Journal of Theoretical and Applied Information Technology
10th February 2016. Vol.84. No.1
© 2005 - 2015 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

REASONING ON STARVATION IN AODV USING ABSTRACT STATE MACHINES

[1]ALESSANDRO BIANCHI, [2]SEBASTIANO PIZZUTILO, [3]GENNARO VESSIO
[1]Assistant Prof., Department of Informatics, University of Bari, Italy
[2]Associate Prof., Department of Informatics, University of Bari, Italy
[3]Ph.D. student, Department of Informatics, University of Bari, Italy
E-mail: [1]alessandro.bianchi@uniba.it, [2]sebastiano.pizzutilo@uniba.it, [3]gennaro.vessio@uniba.it

ABSTRACT

Abstract State Machines (ASMs) are very helpful in analyzing critical and complex systems, but they lack of inherent, domain-independent characterizations of computationally interesting properties. Our long-term research aims at providing an ASM-based characterization of the *starvation-freedom* property. To this end, in the present paper the Ad-hoc On-demand Distance Vector (AODV) routing protocol for Mobile Ad-hoc NETworks (MANETs) is modeled through ASMs, and starvation is studied. This experience suggests us to focus on *vulnerable rules* as the key issue that drives the risk of starvation within the ASM framework.

Keywords: *Abstract State Machines, Verification, Starvation, MANETs, AODV*

1. INTRODUCTION

Several formalisms are successfully applied to the development of critical and complex systems in a wide range of application domains, and to their *ex-ante* and *ex-post* analysis aimed at investigating, verifying and validating functionality and quality issues. Representing the system-under-study at a high level of abstraction allows developers to focus on algorithmic aspects, rather than on specific realizations of solutions at lower levels. Moreover, the mathematical foundation of formal methods provides complete and unambiguous investigations about the properties and the behavior the system-under-study is required to exhibit.

In this context researchers usually distinguish two classes of computationally interesting properties [1]. Safety properties specify that "something bad never happens"; for instance, in a mutual exclusion algorithm the "bad thing" is when two or more processes are in the critical section. Instead, *liveness* properties stipulate that "something good eventually happens"; for instance, in the same algorithm, the "good thing" is that each process eventually enters the critical section. Well-known examples of liveness properties are the *reachability* of a certain state, the *reversibility* to a previous state, *termination*, *starvation-freedom*, and so on.

Some formalisms provide inherent characterizations of properties, in the sense that they can be viewed as independent from the application domain, so that the formal verification of the computationally interesting properties of the modeled systems can be easily conducted. For example, in the Petri Net framework [2] a marking M_i is reachable from an initial marking M_0 if there exists a sequence of transitions such that M_0 is transformed into M_i. If a marking is not reachable, then the transitions it drives are useless and can be deleted. Unfortunately, not all formalisms provide such features: in fact, from this point of view, Abstract State Machines (ASMs) [3] lack.

This paper specifically deals with the starvation-freedom property, here intended as the capability of a process to "make progress infinitely often" [4]. Our long-term research aims at providing an ASM-based characterization of starvation-freedom in order to systematically investigate it. The aim is to enlarge the general body-of-knowledge of the ASM framework and reinforce it as a conceptual tool that developers can find useful and practical for investigating starvation issues in an *operational* fashion.

To this end, in this paper we report on a case study in which an ASM-based model of the Ad-hoc On-demand Distance Vector (AODV) routing protocol [5] for Mobile Ad-hoc NETworks (MANETs) [6] is used for studying starvation. More precisely, the model allows us to identify a *vulnerable rule* which captures starvation risks inside ASMs, and analyze the refinement needed to ensure that starvation-freedom is satisfied. It is worth noting that

AODV is only studied, but no empirical investigation is here presented.

The rest of this paper is structured as follows. The next section is about related work. Then, background on both ASMs and AODV is provided. The core of the paper is the ASM model of AODV, and its discussion. Finally, the paper concludes with the description of future work.

2. RELATED WORK

The ASM framework supports both *manual* and *automatic* formal verification of systems.

Concerning manual analysis, in [7] numerous proofs are provided to illustrate how a modeler can verify properties of a given ASM. Indeed, ASMs are machines equipped with a notion of run that lend themselves to traditional mathematical reasoning or mental simulation. These proofs range from simple to complex and are conceived for being used by human experts. Another approach is provided in [8], where a verification calculus based on the Hoare logic is proposed. However, the calculus only considers partial correctness, i.e. the result of the computation is what was expected, and is only tailored for a specific class of ASMs. Moreover, it is worth noting that a logic for ASMs exists [9]. However, it does not provide characterizations of specific computationally interesting properties, such as starvation-freedom. The ASM notion of run is very helpful for supporting the practitioners' work, independently from the possibility of developing automatic verification mechanisms. Nevertheless, since it requires human effort, the manual approach does not offer absolute guarantee and is error-prone. Moreover, it often requires to deal with theorems, lemmas, etc., that are quite distant from the developers' average background.

