Modeling and Analyzing Mobile Ad hoc Networks in Real-Time Maude

Si Liu[a,*], Peter Csaba Ölveczky[b,*], José Meseguer[a]

[a]*Department of Computer Science, University of Illinois at Urbana-Champaign*
[b]*Department of Informatics, University of Oslo*

Abstract

Modeling and analyzing Mobile Ad-hoc Networks (MANETs) pose non-trivial challenges to formal methods. Time, geometry, communication delays and failures, mobility, and uni- and bidirectional wireless communication can interact in unforeseen ways that are hard to model and analyze by current process calculi and automatic formal methods. As a consequence, current analyses tend to abstract away these physical aspects, so that—although still quite useful in finding various errors—their simplifying assumptions can easily fail to model details of MANET behavior relevant to meet desired requirements. In this work we present a formal framework for the modeling and analysis of MANETS based on Real-Time Maude to address this challenge. Specifically, we show that our framework has good expressive power to model relevant aspects of MANETs, and good compositionality properties, so that a MANET protocol can be easily composed with various models of mobility and with other MANET protocols. We illustrate the use of our framework on two well-known MANET benchmarks: the AODV routing protocol and the leader election protocol of Vasudevan, Kurose, and Towsley. Our formal analysis has uncovered a spurious behavior in the latter protocol that is due to the subtle interplay between communication delays, node movement, and neighbor discovery. This behavior therefore cannot be found by analyses that abstract from node movement and communication delays.

Keywords: Mobile ad-hoc networks, formal specification, model checking, Real-Time Maude.

1. Introduction

Mobile Ad-hoc Networks (MANETs) are self-configuring networks made up of mobile nodes (laptops, smart phones, vehicles, sensors, etc.) connected by wireless links. They are increasingly popular and well suited for deployment in ad-hoc environments. They have an extensive range of applications, including wireless sensor networks, ambient intelligence, personal area networks and wireless local area networks, cooperating "smart" cars, alternative communication infrastructure for emergency response during accidents and natural disasters, which may disable other existing infrastructure, and so on.

The formal analysis of MANETs is challenging, because the relevant requirements for correctness and performance are themselves non-trivial and go beyond the usual requirements for standard network protocols. In particular, both mobility and wireless communication under mobility are essential for MANETs and need to be seriously taken into account when analyzing them under realistic patterns of node movement. Only thus can MANET protocol designers reasonably determine whether or not a MANET protocol implementation will be useful and will meet its desired requirements.

1.1. Formal Modeling and Analysis Challenges

Both mobility and wireless communication in MANETs depend on *physical* characteristics such as: (i) geometric *location* of nodes; (ii) *speed* and *direction* of mobile nodes; (iii) wireless *transmission ranges* based on each node's battery *power* and *distance* between nodes; and (iv) *communication delays* in both the

*Corresponding author

sender and the receiver. Therefore, although usually not described that way, MANETs are in fact *cyber-physical systems* (CPSs). This means that their formal modeling, and the formal verification of their relevant requirements must sufficiently address essential physical characteristics such as mobility and communication and their interactions. For example, a moving node may be within range of another node when the other node started a message send, but may have already moved out of such a range by the time when the message is actually sent. Furthermore, as usual with many other CPSs, there is no easy separation between *correctness requirements*, including safety-critical ones, and *physical behavior*, since timing, distance, motion, and other physical, quantitative aspects may be essential for correct behavior.

All this means that the formal modeling and analysis of MANETs presents a number of nontrivial challenges, including:

1. The need to model node movement realistically.

2. Modeling communication. There is a subtle interaction between wireless communication, which typically is restricted to distances of between 10 and 100 meters, and node mobility. For example, nodes may move into, or out of, the sender's transmission range *during* the communication delay; furthermore, the sender may itself move during the communication. Modeling communication in MANETs is therefore challenging for process languages, which are usually based on fixed communication primitives.

3. Since the communication topology of the network depends on the *physical locations* of the nodes, such locations must be taken into account in the model. However, if not handled carefully this can lead to very large state spaces, which can makes direct model checking analysis unfeasible.

We discuss related work on the formal analysis of MANETs in much more detail in Section 6. Such work includes research where MANETS are expressed in the languages of various model checkers, e.g., [4, 8, 12, 19], and approaches representing MANETs in various process calculi, e.g., [34, 35, 17, 29, 46, 15, 18, 30, 11, 13, 46, 16, 45, 31, 21, 47, 48, 28, 26, 27, 51, 52, 53]. We can summarize our more detailed discussion in Section 6 by stating that in prior work on the formal analysis of MANETs there is still a substantial gap between a more abstract modeling level—at which various physical aspects are omitted—and the actual level at which MANETs need to be analyzed to take into account those physical aspects essential to ensure that relevant requirements are met. Of course, any actual errors found even at a more abstract level are still very valuable. The main issue, however, is that other realistic potential problems may be easily abstracted away when they are not reflected in the given formal model. This fact is illustrated by our analysis in Section 5 of the well-known leader election algorithm by Vasudevan, Kurose, and Towsley [50], where our analysis has uncovered a spurious behavior that is due to a very subtle interplay between node movement, communication delays, and neighbor discovery. This problematic behavior therefore cannot be found using standard formal analysis methods that abstract away node movement and communication delays.

1.2. An Expressive Formal Modeling and Analysis Framework for MANETS

To meet above challenges (1)—(3) a suitable formal and analysis framework for MANETS is needed. To the best of our knowledge this is not yet available in prior work, and is therefore a key motivation behind the present work. We see: (i) *expressiveness* to meet challenges (1)—(3); and (ii) *compositionality* as two main requirements that such a formal framework should meet. Requirement (i) is obvious from the prior discussion. The need for requirement (ii) is amply illustrated throughout the paper but deserves some explanation. The key point is that, typically, the formal requirements of a MANET protocol will need to be analyzed under various assumptions such as:

- various *mobility models*;

- various *wireless communication models*, such as unidirectional or bidirectional communication; and

- possibly in composition with other auxiliary protocols.

To put it briefly, to address relevant requirements formal analyses will almost never consider the given MANET protocol *in isolation*: they will need to consider it *in composition with* other formal models of mobility, communication, and of other MANET protocols. Compositionality, therefore, becomes a highly desirable feature of a MANET formal framework.

Our proposed answer to the *expressiveness* requirement for a MANET formal framework is the use of Real-Time Maude [38]. Because of its expressiveness and flexibility to define models of communication—and to model physical aspects such as geometric location, speed, and distance—as we show in the paper Real-Time Maude is well suited for formally modeling MANETs and, to the best of our knowledge, provides for the first time a reasonably detailed formal modeling framework for them. In particular, we formalize in Real-Time Maude:

- the most popular models for node mobility, and
- geographically bounded wireless communication, which takes into account the interplay between communication delay and mobility,

Concerning Challenge (3) above, we provide a partial answer, yet sufficient to analyze in detail and uncover substantial issues in MANET protocols such as AODV and LE that we discuss in the paper. To make model checking analysis feasible while faithfully representing the relevant physical aspects, we make our models parametric in aspects such as the possible velocities and directions a node can choose. However, even if a node moves slowly, given enough time it may still cover the entire area (and hence contribute to an unmanageable state space). To avoid this second problem we use Real-Time Maude's *time-bounded* model checking command, which allows us to analyze scenarios only up to a certain duration. Admittedly, a full answer to Challenge (3) will require developing suitable abstraction techniques for MANETs that we leave for future research. The point, however, is that, by providing a first reasonably detailed formal model of location-aware MANETs, this paper lays the foundations for developing such abstractions, so that they can still model the relevant features at a more abstract level.

Our proposed answer to the *compositionality* requirement is twofold. On the one hand, we specify all our models modularly and make them parametric; and then exploit Real-Time Maude's powerful composition features such as module hierarchies and object-oriented multiple class inheritance, to define suitable model compositions. On the other hand, we provide, to the best of our knowledge for the first time, a new *compositional methodology* for combining various models of real-time systems in such a way that the time behaviors of each model can be specified in isolation, but can be seamlessly integrated when the various models are composed.

1.3. Our Contributions

We can summarize our contributions as follows:

1. We provide a formal framework for MANETS satisfying the expressiveness and compositionality requirements already discussed in Section 1.2.
2. We illustrate the practical usefulness of the framework and its main features by:
   (a) developing in detail various well-known models of node mobility, of wireless communication, and of the interaction between mobility and communication;
   (b) composing such models with two well-known MANET protocols, namely: (a) the widely used *Ad hoc On-Demand Distance Vector* [40] (AODV) routing protocol for MANETs developed by the IETF MANET working group; and (b) the well-known leader election (LE) protocol of Vasudevan, Kurose, and Towsley [50]; and
   (c) formally analyzing by model checking key properties of both AODV and LE.
3. We present a new compositional methodology for combining various models of real-time systems in such a way that the time behaviors of each model can be specified in isolation, but can be seamlessly integrated when composed, and exploit this methodology when combining MANET protocols with other MANET protocols and with various mobility models.

The present paper is based on two earlier conference papers, one discussing the modeling analysis results for the AODV protocol [24], and another discussing similar modeling and analysis results for LE [25]. Besides giving a more comprehensive account of, and a broader perspective on, the proposed formal framework and its features, and of giving considerably more technical details and explanations, the following contributions are new and cannot be found in those previous papers: (i) the formalization of the random direction mobility model; (ii) a new, formal specification of all the mobility models that makes the various computations much simpler and precise (staying within the rationals for all but one mobility model); and (iii) solving for the first time the problem of how the timed behaviors of multiple Real-Time Maude specifications—including MANET protocols and their mobility models—can be specified in isolation, yet can be seamlessly composed, thus obtaining the timed behavior of the desired composition. Besides being essential for the proposed MANET framework, this new method of combining timed behaviors is totally general and should be very useful for composing larger real-time specifications out of a family of reusable components in many other application areas.

The paper is organized as follows: Section 2 gives some background on Real-Time Maude, popular models for node mobility, and wireless communication in MANETs. Section 3 presents our framework for modeling MANETs in Real-Time Maude, and for composing multiple MANET protocol specifications and mobility models. Sections 4 and 5 illustrate how our framework has been used to mode and analyze the routing protocol AODV and the leader election protocol LE, respectively. Finally, Section 6 discusses related work and Section 7 gives some concluding remarks.

2. Preliminaries

This section gives some necessary background about: (i) Real-Time Maude (Section 2.1), (ii) the most popular models of node mobility used in MANET protocol evaluation (Section 2.2), and (iii) wireless communication in MANETs (Section 2.3).

2.1. Real-Time Maude

Real-Time Maude [38, 36] is a language and tool that extends Maude [9] to support the formal specification and analysis of real-time systems. The specification formalism is based on *real-time rewrite theories* [37]—an extension of *rewriting logic* [5, 32]—and emphasizes *ease* and *generality* of specification. Real-Time Maude specifications are executable under reasonable assumptions, and the tool provides a variety of formal analysis methods, including simulation, reachability analysis, and LTL and timed CTL model checking.

2.1.1. Rewriting Logic Specification in Maude

A *membership equational logic* (MEL) [33] *signature* is a triple $\Sigma = (K, \Sigma, S)$ with K a set of *kinds*, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$ a many-kinded signature, and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A Σ-*algebra* A consists of a set A_k for each kind k, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1\cdots k_n,k}$, and a subset inclusion $A_s \subseteq A_k$ for each sort $s \in S_k$. The set $T_{\Sigma,k}$ denotes the set of ground Σ-terms with kind k, and $T_\Sigma(X)_k$ denotes the set of Σ-terms with kind k over the set X of kinded variables.

A MEL *theory* is a pair (Σ, E) with Σ a MEL-signature and E a finite set of MEL sentences, which are either conditional equations or conditional memberships of the forms:

$$(\forall X)\; t = t' \;\textbf{if}\; \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j, \qquad (\forall X)\; t : s \;\textbf{if}\; \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j,$$

where $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$ for some kind $k \in \Sigma$, the latter stating that t is a term of sort s, provided the condition holds. In Maude, an individual equation in the condition may also be a *matching equation* $p_l := q_l$, which is mathematically interpreted as an ordinary equation. However, operationally, the new variables occurring in the term p_l become instantiated by matching the term p_l against the canonical form of the instance of q_l (see [9] for further explanations). Order-sorted notation $s_1 < s_2$ abbreviates the conditional membership $(\forall x : [s_1])\ x : s_2$ **if** $x : s_1$. Similarly, an operator declaration $f : s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall\, x_1 : [s_1], \ldots, x_n : [s_n])\ f(x_1, \ldots, x_n) : s$ **if** $\bigwedge_{1 \leq i \leq n} x_i : s_i$.

A Maude module specifies a *rewrite theory* [5, 32] of the form $(\Sigma, E \cup A, R)$, where: (i) $(\Sigma, E \cup A)$ is a membership equational logic theory specifying the system's state space as an algebraic data type with A a set of equational axioms (such as a combination of associativity, commutativity, and identity axioms), to perform equational deduction with the equations E (oriented from left to right) *modulo* the axioms A, and (ii) R is a set of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form:

$$l : q \longrightarrow r \ \textbf{if} \ \bigwedge_i p_i = q_i \ \wedge \ \bigwedge_j w_j : s_j \ \wedge \ \bigwedge_m t_m \longrightarrow t'_m,$$

where l is a *label*, and q, r are Σ-terms of the same kind. Intuitively, such a rule specifies a *one-step transition* from a substitution instance of q to the corresponding substitution instance of r, *provided* the condition holds; that is, that the substitution instance of each condition in the rule follows from $\mathcal{R}$.

We briefly summarize the syntax of Maude (see [9] for more details). Sorts and subsort relations are declared by the keywords `sort` and `subsort`, and operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ \ldots \ s_n$ are the sorts of its arguments, and s is its (value) *sort*. Operators can have user-definable syntax, with underbars '`_`' marking each of the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. An operator can also be declared to be a *constructor* (`ctor`) that defines the data elements of a sort. The `frozen` attribute declares which argument positions are *frozen*; arguments in frozen positions cannot be rewritten by rewrite rules [9].

There are three kinds of logical statements in the Maude language, *equations*, *memberships* (declaring that a term has a certain sort), and *rewrite rules*, introduced with the following syntax:

- equations: `eq` $u = v$ or `ceq` $u = v$ `if` *condition*;
- memberships: `mb` $u : s$ or `cmb` $u : s$ `if` *condition*;
- rewrite rules: `rl [`l`]:` u `=>` v or `crl [`l`]:` u `=>` v `if` *condition*.

An equation $f(t_1, \ldots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a term $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied.[1] The mathematical variables in such statements are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they have the form *var* : *sort*. Finally, a comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

2.1.2. Object-Oriented Specification in Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* $\mathcal{R} = (\Sigma, E \cup A, R)$ [37], where:

- $(\Sigma, E \cup A)$ contains an equational subtheory $(\Sigma_{TIME}, E_{TIME}) \subseteq (\Sigma, E \cup A)$, satisfying the *TIME* axioms that specifies sort `Time` as the time domain (which can be discrete or dense). Although a timed module is parametric on the time domain, Real-Time Maude provides some predefined modules specifying useful time domains. For example, the modules `NAT-TIME-DOMAIN-WITH-INF` and `POSRAT-TIME-DOMAIN-WITH-INF` define the time domain to be, respectively, the natural numbers and the nonnegative rational numbers, and contain the subsort declarations `Nat < Time` and `PosRat < Time`. The supersort `TimeInf` extends the sort `Time` with an "infinity" value `INF`.

[1]A specification with `owise` equations can be transformed to an equivalent system without such equations [9].

- The rules in R are decomposed into: (i) "ordinary" rewrite rules specifying the system's *instantaneous* (i.e., zero-time) local transitions, and (ii) *tick (rewrite) rules* that model the elapse of time in a system, having the form l : `{`t`}` $\xrightarrow{u}$ `{`t'`}` **if** *condition*, where t and t' are terms of sort `System`, u is a term of sort `Time` denoting the *duration* of the rewrite, and `{_}` is a built-in constructor of sort `GlobalSystem`. In Real-Time Maude, tick rules, together with their durations, are specified using the syntax

  `crl [`l`]: {`t`} => {`t'`} in time` u `if` *condition*.

The initial state must be reducible to a term `{`t_0`}`, for t_0 a ground term of sort `System`, using the equations in the specification. The form of the tick rules then ensures uniform time elapse in all parts of a system.

