

Fig. 12. Updating the routing table

```
 1 route_state[oip_]=2;
 2 dip_sqn[oip_]=dsn[oip_];
 3 for(int k=0;k<4;k++)
 4 {
 5     if(pre[oip_][k]==true)
 6         { affected_neighbours[k]=true; }
 7 }
 8 for(int j=0;j<4;j++)
 9 {
10     for(int r=0;r<4;r++)
11     {
12         if(nhop[oip_][r]!=-1 &&  nhop[j][0]==nhop[oip_][r])
13         {
14             route_state [j]= 2;
15             dip_sqn[j]=dsn[j];
16             for(int k=0;k<4;k++)
17             {
18                 if(pre[j][k]==true)
19                     { affected_neighbours[k]=true; }
20             }
21             break;
22         }
23     }
24 }
25 multicast(affected_neighbours, rec_rerr (dip_,ip,dip_sqn));
```

Fig. 13. The error recovery procedure

In these cases, if a new discovery message should be generated (when the node has no route as fresh as the route request), the auxiliary boolean variable *gen_msg* is set to $true$. In Fig. 14, after updating the routing table, if a new message should be generated, indicated by *if* (*gen_msg* = *true*), it rebroadcasts the $rreq$ message with the increased hop count if the node is not the destination, lines (51–54). Otherwise, it increases its sequence number and replies to the next hop(s) toward the originator of the route request, *oip_*, based on its routing table. Before unicasting $rrep$ messages, next hops toward the destination, *dip_*, and the sender are set as interested nodes to the route toward the originator, *oip_*, lines (17–22). It unicasts each $rrep$ message to its next hops one by one until it gets an ack from one, lines (23–43); ack reception is modeled implicitly through successful delivery of unicast, i.e., the *succ* part. If it receives an ack, it updates the route state to *valid* and the neighbor status of the next hop to *confirmed* and stops unicasting $rrep$ messages. If it doesn't receive an $RREP\_Ack$ message from the next hop when the route state is *valid*, it initiates the error recovery procedure.

Error recovery procedure The code for this procedure is illustrated in Fig. 13 (its code is the body of *if*-part in line 46 of Fig. 14). As explained earlier, this procedure is initiated when a node doesn't receive an $RREP\_Ack$ message from the next hop of the route with state *valid*. Then, it updates its route state to *invalid* and adds the sequence number of the originator to the array of invalidated sequence numbers, denoted by dip_sqn. Furthermore, it adds all the interested nodes in the current route to the list of affected neighbors, denoted by $affected\_neighbours$, lines (3–7). It invalidates other valid routes that use the same broken next hop as their next hops, adds their sequence numbers to the invalidated array and sets the nodes interested in those routes as affected neighbors, lines (8–24). Finally, it multicasts an $rerr$ message which contains the destination IP address, the node IP address, and the invalidated sequence numbers to the affected neighbors, line 25.

6.4. $rrep$ message server

This message server, shown in Fig. 15, processes the received reply messages and also constructs the route forward to the destination. At first, it updates the routing table and decides whether the message is worth processing, as previously mentioned for $rreq$ messages, and constructs the route, but this time to the destination (its code is similar to the one in Fig. 12 except that $dip\_$ is used instead of $oip\_$, and is place at line 6 of Fig. 15). This message is sent backwards till it reaches the source through the reversed path constructed while broadcasting the $rreq$ messages. When it reaches the source, it can start forwarding data to the destination.

```
 msgsrv rec_rreq(int hops_,int dip_ ,int dsn_ ,int oip_ ,int osn_ ,int sip_,int maxHop)
{
     int []  dip_sqn=new int[4];
     int route_num;
     boolean[] affected_neighbours=new boolean[4];
     boolean gen_msg = false;
     if(ip!=oip_)
     {
          //evaluate and update the routing table
     }
     if(gen_msg==true)
     {
          if(ip==dip_)
          {
               boolean su = false;
               pre[dip_][sip_]=true;
               for(int i=0;i<4;i++)
               {
                    int nh = nhop[dip_][i];
                    if(nh!=-1)
                         { pre[oip_][nh]=true; }
               }
               for(int i=0;i<4;i++)
               {
                    if(nhop[oip_][i]!=-1)
                    {
                         int n_hop = nhop[oip_][i];
                         sn          = sn+1;
                         /* unicast a RREP towards oip of the RREQ */
                         unicast(n_hop,rec_rrep(0 , dip_ , sn , oip_ , self))
                         succ:
                         {
                              route_state[oip_]=1;
                              neigh_state[n_hop]=true;
                              su = true;
                              break;
                         }
                         unsucc:
                         {
                              neigh_state[n_hop]=false;
                         }
                    }
               }
               if(su==false && route_state[oip_]==1)
               {
                    /* error recovery procedure */
               }
          }
          else {
               hops_ = hops_+1;
               if(hops_<maxHop)
                    { rec_rreq(hops_,dip_,dsn_,oip_,osn_, self,maxHop); }
          }
     }
}
```

Fig. 14. The rreq message server

```
 1 msgsrv rec_rrep(int hops_ ,int dip_ ,int dsn_ ,int oip_ ,int sip_) {
 2     int []  dip_sqn=new int[4];
 3     boolean[] affected_neighbours=new boolean[4];
 4     boolean gen_msg = false;
 5     int n_hop,route_num;
 6     /* evaluate and update the routing table */
 7     if(gen_msg==true)
 8     {
 9         if(ip==oip_ )
10         {
11             /* this node is the originator of the corresponding RREQ */
12             /* a data packet may now be sent */
13         }
14         else {
15             hops_   = hops_+1;
16             boolean su = false;
17             pre[oip_][sip_]=true;
18             for(int i=0;i<4;i++)
19             {
20                 n_hop = nhop[oip_][i];
21                 if(n_hop!=-1)
22                     { pre[oip_][n_hop]=true; }
23             }
24             for(int i=0;i<4;i++)
25             {
26                 if(nhop[oip_][i]!=-1)
27                 {
28                     n_hop = nhop[oip_][i];
29                     unicast(n_hop,rec_rrep(hops_,dip_,dsn_,oip_, self))
30                     succ:
31                     {
32                         route_state[oip_]=1;
33                         neigh_state[n_hop]=true;
34                         su = true;
35                         break;
36                     }
37                     unsucc:
38                     {
39                         neigh_state[n_hop]=false;
40                     }
41                 }
42             }
43             if(su==false && route_state[oip_]==1)
44             {
45                 /* error recovery procedure */
46             }
47         }
48     }
49 }
```