Concerning automatic analysis, several examples of model checking techniques applied to ASMs exist, for example [10] and [11]. However, the Turing-completeness of the formalism [3] causes an unavoidable drawback: properties are, in general, undecidable, so the formal verification of ASM specifications cannot be fully automatized [12]. In fact, an algorithm capable of verifying a specific configuration of a given ASM would be able to verify that a certain configuration, e.g. an halting one, is reachable by a Turing machine expressed by means of an ASM. Since the halting problem for Turing machines is undecidable, such an algorithm cannot exist. For this reason, the translation of the given ASM under study into the input required by the adopted model checker may cause a loss of expressive power.

Our long-term study is aimed at proposing an approach to support the properties analysis capable to overcome the previous limitations. On one hand, we want to provide operational characterizations of properties, so that the manual analysis can be perceived more practical when reasoning about the systems' behavior. On the other hand, since the translation of the given ASM under study into a less expressive model is not needed, these properties can be investigated preserving the expressiveness of the model before the application of usual model checking techniques.

Compared to the other well-known approaches to the problem of verifying properties, we focus on ASMs because of the advantages they provide under several viewpoints. When the expressivity is considered, ASMs represent a general model of computation which "subsumes" all other classic computational models [13], [14], [15]. In fact, [7] emphasizes the naturalness with which other computational models, such as Turing machines, can be directly defined as ASM instances without any extraneous encoding (the vice versa is not always true). Thanks to this generality, ASMs suffice to capture the behavior of wide classes of sequential [3] and parallel [16] algorithms, and also a large class of distributed algorithms [17]. Secondly, concerning understandability, the ASM approach provides a way to describe algorithmic issues in a simple abstract pseudo-code, which can be translated into a high level programming language source code in a quite simple manner [7]. Thirdly, considering methodological issues, the ASM formalism has been successfully applied for the design and analysis of critical and complex systems in several domains, and a specific development method got prominence in the last years [7]. Finally, considering the implementation point of view, the capability of translating formal specifications into executable code, in order to conduct simulations of the models, is provided by tools like CoreASM [18].

Concerning the specific MANET context, to the best of our knowledge few works have used ASMs in this domain. In [19] they serve to specify location services and position-based routing. In [20] we adopt them for specifying a variant of AODV aimed at improving the network topology awareness of the network nodes. Finally, in [21] we show the suitability of the ASM-based approach in capturing the specific MANET features (concurrency, communications, mobility, and so on), and for reasoning about them. The present paper moves from the

ASM-based model of AODV we presented there and significantly extends it by focusing on the capability of the formalism to capture starvation issues.

3. BACKGROUND

3.1 Abstract State Machines

Briefly speaking, Abstract State Machines are finite sets of so-called *rules* of the form **if** *condition* **then** *updates* (possibly with the **else** clause in addition) which transform the *abstract* states of the machine [7]. The concept of abstract state extends the usual notion of *state* occurring in finite state machines: it is an arbitrary complex structure, i.e. a domain of objects with functions and relations defined on them. On the other hand, a rule reflects the notion of *transition* occurring in traditional transition systems: *condition* is a first-order formula whose interpretation can be true or false; while *updates* is a finite set of assignments of the form $f(t_1, \ldots, t_n) := t$, whose execution consists in changing in parallel the value of the specified functions to the indicated value.

Pairs of function names together with values for their arguments are called *locations* [7]: they abstract the notion of memory unit. Therefore, the current configuration of locations together with their values determines the current state of the ASM. In order to better understand the semantics of the states with respect to the computational behavior of the modeled system, it is worth remarking that each ASM state can be characterized by one or more predicates over the states. More precisely, in [22] we define a predicate ϕ over an ASM state s as a first-order formula defined over the locations in s, such that that $s \vDash \phi$. In other words, each predicate is expressed by the logical conjunction of the interesting locations' values.

In each state all conditions are checked, so that all updates in rules whose conditions evaluate to true are simultaneously executed, and the result is a transition of the machine from a state to another, i.e. from a configuration of values in locations to another. Moreover, for the unambiguous determination of the next state, updates must be *consistent*, i.e. no pair of updates must refer to the same location.