Real-Time Maude is particularly suitable to formally model distributed real-time systems in an object-oriented style. Each term t in a global system state `{`t`}` is in such cases a term of sort `Configuration` (which is a subsort of `System`), and has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator `_ _` (empty syntax) that is declared associative and commutative and having the `none` multiset as its identity element, so that rewriting is *multiset rewriting* supported directly in Maude.

In object-oriented timed modules one can declare *classes*, *subclasses*, and *messages*. A *class* declaration `class` C `|` att_1 `:` s_1`, . . . ,` att_n `:` s_n declares a class C with attributes $att_1, \ldots, att_n$ of sorts $s_1, \ldots, s_n$. An *object* of class C is represented as a term of sort `Object` and has the form

`<` O `:` C `|` att_1 `:` val_1`, . . . ,` att_n `:` val_n `>`,

where O is the object's identifier, and $val_1, \ldots, val_n$ are its attribute values of sort $s_1, \ldots, s_n$. A *subclass*, introduced with the keyword `subclass`, inherits all the attributes, equations, and rules of its superclasses. A *message* is a term of sort `Msg`, where the declaration `msg` m `:` s_1 `...` s_n `-> Msg` defines the syntax of the message (m) and the sorts (s_1 ... s_n) of its parameters.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rewrite rule

```
rl [l]: m(O,w)
        < O : C | a1 : x,     a2 : O', a3 : z, a4 : y >
      =>
        < O : C | a1 : x + w, a2 : O', a3 : z, a4 : y >
        dly(m'(O',x), y)
```

defines a parametrized family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`, the attribute `a1` of the object `O` is changed to `x + w`, and a new message `dly(m'(O',x), y)` is generated. The latter message has *message delay* `y`, and will become the "ripe" message `m'(O',x)` after time `y` has elapsed. The message `m(O,w)` is *removed* from the state by the rule, since it does *not* occur in the right-hand side of the rule. Likewise, the "delayed" message `dly(m'(O',x), y)` is *generated* by the rule, since it *only* occurs in the right-hand side of the rule. By convention, attributes whose values do not change and do not affect the next state of other attributes or messages, such as `a3` in our example, need not be mentioned in a rule. Similarly, attributes whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, such as `a2` and `a4`, can be omitted from right-hand sides of rules.

2.1.3. Formal Analysis in Real-Time Maude

We summarize below some of Real-Time Maude's analysis commands. Real-Time Maude's *timed fair rewrite* command

`(tfrew` t `in time <=` τ `.)`

simulates *one behavior* of the system within time τ from the initial state t. The *timed search* command

`(tsearch [`n`]` t `=>*` *pattern* `such that` *condition* `in time <=` τ `.)`

analyzes *all possible behaviors* by using a breadth-first strategy to search for n states that are reachable from the initial state t within time τ, match the search *pattern*, and satisfy the search *condition*. The *untimed* search command `utsearch` is similar, but without the time bound.

The Real-Time Maude's *linear temporal logic (LTL) model checker* checks whether each behavior from an initial state, possibly up to a time bound, satisfies an LTL formula. *State propositions* are declared as operators of sort `Prop`, and their semantics are given by equations of the forms

`eq {`*statePattern*`} |=` *prop* `=` *b*   and   `ceq {`*statePattern*`} |=` *prop* `=` *b* `if` *condition*

for b a term of sort `Bool`, which defines the state proposition *prop* to hold in all states `{`t`}` where `{`t`} |=` *prop* evaluates to `true`. An LTL formula is constructed by state propositions and temporal logic operators such as `True`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). Then, the *unbounded* (resp., *time-bounded*) model checking commands

`(mc` t `|=u` φ `.)`   and   `(mc` t `|=t` φ `in time <=` τ `.)`

check whether the formula φ holds in all behaviors from the initial state t (resp., *within time* τ).

Real-Time Maude is also equipped with a *timed CTL* model checker to analyze metric temporal logical properties [22, 23].

2.2. Models of Node Mobility

A key feature of mobile ad hoc networks is node mobility. In order to analyze the performance of MANETs under realistic scenarios, researchers on MANET protocol evaluation have developed a number of node *mobility models*, defining common realistic node movement patterns. Such node mobility models can be divided into *group mobility models*, where the nodes' movements depend on each other, and *entity mobility models*, where a node's movement is independent of the movements of other nodes.

This paper focuses on entity mobility models within a given convex (two-dimensional) area. The main entity mobility models in protocol evaluation are *random walk*, *random waypoint*, and *random direction*, which can be summarized as follows [6]:

Random Walk. A node that moves according to the random walk model moves in "rounds" of fixed durations. A node moves in the same direction and with the same speed throughout a round. At the end of each round, the node randomly selects the *new speed* and *direction* from given ranges of possible velocities and directions, and starts a new round by walking with the new speed in the new direction. If a node reaches the boundary of the area, it "bounces" off the boundary with an angle determined by the incoming direction and continues along this new path [6].

Random Waypoint. A node that moves according to the random waypoint mobility model initially pauses for a fixed duration. It then randomly selects a new destination (from a given set of possible destinations) and a new speed (from a given set of possible velocities), and travels to the selected destination with the selected speed. After arriving, the node again pauses for the fixed duration before a new moving round starts.

Random Direction. A node that moves according to the random direction model first randomly selects the new direction and speed. The node then starts moving in the selected direction with the selected speed until it reaches the boundary of the area. When the node reaches the boundary, it pauses for a fixed duration, and then randomly selects a new direction and new speed and starts moving in the new direction.

These mobility models are illustrated in Fig. 1. In addition, a number of formal methods approaches to MANET analysis also consider the *stationary* (mobility) model, in which the nodes do not move at all (see, e.g., [8, 13, 34, 31]).

Note that the random direction model can be regarded as a special case of the random waypoint model (where the set of possible next destinations are restricted to points on the area boundary). Furthermore, the stationary model is a special case of all models when the range of possible velocities is $[0, 0]$.

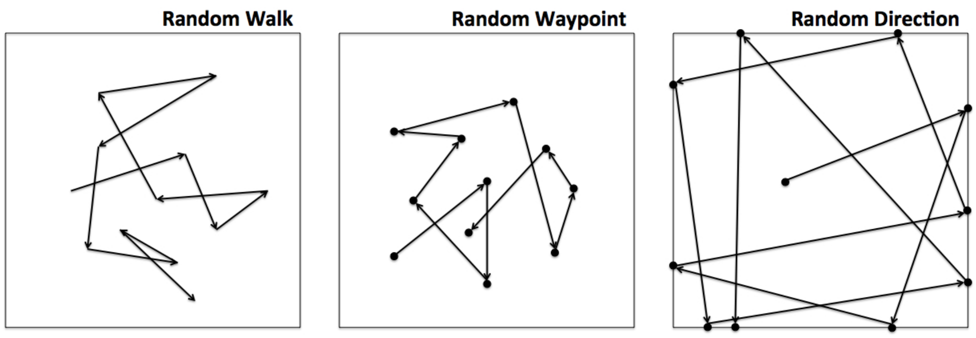

Figure 1: Motion paths of a mobile node in three mobility models, where a bullet • depicts a pause in the movement.

2.3. Wireless Communication in MANETs

One of the important, but subtle, issues in mobile ad hoc networks, in which nodes communicate wirelessly, is the combination of node mobility, wireless transmission range, and communication delay. In particular, both a sender and a potential receiver could move *during* a communication event, so that the potential receiver could possibly move into, or out of, the transmission range of the sender, which itself could be moving. To understand how node movement affects wireless communication, it is therefore necessary to understand the nature of messaging delays in wireless communication.

Wireless Communication. Nodes in a MANET typically communicate wirelessly, e.g., by radio transmission, so that only those nodes that are within the *transmission range* of the sending node can receive a message with sufficient signal strength. Different nodes may transmit with different signal strength, which means that nodes have different transmission ranges. Furthermore, a node may also dynamically adjust its sending power, for example to save energy, which happens in, e.g., topology control algorithms.

Wireless communication supports both (one-hop) broadcast (within the sender's transmission range), (one-hop) multicast, and (one-hop) unicast, since nodes may communicate at different radio frequencies—which may be known to all nodes (broadcast), a group of nodes, or just a pair of nodes (unicast channels). Furthermore, a communication channel can, at a given time, be either *bidirectional* (both nodes can reach each other with messages), or *unidirectional*, if the transmission range of one node is too short for a transmitted message to reach the other node, whereas the transmission range of the other node is broad enough to reach the first node. Figure 2 shows a system with three nodes, A, B, and C, with transmission ranges r_A, r_B, and r_C, respectively. The communication "channel" between A and B would be *bidirectional*, since a signal from A reaches B, and vice versa. The communication "channel" between A and C, however, is *unidirectional*, since a signal from A can reach C, but not vice versa. Finally, there is no communication channel between B and C.

Two different signals in the network at the same time can cause electromagnetic interference. Nodes therefore typically execute a media access control (MAC) protocol before transmitting a message to try to ensure that the appropriate communication channels are clear.

Communication Delay. In a typical wireless transmit/receive process, the per-hop delay, i.e., the communication delay from a transmitter to a receiver, consists of the following five parts [42]:

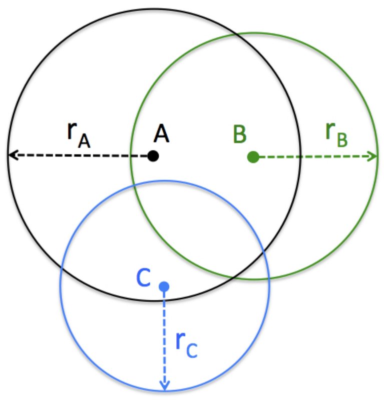

Figure 2: Three nodes and their transmission ranges.

Delay Factor	**Description**
Sender Processing Delay	The duration elapsed on the sender side from the moment a message timestamp is taken to the point the message is buffered in the device.
Media Access Delay	The duration for a message to stay in the radio device buffer; e.g., in a CSMA system, this is the delay waiting for a clear channel to transmit.
Transmit Delay	The duration for a radio device to transmit a message over a radio link.
Radio Propagation Delay	The duration for a message to propagate through the air to a receiver.
Receiver Processing Delay	The duration spent on the receiver side to pass the received message from the device buffer to the application module.

The media access delay depends on the MAC protocol used and the network traffic.

We can typically abstract from the radio propagation delay, since the transmission range in MANETs usually is between 10 and 100 meters, while the radio propagation speed is approximately 3×10^8 meters per second. This means that we can consider the per-hop communication delay to consist of two parts:

1. The delay at the sender side, which includes the sender processing delay, the media access delay, and the transmit delay.

2. The delay at the receiver side, which includes the receiver processing delay.

3. Modeling MANETs in Real-Time Maude

This section shows how MANETs, with nodes that move and communicate as explained in Section 2, can be modeled in Real-Time Maude. This specification of MANETs was used in the two case studies reported in Sections 4 and 5. A key challenge addressed in this section is the formalization of mobility and communication that takes into account the possibility of node movement *during* communication.

In particular, Section 3.2 explains how mobile nodes and the four mobility models (which include the stationary model) in Section 2.2 can be formalized in Real-Time Maude. Section 3.3 shows how we have specified wireless communication in MANETs, including wireless unicast, multicast, and broadcast, in Real-Time Maude.

A key point is that our definition of MANETs communication and mobility models should be *composable* with a specification of a MANET protocol. This would allow a modeler to specify a MANET protocol *without* having to take into account the underlying communication and mobility models; this protocol specification could then be composed with the desired mobility model. Section 3.4 explains how we can compose a protocol specification with one (or more, in case different nodes follow different movement patterns) of our predefined mobility models. In addition, it is sometimes needed to compose *multiple* MANET protocols. For example, the leader election protocol LE must be combined with a routing protocol to support multi-hop communication, a neighbor discovery protocol, and with mobility models. Section 3.4 also explains how multiple MANETs protocol specifications can be easily composed. Such compositions enable a modeling methodology where each protocol can be specification in isolation, without cluttering the specification with mobility and communication issues, or with other protocols.

3.1. Some Basic Data Types

Like other papers we have seen on MANET analysis, we assume that nodes move in a two-dimensional rectangle with size `Xsize` $\times$ `Ysize`. A location in this area is therefore represented as a pair x . y of rational numbers:

```
sorts Point Location .          subsort Location < Point .
op _._ : Rat Rat -> Point [ctor] .

vars X Y : Rat .
cmb X . Y : Location  if 0 <= X and X <= Xsize /\ 0 <= Y and Y <= Ysize .
```

Two of the mobility models select the new speed and direction at the same time. We therefore represent the combination of speed and direction by a vector `<` *xSpeed* `,` *ySpeed* `>` of sort `SpeedVector`, where *xSpeed* denotes the distance traveled along the x-axis during one time unit, and *ySpeed* represents the distance traveled along the y-axis during the same time unit:

```
sort SpeedVector .
op <_,_> : Rat Rat -> SpeedVector [ctor] .
```

We also assume a sort `SpeedVectorRange`, and (for random waypoint mobility) sorts `DestinationRange`, and `VelocityRange`, denoting *sets* of, respectively, `SpeedVector`, `Location`, and nonnegative rational number elements. We do not further specify the different powersets, whose elements could be unions of dense intervals or of single points, or both. Since the nodes may need to nondeterministically select a new speed, a new next destination, and/or a new next direction, we assume for generality's sake that there is an operator `choose` that can be used in rules to nondeterministically select any value in the respective set:

```
op choose : SpeedVectorRange -> Choice [ctor] .
op choose : DestinationRange -> Choice [ctor] .
op choose : VelocityRange -> Choice [ctor] .
op [_] : SpeedVector -> Choice [ctor] .
op [_] : Location -> Choice [ctor] .
op [_] : Rat -> Choice [ctor] .
```

We assume that an element e can be chosen from a set S if `choose(`S`)` rewrites to `[`e`]` in zero or more steps.

In the case studies in this paper, the different ranges are all finite sets of the form e_1 ; e_2 ; $\ldots$; e_n, where the set union operator `_;_` is declared to be associative and commutative. For example, the sort `SpeedVectorRange` is defined as follows in our case studies:

```
sort SpeedVectorRange .
subsort SpeedVector < SpeedVectorRange .
op empty : -> SpeedVectorRange [ctor] .
op _;_ : SpeedVectorRange SpeedVectorRange -> SpeedVectorRange [ctor assoc comm id: empty] .
```

The following rewrite rule then specifies that any element from such a set of speed vectors can be nondeterministically selected:

```
var SV : SpeedVector .   var SVR : SpeedVectorRange .
rl [chooseSpeedAndDirection] :  choose(SV ; SVR) => [SV] .
```

3.2. Modeling Node Mobility

We model a MANET in an object-oriented style, where the state of the system consists of a collection of objects, representing mobile nodes, and a set of messages traveling between these objects. As is common in network simulations, and following the survey [6] on mobility models for mobile ad hoc network research, which does not even mention node collisions, we abstract from the size of the mobile nodes and therefore also abstract from node collisions.

Each mobile node is modeled as an object instance of some subclass of the following class `Node`:

```
class Node | currentLocation : Location, transRange : PosRat .
```

The attributes `currentLocation` and `transRange` denote the node's current location and transmission range, respectively.

A *stationary* node is modeled as an object instance of the subclass `StationaryNode` that does not add any new attributes to `Node`:

```
class StationaryNode .
subclass StationaryNode < Node .
```

A mobile node is modeled as an object instance of a subclass of the class `MobileNode`:

```
class MobileNode | speedVector : SpeedVector,  timer : TimeInf .
subclass MobileNode < Node .
```

where the attribute `speedVector` denotes the node's current speed vector (giving its current velocity and direction). The `timer` attribute is used to ensure that a node changes its movement (or lack thereof) in a timely manner; that is, `timer` denotes the time remaining until some discrete event must take place.

The different mobility models are then specified in subclasses of `MobileNode` as follows.

Random Walk. Recall that a node moving according to the random walk model is moving in time intervals of length `movingTime`. At the end of an interval the node nondeterministically chooses a new speed and a new direction for its next interval. If in the meantime the node hits the boundary of the area, it "bounces off" the boundary like a billiard ball. A random walk node is modeled by an object instance of the following subclass `RWNode`:

```
class RWNode | speedVectorRange : SpeedVectorRange, boundaryTimer : TimeInf .
subclass RWNode < MobileNode .
```

The attribute `speedVectorRange` denotes the set of possible next speed vectors. The `timer` attribute inherited from its superclass denotes the time remaining of its current move period. The `boundaryTimer` attribute denotes the time until the node hits the area boundary. In case the node is not expected to hit the area boundary during its current move, this timer is turned off; i.e., has the "infinity" value `INF`.