Fig. 15. The rrep message server

In case the node is not the originator of the route discovery message, it updates the array of interested nodes, lines (17–23). Then, it unicasts the message to the next hop(s), on the reverse path to the originator, lines (24–42). Based on the AODVv2 protocol, if connectivity to the next hop on the route to the originator is not confirmed yet, the node must request a Route Reply Acknowledgment (*RREP_Ack*) from the intended next hop router. If a *RREP_Ack* is received, then the neighbor status of the next hop and route state must be updated to *confirmed* and *valid*, respectively, lines (30–36), otherwise the neighbor status of the next hop remains *unknown*, lines (37–40). This procedure is modeled through *conditional unicast* which enables the model to react based on the delivery status of the unicast message so that *succ* models the part where the *RREP_ACK* is received while *unsucc* models the part where it fails to receive an acknowledgment from the next hop. In case the unicast is unsuccessful and the route state is valid, the error recovery procedure will be followed, lines (43–46).

```
 1 msgsrv rec_rerr(int source_, int sip_ ,int [] rip_rsn) {
 2     int []  dip_sqn=new int[4];
 3     boolean[] affected_neighbours=new boolean[4];
 4     if(ip!=source_)
 5     {
 6         //regenerate rrer for invalidated routes
 7         for(int i=0;i<4;i++)
 8         {
 9             int rsn=rip_rsn[i];
10             if(route_state[i]==1 && nhop[i][0]==sip_ && dsn[i]<rsn && rsn!=0)
11             {
12                 route_state [i]= 2;
13                 dip_sqn[i]=dsn[i];
14                 for(int j=0;j<4;j++)
15                 {
16                     if(pre[i][j]==true)
17                         { affected_neighbours[j]=true; }
18                 }
19             }
20         }
21         multicast(affected_neighbours, rec_rerr(source_, self,dip_sqn));
22     }
23 }
```

Fig. 16. The rerr message server

6.5. $rerr$ message server

This message server, shown in Fig. 16, processes the received error messages and informs those nodes that depend on the broken link. When a node receives an $rerr$ message, it must invalidate those routes using the broken link as their next hops and sends the $rerr$ message to those nodes interested in the invalidated routes. This message has only two parameters: *sip_* which indicates the IP address of the sender, and *rip_rsn*, which contains the sequence number of those destinations which have become unaccessible from the *sip_*.

For all the *valid* routes to the different destinations, it examines whether the next hop of the route to the destination is equal to *sip_* and the sequence number of the route is smaller then the received sequence number, line 10. In case the above conditions are satisfied, the route is invalidated, lines (11–19), and an *rerr* message is sent to the affected nodes, line 21.

6.6. $newpkt$ message server

Whenever a node intends to send a data packet, it creates a *rec_newpkt* which has only two parameters, *data* and *dip_*. The code for this message server is shown in Fig. 17. If it is the destination of the message, it delivers the message to itself, lines (4–7). Otherwise, if it has a valid route to the destination, it sends data using that route, lines (11–15). If it has no valid route, it increases its own sequence number and broadcasts a route request message, lines (16–25). In addition, if a route to the destination is not found within $RREQ\_WAIT\_TIME$, the node retries to send a new $rreq$ message after increasing its own sequence number. Since we abstracted away from time, we model this procedure through the $resend\_rreq$ message server which attempts to resend an $rreq$ message while the node sequence number is smaller than 3 (to make the state space finite).

7. Evaluation

In this section, we will review the results obtained from efficiently constructing the state spaces for the two introduced wRebeca models, the flooding and AODV protocol. Also, we briefly introduce our tool and its capabilities. Then, the loop freedom invariant is defined and one possible loop scenario is demonstrated. Finally, two properties that must hold for the AODV protocol are expressed that can be checked with regard to the AODV model.

```
msgsrv rec_newpkt(int data ,int dip_) {
     int []  dip_sqn=new int[4];
     boolean[] affected_neighbours=new boolean[4];
     if (ip==dip_ )
      {
          /* the DATA packet is intended for this node */
     }
     else {
          /* the DATA packet is not intended for this node */
          store [dip_]=data;
          if (route_state [dip_]==1)
          {
               /* valid route to dip*/
               /* forward packet */
          }
          else {
               /* no valid route to dip*/
               /* send a new rout discovery request*/
               if (sn<3)
               {
                    sn++;
                    unicast( self ,resend_rreq(dip_)) ;
                    rec_rreq (0,dip_,dsn[dip_], self ,sn, self ,4) ;
               }
          }
     }
}
msgsrv resend_rreq(int dip_)
{
     if (sn<3)
     {
          sn++;
          unicast( self ,resend_rreq(dip_)) ;
          rec_rreq (0,dip_,dsn[dip_], self ,sn, self ,4) ;
     }
}
```

Fig. 17. The rec_newpkt message server

7.1. State-space generation

Static network Consider a network with a static topology, in other words the network constraint is defined so that it leads to only one valid topology. We illustrate the applicability of our counting abstraction technique on the flooding routing protocol. In contrast to the intermediate nodes on a path (the ones except the source and destination), the two source and destination nodes cannot be aggregated (due to their local states). However, in the case of the AODV protocol, no two nodes can be counted together due to the unique variables of IP and routing table of each node. As the number of intermediate nodes with the same neighbors increases, the more reduction takes place. We have precisely chosen four fully connected network topologies to show the power of our reduction technique when the intermediate nodes increase from one to four.

Table 2 illustrates the number of states when running the flooding protocol on different networks with different topologies before and after applying counter abstraction reduction. In the first, second, third, and fourth topology, there are three nodes with one intermediate, four nodes with two, five nodes with three, and six with four intermediates, respectively. By applying counter abstraction reduction, the intermediate nodes are collapsed together as they have the same role in the protocol. However, the effectiveness of this technique depends on the network topology and the modeled protocol.

Dynamic network At these networks, topology is constantly changing, in other words there are more than one possible topology. The resulting state spaces after and before eliminating τ-transitions are compared for the two case studies while the topology is constantly changing for a networks of 4 and 5 nodes, as shown in Table 3. Table 4 depicts the constraints used to generate the state spaces and the number of topologies that each constraint results in. Constraints are chosen randomly here, just to show the effectiveness of our reduction technique.