The formalism also supports the mechanism of procedure calls; this is achieved by the definition of ASM *submachines*, i.e. parameterized rules, which supports the declaration of *local* functions, so that each call of a submachine works with its own instantiation of its local functions.

A generalization of basic ASMs is represented by Distributed ASMs (DASMs) [7], capable of capturing the formalization of multiple agents acting in a distributed environment. Essentially, a DASM is intended as an arbitrary but finite number of independent agents, each executing its own underlying ASM. In a DASM the keyword **self** is used for supporting the relation between local and global states and for denoting the specific agent which is executing a rule.

Moreover, there is a distinction among functions, depending on the different roles that locations can assume in a given ASM [7]. A primary distinction concerns *basic* functions, intended as elementary, and *derived* functions, whose values are defined in terms of other (basic or derived) functions, but neither the ASM nor the environment (and other ASMs in the case of DASMs) can update them: they are automatically updated as a side effect of the updates over the functions from which they derive. In addition, basic functions are classified into *static*, whose values never change during a run, and *dynamic*, for which values change as a consequence of the updates executed by the ASM or by its environment (or other agents). Furthermore, dynamic functions can be: *controlled* if directly updated only by the ASM; *monitored* if directly updated only by the environment or other agents, and only read by the ASM; *shared*, which are both controlled and monitored; *out*, which are updated, but never read by the ASM.

Finally, the *ASM Method* defined in [7] encloses development phases from requirements capture to implementation in a unique ASM-based framework. Requirements can be captured by constructing so-called *ground models*, i.e. representations at high level of abstraction that can be graphically depicted; then, starting from ground models, a hierarchy of intermediate models is constructed by *stepwise refinements*, leading to executable code: each refinement describes the same system at a finer granularity. The method then supports both verification, through formal proof, and validation, through simulation, although it lacks of inherent characterizations of properties as in other formalisms.

3.2 AODV Routing Protocol for MANETs

A Mobile Ad-hoc NETwork [6] is a wireless network designed for communications among nomadic hosts in absence of fixed infrastructure. These networks are useful, sometimes necessary, for allowing hosts to communicate when fixed infrastructures cannot be used, for example for supporting rescue teams operating where pre-existing infrastructures are not reliable [23]. Hosts are intended as autonomous agents: they can dispose without according to a predefined topology; moreover, during their lifetime, they can enter or leave the network at will and continuously change their relative position.

The twofold role played by hosts (which can act both as end-point and intermediate router), as well as the continuous change of the network topology due to movement, poses the need to define specific routing protocols for properly managing the lack of fixed infrastructure. In fact, since each host can directly communicate only within the area established by its transmission range, these protocols need to take into account the contribution of intermediate hosts for realizing communications. The literature proposes several protocols, and among them Ad-hoc On-demand Distance Vector is one of the most popular and simple.

AODV is a *reactive* protocol that discovers and maintains routes on-demand [5]. Routes are built only as desired by initiator nodes using a route request/route reply cycle, which allows each node to update its own routing table. When an initiator wants to start a communication session to a destination, and a proper route is not known, it broadcasts a route request (RREQ) packet to all its neighbors. An RREQ packet includes among the others: *initiator address* and *broadcast id* (this pair uniquely identifies the packet); *destination address*; *destination sequence number*, which expresses the freshness of the information about destination; and *hop count*, initially set to 0, and increased by each intermediate node, for expressing the distance. Because of broadcast transmissions, each intermediate node can receive several instances of a given RREQ from different neighbors: possible duplications of RREQs are discarded.

Knowledge of routes is stored into routing tables. A routing table in a node lists all other (known) nodes in the network, and the best (known) routes to reach them. To this end, each entry in the table includes the address of the node, its sequence number, the hop count to reach it, and the *next hop* field identifying the next node in the route to reach it.

When a node receives an RREQ, it checks if one of the following holds: destination is one of its neighbors; or it knows a route to destination with corresponding sequence number greater than or equal to the one contained into the RREQ (this means that its knowledge about the route is recent). If so, it unicasts a route reply (RREP) packet back to initiator; otherwise, it updates the hop count field and rebroadcasts the RREQ to its neighbors, so that the process is reiterated. An RREP packet contains: *initiator* and *destination address*, *destination sequence number*, and *hop count*. While the RREP travels towards initiator, routes are set up inside the routing tables of the traversed hosts. When initiator receives the RREP, communication starts.

The protocol also includes mechanisms for recording the up-to-date information about the broken links, but this issue is outside the scope of the present paper.