The instantaneous behavior of the mobility part of a random walk node can be modeled by the following rewrite rules. In the `startNewMove` rule, the node is finishing one interval (the `timer` attribute is 0), and must select the speed and the direction for its next round, reset the `timer` to expire at the end of the next round, and possible reset also the `boundaryTimer` if the node will hit the boundary during the next round. We also declare the variables used in this section:

```
vars SV NEW-SPEED : SpeedVector .  var SVR : SpeedVectorRange .   var DER : DestinationRange .
var VR : VelocityRange .           vars R X Y X1 Y1 X-SPEED Y-SPEED : Rat .
var VELOCITY : PosRat .            vars L L' CURR-LOC NEXT-LOC NEW-GOAL-LOC : Location .
vars T T' T1 : Time .              vars TI NEW-BOUNDARY-TIMER : TimeInf .
var MOVE-TIME : NzTime .           var MSG : Msg .                var MC : MsgContent .
vars O O' : Oid .                  var OS : OidSet .              var OBJECT : Object .
vars C C1 C2 SYSTEM : Configuration .
```

```
crl [startNewMove] :
    < O : RWNode | timer : 0, speedVectorRange : SVR, currentLocation : CURR-LOC >
  =>
    < O : RWNode | timer : movingTime, speedVector : SV,
                   boundaryTimer : if timeUntilWallHit(CURR-LOC, SV) < movingTime
                                   then timeUntilWallHit(CURR-LOC, SV) else INF fi >
  if choose(SVR) => [SV] .

op timeUntilWallHit : Location SpeedVector -> TimeInf .
eq timeUntilWallHit(X . Y, < X-SPEED , Y-SPEED >)
 = min(--- hit right or left wall:
       if X-SPEED > 0 then (Xsize - X) / X-SPEED   --- hit right wall
       else (if X-SPEED < 0 then (- X) / X-SPEED   --- hit left wall
             else INF fi) fi,                      --- no movement in x-direction
       --- hit upper or lower wall:
       if Y-SPEED > 0 then (Ysize - Y) / Y-SPEED   --- upper wall
       else (if Y-SPEED < 0 then (- Y) / Y-SPEED   --- hit lower wall
             else INF fi) fi) .                    --- no movement in y-direction
```

This definition allows a node to travel *along* a boundary if it so chooses.

The boundaryHit rewrite rule models the "billiard ball bounce" when a node hits a boundary during its move. When a left or right wall is hit, the speed along the x-axis is negated, and when an upper or lower boundary is hit, the speed along the y-axis is negated. (This means that both speed vector components are negated in the unlikely case that a corner is hit.) The boundaryTimer must also be reset in case the node will *again* hit a wall before finishing its current movement period:

```
crl [boundaryHit] :
    < O : RWNode | boundaryTimer : 0,  timer : T,
                   currentLocation : X . Y, speedVector : < X-SPEED , Y-SPEED >  >
  =>
    < O : RWNode | boundaryTimer : NEW-BOUNDARY-TIMER,  speedVector : NEW-SPEED >
  if NEW-SPEED := < if (X == 0 or X == Xsize) then - X-SPEED else X-SPEED fi,
                    if (Y == 0 or Y == Ysize) then - Y-SPEED else Y-SPEED fi >
  /\ NEW-BOUNDARY-TIMER := if timeUntilWallHit(X . Y, NEW-SPEED) < T
                           then timeUntilWallHit(X . Y, NEW-SPEED) else INF fi .
```

(The actual movement of such a node is modeled in Section 3.5, where the behavior in time is discussed.)

Our "speed vector" representation ensures that we stay within the rational numbers, and allows precise computation of quantities, e, e.g., the time until the area boundary is reached. Notice that selecting a desired velocity v and direction α amounts to selecting the speed vector < $v \cdot \cos\alpha$, $v \cdot \sin\alpha$ >.

Random Waypoint. In the random waypoint mobility model, a node alternates between pausing and moving. When it starts moving, it selects a new velocity and a new destination and starts moving towards the destination. Random waypoint is the only mobility model that forces us to leave the precision of the rational numbers, since we must use the square root function to compute the distance to the desired location. However, by approximating such square roots as rationals, we never leave the realm of rational number computations.

A node that moves according to the random waypoint pattern is modeled by an object instance of the subclass RWPNode :

```
class RWPNode | velocityRange : VelocityRange,   destRange : DestinationRange,   status : Status .
subclass RWPNode < MobileNode .

sort Status .
ops pausing moving : -> Status [ctor] .
```

The status attribute is either pausing or moving, and destRange denotes the set of possible goal locations.

The instantaneous behavior of this mobility model is given by the following rewrite rules. First, the startMoving rule considers the case when a node is pausing and the timer expires, the node must get moving by selecting a new velocity and a new desired next location, and resetting the timer so that it expires when the goal location is reached. It is worth remarking that, since a node must pause when it has reached the goal destination, it does not make sense to select the current location as the next goal destination:

```
crl [startMoving] :
    < O : RWPNode | currentLocation : X . Y, status : pausing, timer : 0,
                    velocityRange : VR,  destRange : DER >
  =>
    < O : RWPNode | status : moving,
                    speedVector : < (NEXT-X - X) / MOVE-TIME , (NEXT-Y - Y) / MOVE-TIME >,
                    timer : MOVE-TIME >
  if choose(VR) => [VELOCITY]
     /\ choose(DER) => [(NEXT-X . NEXT-Y)]
     /\ NEXT-X =/= X or NEXT-Y =/= Y
     /\ MOVE-TIME := distance(X . Y, NEXT-X . NEXT-Y) /  VELOCITY .

op distance(X . Y, X1 . X2) = sqrt((X1 - X) * (X1 - X) + (Y1 - Y) * (Y1 - Y)) .
op sqrt : Rat -> Rat .
eq sqrt(R) = rat(sqrt(float(R))) .
```

Notice that VELOCITY is declared to be a (non-zero) positive rational number. We define an approximate square root function on the rationals by first converting a rational number to a floating-point number, computing the square root of the floating-point number, and then converting back the resulting floating-point number to a rational number.

The startPausing rule applies when the timer of a *moving* node expires; then it is time to take a rest for pauseTime time units:

```
rl [startPausing] :
   < O : RWPNode | status : moving, timer : 0 >
 =>
   < O : RWPNode | status : pausing, timer : pauseTime > .
```

Random Direction. A node that moves according to the random direction model nondeterministically chooses a direction and a speed, i.e., a speed vector, and walks in the given direction until it reaches the boundary of the area. It then pauses for some time before starting a new walk. Nodes moving according this mobility pattern should be modeled as object instances of the following subclass RDNode:

```
class RDNode | speedVectorRange : SpeedVectorRange,  status : Status .
subclass RDNode < MobileNode .
```

The instantaneous behaviors are formalized by two rewrite rules; the newRDwalk rule chooses a new speed vector when the node has paused long enough (i.e., when it is pausing and its timer expires):

```
crl [newRDwalk] :
    < O : RDNode | currentLocation : CURR-LOC, speedVectorRange : SVR,
                   timer : 0, status : pausing >
  =>
    < O : RDNode | status : moving, speedVector : SV,  timer : TI >
  if choose(SVR) => [SV]
     /\ TI := timeUntilWallHit(CURR-LOC, SV)
     /\ TI > 0 .
```

The last conjunct in the condition ensures that the selected direction leads inwards towards the area of operation (or that the node will stand still forever).

The `startPausing` rule models the stage when a moving node starts pausing:

```
rl [startPausing] :
   < O : RDNode | status : moving, timer : 0 >
 =>
   < O : RDNode | status : pausing, timer : pauseTime > .
```

3.3. Modeling Communication in Mobile Wireless Systems

This section explains how we model communication in MANETs. We abstract from communication collision detection and avoidance, and refer to [30, 7] for the formal treatment of communication collisions in wireless networks.

As explained in Section 2.3, wireless communication in a MANET supports both broadcast, unicast, and multicast in one “hop.” Our model has three corresponding message types for the sender to use for *one-hop* transmission:

```
msg broadcast_from_ : MsgContent Oid -> Msg .
msg unicast_from_to_ : MsgContent Oid Oid -> Msg .
msg multicast_from_to_ : MsgContent Oid OidSet -> Msg .
```

An element of sort `OidSet` denotes a set of object identifiers. The application-specific sort `MsgContent` denotes the content of the message to be transmitted. When the sender sends a “message” of one of the above kinds, each node within range that should receive the message will receive a message of the form

```
msgContent from sender to receiver.
```

Broadcast. When a node *sender* wants to broadcast some message content *mc* to all nodes reachable in one hop, it generates a “message”

```
broadcast mc from sender.
```

The following equation adds the delay on the sending side, `sendDelay`, to this “broadcast message:”

```
eq broadcast MC from O = dly(transmit MC from O, sendDelay) .

op transmit_from_ : MsgContent Oid -> Msg [ctor] .
op distrMsg : Oid Location Rat MsgContent Configuration -> Configuration [ctor frozen (1)] .
```

The crucial moment is when the sending delay expires and the `transmit` message becomes “ripe.” This corresponds to the time when the message is actually transmitted by the sender. All the nodes that are within the transmission range of the sender *at that moment* should receive the message. This distribution is performed by the function `distrMsg`, where `distrMsg`(*snd*, *tr*, *loc*, *mc*, *conf*) generates a *single* message, with content *mc*, for each node in *conf* that is *currently* within the transmission range *tr* of location *loc*. The expression *loc* `withinTransRange` *tr* `of` *loc'* computes whether the location *loc'* is within distance *tr* from the location *loc*. The transmission range *tr* should be the transmission range of the sender at the moment of transmission (remember that our framework supports dynamically modifying the transmission power). In particular, the following equations, which take place exactly when the sending delay expires, are used to generate a single message *mc* `from` *snd* `to` *rcvr* for each node that is within the transmission range of the sender *snd* when the sending delay expires; it also uses the transmission range of the sender at that moment. The resulting message has a delay `recDelay` modeling the delay at the receiving side:

```
eq {< O : Node | currentLocation : L, transRange : R > (transmit MC from O) C}
 = {< O : Node | >   distrMsg(O, L, R, MC, C)} .

eq distrMsg(O, L, R, MC, < O' : Node | currentLocation : L' > C)
 = < O' : Node | currentLocation : L' > distrMsg(O, L, R, MC, C)
   (if L' withinTransRange R of L then  dly((MC from O to O'), recDelay)  else none fi) .

eq distrMsg(O, L, R, MC, MSG C) = MSG distrMsg(O, L, R, MC, C) .
eq distrMsg(O, L, R, MC, none) = none .

op _withinTransRange_of_ : Location Rat Location -> Bool .
eq (X . Y) withinTransRange R of (X1 . Y1)
 = ((X1 - X) * (X1 - X) + (Y1 - Y) * (Y1 - Y)) <= R * R .
```

The use of the "global state" operator `{_}` in the first equation ensures that `C` includes *all* nodes in the system, except the sender `O`.

Unicast. The distribution/transmission of a unicast is similar to that of a broadcast, except that it is no longer necessary to consider the entire state:

```
op transmit_from_to_ : MsgContent Oid Oid -> Msg [ctor] .

eq (unicast MC from O to O') = dly(transmit MC from O to O', sendDelay) .

eq (transmit MC from O to O')
   < O : Node | currentLocation : L, transRange : R >
   < O' : Node | currentLocation : L' >
 =
   < O : Node | >    < O' : Node | >
   (if L' withinTransRange R of L then dly(MC from O to O', recDelay) else none fi) .
```

Multicast. Multicast can be defined by letting a multicast message be equivalent to a set of unicast messages: one unicast message for each node in the group:

```
eq (multicast MC from O to O' ; OS) = (unicast MC from O to O')  (multicast MC from O to OS) .
eq (multicast MC from O to none) = none .
```

3.4. Composing MANET Protocols and Mobility and Communication Models

Our model of mobile nodes must be easily *composable* with a specification of the MANET protocol to be analyzed. Furthermore, it is often necessary to compose *multiple* MANET protocols (and our model of communication and mobility for nodes). For example, as already mentioned, the leader election protocol of Vasudevan, Kurose, and Towsley should be run in parallel with both neighbor discovery protocol and a routing protocol. The point is that we can specify each protocol separately, without worrying about mobility, communication, or other protocols running in parallel.

A key Real-Time Maude feature that enables us to easily compose our model of mobile nodes with one or more MANET protocol specification(s) is the support for *multiple class inheritance*: one subclass may inherit the attributes and the rewrite rules of *multiple* superclasses. In particular, the objects in a specification of a MANET protocol P should apply to objects instances of a subclass P-`Node` of `Node`:

```
class P-Node | attributes_P .    --- protocol-specific attributes
subclass P-Node < Node .
```

All the rules in the specification of the protocol P should involve objects of class P-`Node`.

A specification of the protocol P can be composed with our model of mobile nodes by executing the system on objects that are instances of both the desired mobility class and the class P-`Node`. For example, a node in the protocol P that moves according to the random walk mobility model would be an object instance of the following subclass:

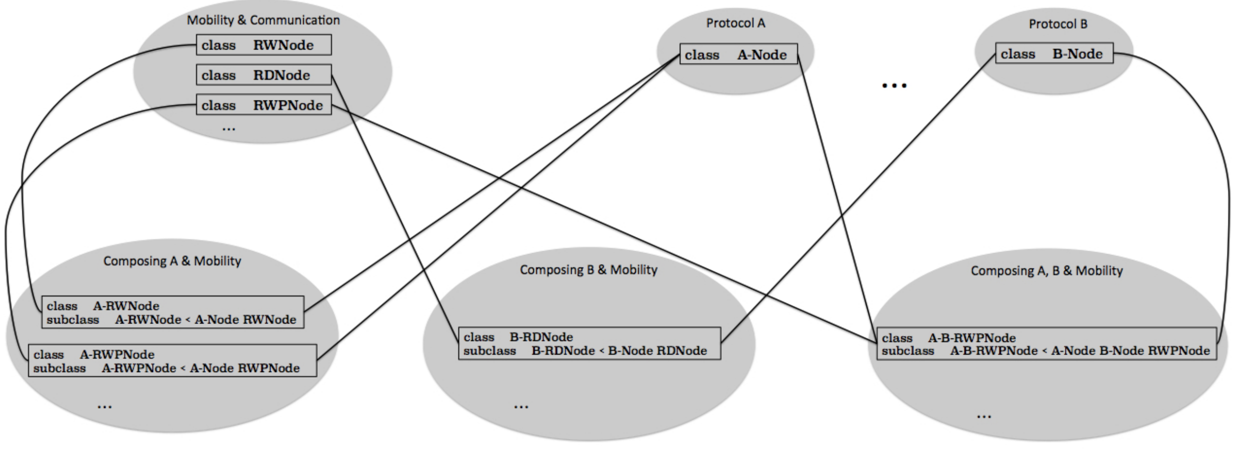

Figure 3: Composing protocols and node mobility and communication.

```
class P-RWNode .              --- no additional attributes
subclass P-RWNode < P-Node  RWNode .
```

In this way, we can also analyze the protocol P when the different nodes in the system may move according to different mobility patterns: some nodes may be stationary, others may move according to the random walk model, and yet others could move according to the random direction model.

Let us now consider the composition of two MANET protocols, say *LE* and *ND*. Assume that these protocols run in parallel and that they share—and interact through—a number of shared attributes, such as, e.g., `neighbors : OidSet`. The protocols *LE* and *ND* should then be defined in terms of objects of the following classes *LE*-`Node` and *ND*-`Node`, respectively:

```
class NeighborsNode | neighbors : OidSet .   --- shared attribute
subclass NeighborNode < Node .

class LE-Node | attributes_LE .       --- LE-specific attributes
subclass LE-Node < NeighborsNode .

class ND-Node | attributes_ND .       --- ND-specific attributes
subclass ND-Node < NeighborsNode .
```

Objects in the composition of both protocols should be instances of a subclass of both *LE*-`Node`, *ND*-`Node`, and the desired mobility model class:

```
class LE-ND-RWNode .
subclass LE-ND-RWNode < LE-Node ND-Node  RWNode .

class LE-ND-RDNode .
subclass LE-ND-RDNode < LE-Node ND-Node  RDNode .

...
```

Figure 3 summarizes the composition of one or more protocol specifications and our generic model of wireless communication and node mobility (shown as "Mobility & Communication" in the figure).

What remains is the subtle issue of being able to define also the timed behavior of each component separately. In Real-Time Maude, the timed behavior of an object-oriented system is typically specified by a tick rule

```
crl [tick] :  {SYSTEM}  => {timeEffect(SYSTEM, T)} in time T  if T <= mte(SYSTEM) .
```

where

- `timeEffect` defines the effect of time elapse on the state of the system, and
- `mte` ("maximum time elapse") defines the maximum time that may elapse until an event must take place.