Table 2. Comparing the size of state spaces with/without applying counter abstraction reduction

No. of intermediate nodes	No. of states before reduction	No. of states after reduction	No. of transitions before reduction	No. of transitions after reduction
1	24	24	36	36
2	226	133	574	276
3	3689	912	13,197	2441
4	71,263	6649	321,419	21,466

Table 3. Comparing the size of state spaces with/without applying τ-transition elimination reduction

	No. of nodes	No. of valid topologies	No. of states before reduction	No. of transitions before reduction	No. of states after reduction	No. of transitions after reduction
Flooding protocol	4	4	2119	11,724	541	1652
	4	8	4431	42,224	567	1744
	4	16	10,255	179,936	655	2192
	4	32	22,255	747,200	710	2765
	4	64	44,495	2,917,728	710	3145
AODV protocol	4	4	3007	16,380	763	1969
	4	8	12,327	113,480	1554	3804
	4	16	35,695	610,816	2245	5549
	4	32	93,679	3,097,792	2942	7596
	4	64	258,447	16,797,536	4053	10,629
	5	16	>655,441	>11,276,879	165,959	598,342

Table 4. Applied network constraints

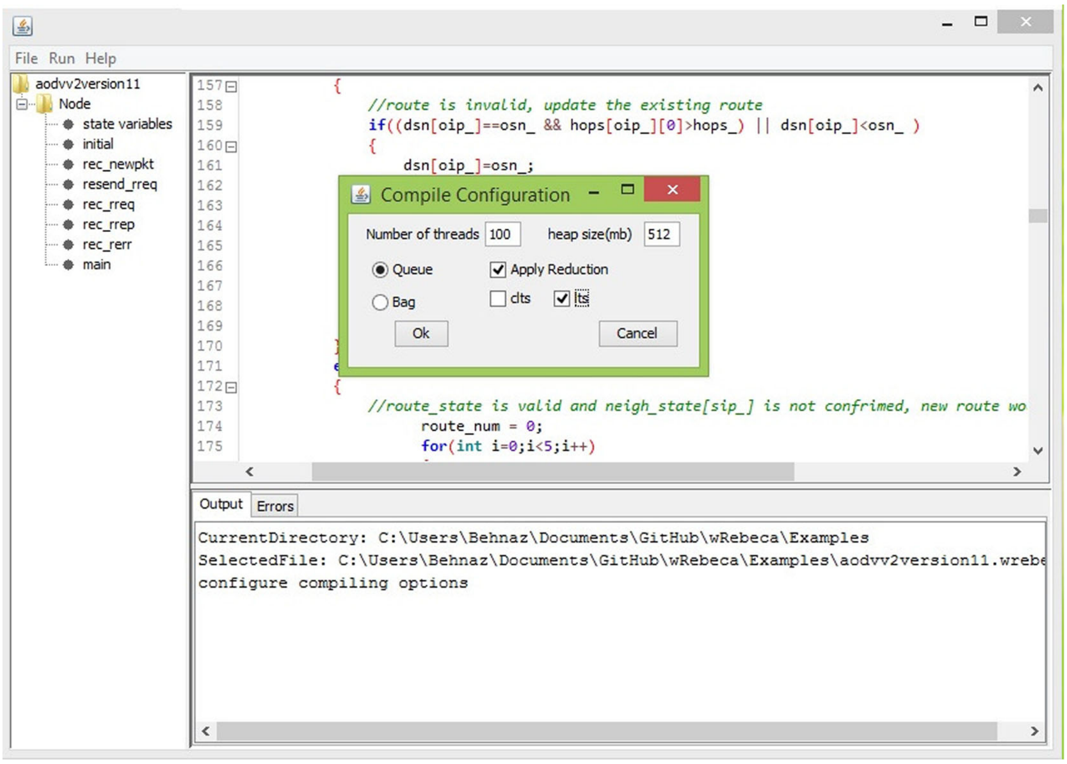
No. of nodes	No. of valid topologies	constraint
4	4	$and(and(con(node0, node1), con(node0, node3)), and(con(node2, node3), con(node1, node3)))$
4	8	$and(and(con(node0, node1), con(node0, node3)), con(node2, node3))$
4	16	$and(con(node0, node1), con(node2, node3))$
4	32	$con(node0, node1)$
5	16	$and(and(con(node0, node1), and(con(node0, node3), con(node4, node1))), and(con(node2, node3), and(con(node1, node3), con(node2, node4))))$

To this aim, we have randomly removed a (fixed) link from the network constraints. Nevertheless, constraints can be chosen wisely to limit the network topologies to those which are prone to lead to an erroneous situation, i.e., violation of a correctness property like loop freedom. However, it is also possible to check the model against all possible topologies by not defining any constraint. In other words, a modeler at first can focus on some suspicious network topologies and after resolving the raised issues it checks the model for all possible topologies. There are also some networks which have certain constraints about how the topology can change, e.g., node 1 can never get into the communication range of node 2. These restrictions on topology changes can be reflected through constraints too. The sizes of state spaces are compared under different network constraints resulting in different number of valid topologies. Eliminating τ-transitions and topology information manifestly reduces the number of states and transitions even when all possible topologies are not restricted. Therefore, it makes MANET protocol verification possible in an efficient manner. Note that in case the size of the network was increased from four to five, we couldn't generate its state space without applying reduction due to the memory limitation on a computer with 8GB RAM.

7.2. Tool support

The presented modeling language is supported by a tool[4], providing a number of options to generate the state space. A screen-shot of this tool is given in Fig. 18. This tool supports both bRebeca and wRebeca models characterized by different file types. After opening a model, the tool extracts the information of the reactive classes, such as the state variables and message servers, and also the main part including the rebec declarations and the network constraint.

[4] Available at https://github.com/b-yousefi/wRebeca.



```
            {
                //route is invalid, update the existing route
                if((dsn[oip_]==osn_ && hops[oip_][0]>hops_) || dsn[oip_]<osn_ )
                {
                    dsn[oip_]=osn_;
            {
                //route_state is valid and neigh_state[sip_] is not confrimed, new route wo
                    route_num = 0;
                    for(int i=0;i<5;i++)
```

Compile Configuration

Number of threads 100 heap size(mb) 512

[✓ Queue] Bag

[✓ Apply Reduction]

dts [✓ lts]

Ok Cancel

```
CurrentDirectory: C:\Users\Behnaz\Documents\GitHub\wRebeca\Examples
SelectedFile: C:\Users\Behnaz\Documents\GitHub\wRebeca\Examples\aodvv2version11.wrebe
configure compiling options
```

Fig. 18. A screen-shot of the wRebca tool with the *compilation info* window to configure the state-space generator

Then it generates several classes in the Java language based on the obtained information and compiles them together with some abstract and base classes (common in all models), for example *global state* and *topology*, to build an engine that constructs the model state space upon its execution. Before compiling, a user can decide about rebecs message processing method, in a FIFO manner (queue) or in an arbitrary way (bag), and if the reduction should be applied. To take advantage of all hardware capabilities, we have implemented our state-space generation algorithm in a multi-threaded way to leverage the power of multi-core CPUs.