4. ASM MODEL OF AODV

A MANET adopting AODV can be modeled by a DASM including a set of *agents* = $\{a_1, \ldots, a_n\}$, where each agent models the behavior of a node executing the protocol. We can think that each a_i is univocally identified by the IP address. Note that in general a host is characterized by more features, e.g. the amplitude of the area in which it is able to transmit, the direction and the speed of its movement, and so on. However, since our purpose is to focus on the route discovery process, these features are abstracted away: for each agent we only take into account its neighborhood. In the following, we firstly focus on the main algorithm implementing the protocol, so obtaining a starvation-prone ASM (*sp*-ASM in the following); then the refinement able to overcome starvation risk is modeled, and a starvation-free ASM (*sf*-ASM) is obtained.

The case study here described is elaborated with respect to both the original, abstract specification of the protocol [5], and its discussion in terms of ASMs [21] with the inclusion of aspects tailored to our purposes.

4.1 Starvation-prone ASM

Since all agents implement the same protocol, each agent behaves according to the same ASM, so only one ASM is discussed in the following.

Each ASM can be in one of several states expressing the parallel computational activities of each host in the MANET. For the purposes of the present work, in order to define the predicates over these states [22], the interesting locations' values deal with the following functions:

- *wishToInitiate*: *agents* × *agents* → *boolean*, which is a shared function indicating whether a new communication session to *dest* is required by the environment;
- *receivedRREQ*: *agents* × *agents* → *boolean*, which is a controlled function acting as a flag indicating whether an RREQ packet has been received.

They allow us to define the following predicates:

- `idle`: the agent is inactive, i.e. it does not need to start a new communication session and it is not involved in a route discovery process. This predicate is expressed by the following locations' values: *wishToInitiate*(**self**, *dest*) = false; *receivedRREQ*(**self**, *dest*) = false, with *dest* ∈ *agents*;
- `router`: the agent has received an RREQ, so it acts as a router supporting a route discovery process initiated by another host. It is expressed by the value true for *receivedRREQ*(**self**, *dest*), regardless of the value of *wishToInitiate*(**self**, *dest*);
- `initiator`: the agent has to start a new communication session, so acting as an initiator host. If the desired destination is not in its neighborhood and a route to it is not in its routing table, initiator starts a route discovery process. This predicate is expressed by the value true for *wishToInitiate*(**self**, *dest*), regardless of the value of *receivedRREQ*(**self**, *dest*).

When the MANET starts operating, each agent is idle, i.e. for each agent both *wishToInitiate*(**self**, *dest*) and *receivedRREQ*(**self**, *dest*) evaluate to false for each *dest*. During the normal execution of an agent, it can fulfill the `idle` predicate with respect to a destination, but at the same time it can fulfill different predicates for other destinations: the value of the parameter *dest* is used for distinguishing these cases.

In addition, each ASM includes the following functions:

- *neighb*: *agents* → PowerSet(*agents*), which is a monitored function specifying the nodes in the neighborhood of each agent. It is monitored because only the environment sets it;
- *routingTable*: *agents* → PowerSet(*records*), which is a controlled function representing the information about the nodes recorded into the agent's routing table. So, each *record* corresponds to an entry of the routing table.

The values of these two functions, as well as the set *agents*, depend on the particular scenario: they are dynamically set according to the MANET evolution, with respect to both the host mobility and the computational history. Moreover, in order to check if information about a host is stored into the agent's routing table, the derived function *hostInRT*: PowerSet(*records*) → PowerSet(*agents*) is defined: it returns the set of the agents stored in a given routing table.

Each agent is associated with two types of queues of messages: *requests* and *replies*, which include RREQ and RREP packets, respectively. This allows us to model sending/receiving of packets by means of enqueuing/dequeuing abstract messages into the corresponding queue. Both *requests* and *replies* are shared functions because updated by the corresponding ASM and by the other agents. These queues are managed by the derived function *isEmpty*, which states if a queue is empty or not, and by some specific ASM framework constructs: *enqueue*, *dequeue*, *empty*, and *top*, which adds an element to a queue, removes an element from a queue, removes all elements from a queue, and returns the top element of a queue, respectively.

Note that the *routingTable* function, even if controlled, indirectly depends on the value of the *requests* and *replies* queues. In fact, whenever a node receives an RREQ or an RREP, it updates its routing table according to the content of the received packet.

Each RREQ, RREP, or record is built by concatenation of the information listed in the description of AODV (in the pseudo-code the dot notation helps in identifying the value of a specific field of a packet).