These functions distribute over the objects and messages in the state in the expected way:

```
op timeEffect : Configuration Time -> Configuration [frozen (1)] .

ceq timeEffect(C1  C2, T) = timeEffect(C1, T)  timeEffect(C2, T) if C1 =/= none and C2 =/= none .
eq timeEffect(none, T) = none .
```

Furthermore, time affects messages by decreasing the remaining message delay according to the time elapsed:

```
eq timeEffect(dly(MSG, T1), T) = dly(MSG, T1 monus T) .
```

where x `monus` $y = \max(0, x - y)$. The message delay operator `dly` is declared to have right identity `0`, which means that a message m and `dly(`m`, 0)` are considered to be identical in Real-Time Maude.

The function `mte` also distributes over the elements in a configuration in the expected way:

```
op mte : Configuration -> TimeInf [frozen (1)] .
ceq mte(C1  C2) = min(mte(C1), mte(C2))  if C1 =/= none and C2 =/= none .
eq mte(none) = INF .    --- "infinity" value
```

Time advance must stop when a message becomes ripe:

```
eq mte(dly(MSG, T)) = T .
```

Since a "ripe" message m is identical to `dly(`m`, 0)`, this equation implies that time cannot advance when there is a ripe message in the configuration; each message must therefore be treated/consumed at the moment when its delay expires.

What still remains is to define `timeEffect` and `mte` on single objects. In other large Real-Time Maude applications (see [36]), these functions could be defined directly on the objects:

```
eq timeEffect(< O : C | ... >, T) = < O : C | ... > .
eq mte(< O : C | ... >) = ... .
```

This is no longer possible if the timed behavior of each component should be specified separately. Instead, the way to achieve such modular specification of the timed behavior of each protocol is to define, for each protocol P_i, the functions

```
op timeEffect-Pi : Object Time -> Object [frozen (1)] .
op mte-Pi : Object -> TimeInf [frozen (1)] .
```

Likewise, the functions

```
op timeEffect-Mob : Object Time -> Object [frozen (1)] .
op mte-Mob : Object -> TimeInf [frozen (1)] .
```

should define the timed behavior of node mobility and communication. The point is that this approach allows us to define the timed behavior of each protocol, as well as the timed behaviors of node mobility and communication, separately, by just defining the functions `timeEffect-`P_i and `mte-`P_i on objects of class P_i`-Node`, for each protocol P_i, and by defining `timeEffect-Mob` and `mte-Mob` on our model of node mobility and communication.

The only definition needed in the composition of multiple MANET protocol specifications is to define the "global" function `timeEffect` as the "composition" of the functions `timeEffect-`P_1, `timeEffect-`P_2, ..., `timeEffect-`P_n, and `timeEffect-Mob`, and to define the function `mte` as the "composition" of the functions `mte-`P_1, `mte-`P_2, ..., `mte-`P_n, and `mte-Mob`:

```
  eq mte(OBJECT) = min(mte-P1(OBJECT), ..., mte-Pn(OBJECT), mte-Mob(OBJECT)) .
  eq timeEffect(OBJECT, T) = timeEffect-Pn(...(timeEffect-P1(timeEffect-Mob(OBJECT, T), T), ...), T) .
```

This definition of `timeEffect` is confluent *provided at most one function* `timeEffect-`P_i *updates the same attribute*. To the best of our knowledge, and despite the wealth of large Real-Time Maude applications, this is the first time that a method is shown for composing Real-Time Maude specifications where each single specification also defines the timed behavior of the subsystem/protocol.

To summarize, a module defining the composition of two protocols `LE` and `ND` as well as an initial state in such a composition should have the following form:

```
(tomod LE+ND is
  including MOBILE-NODE .     --- specifying mobile nodes, tick rule, etc.
  including LE .              --- specifying the protocol LE
  including ND .              --- specifying the protocol ND

  ...           --- specify the sorts SpeedVectorRange, VelocityRange, and DestinationRange
  ...           --- and the choose rule on these domains

  class LE-ND-RWNode .        --- random walk
  subclass LE-ND-RWNode < LE-Node ND-Node RWNode .

  class LE-ND-RDNode .        --- random direction
  subclass LE-ND-RDNode < LE-Node ND-Node RDNode .
  ...
  --- define timed behavior
  var OBJECT : Object .   var T : Time .
  eq timeEffect(OBJECT, T) = timeEffect-ND(timeEffect-LE(timeEffect-Mob(OBJECT, T), T), T) .
  eq mte(OBJECT) = min(mte-Mob(OBJECT), mte-LE(OBJECT), mte-ND(OBJECT)) .

  --- define suitable initial state
  ops node1 node2  ... node-k : -> Oid [ctor] .  --- node names
  op init : -> GlobalSystem .
  eq init = {< node1 : LE-ND-RWNode | attributesWithInitialValues >
             < node2 : LE-ND-RDNode | attributesWithInitialValues' >
              ...
             < node-k : LE-ND-RWNode | attributesWithInitialValues'' >} .
endtom)
```

The following Section 3.5 defines the timed behavior of node mobility and communication.

3.5. Timed Behavior of Mobility and Communication

As explained in Section 3.4, the timed behavior of node mobility is specified by defining the functions `timeEffect-Mob` (how does the passage of a certain amount of time affect the state of a mobile node?) and `mte-Mob` (how much time can advance before some "event" must take place?) on mobile nodes. The definition of the functions `timeEffect` and `mte` in Section 3.4 defines the timed behavior of different kinds of "messages": a message must be treated as soon as its delay has expired.

The mobility-specific parts of a stationary node are not affected by the elapse of time, and do not impose any constraints on how much time can advance:

```
eq timeEffect-Mob(< O : StationaryNode | >, T) = < O : StationaryNode | > .
eq mte-Mob(< O : StationaryNode | >) = INF .
```

The mobility-specific parts of a random walk node allows time to advance until one of its timers expires:

```
var TI : TimeInf .
eq mte-Mob(< O : RWNode | boundaryTimer : TI, timer : T >) = min(TI, T) .
```

Time affects a random walk node by decreasing the timers according to the elapsed time, and by moving the node to a new location:

```
eq timeEffect-Mob(< O : RWNode | boundaryTimer : TI, timer : T,
                                 currentLocation : X . Y,
                                 speedVector : < X-SPEED , Y-SPEED >  >,  T')
 = < O : RWNode | boundaryTimer : TI monus T', timer : T monus T',
                  currentLocation : (X + X-SPEED * T') . (Y + Y-SPEED * T') > .
```

We next discuss how time affects the mobility-specific parts of a *random waypoint* node. The next "event" (either start pausing or start moving) takes place when the `timer` expires:

```
eq mte-Mob(< O : RWPNode | timer : T >) = T .
```

The passage of time affects a `pausing` node by decreasing its `timer` value according to the elapsed time:

```
eq timeEffect-Mob(< O : RWPNode | status : pausing, timer : T >, T')
 = < O : RWPNode | timer : T monus T' > .
```

Time elapse affects a `moving` random waypoint node by decreasing its `timer` and moving the node:

```
eq timeEffect-Mob(< O : RWPNode | status : moving, timer : T,
                                  currentLocation : X . Y,
                                  speedVector : < X-SPEED , Y-SPEED >  >, T') =
 = < O : RWPNode | timer : T monus T',
                   currentLocation :  (X + X-SPEED * T') . (Y + Y-SPEED * T') > .
```

The timed behavior of a random direction node is defined in exactly the same way.

The timed behavior of communication is straightforward. The sending delay is modeled as follows in Section 3.3: A `broadcast` or `unicast` message is immediately transformed into a "message" `dly(transmit ..., sendDelay)` with delay `sendDelay`. When this delay expires, the resulting `transmit` message is transformed into a message of the form `dly(`M `from` *sender* `to` *receiver*`, recDelay)` with delay `recDelay`. When time `recDelay` has passed, this message becomes the undelayed/ripe message M `from` *sender* `to` *receiver* which is read by the receiver. The equation

```
eq timeEffect(dly(MSG, T1), T) = dly(MSG, T1 monus T) .
```

in Section 3.4 is the one that reduces the remaining messaging delay according to the elapsed time `T`, and the equation

```
eq mte(dly(MSG, T)) = T .
```

ensures that time cannot advance when there is a ripe message in the system, since an undelayed message M is equivalent to `dly(`M`, 0)`. Ripe messages must therefore be read immediately.

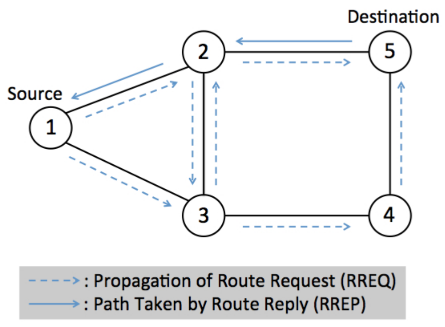
Destination
Source
1
2
3
4
5
: Propagation of Route Request (RREQ)
: Path Taken by Route Reply (RREP)

Figure 4: Route discovery process

4. Case Study 1: Route Discovery in AODV

This section shows how our framework has been used to formalize and analyze the well-known *Ad hoc On-Demand Distance Vector* (AODV) [40] routing protocol for MANETs. Section 4.1 gives a brief overview of the AODV protocol. The protocol is intended to first establish a route between a source node and a destination node, and then maintain a route between the two nodes during topology changes caused by node movement. We focus in this paper on the route discovery process, and present our Real-Time Maude model of the AODV route discovery process in Section 4.2. Section 4.3 explains how the main property of any routing protocol, that a route will be established between source and destination, can be analyzed under different mobility models. Our analysis shows that a previously known flaw in AODV due to link failures in a static setting can also be caused by mobility in a mobile setting. The entire executable Real-Time Maude specification is available at `https://sites.google.com/site/siliunobi/wrla14`.

4.1. The AODV Protocol

AODV [40] is a widely used protocol developed by the IETF MANET working group for routing messages between mobile nodes which dynamically form an ad hoc network. AODV allows a source node to initiate a route discovery process to establish a route to a given destination node. Each node contains the distances, measured in hops, to other nodes in its routing table. Since AODV works *on-demand*, routers only maintain distance information for nodes reached during route discovery. AODV consists of two main parts: *route discovery* and *route maintenance*.

Route Discovery. When a source node S wants to establish a route to a given destination node D, the source node initiates a route discovery process by broadcasting a route request (RREQ) message to its neighbors. When a node N receives such a RREQ message, it unicasts a route reply (RREP) message to the source if a valid route to the destination D can be found in its local routing table; otherwise, the node N re-broadcasts the received RREQ message to its own neighbors. As the RREQ messages travel from S to D, reverse paths from all nodes back to `S` are automatically set up. Eventually, when a RREQ message reaches D for the first time, D sends a RREP message back along the reverse path, which also gives the desired route between S and D. A node that receives multiple RREQ messages drops the subsequent ones if the same RREQ was recorded previously. To ensure loop-free routing, AODV employs a sequence number to represent how fresh the received information is. The higher a sequence number is, the fresher a route will be. Therefore a requesting node is required to select the one with the greatest sequence number.

Figure 4 shows a topology of five nodes with node `1` the source and node `5` the destination. Dashed arrows and solid arrows show the RREQ and RREP flows, respectively. After the route discovery process, a route $1 \longrightarrow 2 \longrightarrow 5$ is established. Note that the route $1 \longrightarrow 3 \longrightarrow 4 \longrightarrow 5$ may be established instead, for example due to message loss or high latency along the path $1 \longrightarrow 2 \longrightarrow 5$; AODV does not guarantee that the route established is the shortest one.

Route Maintenance. The purpose of route maintenance is to provide feedback to the sender in case a router (i.e., a node on the route from source to destination) is lost or a link breaks, in order to trigger route re-discovery. In a mobile setting, node movement could make a router unable to forward the message. If an intermediate node moves away, all its neighbors should be informed of the fact, which is also further forwarded to all the other hops in order to delete the associated route. As a result, the source node must then re-discover a new route. AODV uses a simple neighbor discovery protocol to detect topology changes.

4.2. Modeling AODV Route Discovery in Real-Time Maude

This section presents our Real-Time Maude specification of the AODV route discovery process. We discuss 5 of the 16 rewrite rules in our specification, and refer to https://sites.google.com/site/siliunobi/wrla14 for the full specification.

In AODV each run of the route discovery process is identified by the initiator of the route discovery and its current request identifier. In addition, each node has a sequence number used to determine how fresh a route to the destination must be before it can be accepted by the source [41].

Nodes. We model AODV nodes as objects of a subclass `AODV-Node` of class `Node`. The new attributes are the following:

- `rreqId` denotes the node's current request identifier.
- `sequenceNumber` denotes the node's current sequence number;
- `routingTable` denotes the node's routing table of the predefined Maude data type `MAP`. Each entry in the routing table has the form n `|-> tuple3(`n'`, `d`, `s`)`, where n is a destination node, n' is the next hop towards the destination n, d is the distance to the destination, and s is the sequence number of the node n.
- `requestBuffer` is a set of received routing requests of the form o `~` n, where o is the initiator of the request and n is its request id, uniquely determining the request. This is needed to know which requests the node has already treated.

```
class AODV-Node | rreqId : Nat,                sequenceNumber : Nat,
                  routingTable : RouteTable,  requestBuffer : RreqBuffer .
subclass AODV-Node < Node .
```

Messages. In the AODV route discovery process there are mainly two kinds of messages: RREQ messages, that are broadcast (in one hop), and RREP messages, that are unicast in a hop-by-hop fashion to the source:

```
op rreq : Oid Nat Nat Oid Sqn Nat Oid -> MsgContent [ctor] .
--- source, source sqn, request id, destination, destination sqn, current number of hops, router
op rrep : Oid Oid Sqn Nat Oid         -> MsgContent [ctor] .
--- source, destination, destination sqn, current number of hops, router
```

In addition, we initiate a round of the protocol using a message `initiateRouteDiscovery(`*`source`*`,` *`destination`*`)`:

```
msg initiateRouteDiscovery : Oid Oid -> Msg .
```

Timed Behavior. Since we focus on route discovery, we will not handle the timing issues in the route maintenance process. Our AODV model therefore does not impose additional timing constraints:

```
eq timeEffect-AODV(< O : AODVNode | >, T)  =  < O : AODVNode | > .
eq mte-AODV(< O : AODVNode | >) = INF .
```

4.2.1. Modeling Dynamic Behaviors

The route discovery process in AODV consists of three parts: initiating route discovery, route request handling, and route reply handling.

Initiating Route Discovery. At the start of the route discovery process, a source node checks its local routing table for an entry towards the destination (`inRT(RT, DEST)`). If such an entry does not exist, the source initiates the route discovery process by broadcasting a RREQ message. Before broadcasting, the originator needs to increase the local routing request ID and its own sequence number, and add the outgoing RREQ id to the request buffer:[2]

```
 rl [init-route-discovery] :
    initiateRouteDiscovery(O, DEST)
    < O : AODV-Node | rreqID : RREQID,   sequenceNumber : SN,
                      routingTable : RT, requestBuffer : RB >
  =>
    if inRT(RT, DEST)
       then < O : AODV-Node | >    --- route exists; do nothing
       else < O : AODV-Node | rreqId : RREQID + 1, sequenceNumber : SN + 1,
                              requestBuffer : (O ~ RREQID , RB) >
            (broadcast rreq(O, SN + 1, RREQID, DEST, 0, 0, O) from O) fi .
```

Route Request Handling. When a node receives a RREQ message, it first checks whether a request with the same source and request ID has already been received and stored in the request-buffer. If so, the request can be ignored and the local routing table is updated by adding a routing table entry towards the sender.

If the request has not been received before (`not (SOURCE ~ RREQID) in RB`), the receiving node adds the new route request identifier to the request buffer, and takes further actions as explained next.