During state-space generation, information about the state variables and transitions are stored as an LTS in the Aldebaran format[5]. This LTS can be evaluated by tools such as the *mCRL2* toolset[6]. For example, one can express desired properties in μ-calculus [MS03] and verify them. Also, as explained in Sect. 5, labels are extended with network constraints as defined in [GFM11] so that the reduced LTS can be model checked with respect to underlying topology [GAFM13].

7.3. Model checking of the AODV protocol properties

There are different ways to check a given property on a wRebeca model. Invariant properties can be evaluated while generating the state space by checking each reached global state against defined invariants. Furthermore, the resulting state space can be model checked by tools supporting Aldebaran format such as mCRL2 and CLTS model checker.

7.3.1. *Checking the loop freedom invariant*

Loop freedom is one of the well-known property which must hold for all routing protocols such as the AODV protocol.

[5] http://cadp.inria.fr/man/aldebaran.html.
[6] http://www.mcrl2.org/.

```
bool loop_freedom(des:int, cur:int, visited:Set<int>){
    for(int i=0; i<n; i++)
        if ((state.node(cur).nhops[des][i]!=-1) && (!visited.contains(state.node(cur).nhops[des][i]))
                && loop_freedom(des,i,visited.add(i)))
            return true;
        else
            return false;
}
```

Fig. 19. Checking loop freedom property on a global state: we have used a dot notation to access the array *nhops* of the rebec with the identifier i, i.e., $state.node(i)$, where $state$ is the newly generated global state

For example, consider the routes to a destination x in the routing tables of all nodes, where $node_0$ has a route to x with the next hop $node_1$, $node_1$ has a route to x with the next hop $node_2$, and $node_2$ has a route to x with the next hop $node_0$. The given example constructs a loop which consists of the three nodes, $node_0$, $node_1$, and $node_2$. A state is considered *loop free* if the collective routing table entries of all nodes for each pair of a source and destination do not form a loop. As it was mentioned earlier in AODV-v2-11, each route may have more than one next hop when the adjacency states of the next hops are *unconfirmed*. Therefore, while the loop freedom of a state is checked, one must take into account all next hops stored for each route. Then, for each next hop it must be checked whether it leads to a loop or not. A routing protocol deployed on a network is called loop free if all of its reachable states are loop free. In other words, loop freedom property of a protocol is an invariant (which can be easily specified by the ACTL-X fragment of μ-calculus, and hence, is preserved by the reduced semantic model). However, We have extended our state-space generator engine to check invariants (specified by functions) over each newly generated global state on the fly by calling the functions provided by a user, i.e., the invariants. To this aim, we have specified the loop freedom invariant by a recursive function, to inspect for a given global state whether the next hops in the routing table entries of nodes collectively lead to a loop-formation scenario, as shown in Fig. 19. Therefore, whenever the state-space generator reaches to a new state, before proceeding any further, it checks whether any loop is formed on the forward/backward routes between the source and destination, by calling $loop\_freedom(4, 1, \text{new Set}\langle int\rangle(1))$ and $loop\_freedom(1, 4, \text{new Set}\langle int\rangle(4))$, as $node_4$ and $node_1$ are the destination and source respectively. If the loop freedom condition is violated, the $loop\_freedom$ function returns $false$, and the state generator engine stops while it returns the path which has led to the global state under consideration as a counter example. The function $loop\_freedom$ has three parameters: des refers to the destination of the route, cur refers to the IP address of the current node which is going to be processed and $visited$ is the list of IP addresses of those nodes which have been processed.

Although keeping more than one next hop for each route may increase the route availability, it compromises its validity by violating the loop freedom invariant in a network of at least four nodes with a dynamic topology. Consider the network topology shown in Fig. 2a. The following scenario explains steps that lead to the invariant violation.

1. $node_2$ initiates a route discovery procedure for destination $node_3$ by broadcasting a *rreq* message.
2. $node_1$ and $node_4$ upon receiving the *rreq* message, add a route to their routing tables towards $node_2$ and store $node_2$ as their next hop. Since it is the first time that these nodes have received a message from $node_2$, the neighbor state of $node_2$ is set to *unconfirmed*. Therefore, the route state is *unconfirmed*.
3. As $node_1$ and $node_4$ are not the intended destination of the route request, they rebroadcast the *rreq* message.
4. $node_1$ receives the *rreq* message sent by $node_4$ and since the route to $node_2$ is *unconfirmed* it adds $node_4$ as a new next hop to $node_2$.
5. $node_4$ also adds $node_1$ as the new next hop towards $node_2$ after processing the *rreq* sent by $node_1$. At this point a loop is formed between $node_1$ and $node_4$.
6. $node_3$ receives the *rreq* message sent by $node_1$ and since it is the destination, it sends a *rrep* message towards $node_1$.
7. $node_2$ moves out of the communication ranges of $node_1$ and $node_4$.
8. $node_1$ receives the *rrep* message sent by $node_3$ and as the route state towards $node_2$ is *unconfirmed* it unicasts the *rrep* message one by one to the existing next hops, $node_2$ and $node_4$, till it gets an ack.

$$\forall x, y : \mathbb{N}((x > 0 \land x < 4 \land y > 0 \land y < x) \Rightarrow [src\_sn(x).true^*.src\_sn(y)]false)$$

$$\begin{aligned}\forall x, y, m, n : \mathbb{N}(&x \geq 0 \land x < 4 \land y \geq 0 \land y < 4 \land\\ &m \geq 0 \land m < 4 \land n \geq 0 \land n < 4 \land\\ &(m < x \lor n < y)) \Rightarrow [true^*.info\_i\_dsn(x, y).true^*.info\_i\_dsn(m, n)]false)\end{aligned}$$

Fig. 20. μ-calculus properties verified by mCRL2

Since $node_2$ has moved out of the communication ranges of $node_1$, no ack is received from $node_2$ and $node_2$ gets removed from the routing table as the next hop to $node_2$. Then, another *rrep* is sent to $node_4$. Since $node_4$ is adjacent to $node_1$, it receives the message and then sends an ack to $node_1$. Therefore, $node_1$ sets the neighbor state of $node_4$ to *confirmed* and subsequently the route state towards $node_2$ to *valid*.

9. $node_4$ by receiving the *rrep* message from $node_1$ unicasts it to its next hops $node_1$ and $node_2$ similar to $node_1$. Since it fails to receive an ack from $node_2$ and receives one from $node_1$, it updates its routing table by validating $node_1$ as its next hop to $node_2$.