The ASM pseudo-code of the *i*-th agent is:

```
AgentProgram($a_i$) =
    if ¬(isEmpty(requests(self))) then {
        dest = top(requests(self)).dest
        receivedRREQ(self, dest) := true
        Router(dest)
    }
    if wishToInitiate(self, dest) = true then
        Initiator(dest)
```

Informally speaking, each agent is inactive until its computation is solicited by the receipt of an RREQ or because a new communication session is required by the environment. Activation of an agent unfolds two different computational branches which lead to the execution of the *Router* or *Initiator* submachine. An instance of a new *Router* or a new *Initiator* submachine is created whenever a new route discovery process is needed or a new communication session is desired, respectively. It is worth noting that both submachines evolve concurrently, and in each of them rules are executed as soon as they become applicable.

The pseudo-code of the *Router* submachine is:

```
Router(dest) =
    if dest = self ∨ dest ∈ neighb(self) ∨
    dest ∈ hostInRT(routingTable(self)) then {
        init = top(requests(self)).init
        UnicastRREP(init)
        dequeue top(requests(self))
        receivedRREQ(self, dest) := false
    }
    else {
        BroadcastRREQ
        dequeue top(requests(self))
        receivedRREQ(self, dest) := false
    }
```

The *Router* submachine includes the `endRouting` predicate, which specifies that the execution of the routing activities due to the route discovery process is completed. This predicate is only expressed by the value false for the *receivedRREQ*(**self**, *dest*) function. If router is the destination of the received RREQ (*dest* evaluates to **self**) or if it knows a fresh route to *dest* (*dest* ∈ *neighb*(**self**) ∨ *dest* ∈ *hostInRT*(*routingTable*(**self**))), then it unicasts an RREP packet back to initiator. Otherwise, it re-broadcasts the RREQ to all its neighbors. In both cases the computation evolves to the state satisfying `endRouting` for that value of *dest*.

For the sake of brevity, the analysis of the "freshness" of information in routing tables is not described.

The pseudo-code of the *BroadcastRREQ* and *UnicastRREP* rules is:

```
BroadcastRREQ =
    forall n ∈ neighb(self) do {
        forall r ∈ requests(n) do {
            if RREQ.dest = r.dest ∧ RREQ.id =
            r.id then
                discard RREQ
        }
        increase RREQ.hopCount
        enqueue RREQ into requests(n)
    }
UnicastRREP(i) =
    r = top(requests(self))
    select c from routingTable(self) with
        c.dest = r.dest and c.destSeqNum ≥
        r.destSeqNum
    enqueue RREP with dest = c.dest and
        destSeqNum = c.destSeqNum
        into replies(i)
```

It is also worth specifying that, in order to reach its destination, an RREP packet must travel across several intermediate nodes, each of them executing some computation for updating routing tables. But, for simplicity, these functions have been abstracted away. The interested reader can find the full specification of the protocol and the prove of its correctness in [21].