The rewrite rule `receiving-rreq-2` shows the case when the receiver is the desired destination node. The receiver then generates a route reply message `rrep(SOURCE, DEST, SN', 0, DEST)`, where `SN'` is the maximum of the current sequence number and the destination sequence number in the RREQ [40]. The hop count is obviously `0`. The destination should also update the routing table entry for the sender, as well as the source node, in its local routing table (`RT'` and `RT"`). Then the RREP is unicast to the next hop along the route back to the source node (`nexthop(RT"[SOURCE])`) by looking up the updated routing table `RT"`.

```
 crl [receiving-rreq-2] :
     (rreq(SOURCE, OSN, RREQID, DEST, DSN, HOPS, SENDER) from SENDER to DEST)
     < DEST : AODV-Node | sequenceNumber : SN,  routingTable : RT,  requestBuffer : RB >
  =>
     < DEST : AODV-Node | sequenceNumber : SN',  routingTable : RT'',
                          requestBuffer : (SOURCE ~ RREQID , RB) >
     (unicast rrep(SOURCE, DEST, SN', 0, DEST) from DEST to nexthop(RT''[SOURCE]))
  if RT'  := update(SENDER, SENDER, 1, 0, RT) /\
     RT'' := update(SOURCE, SENDER, HOPS + 1, OSN, RT') /\
     SN'  := max(SN, DSN + 1) /\
     not (SOURCE ~ RREQID) in RB .
```

If the receiving node `O` is *not* the desired destination `DEST`, but an intermediate node, then it either: (a) generates a route reply to the sender, or (b) re-broadcasts the received RREQ to its neighbors. Action (a), as the `receiving-rreq-3` rule shows, happens only when `O`'s local information is fresher than that in the RREQ; that is, its local sequence number for the destination is greater than or equal to the received sequence number (`DSN <= localdsn(RT[DEST])`). In this case, `O` unicasts the route reply with the fresher destination sequence number and its distance in hops from the destination along the route back to the source node:

```
  crl [receiving-rreq-3] :
      (rreq(SOURCE, OSN, RREQID, DEST, DSN, HOPS, SENDER) from SENDER to O)
```

[2]We do not show the declarations of the mathematical variables used in the case studies; they follow the Maude convention that such variables are written with capital letters.

```
    < O : AODV-Node | sequenceNumber : SN,  routingTable : RT,  requestBuffer : RB >
  =>
    < O : AODV-Node | sequenceNumber : SN,  routingTable : RT'',
                      requestBuffer : (SOURCE ~ RREQID , RB) >
    (unicast rrep(SOURCE, DEST, localdsn(RT''[DEST]), hops(RT''[DEST]), O)
        from O to nexthop(RT''[SOURCE]))
  if DEST =/= O /\ inRT(RT,DEST) /\ not (SOURCE ~ RREQID) in RB /\
     DSN <= localdsn(RT[DEST]) /\
     RT'  := update(SENDER, SENDER, 1, 0, RT) /\
     RT'' := update(SOURCE, SENDER, HOPS + 1, OSN, RT') .
```

Action (b) happens if local information is not fresh enough, or if no routing table entry for DEST can be found. O should then re-broadcast the received RREQ with the hops increment and the maximal destination sequence number in the message content.

Route Reply Handling. If the receiver of an RREP message is in fact the originating source of the route discovery process, the RREP is consumed and a routing table entry for the destination is created or updated. We can consider three cases: (a) the destination sequence number in the originator's existing routing table is smaller then the one in the received RREP; (b) the two destination sequence numbers are the same, but the increased hop count in the received RREP is smaller than the one in the local routing table (`hops(RT[DEST]) > HOPS + 1`); or (c) no route table entry for the destination can be found locally. In all other cases, the received RREP can be silently ignored. The `receiving-rrep-3` rule shows the specification of case (b):

```
crl [receiving-rrep-3] :
    (rrep(SOURCE, DEST, DSN, HOPS, SENDER) from SENDER to SOURCE)
    < SOURCE : AODV-Node | sequenceNumber : SN,  routingTable : RT,  requestBuffer : RB >
  =>
    < SOURCE : AODV-Node | routingTable : update(DEST, SENDER, HOPS + 1, DSN, RT) >
  if inRT(RT, DEST) /\ DSN == localdsn(RT[DEST]) /\ hops(RT[DEST]) > HOPS + 1 .
```

When the receiver is an intermediate node, the RREP can be forwarded if one of the above cases (a), (b) or (c) is satisfied, and can be silently ignored otherwise. The `receiving-rrep-6` rule formalizes case (c):

```
crl [receiving-rrep-6] :
    (rrep(SOURCE, DEST, DSN, HOPS, SENDER) from SENDER to O)
    < O : AODV-Node | routingTable : RT >
  =>
    < O : AODV-Node | routingTable : RT' >
    (unicast rrep(SOURCE, DEST, DSN, HOPS + 1, O) from O to nexthop(RT'[SOURCE]))
  if SOURCE =/= O /\ not inRT(RT,DEST) /\
     RT' := update(DEST, SENDER, HOPS + 1, DSN, RT) .
```

4.3. Formal Analysis of Route Discovery in AODV

To the best of our knowledge AODV has only been formally analyzed in static topologies with random link failures, or in an arbitrary mobility setting. Our framework allows us to analyze AODV under realistic mobility and communication models.

Combining AODV and Mobility. As explained in Section 3.4, an AODV node whose mobility follows a certain mobility pattern is modeled by an object instance of a subclass of both the mobility model class and the class AODV-Node. For example a node that moves according to the random walk pattern should be modeled as an object instance of the AODV-RWNode:

```
class AODV-RWNode .
subclass AODV-RWNode < AODV-Node RWNode .
```

```
class AODV-RWPNode .
subclasses AODV-RWPNode < RWPNode AODVNode .

class AODV-StationaryNode .
subclass AODV-StationaryNode < AODV-Node StationaryNode .
```

The composition of the two specifications must also define the functions `timeEffect` and `mte` on objects as explained in Section 3.4:

```
eq mte(OBJECT) = min(mte-AODV(OBJECT), mte-Mob(OBJECT)) .
eq timeEffect(OBJECT, T) = timeEffect-AODV(timeEffect-Mob(OBJECT, T), T) .
```

Initial States. We make the following assumptions in our experiments:

- The transmission range is 10 meters, and the area under consideration is 100×100.
- The sending and receiving delays are 10 and 5 time units, respectively, in all scenarios except for the scenarios IV and V below.

We define a parametric initial configuration `init1(`*source*`,` *destination*`)`, corresponding to the setting in the lefthand side of Fig. 5 (a solid circle refers to the initial location of a node, while a dash circle refers to some point along the motion path of a node). This setting has four stationary nodes (`1`, `3`, `4`, and `5`), and one random waypoint node (2) that can only choose to move up to location $(50, 60)$. The five nodes are initially located at positions $(45, 45)$, $(50, 50)$, $(50, 40)$, $(60, 40)$, and $(60, 50)$, respectively.

```
op init1 : Oid Oid -> Configuration .
subsort Nat < Oid .
vars SOURCE DEST : Oid .

eq init1(SOURCE, DEST)
 = initiateRouteDiscovery(SOURCE, DEST)
   < 1 : AODV-StationaryNode | currentLocation : 45 . 45,  transRange : 10,  rreqId : 10,
                               sequenceNumber : 1, routingTable : empty,  requestBuffer : empty >
   < 2 : AODV-RWPNode | currentLocation : 50 . 50,  transRange : 10, speedVector : < 0 , 0 >,
                        timer : pauseTime,  velocityRange : (1),  destRange : (50 . 60),
                        status : pausing,  rreqID : 20,  sequenceNumber : 1,
                        routingTable : empty,  requestBuffer : empty >
   < 3 : AODV-StationaryNode | currentLocation : 50 . 40,  transRange : 10, rreqId : 30,
                               sequenceNumber : 1, routingTable : empty,  requestBuffer : empty >
   < 4 : AODV-StationaryNode | currentLocation : 60 . 40,  transRange : 10, rreqID : 40,
                               sequenceNumber : 1, routingTable : empty,  requestBuffer : empty >
   < 5 : AODV-StationaryNode | currentLocation : 60 . 50,  transRange : 10, rreqID : 50,
                               sequenceNumber : 1, routingTable : empty,  requestBuffer : empty > .

rl [chooseDestination] : choose(D ; DR) => [D] .   --- any element in the set can be chosen
rl [chooseVelocity] : choose(V ; VR) => [V] .
eq pauseTime = 10 .    --- also experimented with 30 and 60
```

The following parametrized initial state denotes the MANET in the upper part of Fig. 6: it has three stationary nodes, and one random walk node (node 2) that can move up or down:

```
op init3 : Oid Oid -> Configuration .

eq init3(SOURCE, DEST)
 = initiateRouteDiscovery(SOURCE, DEST)
   < 1 : AODV-StationaryNode | currentLocation : 40 . 50,  transRange : 10,  rreqId : 10,
                               sequenceNumber : 1, routingTable : empty,  requestBuffer : empty >
```

```
    < 2 : AODV-RWNode | currentLocation : 50 . 40,  transRange : 10, speedVector : < 0 , 0 >,
                        timer : 0, speedVectorRange : (< 0 , 1> ; < 0 , -1 >),  boundaryTimer : INF,
                        rreqID : 20,  sequenceNumber : 1,
                        routingTable : empty,  requestBuffer : empty >
    < 3 : AODV-StationaryNode | currentLocation : 60 . 50,  transRange : 10, ... >
    < 4 : AODV-StationaryNode | currentLocation : 70 . 50,  transRange : 10, ... > .

  rl [chooseSpeedVector] : choose(SV ; SVR) => [SV] .
```

Property to Analyze. We formally analyze the main property of a routing protocol: a route will eventually be established between the source node and the destination node.

We use Real-Time Maude's LTL model checker to analyze this property, and define the parametrized atomic proposition `route-found(`*source*`,` *destination*`)` to hold if and only if there is an entry towards the destination node *destination* in the routing table of the the node *source*:

```
op route-found : Oid Oid -> Prop [ctor] .
eq {CONFIG  < SOURCE : AODV-Node | routingTable : (RT , DEST |-> TP) >}
   |= route-found(SOURCE, DEST) = true .
```

The following time-bounded model checking command then checks whether a route from node `1` to node `5` can always be found from initial state `init1(1,5)` within time 100:

```
(mc {init1(1, 5)} |=t <> route-found(1, 5) in time <= 100 .)
```

The model checking command returns `true` if the property holds for all behaviors; otherwise, a trace illustrating a counterexample is shown.

In addition, we also use search to search for states where the desired route has been found, to: (i) output the routing table, and (ii) analyze whether it is possible to arrive at a state where a route has been established, even if not all behaviors satisfy the desired property.[3] The following time-bounded search command is then used:

```
(tsearch {init1(1, 5)} =>* {C:Configuration} such that routeFound(1, 5, C:Configuration)
    in time <= 100 .)
```

where `routeFound` is defined as expected:

```
op routeFound : Oid Oid Configuration -> Bool [frozen (3)] .
eq routeFound(SOURCE, DEST, < SOURCE : AODV-Node | routingTable : (RT , DEST |-> TP) > C) = true .
eq routeFound(SOURCE, DEST, C) = false [owise] .
```

In this case study, we use the natural numbers as the time domain.

Scenarios and Analysis Results. We have analyzed AODV in five different scenarios.

Scenario I, shown in Fig. 4, corresponds to the state `init1(1, 5)`, except that all nodes are stationary nodes. The model checking result shows that a desired route is always established. The search command shows a routing table entry `5 |-> tuple3(2, 2, 1)` in node `1`'s routing table, indicating that a 2-hop route with the next hop `2` is built towards node `5`. By checking node `2`'s routing table, we obtain the route $1 \longrightarrow 2 \longrightarrow 5$.

Scenario I corresponds to the initial state `init1(1, 5)`. We set its pause time to: (a) 10 time units, (b) 30 time units, or (c) 60 time units. The analysis results show that:

- In Case (a), the property holds, and we find a routing table entry `5 |-> tuple3(3, 3, 1)` in node `1`'s routing table, and the whole route is $1 \longrightarrow 3 \longrightarrow 4 \longrightarrow 5$.

[3]The same analysis can be performed by time-bounded model checking of the formula `[] ~ routeFound(`*source*`,`*destination*`)`.

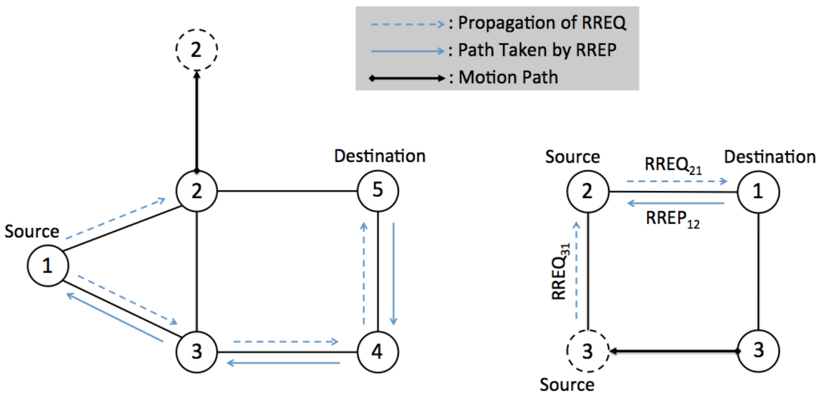

Figure 5: Topologies of Scenarios II and III.

- In Case (b), the property does not hold, and the `tsearch` command returns no solution, meaning that there is no possibility that a route can be built between the nodes `1` and `5` within 100 time units.
- In Case (c), the property holds, and we can obtain the same route as in Case (a).

In Case (b), the pause time (30) allows node `2` to forward the RREQ message to node `5`. However, node `2` cannot receive the RREP message from node `5` due to its movement (the dash circle in this case is at $(50, 60)$). Meanwhile, since node `5` has already recorded node `1`'s RREQ from node `2`, it silently ignores the one from node `4`. Thus, in this case, no route can be established between nodes `1` and `5`.

Scenario III is shown in the righthand side graph in Fig. 5 and considers three nodes. The point now is that *both* node `2` (located at $(40, 50)$) and node `3` (a random waypoint node located at $(50, 40)$) want to establish a route to node `1` (located at $(50, 50)$). Before sending out the RREQ message, node `3` moves left to the new location $(40, 40)$, which is within the transmission range of node `2`. Thus, to establish the route to node `1`, node `3`'s RREQ message needs to be forwarded by node `2`. However, our analysis shows that route discovery for node `3` fails: no route can be found between nodes `3` and `1`, though obviously node `2` succeeds in building a route to node `1`.

This problem arises due to the discarding of the RREP message. As stated in [40], an intermediate node forwards a RREP message only if the RREP message serves to update its routing table entry towards the destination. However, in this case, node `2` has already secured an optimal route to node `1` before receiving the RREQ message from node `3`. The paper [12] also points out this problem, but in a static linear topology with three nodes and two links. Our scenario, besides uncovering the flaw in AODV, shows in a more realistic setting that node mobility may cause failure in the AODV route discovery.

Scenario IV corresponds to the initial state `init3(1, 4)` and is shown in Fig. 6 (top). We set the sender delay and the receiver delay to 10 and 0, respectively, and set node `2`'s moving time to 10. Thus, it can receive the RREQ message from node `1`, once it reaches up to the intermediate point $(50, 50)$ between nodes `1` and `3`.

The experimental results show that route discovery fails. The reason is that when node `2` is ready to send out the RREQ message after both delays expire, it has moved to $(50, 60)$, indicated by the uppermost dash circle, which is beyond the transmission range of node `3`, and therefore no RREQ message will be delivered to node `3`, not to mention the destination node `4`.

Scenario V has the same topology and setting as Scenario IV, except that node `2` is a random waypoint node with pause time 10, so that it can receive the RREQ message from node `1`, once it moves to the intermediate point $(50, 50)$. Model checking shows that the property does not always hold. However, search finds that a route between nodes `1` and `4` can sometimes be successfully established. The reason is that node `2` has, for each moving step, two nondeterministic choices (up or down). A route cannot be established for some moving choices, like (up, down, down, down). However, AODV could be lucky and establish a route with different movements. For example, node `2` may pause at the $(50, 50)$ for a time interval that equals

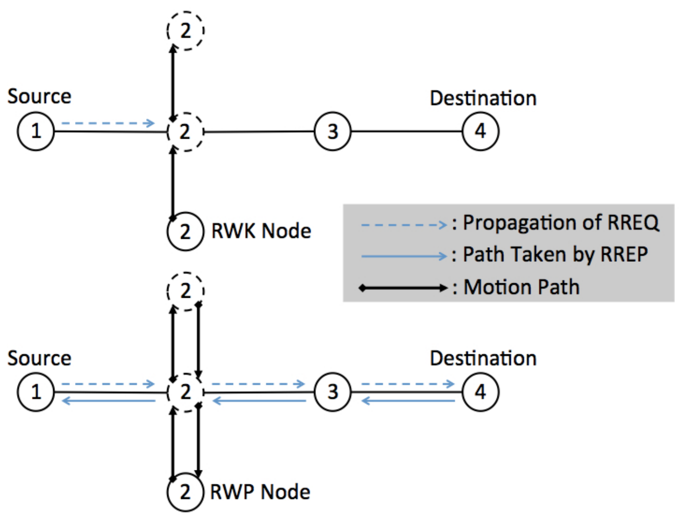

Figure 6: Topologies of Scenarios IV and V.

the sum of both delays at its side, thus generating the RREQ message for node `3` before it moves away. Likewise, when node `2` returns to the intermediate point, it happens to receive the RREP message from node `3`. Despite delays, it takes advantage of the pause time to forward the RREP message back to the source node `1`.