We have found the scenario in the wRebeca model with the network constraint resulting four topologies as indicated in Table 4. However, this scenario was also found for all the network constraints described in the table. Furthermore, we can generalize the scenario to all networks with the same connectivity when the communications occur, and the same mobility scenario.

7.3.2. *Checking the properties by mCRL2*

Sequence numbers are used frequently by the AODV protocol to evaluate the freshness of routes. Therefore, it is important that each node's sequence number increases monotonically. To this end, we manually configured the state generator to add two self-loops to each state with the label $src\_sn(x)$ to monitor the sequence number of the source node, where x is *sn* of the source node, and the label $info\_i\_dsn(y, z)$ to trace the destination sequence number of routes to the source and destination for each node i (i.e., the backward and forward routes to the destination of our model), where y and z are $dsn[src]$ and $dsn[dst]$ of node i, respectively. These properties are expressed through the ACTL-X fragment of μ-calculus as shown in Fig. 20. The first formula asserts a monotonic increase of the source sequence number. The second formula assures the destination sequence numbers stored in the routing table of $node_i$ are increased monotonically, and must hold for the nodes in the model.

7.3.3. *Checking packet delivery property by the CLTS model checker*

The CLTS model checker can be used to express and verify interesting properties of MANET protocols dependent to the underlying topology specified in Constrained Action Computation Tree Logic (CACTL) [GAFM13], an extension of Action CTL [DV90]. The path quantifier *All* in CACTL is parametrized by a multi-hop constrain over the topology, which specifies the pre-condition required for paths of a state to be inspected. Therefore, a state satisfies $\mathbf{A}^{\mu}\varphi$ if its paths over which the multi-hop constraint μ holds, also satisfy φ. It also contains the two temporal operators *until* and *weak until* to specify the path formulae $\phi\,{}_{\chi}\mathbf{U}_{\chi'}\,\phi'$ and $\phi\,{}_{\chi}\mathbf{W}_{\chi'}\,\phi'$ to denote a path over which states satisfying ϕ are met by actions of χ until a state satisfying ϕ' is met by actions of χ' (in case of weak until, the state satisfying ϕ' can never be met).

The important property of *packet delivery* in routing or information dissemination protocols in the context of MANETs becomes: if there exists an end-to-end route (multi-hop communication path) between two nodes A and C *for a sufficiently long period of time*, then packets sent by A will eventually be received by C [FVGH+13]. To specify such the property, inspired from [FVGH+13] we revised our specification to include data packet handling (to forward the packet to its next hop towards the destination) in addition to the route discovery packets and their corresponding handlers. Therefore, whenever a node, source, discovers a route to an intended destination, it starts forwarding its data packet through the next hop specified in its routing table. The data packet is forwarded by intermediate nodes to their next hops. When the data packet reaches the intended destination, it delivers the data to itself by unicasting the $deliver$ message to itself. In case an intermediate node fails to forward the message, the error recovery procedure is followed as explained in Sect. 6. Consequently, using the following formula, we can verify packet delivery property:

$$\mathbf{A}^{true}(true\;{}_{\neg rec\_newpkt(0,4)}\mathbf{W}_{rec\_newpkt(0,4)}\;\mathbf{A}^{n_1 \dashrightarrow n_4 \land n_4 \dashrightarrow n_1}(true\;{}_{\tau}\mathbf{U}_{deliver()}\;true))$$

It expresses that as long as there is a stable multi-hop path from n_1 to n_4 and vice versa (specified by $n_1 \dashrightarrow n_4 \wedge n_4 \dashrightarrow n_1$), any $rec\_newpkt(0, 4)$ message is proceeded by a $delivery()$ message after passing τ-transitions which abstract away from other message communications. By model checking the resulting CLTS of the AODVv2 model, we found a scenario in which the property does not hold. We explain this scenario in a network of three nodes N_1, N_2 and N_3, where node N_3 is always connected to the nodes N_1 and N_2, while the connection between the nodes N_1 and N_2 is transient. Therefore, the mobility of nodes leads to the topologies shown in Fig. 10b and Fig. 10c. Assume the topology is initially as the one in Fig. 10b:

- Node N_1 unicasts a $rec\_newpkt(data, N_2)$ to itself, indicating that it wants to send $data$ to node N_2.
- Node N_1 initiates a route discovery procedure by broadcasting an $rreq_{N_1,0}$ message to its neighbors, i.e., nodes N_3 and N_2. Note that $rreq_{a,i}$ refers to an $rreq$ message received from node a with the hop count of i. Each *rreq* message has more parameters but here only these two parameters are of interest and the other parameters are assumed to be equal for all the *rreq* messages, i.e., the destination and source sequence numbers, and the source and destination IP addresses.
- Node N_3 processes the $rreq_{N_1,0}$ and since it is not the destination and has no route to N_2 in its routing table, rebroadcasts the $rreq_{N_3,1}$ message to its neighbors, nodes N_1 and N_2, after increasing the hop count. At this point, node N_2 has two messages in its queue, $rreq_{N_1,0}$ and $rreq_{N_3,1}$.
- Node N_1 moves out of the communication range of node N_2, resulting the network topology shown in Fig. 10c.
- Node N_2 takes $rreq_{N_1,0}$ from the head of its queue and updates its routing table by setting N_1 as the next hop in the route towards N_1. As node N_2 is the intended destination for the route discovery message, it unicasts an $rrep$ message towards the originator, N_1, indicating that the route has been built and it can start forwarding the data. Therefore, node N_2 attempts to unicast an $rrep$ message to node N_1, i.e., its next hop towards the originator.
- Since the connection between the nodes N_1 and N_2 is broken, it fails to receive an ack from N_1 and marks the route as *invalid*.
- Node N_2 takes $rreq_{N_3,1}$ from its queue and since the route state towards N_1 is *invalid*, it evaluates the received route to determine whether it is loop free. Updating the routing table with the received route is said to be “loop free", if the received message cost, e.g., the hop count is less than or equal to the existing route cost. Since the hop count of the received message is greater than the existing one, it does not update the existing route and the message is discarded.

Although the route through node N_3 to node N_1 seems to be valid, the protocol refuses to employ it to prevent possible loop formation in the future.

8. Related work

A large number of studies has been conducted for modeling and verification of MANET protocols using different approaches to tackle its specific challenges. These challenges, as discussed in Sect. 3, are modeling the underlying topology, mobility and local broadcast.