The pseudo-code of the *Initiator* submachine is:

```
Initiator(dest) =
    if dest ∈ neighb(self) ∨ dest ∈
    hostInRT(routingTable(self)) then {
        StartCommunicationSession(dest)
        wishToInitiate(self, dest) := false
        sentRREQ(self) := false
    }
    else {
        BroadcastRREQ
        sentRREQ(self) := true
    }
    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    if sentRREQ(self) ∧ (isEmpty(replies(self)))
    then {
        select r from replies(self) with maximum
            destSeqNum
        StartCommunicationSession(dest)
        empty replies(self)
        wishToInitiate(self, dest) := false
        sentRREQ(self) := false
    }
    else if sentRREQ(self) ∧
    isEmpty(replies(self)) then
        skip
    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

In the pseudo-code above: **skip** means that no computational activity is executed when the corresponding rule fires; *BroadcastRREQ* behaves as well as in the *Router* submachine; *StartCommunicationSession*(*dest*) is not described because it is not strictly part of the protocol.

The *Initiator* submachine is characterized by two local functions: *sentRREQ*: *agents* → *boolean*, which is a controlled function indicating whether an RREQ has been sent; and the aforementioned *replies*. This means that a new queue of *replies* is instantiated for each specific communication session. This submachine includes additional states characterized by the following predicates over the states:

- `waiting`: it indicates that the agent is waiting for responses concerning that *dest* from the other agents. It is expressed by: *wishToInitiate*(**self**, *dest*) = true; *sentRREQ*(**self**) = true; *isEmpty*(*replies*(**self**)) = true;
- `endInitiating`: it indicates that the computational activities executed by initiator, con-

cerning the route discovery for that *dest*, are completed. It is expressed by the following locations' values: *wishToInitiate*(**self**, *dest*) = false; *sentRREQ*(**self**) = false.

If a route to *dest* is known, then the communication session simply starts; otherwise, *BroadcastRREQ* is executed. Its result consists in inserting a new RREQ into the *requests* queue of all the agent's neighbors and in satisfying the `waiting` predicate. When an RREP is received (i.e. *isEmpty*(*replies*(**self**)) evaluates to false), then the computation continues: the communication session starts and then the *replies* queue is emptied; otherwise nothing happens, i.e. the node doesn't take any action regarding this particular route discovery attempt, but it simply waits.

The dashed lines in the model above enclose the code fragment stating that the agent must wait until it does not receive an RREP corresponding to the RREQ previously sent. This is the "vulnerable area" of the algorithm which can make the agent starve, because it cyclically returns to states satisfying the `waiting` predicate. If no other ASM sends an RREP back to the agent, the function *isEmpty*(*replies*(**self**)) never changes its value to false, so the agent's computation cannot evolve. This simple observation suggests to focus on this area for changing the algorithm, so that the computation executed by an agent can continue.

Note that the starvation issue concerns only the route discovery process needed when initiator does not know a way to reach a specific destination. In all other cases the agent behaves normally: it processes all AODV control packets coming in, and is receptive to requests for other destinations.

One more comment concerns the permanence in the idle mode: it is not a case of starvation, but simply indicates that the agent does not need to execute any activity. In ASM terms: the system evolution only depends on the shared function *wishToInitiate*(**self**, *dest*) and on the derived function *isEmpty*(*requests*(**self**)). If none of them changes its value means that the agent is not proactive (it does not need to start a new communication) neither reactive (it has not received a request).

4.2 Refinement for Overcoming Starvation

The starvation issue of initiator can be solved by a refinement in accordance with the original formulation of the protocol [5], where the authors have introduced a timeout for escaping infinite waiting in case a route is not found within a specified amount of time. This solution drives to a timeout-based synchronization.

From the ASM point of view, the modification only affects the *Initiator* submachine. It consists in adding the local controlled function *timeout*: *agents* → *integers*, which models the maximum waiting time for an RREP, and the configuration of the `waiting` predicate now also includes *timeout*(**self**) > 0.

The change in the pseudo-code consists in adding the following rule:

```
if sentRREQ(self) ∧
¬(isEmpty(replies(self))) then {
    select r from replies(self) with maximum
        destSeqNum
    StartCommunicationSession(dest)
    empty replies(self)
    wishToInitiate(self, dest) := false
    sentRREQ(self) := false
}
if sentRREQ(self) ∧ isEmpty(replies(self)) ∧
¬(timeout(self) = 0) then
    timeout(self) := timeout(self) – 1
if sentRREQ(self) ∧ isEmpty(replies(self)) ∧
timeout(self) = 0 then {
    wishToInitiate(self, dest) := false
sentRREQ(self) := false
}
```

Note that the model does not include the notification to the environment about the timeout expiration. In order to acknowledge the impossibility to start the communication, the ASM can easily be refined, but this feature is outside our scope.

The proof of the absence of starvation in this refinement is quite straightforward. A generic agent a_i avoids starvation due to the waiting for one or more RREPs concerning *dest* when its computation is enabled to evolve to a state satisfying `endInitiating`. There are two paths for reaching this: when ¬(*isEmpty*(*replies*(**self**))) is satisfied or when *timeout*(**self**) = 0 is satisfied. The former corresponds to the exit from the vulnerable area because of the receipt of at least one RREP, and it is the same as in *sp*-ASM case. The latter is due to the refinement and states the exit from the same area because of the timeout expiration. In ASM terms, this happens when both *wishToInitiate*(**self**, *dest*) and *sentRREQ*(**self**) evaluate to false, regardless of the values of the other functions. This configuration is surely set within a finite amount of time, since, under the assumption that the initial value of *timeout* is greater than 0, it decreases at each iteration, so converging to 0.

5. DISCUSSION

The described experience on AODV allows us to derive some provisional results to address starvation. The vulnerable area in the *sp*-ASM is characterized by a rule whose condition depends on both the controlled function *sentRREQ*(**self**) and the derived function *isEmpty*(*replies*(**self**)), linked together by a logical conjunction. The former evaluates to true, so it does not affect the condition satisfaction, therefore it is important to focus on the latter. If the *replies* queue contains elements, then the computation evolves outside the vulnerable area; otherwise, the **skip** rule is executed, the values of the locations are unchanged, and the computation loops inside the vulnerable area.

Conversely, in the *sf*-ASM, the condition guarding the model fragment previously affected by starvation is the same as in the *sp*-ASM, but it is enriched with the controlled *timeout*(**self**) function (in logical conjunction to *sentRREQ*(**self**) and *isEmpty*(*replies*(**self**))). Moreover, the `waiting` predicate also considers the *timeout*(**self**) function. If the computation loops within this model fragment, the value of the locations changes because of the update decreasing the timeout, which surely will converge to 0. In this way it is guaranteed that the state of the ASM will change, so allowing the computation to evolve.

The above observations suggest that the area of the ASM subject to the risk of starvation is a *vulnerable rule*, of the form **if** *cond* **then** *update-true* **else** *update-false*, characterized by the following features:

1. The truth value of *cond* depends on one or more *risky* functions;
2. a) One update between *update-true* and *update-false* generates a computation that does not change the value of the predicate over the states representing the waiting issue, so determining a cyclical return to states characterized by the same predicate;

   b) The computation evolves to a subsequent state through the other update.

The functions in feature (1) are "risky" because the *risk* to starve the system depends on their values, which can be set by the ASM as well as by the environment and other ASMs in the same system. Concerning the class the risky functions belong to, let's note that out functions surely do not impact starvation, because their values are produced by the ASM, but never used. Secondly, monitored and shared functions surely need special attention, because their presence indicates that the ASM behavior is affected by the environment or the other agents in the same system. Furthermore, we cannot exclude the role of controlled functions in driving to starvation: their values can be managed by the ASM because set inside it, but they can be risky because their values can also depend on some monitored or shared function. If so, their usage in *cond* can be as risky as using monitored/shared functions: for example, if *c* is a controlled function depending on a monitored function *m*, and *c* is used in the rule guard, there is the same risk of starvation as *m* is used in the guard. Finally, more detailed analysis should be executed when derived functions appear in conditions of vulnerable rules. In fact, since these functions cannot be managed inside the ASM, but their values depend on other functions, the latter must be investigated.

Note that the values of controlled functions *indirectly* depend on other functions, and their updates are an effect of updates over the latter; instead, derived functions *directly* depend on other functions and their updates are automatic. Therefore, in order to investigate controlled and derived functions, their dependency graph must be analyzed.

Concerning feature (2a), an important issue is related to the granularity used for defining the states in which the computation cyclically returns: if it is finer, then the cyclical return can go through several intermediate states, but if it is coarser, the rule execution could not produce any appreciable change of the ASM state. Stepwise refinements intrinsic to the ASM Method make the definition at the desired granularity easy. Moreover, an adequate stepwise refinement can be defined so that the case of vulnerable areas including just one rule can be generalized to any finite number of rules, in which at least one rule satisfies the three features above. Finally, it is worth noting that the update allowing the computation to exit from the vulnerable area is the "good thing" stated in [1].

In the case of presence of vulnerable rules, in order to avoid the starvation risk, the algorithm must be restructured with the aim to guarantee that the agent does not stuck in an infinite loop. The timeout-based solution presented above is only one of the possible ways to reach this goal, but other methods can be suggested by the specific problem under study.