Our analysis is summarized in Table 1; essentially we find that the previously known error in AODV due to link failure in a static topology can also appear when two mobile nodes want to find routes at the same time, under various standard mobility models.

Table 1: Summary of the results and execution times of our analyses of the different scenarios. For model checking (`mc`), the table shows ✓ if the property holds, and shows × otherwise. For the `tsearch` command, ✓ shows that a route can be found. All experiments in this paper were carried out on a 2.90GHz Intel Core i7-3520M with 3.7GB RAM.

Scenario	I	II-(a)	II-(b)	II-(c)	III	IV	V
`mc` result	✓	✓	×	✓	×	×	×
`mc` execution time (ms)	196	152	252	200	256	288	272
`tsearch` result	✓	✓	×	✓	×	×	✓
`tsearch` execution time (ms)	272	284	324	332	212	528	1328

5. Case Study 2: The LE Leader Election Algorithm

This section gives an overview of how our framework has been used to formalize and analyze the well-known leader election algorithm for MANETs by Vasudevan, Kurose, and Towsley [50, 49]. We call their leader election algorithm LE.

LE uses source-to-destination (i.e., “multi-hop”) unicast, and therefore relies on the network infrastructure being able to transport messages to a given node. This can be achieved by composing LE with some routing protocol, such as AODV, and some transport protocol, such as UDP or TCP, that uses the routing information obtained by the routing protocol to transport messages from source to destination. However, the detailed description of LE [49] does not specify any routing or message transport protocol; it just assumes that “a message sent by a node is eventually received by the intended receiver, provided that the two nodes remain connected forever starting from the instant the message is sent.” We follow a similar approach. Instead of using AODV or another concrete routing algorithm, we model multi-hop communication in the following much more abstract way that satisfies the assumption above: if there exists a communication path from source to destination at a certain time after the message is sent, the message is delivered to the receiver.

LE also assumes that each node knows its neighbors, and that new links formed by node mobility are detected somehow. However, LE does not specify any neighbor discovery algorithm, nor does it make explicit the assumptions/requirements on the discovery of new links. The exception is that an explicit "probe" protocol is used to discover the loss of connection to a node from which a node awaits an *ack* message. We model neighbor discovery abstractly by periodically letting each node know which nodes are currently within its transmission range.

We formally analyze LE in a number of scenarios, including with both bidirectional and unidirectional communication channels and with node mobility. Our analysis has uncovered a spurious behavior in which a continuously reachable node is left out of the election forever. As explained in detail in Section 5.3, this behavior is caused by a very subtle interplay between node mobility, communication delays, and neighbor discovery. The point is that this spurious behavior could never be found by standard formal analysis methods for MANETs that abstract away communication delays and node movements. LE has been subjected to a large number of formal analyses that have failed to find flaws in the algorithm. We cannot claim that this behavior *invalidates* the LE algorithm, since LE may be based on other unstated assumptions, such as "continuous neighbor discovery" and/or multi-hop communication even to nodes that are immediate neighbors when a sending event begins. However, at the least, our "counterexample" shows the need to make explicit subtle assumptions about the underlying neighbor discovery process, and to make more precise the meaning of sending to "immediate neighbors" when an immediate neighbor may cease to be one *during* the sending process.

Section 5.1 gives a brief overview of the LE protocol. Section 5.2 presents the Real-Time Maude model of LE. Section 5.3 explains how our modeling framework for MANETs can be used to analyze our LE model under realistic mobility and communication models, including both bidirectional links and unidirectional links.

The paper [25] presents our model and analysis in greater detail. The entire executable Real-Time Maude specification is available at `https://sites.google.com/site/siliunobi/leader-election`.

5.1. Overview of LE

Apart from tolerating frequent topology changes, one key feature of LE is that it aims not only at finding *a* leader, but also at electing the *best-valued* node (according to some measure, such as the amount of remaining battery life or the average distance to other nodes) in each connected component as the leader of the connected component.

In a *static* topology, LE works as follows. When an election is triggered at a node, the node broadcasts an *election* message to its immediate neighbors. A node that receives an *election* message for the *first* time, records the sender of the message as its *parent* in the spanning tree under construction, and multicasts an *election* message to its other neighbors. (It is assumed that each node knows its set of neighbors, which can be established with a simple neighbor discovery protocol running in parallel.) When a node receives an *election* message from a node that is not its parent, it immediately responds with an *ack* message. When a node has received an *ack* message from all its children, it sends an *ack* message to its parent. Each *ack* message to a parent includes the identity and value of the best-valued node in the subtree (of the spanning tree defined by the "parent" relation) rooted at the sender. Therefore, when the source node has received an *ack* message from all of its children, it can easily determine the best-valued node in the entire spanning tree (i.e., in its connected component); this best-valued node becomes the elected leader. The source node then broadcasts a *leader* message announcing the identity of the new leader. Figure 7 shows a run of LE under a static topology of five nodes, with node `1` being the source and node `5` the best-valued node (the higher the node number, the better value it has). Starting with the propagation of *election* messages, LE eventually announces, by the propagation of *leader* messages, the best-valued node `5` as the leader, *after* having received all *ack* messages.

Multiple nodes can concurrently initiate *multiple* elections; in this case, only one election should "survive." This is done by associating to each election a *priority*, so that a node already involved in an election ignores

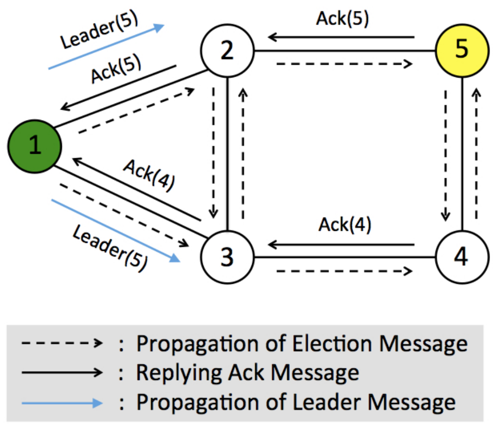
Leader(5)
Ack(5)
Ack(5)
Ack(4)
Ack(4)
Leader(5)
1
2
3
4
5
- - - - -> : Propagation of Election Message
——> : Replying Ack Message
——> : Propagation of Leader Message

Figure 7: An example LE run in a static topology.

incoming elections with lower priority, but participates in an election with *higher* priority after discarding its current election.

To deal with the *dynamic* setting, with node mobility and link failures, the algorithm is extended to handle the following scenarios:

1. When a parent-child pair becomes disconnected during the election process, the parent removes the child from its waiting list of acknowledgements and continues its election. The child terminates the current election by announcing as the leader its maximal downstream node. LE's *probe* mechanism is used to detect a disconnected link between a child and its parent.

2. When a new link forms between two nodes: If one or both nodes are in their elections, the ongoing computation(s) terminate before the exchange of leaders takes place; if both have accomplished their elections and are propagating their own leaders, the exchange of leaders occurs with the *better-valued leader* being the winner.

Node crashes and recoveries can also result in dynamic topologies. LE takes them into account implicitly by treating the former as network partitioning and the latter as network merging.

The main correctness property of LE is that every connected component will select a unique leader, which is the highest-valued node in that component. In [50, 49] the authors prove this correctness property, assuming that the links are bidirectional, but add that "the algorithm should work correctly even in the case of unidirectional links, provided that there is symmetric connectivity between nodes."

5.2. Modeling LE in Real-Time Maude

This section explains how LE has been formalized in Real-Time Maude, and presents 8 of the 23 rewrite rules in our formal model of LE.

5.2.1. Nodes and Messages

We model an LE node as an object of a subclass `LE-Node` of the class `Node`. The new attributes are the identifier of the leader (`leader`), the parent (`parent`), the current best-valued node (`max`), the node's computation number (`number`), its computation index (`src`), which denotes the "index" of the election in which the node is currently participating, the set of neighbors from which the node is expecting, but has yet to receive, an *ack* message (`acks`), a flag indicating whether the node is currently in an election (`eflag`), a flag indicating whether the node has sent an acknowledgement to its parent (`pflag`), the node's (immediate) neighbors (`neighbors`), the new neighbors found by the neighbor discovery process (`newNbs`), and the relevant nodes which can no longer reach the node (`lostConxs`):

```
class LE-Node | leader : Oid,       parent : Oid,         max : Oid,
                number : Nat,       src : CompIndex,      acks : OidSet,
                eflag : Bool,       pflag : Bool,         neighbors : OidSet,
                newNbs : OidSet,    lostConxs : OidSet .
subclass LE-Node < Node .
```

A computation index is a pair o ~ k, with o a node identifier and k a computation number, identifying an election. As in [50], we assume that a node's identifier determines its value.

In LE there are mainly three kinds of messages: `election`, `ack`, and `leader` messages. The first parameter of these messages identifies *which* election the message belongs to. In addition, we use a message `electLeader` to start a run of the leader election process:

```
op election : CompIndex Oid      -> MsgContent [ctor] .   --- computation index
op ack      : CompIndex Bool Oid -> MsgContent [ctor] .   --- highest-valued node
op leader   : CompIndex Oid      -> MsgContent [ctor] .
msg electLeader : Oid -> Msg .
```

5.2.2. Modeling Communication

In LE, nodes broadcast/multicast `election` messages to *immediate neighbors*, and unicast *ack* messages to their parents and "siblings" in the spanning tree under construction. Sending to immediate neighbors may be seen as *one-hop* broadcast/multicast, which we model as explained in Section 3.3: the sender sends a "broadcast message;" after time `sendDly` this broadcast message is distributed to all nodes within transmission range of the sender *at that moment*, and will arrive `rcvDly` time units later.

Unicasting *ack* messages, however, may involve multiple hops, since a node may have moved away from its parent by the time the *ack* message should be sent. As mentioned above, LE does not specify a transport protocol to transmit such messages, but only requires (i) that communication between neighbors is FIFO and (ii) that the destination node must get the message if it is connected to the sender forever from the time when the message is sent. In their simulations, the LE developers execute LE together with the AODV routing protocol and different transport protocols, including TCP, UDP, and UDP broadcast.

In this paper, we abstract from details about how messages are routed, and model multi-hop message transmission as follows:

- the sender sends a `multiHopUnicast` message to the destination node;
- if there *exists* a communication path from source to destination exactly `multiHopSendDelay` time units later, the message will be received by the destination node `multiHopSendDly` + `rcvDly` time units after it was sent.

We model such communication as follows:

```
op multiHopUnicast_from_to_ : MsgContent Oid Oid -> Msg .
op mhTransfer : MsgContent Oid Oid -> Configuration .

vars O O1 O2 : Oid .         var OS : OidSet .         var MC : MsgContent .
var CONF : Configuration .   vars L1 L2 : Location .   var R : Rat .

eq multiHopUnicast MC from O1 to O2 = dly(mhTransfer(MC,O1,O2), multiHopSendDly).
eq {mhTransfer(MC, O1, O2) CONF}
 = if O2 in reachable(O1, CONF) then {dly(MC from O1 to O2, rcvDly) CONF} else {CONF} fi .

op reachable : OidSet Configuration -> OidSet [frozen (2)] .

ceq reachable(O1 ; OS,        --- add O2 to nodes reachable from (O1 ; OS)
              < O1 : Node | currentLocation : L1, transRange : R >
              < O2 : Node | currentLocation : L2 >  CONF)
```

```
   = reachable(O1 ; O2 ; OS,   < O1 : Node | >  < O2 : Node | >  CONF)
     if not (O2 in (O1 ; OS)) /\ L2 withinTransRange R of L1 .

  eq reachable(OS, CONF) = OS [owise] .   --- fixed point reached
```

Since this model abstracts from the actual route by which a message is transported, a message that happens to be transferred in one hop has the same delay as one that uses 10 hops. Our model satisfies the requirement that messages are delivered if there is a path from source to destination forever. However, our model does not guarantee FIFO transmission between neighbors for two reasons:

1. Two one-hop messages sent "at the same time" results in two messages with the same delay, since our model abstracts from details about the buffering of outgoing messages.
2. Since we abstract from routing details, a "multi-hop" message has sending delay `multiHopSendDly` even if it happens to need only one hop, and could be overtaken by a one-hop broadcast message sent later along the same link.

5.2.3. Neighbor and Connectivity Discovery

LE assumes that new (one-hop) links caused by node movement are detected. We model such neighbor discovery abstractly by periodically updating the `newNbs` attribute of each node with those nodes that are within transmission range but are not included in the node's `neighbors` attribute, and by removing from `neighbors` those nodes that are no longer within transmission range.

In LE, the leader of a component "periodically sends out a heartbeat message to other nodes," which can then discover whether they are disconnected from the leader. Each node n also periodically sends a *probe* message to each node n' from which it awaits an *ack* message. If n does not receive a *reply* message from n' within certain time, it assumes that the connection to n' is lost. Finally, LE assumes that a node knows when it becomes disconnected from its parent. We abstract from heartbeat and probe/reply protocols, and instead periodically check whether a connection is lost to nodes in `acks`, the leader, or the parent.

We include in the state a timer object `< 100 : GlobalND | timer :` t `>` that triggers both the neighbor discovery process and the lost connectivity process, periodically, each time the timer expires:

```
  rl [computeNewNbsAndLostConnections] :
     {< O : GlobalND | timer : 0 >   CONF}  =>
     {< O : GlobalND | timer : period >  updateNbsAndAck(CONF)} .
```

where, for each node object o in `CONF`, `updateNbsAndAck`:

1. sets o's `newNbs` attribute to o's current immediate neighbors minus the nodes already in o's `neighbors` attribute;
2. removes all nodes which are no longer o's neighbors from o's `neighbors` attribute;
3. sets o's `lostConxs` attributes to those relevant nodes that cannot reach o (in multiple hops).

We refer to the online specification for the definition of this function.

5.2.4. Timed Behavior

Since we integrate neighbor discovery into our LE model, we will not explicitly handle the timing issue. For example, we do not model a timeout for an awaited acknowledgement.

5.2.5. Modeling the Behavior of LE

Our model of LE consists of five parts: initiating leader election, handling an *election* message, handling an *ack* message, handling a *leader* message, and dealing with new neighbors and lost connections.

Starting Leader Election. A “message” `electLeader(`o`)` kicks off a run of LE with node o as initiator. Node o multicasts a message `election(`o `~` n`)` to all its immediate neighbors, where n is the latest computation number.[4] The source will then wait for the `ack` messages from those neighbors by setting `acks` to `OS`. Moreover, it sets `eflag` to `true`, indicating that it is currently in an election:

```
vars O O' O1 O2 SND LID M M' P : Oid .     vars OS OS1 OS2 : OidSet .
var N : Nat .      var I : CompIndex .     vars B FL : Bool .

rl [init-leader-election] :
   electLeader(O)
   < O : LE-Node | eflag : false,  neighbors : OS,  number : N,  leader : O2 >
 =>
   < O : LE-Node | acks : OS,     src : O ~ N,    number : N + 1,
                   eflag : true,  pflag : false,  parent : O,   max : O >
   (multicast election(O ~ N, O2) from O to OS) .
```

Receiving an Election Message. When a node that is not involved in an election (`eflag` is `false`) receives an `election` message from `SND`, the node sets `SND` as its `parent`, and sets its `src`, `eflag`, and `pflag` attributes accordingly. The node multicasts an `election` message to all its neighbors except the parent:

```
crl [join-1] :
    (election(I, LID) from SND to O)
    < O : LE-Node | eflag : false, leader : LID, neighbors : OS1 >
  =>
    < O : LE-Node | src : I, acks : OS2, eflag : true, pflag : false,
                    parent : SND, max : O >
    (multicast election(I, LID) from O to OS2)
  if OS2 := delete(SND, OS1) .
```

Receiving Ack Messages. When a node receives an `ack` message for the current computation `I`, from a node `SND`, it deletes `SND` from the set `acks`. If the reported best node `M'` is better than the node's current best-valued node `M`, then the `max` attribute is also updated accordingly:

```
rl [update-acks] :
   (ack(I, FL, M') from SND to O)
   < O : LE-Node | pflag : false, src : I, acks : OS, max : M >
 =>
   < O : LE-Node | acks : delete(SND, OS),  max : (if FL and M' > M then M' else M fi) > .
```

All `acks` Received. When a node is no longer waiting for any `ack` message (`acks` is `empty`), and it has not yet sent an `ack` to its parent (`pflag` is `false`), it sends an `ack` message to its parent, with its best-valued node `M`. However, if the node initiated this round of the protocol (and therefore is the root node) it starts propagating the leader `M` to its immediate neighbors:

```
rl [all-acks-received-1] :
   < O : LE-Node | acks : empty, src : (O' ~ N), pflag : false,
                   parent : P,   max : M,        neighbors : OS >
 =>
   if O =/= O'   --- not root node
   then < O : LE-Node | pflag : true >
        (multiHopUnicast ack(O' ~ N, true, M) from O to P)
   else < O : LE-Node | eflag : false, leader : M >
        (multicast leader(O' ~ N, M) from O to OS)  fi .
```

[4] In case there are multiple concurrent runs of the protocol, this index helps deciding which run should continue.