Some works model and analyze the correctness of MANET protocols using existing formal frameworks such as SPIN [DRA04, WPP04] and UPPAAL [FvGH⁺12, MF06, WPP05]. In a SPIN model, node connectivity is modeled with the help of PROMELA channels, one for each node. Also, mobility is modeled by *case selection* instruction provided by PROMELA, for modeling nondeterminism. In the initialization section, possible links to other neighbors are defined as different cases that all will be checked for a model. Since it does not provide a specific technique to reduce the state space, its state space grows very fast and it is only feasible to check small topologies. Therefore, models would be limited to fewer nodes. In UPPAAL, connectivity is modeled through a set of arrays of booleans, while changing topology is modeled by a separate automaton which manipulates the arrays. In [WPP04], a case study was carried out to evaluate two model checkers, SPIN and UPPAAL. Due to state-space explosion, the analysis was limited to some special mobility scenarios (as a part of the specification). However, our reduction technique makes it possible to verify a MANET for all possible topology changes to find an error.

As explained in [EM99], from a theoretical point of view, compositionality is not preserved if broadcast is encoded based on point-to-point communications. Lack of support for compositional modeling and arbitrary topology changes has motivated new approaches with a primitive for local broadcast and support of arbitrary mobility in an algebraic way. These approaches include CBS# [NH06], CWS [MS06], CMN [Mer09], the ω-calculus

[SRS10], bAπ [God10], CMAN [God07, God09], RBPT [GFM08] and the bpsi-calculi [BHJ+15, PBPR13]. Each of these proposed frameworks overcome the modeling difficulties such as local broadcast and its message delivery guarantee property and mobility in different ways. All these approaches, except CBS#, CWS, AWN and bpsi-calculi, suffer from lack of message delivery guarantee that makes them inappropriate for analyzing properties such as packet delivery [FVGH+13]. They model broadcast through either an enforced synchronized or lossy communication. When communications are lossy, a node may not receive a message although it is in the transmission range of the sender. CBS# and CWS use enforced synchronization for broadcast to make sure that all ready nodes within the transmission range of a sender will receive the message. Although they guarantee message delivery to the ready receivers, it is not possible to define meaningful nodes (which can successfully receive messages while they are processing another message) in their syntax which are always ready (i.e, *input-enabled*) [FVGH+13]. The process algebra AWN is proposed particularly for modeling wireless mesh network (WMN) routing protocols which uses local broadcast with message delivery guarantee. It defines its own data structures to model routing tables and other necessary data types to model the AODV protocol. In addition, conditional unicast is introduced for modeling the procedure to act based on the message delivery acknowledgment. In all these approaches, while a node is busy processing a message, it fails to receive messages from other nodes. Therefore, either nodes are defined to be input-enabled at the semantics as in CBS# and CWS or a process with a queue that concurrently stores new messages should be specified at the syntax as in AWN and bpsi-calculus. Almost all these languages model mobility of nodes in their semantics through arbitrary changes of the topology with the exception that it is modeled through different generations of assertions on connectivity information in [BHJ+15]. In wRebeca, communications are asynchronous and received messages are stored in queues implicitly at the semantic level (without the need to make nodes explicitly input-enabled). Furthermore, the atomic execution of message handlers, which avoids unnecessary interleaving of the node behaviors, together with topology abstraction through τ-elimination technique, where the topology changes are a source of state-space explosion in the process calculi approaches, make our framework applicable to the verification of real-world yet complex protocols such as AODV. We remark that unnecessary interleaving of behaviors can be handled in bpsi-calculus by means of priorities.

There are different approaches [DSTZ12, DSZ11, ADR+11] with the aim to analyze networks with an infinite number of nodes, where nodes execute an instance of a network process. A network configuration is represented as a graph in which each individual node represents a state of the process. The behavior of a process is modeled by an automaton. The network configuration transforms due to either the process evolution at a network node or the topology reconfiguration. Verification of safety properties, reaching to an undesirable configuration starting from an initial configuration, is parameterized due to any possible number of nodes and connectivity among them. It is proved that the problem of parameterized safety properties, the so-called *control states reachability problem*, is undecidable. However, that problem turns out to be decidable for the class of bounded path graphs [DSZ11, DSTZ12]. Decidability of the problem was also considered when configurations evolve due to discrete/continuous clocks at processes [ADR+11]. Furthermore, an inductive approach based on reduction to prove compositional invariants for the dynamic process networks was presented in [NT15a]. This approach reduces the calculation of a compositional invariant to a smallest representative network through setting up a collection of local symmetry relations between nodes, specifically defined for each problem. The computed non-dynamic compositional invariant on the representative network is generalized for the entire dynamic network family when the non-dynamic invariant is preserved by any reaction to a dynamic change in the network. Then, they proved loop freedom of AODVv2-04 for an arbitrary number of nodes in [NT15b] through an inductive and compositional proof: It provides an inductive invariant and proves that it is held initially and also preserved by every action, either a protocol action or a change in the network, similar to the approach of [vGHPT16]. They have reported two loop-formation scenarios due to inappropriate setting of timing constants and accepting any valid route when the current route is broken without any further evaluation (to ensure loop formation). Another approach is based on graph transformation systems, where network configurations are hypergraphs and transitions are specified by graph rewriting rules, modeling the dynamic behavior of a protocol. Safety properties are symbolically specified by graph patterns, a generalized form of hypergraphs with negative conditions, through a *symbolic backward reachability analysis* which is not guaranteed to terminate due to the undecidability of the problem [SWJ08]. To this aim, an over-approximation of the set of configurations preceding a bad configuration are computed by using a fixed point analysis, and then check that this set contains no initial configuration. While these approaches are scalable to prove a property for MANETs with a potentially unbounded number of nodes with an exhaustive effort, our approach is valuable to easily examine confirmation and diagnostics of suspected errors in the early phase of protocol development for a limited number of nodes. In other words, our efficient model checking tool can be used as an initial step before involving to generalize a property for an arbitrary sized network.

9. Conclusion and future work

In this paper we extended the syntax and semantics of bRebeca, the actor-based modeling language for broadcasting environment, to support wireless communication in a dynamic environment. We addressed the key features of wireless ad hoc networks, namely reliable local broadcast, conditional unicast, and last but not least mobility. The reliable asynchronous local broadcast/unicast communication, and implicit support of message storages make our framework suitable to analyze MANETs with respect to different mobility scenarios. A modeler only focuses on how to decompose a protocol into a set of communicating actors to cover functionalities of the protocol under investigation.

To overcome the state-space explosion, we leveraged the counter abstraction technique to analyze ad hoc networks with static topologies. Our reduction technique performs well on protocols with no specific state variable that distinguishes each rebec, and topologies with many topologically equivalent nodes. We demonstrated the effectiveness of our approach on the flooding protocol in different network settings. However, mobility ruins the soundness of our counting abstraction. To this end, we eliminated τ-transitions while topology information was removed from the global states to considerably reduce the size of the state space. We integrated the proposed reduction techniques into a tool customizable in verifying wRebeca models for different message storage policies and the topology dynamism. Invariants can be checked during the state-space generation while the resulting output can be fed into the existing model checking tools such as mCRL2 and CLTS model checker.