6. CONCLUSION

In this paper we have reported on an experience aimed at studying starvation risks in the Ad-hoc On-demand Distance Vector routing protocol for Mobile Ad-hoc NETworks using Abstract State Machines. The starvation-freedom property is an instance of the class of liveness properties, that are undecidable as well as safety properties. Nevertheless, these properties are semi-decidable and the results discussed in this paper aim at enforcing the ASM framework as a conceptual tool for studying them.

In order to investigate this issue, two formalizations of AODV have been given: the first one is starvation-prone because of the lack of a proper path between the initiator of a route discovery process and the desired destination; the second one is a refinement which makes the system starvation-free thanks to the adoption of a timeout, as stated in the original specification of the protocol. The analysis of both models allowed us to provide the ASM-based definition of vulnerable rules, for capturing starvation risks, upon which modelers must focus their attention. The obtained results are encouraging for the purposes of our research, because the entirely operational definition of vulnerable rules allows modelers to treat the analysis of starvation inside the ASM framework before adopting it in conjunction with *hybrid* model checking approaches.

From a methodological point of view, the vulnerable rules definition implicitly suggests some tasks a modeler can execute to determine the possible presence of starvation risks: analyze the model, looking for cyclical returns to states characterized by the same predicate over the states; if so, the modeler must check if it is driven by conditions whose truth value depends on some risky function; in this case, the corresponding updates must be studied for investigating their effects. Some of these activities could be supported by automatic tools, such as parsers or dependency graph analyzers.

Nevertheless, our approach also presents drawbacks. As other manual techniques, since it is human-based, it is error-prone and requires expertise in order to find an appropriate abstraction of the system to be verified. In other words, any analysis is as good as the model is. Furthermore, because of decidability issues, it cannot be completely automatized, even if, as previously mentioned, automatic tools can support it.

The research will continue with the aim to generalize the finding of the present paper and to formally prove the necessary conditions that enable starvation inside ASMs. In order to achieve this goal, we will deepen into the relationship between the syntactic notion of starvation-freedom inside ASMs and the semantic notion of starvation in literature.

REFERENCES:

[1] E. Kindler, "Safety and Liveness Properties: A Survey", *EATCS Bulletin*, 53, 1994, pp. 268-272.

[2] R. David and H. Alla, "Discrete, Continuous and Hybrid Petri Nets", *Springer-Verlag*, 2005.

[3] Y. Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms", *ACM Transactions on Computational Logic*, 1(1), 2000, pp. 77-111.

[4] B. Alpern and F.B. Schneider, "Defining Liveness", *Information Processing Letters*, 21(4), 1985, pp. 181-185.

[5] C.E. Perkins, E.M. Belding-Royer and S.R. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing", *RFC 3561*, 2003, http://tools.ietf.org/html/rfc3561.

[6] D.P. Agrawal and Q.A. Zeng, "Introduction to Wireless and Mobile Systems", *Thomson Brooks/Cole*, 2003.

[7] E. Börger and R. Stärk, "Abstract State Machines: A Method for High-Level System Design and Analysis", *Springer-Verlag*, 2003.

[8] W. Gabrisch and W. Zimmermann, "A Hoare-Style Verification Calculus for Control State ASMs", *Proceedings of the 5th Balkan Conference on Informatics*, 2012, pp. 205-210.

[9] R.F. Stärk and S. Nanchen, "A Logic for Abstract State Machines", *Journal of Universal Computer Science*, 7(11), 2001, pp. 981-1006.

[10] G. Del Castillo and K. Winter, "Model Checking Support for the ASM High-Level Language", *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2000, pp. 331-346.

[11] P. Arcaini, A. Gargantini and E. Riccobene, "AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications", *Proceedings of the 2nd International Conference on Abstract State Machines, Alloy, B and Z*, 2010, pp. 61-74.

[12] M. Spielmann, "Automatic Verification of Abstract State Machines", *Proceedings of the 11th International Conference on Computer Aided Verification*, 1999, pp. 431-442.

[13] Y. Gurevich, "A New Thesis", *American Mathematical Society Abstracts*, 1985, p. 317.

[14] W. Reisig, "The Expressive Power of Abstract State Machines", *Computing and Informatics*, 22, 2003, pp. 209-219.

[15] N. Dershowitz, "The Generic Model of Computation", *Electronic Proceedings of Theoretical Computer Science*, 2013.

[16] A. Blass and Y. Gurevich, "Abstract State Machines Capture Parallel Algorithms", *ACM Transactions on Computational Logic*, 4(4), 2003, pp. 578-651.

[17] A. Glausch and W. Reisig, "An ASM-Characterization of a Class of Distributed Algorithms", in J.R. Abrial and U. Glässer, eds., *Rigorous Methods for Software Construction and Analysis*, 2009, pp. 50-64.

[18] R. Farahbod, V. Gervasi and U. Glässer, "CoreASM: An Extensible ASM Execution Engine", *Fundamenta Informaticae*, 77(1-2), 2007, pp. 71-103.

[19] A. Benczur, U. Glässer and T. Lukovskzi, "Formal Description of a Distributed Location Service for Mobile Ad-hoc Networks", in E. Börger, A. Gargantini and E. Riccobene, eds., *Abstract State Machines 2003 – Advances in Theory and Applications*, 2003, pp. 204-217.

[20] A. Bianchi, S. Pizzutilo and G. Vessio, "Preliminary Description of NACK-based Ad-hoc On-demand Distance Vector Routing Protocol for MANETs", *Proceedings of the 9th International Conference on Software Engineering and Applications*, 2014, pp. 500-505.

[21] A. Bianchi, S. Pizzutilo and G. Vessio, "Suitability of Abstract State Machines for Discussing Mobile Ad-hoc Networks", *Global Journal of Advanced Software Engineering*, 1, 2014, pp. 29-38.

[22] A. Bianchi, S. Pizzutilo and G. Vessio, "Applying Predicate Abstraction to Abstract State Machines", *Enterprise, Business-Process and Information Systems Modeling*, LNBIP 214, Springer, 2015, pp. 283-292.

[23] Y.N. Lien, H.C. Jang and T.C. Tsai, "A MANET Based Emergency Communication and Information System for Catastrophic Natural Disasters", *Proceedings of the 29th International-Conference on Distributed Computing Systems Workshops*, 2009, pp. 412-417.