Leader Message Handling. If a node already in an election receives a `leader` message for the first time, it just updates the local `leader`, clears the `eflag` (its election is over), and propagates the received message:

```
crl [adopt-new-leader-1] :
    (leader(I, LID) from SND to O)
    < O : LE-Node | pflag : true, eflag : true, max : M, neighbors : OS >
  =>
    < O : LE-Node | leader : LID, eflag : false, src : I >
    (multicast leader(I,LID) from O to OS)   if M <= LID .
```

New Links. If one or more new neighbors have been found from a node `O` that has already finished its election, then the node multicasts the leader message to the new immediate neighbors:

```
rl [new-links-found] :
   < O : LE-Node | newNbs : O' ; OS, eflag : B, src : I, leader : LID >
 =>
   < O : LE-Node | newNbs : empty >
   (if not B and LID =/= 0 then (multicast leader(I,LID) from O to (O' ; OS)) else none fi) .
```

Lost Connections. If a node gets disconnected from its parent while still in an election, it terminates the diffusing computation by announcing its maximal downstream node as the leader:

```
rl [disconnected-from-parent] :
   < O : LE-Node | lostConxs : OS ; P, pflag : true, eflag : true,
                   parent : P, max : M, src : I, neighbors : OS2 >
  =>
   < O : LE-Node | lostConxs : OS,  eflag : false, leader : M >
   (multicast leader(I, M) from O to OS2) .
```

5.3. Formal Analysis of LE

We compose our MANETs model with our model of the LE protocol model (and our abstract models of neighbor discovery and multi-hop unicast) to analyze LE under realistic mobility and communication models. This includes analyzing LE with both bidirectional links and unidirectional links; the latter are a consequence, e.g., of nodes sending with different signaling power, which could be due to different device capabilities and/or different amounts of remaining energy.

Although many papers (e.g., [50, 13, 46, 16, 45, 21, 47, 48, 31, 14]) have studied LE, it seems that very little is known by way of formal analysis about how it behaves with unidirectional connections and under realistic mobility scenarios. We analyze LE under *four different settings*:

I. single connected component (i.e., each node can reach any other node by at least one (possibly directed) path); only stationary nodes; bidirectional communication channels;

II. single connected component; only stationary nodes; both bidirectional and unidirectional communication channels;

III. single connected component *throughout* the entire leader election process; both stationary and mobile nodes; bidirectional communication channels;

IV. *two* connected components that repeatedly merge and partition because of node movement.

LE Nodes. As explained in Section 3.4, we can combine our protocol specification with a node mobility model by having nodes as object instances of a subclass of both `LE-Node` and a mobility class, such as `RWPNode` for random waypoint mobility:

```
class LE-RWPNode .
subclass LE-RWPNode < RWPNode LE-Node .

class LE-StatNode .
subclass LE-StatNode < StationaryNode LE-Node .
```

We use the numbers 1, 2, ... as node identifiers, and use the value 0 as the undefined/"null" node:

```
subsort Nat < Oid .
```

Formalizing the Correctness Property. We use model checking to analyze the main correctness property of LE, as described in [50]:

"Given a network of mobile nodes each with a value, after a finite number of topological changes, every connected component will eventually select a unique leader, which is the most-valued-node from among the nodes in that component."

The following atomic proposition `unique-leaders` holds if and only if all nodes in a connected component have the same leader, which is, furthermore, the highest-valued node in that connected component:[5]

```
op unique-leaders : -> Prop [ctor] .

eq {< O : LE-Node | leader : 0 >  REST} |= unique-leaders = false .
--- no leader ('O') selected by some node

ceq {< O : LE-Node | leader : O' >  REST}
    |= unique-leaders = false if O' < O .  --- O better than its leader

ceq {< O1 : LE-Node | leader : O' >  < O2 : LENode | >  REST}
    |= unique-leaders = false
   if O' < O2    --- wrong leader selected by O1
      /\ O2 in reachable(O1, < O1 : LENode | >  < O2 : LENode | >  REST) .

eq {SYSTEM} |= unique-leaders = true [owise] .
```

The main correctness property can then be formalized as the LTL formula `<> unique-leaders`. Given an initial state `init`, the following commands check whether the property holds (possible up to the duration of the test round, `roundTime`). If the property does not hold, a trace illustrating a counterexample is shown.

```
(mc {init} |=u <> unique-leaders .)
(mc {init} |=t <> unique-leaders in time <= roundTime .)
```

We can use unbounded model checking for *static* topologies. In dynamic topologies, the locations of the moving nodes contribute to an infinite reachable state space, and time-bounded model checking is therefore required to have a finite reachable state space, which implies that model checking terminates.

5.4. Scenarios and Analysis

We make the following assumptions in our experiments:

- The transmission range of a node is 10 meters, and the test area is $100m \times 100m$.
- The one-hop delays at the sender side and at the receiver side are 1 and 0, respectively. The multi-hop "send" delay is 2 time units.
- `roundTime` (i.e., the time bound in the model checking) is 20.
- All nodes are initially aware of their one-hop neighbors. `leader` and `parent` are initially set to `0`, and `max` is initialized to the node's identifier.

[5]Remember that the value of a node is given by its identifier.

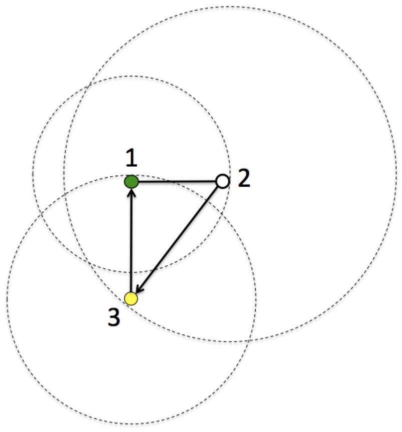

Figure 8: Topology in Scenario II.

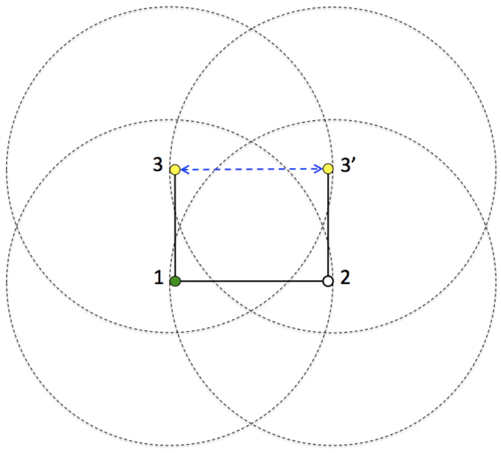

Figure 9: Topology in Scenario III.

Scenario I. Scenario I corresponds to the topology in Fig. 7, and consists of five stationary nodes with bidirectional connections. The nodes `1`, `2`, `3`, `4`, and `5` are located at `(45 . 45)`, `(50 . 50)`, `(50 . 40)`, `(60 . 40)`, and `(60 . 50)`, respectively. We consider two sub-scenarios: (a) only node `1` initiates a round of the leader election protocol; and (b) all five nodes initiate a run of the protocol. Time-bounded model checking shows that the property holds: all five nodes elect the best-valued node `5` as their leader within 20 time units; the execution times of the analyses are 150 milliseconds (ms) and 4500 ms, respectively.

Scenario II. This scenario, shown in Fig. 8 (where a solid line denotes a bidirectional link, an arrow denotes a unidirectional link, and a dashed circle shows a node's transmission area), considers a topology with three stationary nodes, where the links between nodes `2` and `3` and between `3` and `1` are unidirectional. This scenario defines a single connected component in the sense that there is a directed path from each node to any other node. To form such a *unidirectional* but *connected* component, we set the transmission ranges of the source `1` and other two nodes `2` and `3` to $10\,m$, $30\,m$, and $20\,m$, respectively.

Real-Time Maude model checking shows (in 100 ms CPU time) that the desired property is satisfied in this topology with the above system parameters.

Scenario III. Scenario III, shown in Fig. 9 (where a dashed arrow denotes a node's motion path), considers a bidirectional *dynamic* topology with three nodes, where the source node 1 is located at (50 . 50), and nodes 2 and 3 are initially at (60 . 50) and (50 . 55), respectively. Node 3 is a *random waypoint* node that moves back and forth along the dashed arrow with end points (50 . 55) and (60 . 55) (denoted by 3'). Note that the topology remains a connected component despite node 3's movement. We set the pause time of the moving node to 0 and the period of the neighbor/connectivity discovery process to 2.

We experiment with three sub-scenarios: (a) the speed of the moving node is 10; i.e., the speedRange attribute is the singleton 10; (b) the speed is 5; and (c) the speed is again 10, but now the pause time is 1 time unit. The initial state of Scenario III-a is given by the term (with parts of the term replaced by '...'):

```
eq period = 2 .
eq pauseTime = 0 .

eq initConfig
 = electLeader(1)
   < 100 : GlobalND | timer : period >
   < 1 : LE-StatNode | currentLocation : 50 . 50,  transRange : 10,  leader : 0,  max : 0,
                       neighbors : (2 ; 3),  parent : 0,  number : 100,  src : 0 ~ 0,
                       acks : empty,  eflag : false,  pflag : false,  newNbs : empty,
                       lostConxs : empty >
   < 2 : LE-StatNode | currentLocation : 60 . 50,  transRange : 10,  leader : 0,  max : 0,
                       neighbors : 1,  parent : 0,  ...  >
   < 3 : LE-RWPNode | currentLocation : 50 . 55,  transRange : 10,  speed : 0,
                      speedVector : < 0 , 0 >,  speedRange : 10,
                      destRange : (60 . 55) ; (50 . 55),  timer : pauseTime,
                      status : pausing,  leader : 0,  neighbors : 1,  ...  > .
```

Real-Time Maude model checking of Scenario III-a shows (in 240 ms CPU time) that the desired property is *not* satisfied: Node 3 moves away from Node 1 *during* Node 1's multicast to "immediate neighbors," and is not within Node 1's transmission range when the sending delay of the one-hop multicast of election messages to "immediate neighbors" *expires*. Therefore, Node 3 does not get this message. Furthermore, the neighbor discovery process takes place every 2 time units, which exactly coincides with the moments when Node 3 is close to Node 1! The neighbor discovery process therefore never discovers that Node 3 is *not* an immediate neighbor of Node 1, and will hence never discover that Node 3 is a *new* neighbor. Node 3 will therefore be left out of the election process forever.

We cannot claim that this behavior *invalidates* the LE protocol, since LE may be based on other assumptions, such as "continuous neighbor discovery" and/or *multi-hop* communication even to nodes that are immediate neighbors when a sending event begins. However, this spurious behavior shows (at least) the need to make explicit subtle requirements of the underlying neighbor discovery process, and to make more precise the meaning of sending to "immediate neighbors" when an immediate neighbor may cease to be one *during* the sending process.

Real-Time Maude model checking of Scenarios III-b and III-c show that the desired property holds in these scenarios. The only difference between Scenarios III-a and III-c is that pauseTime is 1 in Scenario III-c. This implies that Node 3 takes three time units to move from location 3 to location 3', and back. Since the neighbor discovery process takes place every two time units, it will sooner or later take place when Node 3 is in location 3' in Fig 9, and hence no longer is an immediate neighbor of Node 1. Sometime later, the neighbor discovery will take place when Node 3 is again close to Node 1, and will discover the "new" link between Nodes 1 and 3, and will then involve Node 3 in the election.

In Scenario III-b, it takes Node 3 two time units to move between the locations 3 and 3' in Fig. 9, and the neighbor discovery process will therefore update the neighbor information every time Node 3 reaches one of these end-points.

Scenario IV. Finally, to analyze merge and partition of connected components during the leader election processes in two connected components, we consider the system with two connected components (consisting

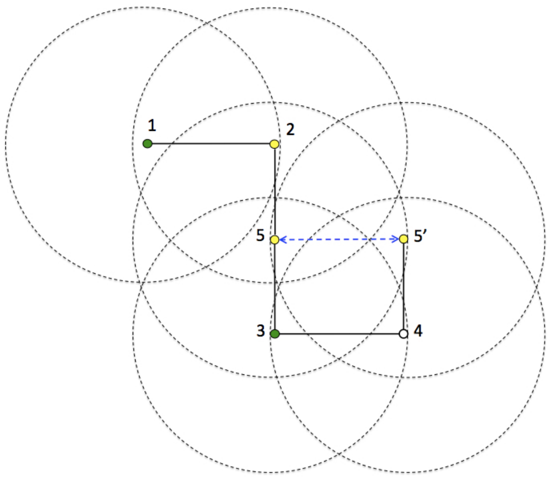

Figure 10: Topology in Scenario IV: two connected components that repeatedly merge and partition.

of Nodes `1` and `2`, and of Nodes `3`, `4`, and `5`, respectively) in Fig. 10. Since Node `5` moves back and forth between position `5` and position `5'` in Fig. 10, the two connected components will repeatedly merge (when Node `5` is close to position `5'`) and partition (when Node `5` is close to position `5`).

Our model checking analysis shows that the property holds when both Node `1` and Node `3` initiate elections at the same time and when `pauseTime` is 8.

6. Related Work

There are a number of formal specification and analysis efforts of MANETs in general, and AODV in particular. Bhargavan et al. [4] use the SPIN model checker to analyze AODV. They only consider a 3-node topology with one link break, but without node movement, and communication delay is not considered. Chiyangwa et al. [8] apply the real-time model checker UPPAAL to analyze AODV. They only consider a static linear network topology. Although they take communication delay into account, the effect of mobility on communication delay is not considered, since the topology is fixed. Fehnker et al. [12] also use UPPAAL to analyze AODV. They also only consider static topologies, or simple dynamic topologies by adding or removing a link, and those topologies are based on the connectivity graph without concrete locations for nodes. Furthermore, no timing issues are considered. Höfner et al. [19] apply statistical model checking to AODV. However, mobility is simply considered by arbitrary instantaneous node jumping between zones that split the whole test grid. Although they take into account the communication delay, the combination of mobility and communication delay is not considered.

None of these studies has built a generic framework for MANETs. Our modeling framework aims at the combination of (bidirectional/unidirectional) wireless communication and mobility, and allows formal modeling and analysis of protocols under realistic mobility models.

On the process algebra side, [34, 35, 17, 29, 46, 15, 18, 30, 11, 13, 46, 16, 45, 31, 21, 47, 48, 28, 26, 27, 51, 52, 53] have been proposed as process algebraic modeling languages for MANETs. These languages feature a form of local broadcast, in which a message sent by a node could be received by other nodes “within transmission range.” However, connectivity is only considered abstractly, without taking into account concrete locations and the transmission range of nodes. Furthermore, [34] only considers fixed network topologies, whereas the others (except [18]) deal with arbitrary changes in topology. Godskesen et al. [18] consider realistic mobility, and propose concrete mobility models. However, no protocol application or automated analysis is given, and communication delay is not taken into account. Merro et al. [30] propose a timed calculus with time-consuming communications, and equip it with a formal semantics to analyze communication collisions. In general, these studies have proposed modeling/analysis frameworks for MANETs, but they lack the expressiveness to model both mobility, bidirectional/unidirectional wireless communication, and/or timing issues.