We presented a complete and accurate model of the core functionalities of a recent version of AODVv2 protocol (version 11). We abstracted optional features and timing aspects to make our model manageable. We verified the loop freedom property in AODVv2-11 and found a scenario in which the property is violated. The scenario was confirmed by the AODV group. Loop freedom has already been proved on various versions of AODV: AODVv1-02 [BOG02], AODV-rfc3561 [FVGH+13, vGHPT16], and AODVv2-04 [NT15b, SWJ08], respectively. Among these only [NT15b] considers the timed behavior of the AODV. The new version differs in the following aspects which distinguish our attempt: in this version multiple next hops are maintained for each destination and consequently the process to update the routing table is completely different; Different statuses are considered for a route in the table of a node regarding the neighbor status of its next hop; Sequence numbers for invalid destinations in intermediate nodes are not increased anymore (like [NT15b, SWJ08], in contrast to others). Although, these approaches focus on providing a general proof for the property, our model checking-based approach detects the error and the scenario that leads to it. Our approach, can be adopted to resolve conceptual/design errors in an iterative way in the early phase of protocol development. The positive result of verifications constitutes a predicate about the protocol for a limited number of nodes. The combination of model checking and theorem proving techniques allows to prove a predicate about a MANET protocol for any number of nodes.

We plan to integrate our state-space generator tool into the verification environment [afra] to take advantage of its model checker. Furthermore, we aim to run more case studies to extend application of our framework. To analyze real-time and probabilistic behaviors of wireless network protocols, wRebeca can be extended in the same way of [KSS+15, VK12]. To this aim, there is a need to examine the soundness of our reduction techniques when probability and time are introduced.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments on the earlier version of the paper, the AODV group for their supports, Mohammad Reza Mousavi for his discussion on the paper, Wan Fokkink and Bas Luttik for their helpful comments on the paper.

References

[ADR$^+$11] Abdulla PA, Delzanno G, Rezine O, Sangnier A, Traverso R (2011) On the verification of timed ad hoc networks. In: 9th international conference on formal modeling and analysis of timed systems, volume 6919 of LNCS, Springer, pp 256–270

[afra] Rebeca formal modeling language. http://www.rebeca-lang.org/wiki/pmwiki.php/Tools/Afra

[Agh90] Agha GA (1990) ACTORS—a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence. MIT Press, Cambridge, MA

[BG92] Bertsekas DP, Gallager RG (1992) Data networks. Prentice Hall, Upper Saddle River, NJ

[BHJ$^+$15] Borgström J, Huang S, Johansson M, Raabjerg P, Victor B, Pohjola JÅ, Parrow J (2015) Broadcast psi-calculi with an application to wireless protocols. Softw Syst Model 14(1):201–216

[BMWK09] Basler Gérard, Mazzucchi Michele, Wahl Thomas, Kroening Daniel (2009) Symbolic counter abstraction for concurrent software. In *Computer Aided Verification*, Springer pp 64–78

[BOG02] Bhargavan K, Obradovic D, Gunter CA (2002) Formal verification of standards for distance vector routing protocols. J ACM 49(4):538–576

[CCH07] Cui T, Chen L, Ho T (2007) Distributed optimization in wireless networks using broadcast advantage. In: Decision and control. IEEE, pp 5839–5844

[CEJS98] Clarke EM, Emerson EA, Jha S, Sistla AP (1998) Symmetry reductions in model checking. In: Hu AJ, Vardi MY (eds) Computer aided verification. Springer, Berlin, pp 147–158

[DK86] Dechter R, Kleinrock L (1986) Broadcast communications and distributed algorithms. IEEE Trans Comput 35(3):210–219

[DRA04] De Renesse R, Aghvami AH (2004) Formal verification of ad-hoc routing protocols using spin model checker. In 12th IEEE mediterranean, electrotechnical conference, volume 3. IEEE, pp 1177–1182

[DSTZ12] Delzanno G, Sangnier A, Traverso R, Zavattaro G (2012) On the complexity of parameterized reachability in reconfigurable broadcast networks. In: Annual conference on foundations of software technology and theoretical computer science, volume 18 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp 289–300

[DSZ11] Delzanno G, Sangnier A, Zavattaro G (2011) Parameterized verification of safety properties in ad hoc network protocols. In: First international workshop on process algebra and coordination, volume 60 of EPTCS, pp 56–65

[DV90] De Nicola R, Vaandrager FW (1990) Action versus state based logics for transition systems. In: Semantics of systems of concurrent processes, volume 469 of Lecture notes in computer science. Springer, pp 407–419

[EM99] Ene C, Muntean T (1999) Expressiveness of point-to-point versus broadcast communications. In: Ciobanu G, Păun G (eds) Fundamentals of computation theory. FCT 1999, volume 1684 of LNCS. Springer, Berlin

[ET99] Emerson EA, Trefler RJ (1999) From asymmetry to full symmetry: new techniques for symmetry reduction in model checking. In: Pierre L, Kropf T (eds) Correct hardware design and verification methods. Springer, Berlin, pp 142–156

[FvGH$^+$12] Fehnker A, van Glabbeek R, Höfner P, McIver A, Portmann M, Tan WL (2012) Automated analysis of AODV using Uppaal. In: Tools and algorithms for the construction and analysis of systems, volume 7214 of LNCS. Springer, Berlin, pp 173–187

[FVGH$^+$13] Fehnker A, Van Glabbeek R, Höfner P, McIver A, Portmann M, Tan WL (2013) A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. arXiv preprint arXiv:1312.7645

[GAFM13] Ghassemi F, Ahmadi S, Fokkink W, Movaghar A (2013) Model checking MANETs with arbitrary mobility. In: Arbab F, Sirjani M (eds) Fundamentals of software engineering. Springer, Berlin, pp 217–232

[GFM08] Ghassemi F, Fokkink W, Movaghar A (2008) Restricted broadcast process theory. In: Sixth IEEE international conference on software engineering and formal methods (SEFM). IEEE Computer Society, pp 345–354

[GFM11] Ghassemi F, Fokkink W, Movaghar A (2011) Verification of mobile ad hoc networks: an algebraic approach. Theor Comput Sci 412(28):3262–3282

[God07] Godskesen JC (2007) A calculus for mobile ad hoc networks. In: Murphy AL, Vitek J (eds) Coordination models and languages, volume 4467 of LNCS. Springer, Berlin, pp 132–150