The leader election protocol LE has also been subjected to a number of formal analysis efforts. Gelastou et al. [13] specify and verify LE using both I/O automata and process algebra. They only consider static bidirectional topologies with non-lossy channels, and communication delay is not taken into consideration. Singh et al. [46] present the ω-calculus for formally modeling and reasoning about MANETs, and illustrate their techniques by developing and analyzing a formal model of LE. They only consider dynamic bidirectional topologies where a node is free to move as long as the network remains connected, without taking into account unidirectional scenarios, communication delay, and message loss. Ghassemi et al. [16] provide a framework for modeling and analyzing both qualitative and quantitative aspects of MANET protocols, where communication delay and dynamic topology (modeled by probabilistic message loss) are considered. They focus on the performance of LE under various parameters without giving any qualitative results. Sibilio et al. [45, 31] propose a calculus for trustworthy MANETs with which they analyze a secure version of LE (neighbors trust each other at some security level) with three stationary nodes connected by bidirectional links. Kouzapas et al. [21] propose a calculus of dynamic networks whose semantics contain rules mimicking the behavior of a neighbor discovery protocol (just as we integrate the underlying neighbor discovery into our model). They analyze LE with an arbitrary derivative of the initial state based on bisimilarity under the assumption of no message loss. Song et al. [47, 48] introduce a stochastic broadcast calculus for MANETs with mobility modeled stochastically. They analyze a simplified model of LE with four nodes, where the mobility of (only) one node affects the transmission probability. In [14], Ghassemi *et al.* introduce both constrained labeled transition systems to represent mobility and a branching-time temporal logic to model check MANET protocols. They specify the correctness property of LE, but do not verify it in detail. Finally, the developers of LE present in their accompanying technical report [49] a "formal" specification of LE and use temporal logic to prove the correctness of the protocol, assuming bidirectional connections and no message loss.

The work presented in this paper distinguishes itself by modeling node locations, transmissions ranges, message loss, communication delay, well known mobility models, neighbor discovery, and uni/bidirectional connectivity, as well as their interrelations. From a modeling perspective:

- Related work does not model node locations explicitly, but represent the topologies abstractly as "neighborhood graphs."

- Related work therefore does not consider *realistic mobility models*, but only static topologies or simple dynamic topologies with arbitrary link breaks.

  In particular, related work on defining a modeling framework for MANETs or targeting specific protocols such as AODV or LE, model the network topology or neighborhood connectivity in an abstract way without representing the actual location of a node, the distance between nodes, or the transmission range of a node. Instead, the topology is described as a set of connectivity graphs denoting the possible connectivities between nodes within the network.

  The following related work encodes node locations and/or transmission ranges abstractly: [29] encodes a node's location and radius into the syntax of the calculus, but does not consider concrete locations or actual distances between nodes; [18] describes mobility models using abstract functions on vectors, but, again, does not model distances between nodes or transmission range; [19] models the network topology over a 2D rectangular grid, provides each node with transmission range, and uses a topology-based mobility model where only the change of the connectivity graph due to node movement is considered; [34] models node transmission range and locations abstractly and uses abstract functions to compute distances between nodes.

- Related work does not consider *unidirectional connectivity*.

- Regarding *neighbor discovery*, though most related work considers neighbor connectivity, they implicitly model its behaviors as arbitrary changing of connections (though [21] mentions that the modeling framework is embedded with the underlying neighbor discovery, it handles neighborhood changes as others). We explicitly integrate an abstract neighbor (and multi-hop connectivity) discovery process into our model, and model its dynamic behaviors accordingly along with node mobility.
- Regarding *transmission mode*, most related work considers broadcast and unicast. Only [45, 47, 48, 31] consider groupcast.
- Regarding *message loss*, [47, 48, 16] consider probabilistic message transmission, and [14] considers lossy communication and loss of connectivity, whereas message loss in our model comes from node mobility and unidirectional connections.
- Only [16, 47, 48] consider *communication delay*.
- To the best of our knowledge no related work considers the *the interplay of all the above ingredients*.

Finally, Maude and Real-Time Maude have been applied to analyze wireless systems. Our previous work [24, 25] builds the modeling framework that serves as the basis for this paper, and analyzes AODV and LE under different mobility models. The papers [20, 39] model wireless sensor networks in (Real-Time) Maude, but do not consider node mobility.

7. Conclusions

The formal modeling of MANETs presents non-trivial challenges, because physical aspects such as geometrical location, mobility, and wireless communication need to be taken into account to faithfully specify and verify their requirements. Although, as discussed in Section 6, there is already a substantial body of work on the formal analysis of MANETs, there is still a substantial gap between the more abstract level at which those analyses have been carried out and the more detailed level at which various physical aspects having a direct bearing on required behavior have to be considered. To address this problem we have presented a new formal framework for MANETs based on Real-Time Maude and having good expressiveness and compositionality features. We have also shown how the most commonly used mobility models as well as wireless communication and its interactions with mobility can be naturally expressed in the framework. Furthermore, we have presented two case studies, for the AODV and LE protocols, showing the effectiveness of the formal framework and of Real-Time Maude in analyzing MANETs in sufficient detail and uncovering problematic behaviors due to mobility. In particular, our analysis uncovered a previously unreported spurious behavior in the well-analyzed LE protocol that is due to the subtle interplay between neighbor discovery, node movement, and communication delay. This behavior would be impossible to find if those aspects are abstracted away. Our analysis does not explicitly falsify LE—due to the vague description of the assumptions on communication and neighbor discovery in [49]—but at least shows a potentially troubling behavior and the need to make certain assumptions more precise.

As usual, much work remains ahead. First, the ground is now ready for analyzing various compositions not only of one MANET protocol with various mobility models, but also of composed MANET protocols with such models. Second, a broader collection of MANET protocols should be analyzed and the experience gained should be used to improve the framework and its methodology. Third, suitable abstraction methods that can drastically reduce the state space while still modeling relevant physical aspects at a more abstract level should be developed. Fourth, we analyze the systems from particular *single* initial states, with given (initial) locations, speed ranges, neighbor discovery time intervals, and so on. It would be desirable to somehow cover entire parameter spaces. Recent developments in combining rewriting with SMT solving [43] might make such analysis possible and should be further investigated. Fifth, various performance and Quality of Service (QoS) requirements for MANETs should also be supported by the formal framework. For this, the use of probabilistic rewrite theories [1] together with statistical model checking tools e.g., [44, 1, 2] have already been shown to be quite well suited for analyzing various network protocol applications, including sensor networks (see, e.g., [3, 20, 10]), so the extension of these methods to MANETs should be straightforward. Last, but not least, the framework and its methodology should be used not just for *post mortem* analysis of existing MANET protocols, but also as an integral part of the design of new MANET protocols. In particular, its usefulness in identifying design errors early in the design process, shortening its length, and resulting in MANET designs of higher quality and reliability should be evaluated.

Acknowledgements. We thank the anonymous reviewers for their very insightful comments on an earlier version of this paper that has led to substantial improvements. This work was partially supported by NSF grant CNS 13-19109.

References

[1] G. Agha, J. Meseguer, K. Sen, PMaude: Rewrite-based specification language for probabilistic object systems, ENTCS 153 (2006) 213–239.

[2] M. AlTurki, J. Meseguer, PVeStA: A parallel statistical model checking and quantitative analysis tool, in: Proc. CALCO'11, volume 6859 of *Lecture Notes in Computer Science*, Springer, 2011.

[3] M. AlTurki, J. Meseguer, C. Gunter, Probabilistic modeling and analysis of DoS protection for the ASV protocol, ENTCS 234 (2009) 3–18.

[4] K. Bhargavan, D. Obradovic, C. Gunter, Formal verification of standards for distance vector routing protocols, Journal of the ACM 49 (2002) 538–576.

[5] R. Bruni, J. Meseguer, Semantic foundations for generalized rewrite theories, Theoretical Computer Science 360 (2006) 386–414.

[6] T. Camp, J. Boleng, V. Davies, A survey of mobility models for ad hoc network research, Wireless Communications and Mobile Computing 2 (2002) 483–502.

[7] A. Cerone, M. Hennessy, M. Merro, Modelling MAC-layer communications in wireless systems, in: Proc. COORDINATION'13, volume 7890 of *Lecture Notes in Computer Science*, Springer, 2013.

[8] S. Chiyangwa, M.Z. Kwiatkowska, A timing analysis of AODV, in: Proc. FMOODS'05, volume 3535 of *Lecture Notes in Computer Science*, Springer, 2005.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude, volume 4350 of *Lecture Notes in Computer Science*, Springer, 2007.

[10] J. Eckhardt, T. Mühlbauer, M. AlTurki, J. Meseguer, M. Wirsing, Stable availability under denial of service attacks through formal patterns, in: Proc. FASE'12, volume 7212 of *Lecture Notes in Computer Science*, Springer, 2012.

[11] A. Fehnker, R. van Glabbeek, P. Höfner, A. McIver, M. Portmann, W. Tan, A Process Algebra for wireless mesh networks used for modelling, verifying and analysing AODV, Technical Report 5513, NICTA, 2012.

[12] A. Fehnker, R.J. van Glabbeek, P. Höfner, A. McIver, M. Portmann, W.L. Tan, Automated analysis of AODV using Uppaal., in: Proc. TACAS'12, volume 7214 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 173–187.

[13] M. Gelastou, C. Georgiou, A. Philippou, On the application of formal methods for specifying and verifying distributed protocols, in: NCA, IEEE, 2008.

[14] F. Ghassemi, S. Ahmadi, W. Fokkink, A. Movaghar, Model checking MANETs with arbitrary mobility, in: Proc. FSEN'13, volume 8161 of *Lecture Notes in Computer Science*, Springer, 2013.

[15] F. Ghassemi, W. Fokkink, A. Movaghar, Restricted broadcast process theory, in: Proc. SEFM '08, IEEE, 2008.

[16] F. Ghassemi, M. Talebi, A. Movaghar, W. Fokkink, Stochastic restricted broadcast process theory, in: EPEW, volume 6977 of *Lecture Notes in Computer Science*, Springer, 2011.

[17] J.C. Godskesen, A calculus for mobile ad hoc networks, in: Proc. Coordination'07, volume 4467 of *Lecture Notes in Computer Science*, Springer, 2007.

[18] J.C. Godskesen, S. Nanz, Mobility models and behavioural equivalence for wireless networks, in: Proc. Coordination'09, volume 5521 of *Lecture Notes in Computer Science*, Springer, 2009.

[19] P. Höfner, M. Kamali, Quantitative analysis of AODV and its variants on dynamic topologies using statistical model checking, in: Proc. FORMATS'13, volume 8053 of *Lecture Notes in Computer Science*, Springer, 2013.

[20] M. Katelman, J. Meseguer, J.C. Hou, Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking, in: Proc. FMOODS'08, volume 5051 of *Lecture Notes in Computer Science*, Springer, 2008.

[21] D. Kouzapas, A. Philippou, A process calculus for dynamic networks, in: Proc. FMOODS/FORTE'11, volume 6722 of *Lecture Notes in Computer Science*, Springer, 2011.

[22] D. Lepri, E. Ábrahám, P.C. Ölveczky, A timed CTL model checker for Real-Time Maude, in: Proc. CALCO'13, volume 8089 of *Lecture Notes in Computer Science*, Springer, 2013.

[23] D. Lepri, E. Ábrahám, P.C. Ölveczky, Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories, Science of Computer Programming 99 (2015) 128–192.

[24] S. Liu, P. Ölveczky, J. Meseguer, A framework for mobile ad hoc networks in Real-Time Maude, in: Proc. WRLA'14, volume 8663 of *Lecture Notes in Computer Science*, Springer, 2014.

[25] S. Liu, P.C. Ölveczky, J. Meseguer, Formal analysis of leader election in MANETs using Real-Time Maude, in: Software, Services, and Systems: Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering, volume 8950 of *Lecture Notes in Computer Science*, Springer, 2015.

[26] S. Liu, X. Wu, Q. Li, H. Zhu, Q. Wang, Formal approaches to wireless sensor networks, in: SSIRI 2011, IEEE Computer Society, 2011, pp. 11–18.

[27] S. Liu, Y. Zhao, H. Zhu, Q. Li, A calculus for mobile ad hoc networks from a group probabilistic perspective, in: HASE 2011, IEEE Computer Society, 2011, pp. 157–162.
[28] S. Liu, Y. Zhao, H. Zhu, Q. Li, Towards a probabilistic calculus for mobile ad hoc networks, in: TASE 2011, IEEE Computer Society, 2011, pp. 195–198.
[29] M. Merro, An observational theory for mobile ad hoc networks (full version), Information and Computation 207 (2009) 194–208.
[30] M. Merro, F. Ballardin, E. Sibilio, A timed calculus for wireless systems, Theoretical Computer Science 412 (2011) 6585–6611.
[31] M. Merro, E. Sibilio, A calculus of trustworthy ad hoc networks, Formal Aspects of Computing 25 (2013) 801–832.
[32] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1992) 73–155.
[33] J. Meseguer, Membership algebra as a logical framework for equational specification, in: Proc. WADT'97, volume 1376 of *Lecture Notes in Computer Science*, Springer, 1998.
[34] N. Mezzetti, D. Sangiorgi, Towards a calculus for wireless systems, ENTCS 158 (2006) 331–353.
[35] S. Nanz, C. Hankin, A framework for security analysis of mobile wireless networks, Theoretical Computer Science 367 (2006) 203–227.
[36] P.C. Ölveczky, Real-Time Maude and its applications, in: Proc. WRLA'14, volume 8663 of *Lecture Notes in Computer Science*, Springer, 2014.
[37] P.C. Ölveczky, J. Meseguer, Specification of real-time and hybrid systems in rewriting logic, Theoretical Computer Science 285 (2002) 359–405.
[38] P.C. Ölveczky, J. Meseguer, Semantics and pragmatics of Real-Time Maude, Higher-Order and Symbolic Computation 20 (2007) 161–196.
[39] P.C. Ölveczky, S. Thorvaldsen, Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude, Theoretical Computer Science 410 (2009) 254–280.
[40] C. Perkins, E. Belding-Royer, S. Das, Ad hoc on-demand distance vector (AODV) routing. RFC 3561 (experimental), http://www.ietf.org/rfc/rfc3561, 2003.
[41] C.E. Perkins, E.M. Belding-Royer, Ad-hoc on-demand distance vector routing, in: 2nd Workshop on Mobile Computing Systems and Applications (WMCSA '99), IEEE Computer Society, 1999, pp. 90–100.
[42] S. Ping, Delay Measurement Time Synchronization for Wireless Sensor Networks, Technical Report IRB-TR-03-013, Intel Research Berkeley, 2003. http://www.intel-research.net/Publications/Berkeley/081120031327_137.pdf.
[43] C. Rocha, J. Meseguer, C.A. Muñoz, Rewriting modulo SMT and open system analysis, in: Proc. WRLA'14, volume 8663 of *Lecture Notes in Computer Science*, Springer, 2014.
[44] K. Sen, M. Viswanathan, G. Agha, On statistical model checking of stochastic systems, in: Proc. CAV'05, volume 3576 of *Lecture Notes in Computer Science*, Springer, 2005.
[45] E. Sibilio, Formal Methods for Wireless Systems, Ph.D. thesis, University of Verona, 2011.
[46] A. Singh, C.R. Ramakrishnan, S.A. Smolka, A process calculus for mobile ad hoc networks, Science of Computer Programming 75 (2010) 440–469.
[47] L. Song, Probabilistic Models and Process Calculi for Mobile Ad Hoc Networks, Ph.D. thesis, IT University of Copenhagen, 2012.
[48] L. Song, J.C. Godskesen, Broadcast abstraction in a stochastic calculus for mobile networks, in: Proc. IFIP TCS'12, volume 7604 of *Lecture Notes in Computer Science*, Springer, 2012.
[49] S. Vasudevan, J.F. Kurose, D.F. Towsley, Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks, Technical Report UMass CMPSCI 03-20, University of Massachusetts, 2003.
[50] S. Vasudevan, J.F. Kurose, D.F. Towsley, Design and analysis of a leader election algorithm for mobile ad hoc networks, in: Proc. ICNP'04, IEEE, 2004.
[51] X. Wu, S. Liu, H. Zhu, Y. Zhao, Reasoning about group-based mobility in manets, in: PRDC 2014, IEEE Computer Society, 2014, pp. 244–253.
[52] X. Wu, S. Liu, H. Zhu, Y. Zhao, L. Chen, Modeling and verifying the ariadne protocol using CSP, in: ECBS 2012, IEEE Computer Society, 2012, pp. 24–32.
[53] X. Wu, H. Zhu, Y. Zhao, Z. Wang, S. Liu, Modeling and verifying the ariadne protocol using process algebra, Comput. Sci. Inf. Syst. 10 (2013) 393–421.