[God09] Godskesen JC (2009) A calculus for mobile ad-hoc networks with static location binding. Electr Notes Theor Comput Sci 242(1):161–183

[God10] Godskesen JC (2010) Observables for mobile and wireless broadcasting systems. In: Coordination models and languages, volume 6116 of LNCS. Springer, Berlin, pp 1–15

[Hew77] Hewitt C (1977) Viewing control structures as patterns of passing messages. Artif Intell 8(3):323–364

[JSM$^+$10] Jaghoori MM, Sirjani M, Mousavi MR, Khamespanah E, Movaghar A (2010) Symmetry and partial order reduction techniques in model checking Rebeca. Acta Inform 47(1):33–66

[Kat11] Katoen J-P (2011) Model checking: one can do much more than you think! In: Arbab F, Sirjani M (eds) Fundamentals of software engineering. Springer, Berlin, pp 1–14

[KLN11] Kuhn F, Lynch NA, Newport CC (2011) The abstract MAC layer. Distrib Comput 24(3–4):187–206

[KSS$^+$15] Khamespanah E, Sirjani M, Sabahi-Kaviani Z, Khosravi R, Izadi M (2015) Timed rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. Sci Comput Program 98:184–204

[Mer09] Merro M (2009) An observational theory for mobile ad hoc networks (full version). Inf Comput 207(2):194–208, Special issue on Structural Operational Semantics (SOS)

[MF06] Mclver AK, Fehnker A (2006) Formal techniques for the analysis of wireless networks. In: Second international symposium on leveraging applications of formal methods, verification and validation. IEEE, pp 263–270

[MKKAR06] Mahmud SA, Khan S, Khan S, Al-Raweshidy H (2006) A comparison of manets and wmns: commercial feasibility of community wireless networks and manets. In: 1st international conference on access networks. ACM

[MS03] Mateescu R, Sighireanu M (2003) Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Sci Comput Program 46(3):255–281

[MS06] Mezzetti N, Sangiorgi D (2006) Towards a calculus for wireless systems. Electr Notes Theor Comput Sci 158(0):331–353

[NH06] Nanz S, Hankin C (2006) A framework for security analysis of mobile wireless networks. Theor Comput Sci 367(1–2):203–227

[NT15a] Namjoshi KS, Trefler RJ (2015a) Analysis of dynamic process networks. In: Baier C, Tinelli C (eds) Tools and algorithms for the construction and analysis of systems, volume 9035 of LNCS. Springer, Berlin, pp 164–178

[NT15b] Namjoshi KS, Trefler RJ (2015b) Loop freedom in aodvv2. In: Graf S, Viswanathan M (eds) Formal techniques for distributed objects, components, and systems, volume 9039 of LNCS. Springer, Cham, pp 98–112

[PB99] Perkins CE, Belding-Royer EM (1999) Ad-hoc on-demand distance vector routing. In: 2nd workshop on mobile computing systems and applications. IEEE Computer Society, Washington, DC, pp 90–100

[PBPR13] Pohjola JÅ, Borgström J, Parrow J, Raabjerg P (2013) Negative premises in applied process calculi. Technical report, Department of Information Technology, Uppsala University

[Pen08] Peng J (2008) A new arq scheme for reliable broadcasting in wireless lans. IEEE Commun Lett 12(2):146–148

[Plo81] Plotkin GD (1981) A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus

[PXZ02] Pnueli A, Xu J, Zuck LD (2002) Liveness with (0, 1, infty)-counter abstraction. In: 14th international conference on computer aided verification, CAV '02, Springer, pp 107–122

[RSA$^+$14] Reynisson AH, Sirjani M, Aceto L, Cimini M, Jafari A, Ingólfsdóttir A, Sigurdarson SH (2014) Modelling and simulation of asynchronous real-time systems using Timed Rebeca. Sci Comput Program 89:41–68

[SJ11] Sirjani M, Jaghoori MM (2011) Ten years of analyzing actors: Rebeca experience. In: Agha G, Meseguer J, Danvy O (eds) Formal modeling: actors, open systems, biological systems. Springer, Berlin, pp 20–56

[SK13] Sabouri H, Khosravi R (2013) Delta modeling and model checking of product families. In: Arbab F, Sirjani M (eds) Fundamentals of software engineering. Springer, Berlin, pp 51–65

[SL04] Si W, Li C (2004) RMAC: A reliable multicast MAC protocol for wireless ad hoc networks. In: 33rd international conference on parallel processing (ICPP 2004). IEEE Computer Society, pp 494–501

[SMSdB04] Sirjani M, Movaghar A, Shali A, de Boer FS (2004) Modeling and verification of reactive systems using Rebeca. Fundam Inform 63(4):385–410

[SRS10] Singh A, Ramakrishnan CR, Smolka SA (2010) A process calculus for mobile ad hoc networks. Sci Comput Program 75(6):440–469

[SS10] Sabouri H, Sirjani M (2010) Slicing-based reductions for rebeca. Electr Notes Theor Comput Sci 260:209–224

[SWJ08] Saksena M, Wibling O, Jonsson B (2008) Graph grammar modeling and verification of ad hoc routing protocols. In: 14th international conference on tools and algorithms for the construction and analysis of systems, volume 4963 of LNCS. Springer, pp 18–32

[vGHPT16] van Glabbeek RJ, Höfner P, Portmann M, Tan WL (2016) Modelling and verifying the AODV routing protocol. Distrib Comput 29(4):279–315

[vGW96] van Glabbeek R, Weijland WP (1996) Branching time and abstraction in bisimulation semantics. J ACM 43(3):555–600

[VK12] Varshosaz M, Khosravi R (2012) Modeling and verification of probabilistic actor systems using pRebeca. In: Aoki T, Taguchi K (eds) Formal methods and software engineering. Springer, Berlin, pp 135–150

[WPP04] Wibling O, Parrow J, Pears A (2004) Automatized verification of ad hoc routing protocols. In: de Frutos-Escrig D, Núñez M (eds) Formal techniques for networked and distributed systems, volume 3235 of LNCS. Springer, Berlin, pp 343–358

[WPP05] Wibling O, Parrow J, Pears A (2005) Ad hoc routing protocol verification through broadcast abstraction. In: Wang F (ed) Formal techniques for networked and distributed systems-FORTE 2005. Springer, Berlin, pp 128–142

[YGK15] Yousefi B, Ghassemi F, Khosravi R (2015) Modeling and efficient verification of broadcasting actors. In: In pre-proceeding of 6th IPM international conference on fundamentals of software engineering, pp 114–128

Received 30 April 2016
Accepted in revised form 13 March 2017 by Joachim Parrow
Published online 19 April 2017